Mixed Inductive-Coinductive Reasoning

Types, Programs and Logic

Proefschrift
ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op donderdag 19 april 2018
om 16:30 uur precies

door

Henning Basold

geboren op 16 juni 1986
te Braunschweig (Duitsland)

Promotoren
Prof. dr. Jan Rutten
Prof. dr. Herman Geuvers

Copromotor
Dr. Helle Hvid Hansen (Technische Universiteit Delft)

Manuscriptcommissie
Prof. dr. Bart Jacobs (Voorzitter)
Prof. dr. Neil Ghani (University of Strathclyde, Verenigd Koninkrijk)
Prof. dr. Tarmo Uustalu (Tallinn University of Technology, Estland)
Dr. Andreas Abel (Chalmers University of Technology en Götenborgs Universitet, Zweden)
Dr. Ekaterina Komendantskaya (Heriot-Watt University, Verenigd Koninkrijk)

Radboud University

Centrum Wiskunde & Informatica

This research has been carried out under the auspices of the iCIS (institute for Computing and Information Science) of the Radboud University Nijmegen, the Formal Methods Group of the CWI, and the research school IPA (Institute for Programming research and Algorithmics).

Netherlands Organisation for Scientific Research

This research has been supported by NWO (grant number 612.001.021).

ISBN 978-0-244-67206-5
Typeset with XeLaTeX
Printed and published by Lulu

Contents

Acknowledgements

Ich sage: laßt alle Hoffnung fahren, ihr, die ihr in die Beobachtung eintretet.
(Eng.: I say, abandon all hope, you who enter the realm of observation.)

— Galileo Galilei in Bertolt Brechts "Leben des Galilei", Akt 9.

Before we start with the technical content of the thesis, I would like to thank a few people that contributed to its development in one way or another.

First and foremost, I would like to thank my supervisor Helle Hvid Hansen. Her infinite patience, her vast knowledge of scientific topics and the English language, and her attention to detail made me not only a better researcher but also a much better writer. Before I started my PhD, I did an internship at the CWI in Amsterdam with Marcello Bonsangue and Jan Rutten, who then also became my promoter. It was there that I met Helle the first time and we started to have scientific and non-scientific conversations. We also shared good evenings outside of work, through which I learned about very nice restaurants in Amsterdam, and one or two tricks in the kitchen. Both our discussions and Helle's incredibly detailed feedback improved my research, mathematical and writing skills tremendously, and without them, my thesis would be far worse than it is now.

As I already mentioned, I did an internship with Jan before he became my promoter. Already at that time it became clear that Jan finds a good balance between giving one a lot of freedom and guidance. But not just that, Jan has also the ability, which amazes me every time, to make suggestions that open up new paths or lead to vast simplifications. This impression proved to be right, and I am greatly indebted to Jan in his role as my supervisor. Without him, I would have neither arrived at the research topics of my thesis nor at all where I am today. His constant support, his suggestions, feedback and his ability to listen are invaluable.

Last but not least, Herman Geuvers entered the team when my research shifted towards type theory. I learned a lot from Herman about logic and type theory. He has an enormous knowledge about these fields, and I could always come by and get an extensive answer. If the answer would be too complicated to be answered on the spot, he would go to his cabinet or bookshelf and take a printed publication, sometimes his own, that would answer the question. Indeed, the strong normalisation proof in the last chapter would not be there, had he not given me his '94 paper on strong normalisation of the calculus of constructions. Making a long story short, Herman is another cornerstone in my development as a researcher, and I would like to wholeheartedly thank him for everything he did and the positive atmosphere he creates.

Overall, I am equally indebted to all of you, Helle, Jan and Herman, for your support, your kindness and what you have taught me. You were the best team of supervisors that I could envision!

I would like to thank other people that have directly contributed to the content of this thesis. First, there is the reading committee consisting of Andreas Abel, Neil Ghani, Bart Jacobs, Ekaterina Komendantskaya and Tarmo Uustalu. I am grateful for all the time and effort they put into reviewing this, fairly lengthy, thesis and all the suggestions they made. In particular, I am happy that a fundamental flaw was found by Andreas before publication.

Next, there are all the people that I have collaborated with on publications: Damien, Helle, Henning G., Herman, Jan, Jean-Éric, Jurriaan, Katya, Marcello, Michaela, Niels and Stefan. I would like to thank all of them for being fantastic collaborators.

Finally, there are a few other people that I would like to mention because they had a direct influence on the content and even the existence of this thesis. I am especially indebted to Stefan Milius, who brought me into contact with Jan and thus opened up the path to my PhD. This resulted in an internship with Jan and Marcello. During this internship and afterwards, I had some outstanding discussions with Marcello, which resulted in my first publication. I am grateful for Marcello's scientific guidance, which laid the foundations for my later work in theoretical computer science, particularly in the field of coalgebra.

Science demands a certain amount of dedication, a demand that can be very high at times and one that can only be fulfilled if we have people of support and collaboration around us. It is these people that I would like to dedicate the opening quote to. Brecht derived it from the inscription "Abandon all hope, ye who enter here." above the entrance to hell in Dante Alighieri's *La Divina Commedia*. The phrase is used by Galilei in Brecht's play when he picks up again his studies on the rotation of the sun and describes his approach to science: Work slowly, question everything, repeat experiments and compare the outcomes, distrust everything that fits your beliefs, and only accept a result if all other possibilities can be excluded. This is a daunting task, which can not only be lonely at times but also carries the danger of becoming ignorant of the surrounding world. However, I was very lucky and met some fantastic people along the way, who reminded me about what is important in life and who were willing to share this daunting task. I mentioned my scientific collaborators and guides already above, but there are further important people in my life.

In particular, I would like to thank Pauline Chew, Simone Lederer and Jurriaan Rot. Pauline was always around with continuous support and for, often intense, conversations over food or late at night. Thank you, Pauline, for all this and, above all, for making me grow as a person. With Simone I had a perfect travel partner, who was always up for spontaneous activities in the weekend and for good conversations. To you also a big thank you, Simone, for the good time and the different perspectives you offered. And, I would like to thank you both, Pauline and Simone, for being my paranimphs. I met Jurriaan at the CWI, from where we became friends and research partners. Among the many interests that we share, our running sessions and bike rides, cooking and, of course, coalgebras are the most important ones to me. Thank you, Jurriaan, for all the good cooperation, and for your help with my thesis and articles.

The next shout goes out to my friends in Nijmegen, with whom I spent a great deal of my spare time, be it at dinners, watching films, going to concerts or to museums: Alexis, Dario, Elena, Gabriel, Jacopo, Joshua, Michael, Michele, Pauline, Robbert, Rui Fei, Simon, Sjef, Steffen and Tim, and my flatmate Katja. Some friends I also found in Amsterdam, like Nick, who was my flatmate for two years, and with whom I shared meals and good conversations in the evenings. Sung is another great person that I met at the CWI. We had many discussions about Reo and life in general. Thank you all for the pleasant time I had in the Netherlands, each of you knows what we share.

Among the fantastic people I met, there are also those that I am lucky to have or have had as colleagues. My first job was the civil service in the hospital in Braunschweig, where I found in Thomas Joosten and Jürgen Feß people, who supported me in my choice to study but also made me aware of possible pitfalls. I am thankful that I could work before and during my studies with Udo Hanfland and the people at BBR in Braunschweig, where I learned a lot about programming, project management and collaboration.

At the university in Braunschweig, I am indebted to Jiří Adámek, Michaela Huhn and Stefan Milius for their supervision of my master's thesis. Also, I would like to thank Rainer Löwen for his lectures,

which shaped my mathematical mind, and Fiona Gottschalk, Nadine Hattwig, Sebastian Struckmann and Kristof Teichel for the good time we had racking our brains over (algebraic) topology.

The next stage was my time at the CWI, where I met many great people. The first was probably Alexandra Silva. I owe her a lot, as she brought me in contact with Nick, gave us some basic furniture and later often provided me with shelter in Nijmegen (thanks also to Neko for being such a good host!). Besides being an excellent cook, Alexandra was a good friend and colleague in Nijmegen. At CWI, I also got in contact with Matteo Mio with whom I have, on and off, good discussions about mathematics, logic, geeky topics but also some nice evenings out. Also Enric Cosme-Llópez was over at CWI as visitor and we had, both, in Amsterdam and in Lyon, a few good evenings. Since we are at it, I would like to thank also all the other people that I met at CWI: Dominik, Erik, Farhad (especially for the stories and film recommendations!), Frank (for the jokes and famous FM dinners), Joost, Julian, Kasper, Michiel, Nikos, Stijn and Vlad.

At the Radboud University itself, I crossed paths with many people that I would like to thank as well. One part of working at the university is teaching. I was lucky enough to assist in the combinatorics course given by Engelbert, who is a fabulous teacher and prepares everything so meticulously that there is nothing more that an assistant could wish for. Another part of working at the university is research and writing. I would like to thank Bart for his scientific and initial financial support, for his feedback and for managing the reading committee of my thesis. The last part of working is, of course, the environment. I am grateful for all the people at the Radboud university that made working there a pleasant experience: Aleks, Arjen, Baris, Bart, Bas S., Bas W., Bram, Camille, Dan, Elena M., Elena S., Engelbert, Erik, Fabian, Fabio, Freek, Frits, Gabriel, Guillaume, Hans, Henk, Herman, Ingrid, Irma, Jacopo, Jonce, Josef, Joshua, Jurriaan, Kasper, Kenta, Maaike, Markus, Mathys, Matteo S., Max, Maya, Michael, Michiel, Mohsen, Nicole, Niels, Paul, Perry, Peter, Ralf, Rick, Ridho, Robbert, Robert, Robin, Ronny, Ruifei, Saskia, Simone L., Simone M., Suzan, Tim, Tom C., Tom H., Twan, Zaheer. In particular, I would like to thank the Data Science group that I could still feel being a part of, even though I was technically in another group after the reorganisation.

And then there are the people that I met "in the wild", that is, at conferences, workshops or other occasions. I am happy to be part of the coalgebra and TYPES community and would like to thank them for their very warm and welcoming atmosphere. Apart from the people that I have mentioned already, I would like to especially thank Andreas for our discussions that led me to the topics of the last two chapters; Clemens for hosting me in Edinburgh and for being a friend in general; Filippo for being a great roommate on Barbados and for initialising my contact to work in Lyon; Katya for our discussions and her invitations to Scotland; Neil for our joint workshop and his insights into category theory; Prakash for bringing together researchers through workshops and SIGLOG; Tarmo for my fantastic first experience of TYPES in Tallinn and our discussions; and finally Daniela for being a perfect flatmate and friend, who gave me also many scientific insights.

I want to end by coming back to my roots in Braunschweig, where I was happy to have found some very good friends: Anja, Christoph, Christian, Lea & Patrick and Philipp & Kathrin. Thank you all for your friendship and the great time we spent together. I would also like to express my gratitude in memory to Rudolf, Anselm, Inge, Helga & Alfred. And finally, I would like to thank my family, my parents and Virginie for their support and love.

Henning Basold, February 2018, Lyon.

Introduction

> *Thought must never submit, neither to a dogma, nor to a party, nor to a passion, nor to an interest, nor to a preconceived idea, nor to whatever it may be, save to the facts themselves, because, for thought, submission would mean ceasing to be.*
>
> — Henri Poincaré, 1909.[1]

The purpose of this thesis is to systematically study languages for specifying and reasoning about mixed inductive-coinductive structures and processes. We will focus mostly on type theoretic and category theoretic languages, but some of the reasoning principles are based on standard bisimulations and up-to techniques. In the course of this thesis, we will analyse existing simple type theories that allow the specification of inductive-coinductive processes, and we will exhibit several reasoning principles for these type theories: through category theory, by using coinductive predicates and relations combined with up-to techniques, in form of a logic, and in certain cases by automatic decision procedures. Moreover, we will develop a dependent type theory, both category theoretically and as a syntactic calculus, that is based solely on inductive-coinductive types. As we will see, this type theory can serve as a framework for quite general inductive-coinductive definitions and proofs, and forms the pinnacle in expressivity of this thesis.

In the remainder of this introduction, we will motivate the study of inductive-coinductive reasoning and provide an overview of the developments that happen throughout this thesis. To illustrate why it is important to study inductive-coinductive reasoning in its own right, we will first discuss an example of an inductive-coinductive property that pervades this thesis and that illustrates beautifully how inductive-coinductive reasoning arises naturally in Mathematics and Computer Science. Afterwards, we will give a brief historical overview, discuss the problems that will be tackled in this thesis and detail the approach that we take. We will finish the introduction with a discussion of the contributions and outline of this thesis.

Background and Motivation

Are you sitting comfortably? Then let us dive into the story of mixed induction-coinduction and how it can help us in the practice of Mathematics and Computer Science.

Induction and coinduction are threads that cross the landscape of Mathematics and Computer Science as methods to define objects and reason about them. Of these two, induction is by far the better known technique, although disguised coinduction has always been around. It was only in recent years that we began to see through this disguise and developed coinduction as a technique in its own right. This led to some remarkable theory under the umbrella of coalgebra and to striking applications of coinduction.

One of the topics in coalgebra that is actively researched are so-called *behavioural differential equations* (BDE) [Rut03; Rut05], which allow for very concise and intuitive process specifications, analogous to the differential equations from Mathematical Analysis. However, BDEs also inherit

from their analytic counterpart the problem that a system of equations may specify impossible behaviour, and therefore may not have a unique solution or may have no solution at all. The starting point of the research, which led up to this thesis, was thus to extend the known formats for BDEs to cover wider application areas, while guaranteeing that the specified behaviour in these more general formats is still well-defined. Once we have the ability to specify processes, it is natural to also investigate reasoning principles for such processes to, for example, to be able to compare their behaviour. Generalising process specifications and exhibiting reasoning principles for these processes were the two main goals of the NWO project "Behavioural Differential Equations" (612.001.021), in which part of the research for this thesis has been conducted. So how does mixed induction-coinduction fit into this agenda?

As it turns out, induction and coinduction are complementary techniques, they are *dual* in a precise sense that we will discuss later. Being complementary, it is often necessary to use both techniques or even intertwine them. The combination of induction and coinduction allows us, for instance, to construe forms of behavioural differential equations whose expressiveness exceeds that of the currently available forms. We will also see that combined induction-coinduction is often used implicitly, just like induction and coinduction used to be. Thus, the purpose of this thesis is to systematically study the combination of induction and coinduction, which I hope, if anything, inspires others to work on and use inductive-coinductive techniques.

The perspective that we will take in this thesis is that of logic and computation. However, this should not limit the applicability of the results presented in this thesis to Computer Science. Rather, the logics developed here are largely independent of the field, as are many of the ideas. I think that many objects that occur in Mathematics, Computer Science, and other branches of science, in which we build formal models, lend themselves to a mixed inductive-coinductive description. Therefore, I wish to demonstrate, besides presenting general theory, also the use of inductive-coinductive objects in an accessible way. That being said, some parts of this thesis presuppose knowledge of category theory (Section 4.2 and Chapter 6) and an understanding of dependent type theory (Chapter 6 and Chapter 7). To support a reader unfamiliar with any of these, we provide in Section 6.1 a short introduction to dependent type theory, and introduce some notations and non-standard bits of category theory in Chapter 2. Despite these requirements, I hope that the reader may still find pleasure in reading this thesis, and may obtain new insights from examples like that in Section 7.5.

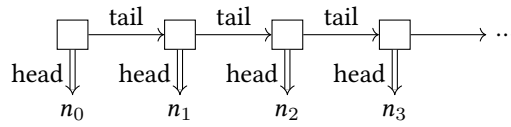
Induction and Coinduction

So what are induction and coinduction? And how can their combination contribute to our understanding of problems in Mathematics and Computer Science and help solving these problems? The way we will approach these questions is through the slogan that[2]

> inductive objects describe terminating computations and values, whereas coinductive objects describe the behaviour of observable processes.

Consider, for instance, natural numbers and infinite sequences, from here on called streams, of natural numbers. The former illustrate the idea of *terminating computations* that result in *values*: every computation on natural numbers terminates in either *zero* or the *successor* of another natural number. For instance, the number "1" is a value, since it is the successor of zero, whereas "$1 + 2$" is not a value but it computes to the value "3". Streams of natural numbers, on the other hand, are

observable processes. To illustrate this, let us picture a box with a screen labelled "head" that displays a natural number and a button labelled "tail". If we push the button "tail", the state of the box may change and a new number may be displayed on the box. We can observe the behaviour of the box by repeatedly pushing the "tail" button and noting down all the numbers that we see, thereby obtaining an arbitrarily long sequence of numbers. This is illustrated in the following diagram, where each square represents the unknown state of the box, the double arrows labelled "head" point to the displayed values, and "tail"-labelled arrows signify state changes through button pushes.



In fact, we can use these descriptions to characterise natural numbers and streams as follows. First of all, for a given type[3] X, we will write $x : X$ if x is an element of type X. Let N be a type of natural numbers and N^ω a type of streams over natural numbers, both of which we will characterise now without further specifying their internal structure.[4] We have already said that the natural numbers are characterised by having an element zero and a successor map, that is, there is an element $0 : \mathrm{N}$ and a map $\mathrm{suc} \colon \mathrm{N} \to \mathrm{N}$. In contrast to this, the streams come with two maps $\mathrm{hd} \colon \mathrm{N}^\omega \to \mathrm{N}$ and $\mathrm{tl} \colon \mathrm{N}^\omega \to \mathrm{N}^\omega$ that allow us to obtain the head and the tail of a stream. Note that the structure on N allows us to construct elements of N, hence we will refer to 0 and suc generally as *constructors*. Streams have the *dual* property, their structure is determined by maps out of N^ω, the *observations* hd and tl that we can make on streams. Suppose now that X is a type that comes with a distinguished element $x_0 : X$ and a map $s \colon X \to X$. We will compactly denote this as a triple (X, x_0, s) and call it an *algebra*. What makes N special among such algebras is that there is a unique map $f \colon \mathrm{N} \to X$, such that, for all $n : \mathrm{N}$,

$$f(0) = x_0 \qquad \text{and} \qquad f(\mathrm{suc}(n)) = s(f(n)). \tag{1.1}$$

This is summed up by saying that $(\mathrm{N}, 0, \mathrm{suc})$ is *initial* among all algebras $(X, x_0 : X, s \colon X \to X)$. Dually, streams are characterised by the property that $(\mathrm{N}^\omega, \mathrm{hd}, \mathrm{tl})$ is *final* among all *coalgebras* $(Y, h \colon Y \to \mathrm{N}, t \colon Y \to Y)$, which means that there is a unique map $g \colon Y \to \mathrm{N}^\omega$ such that, for all $y : Y$

$$\mathrm{hd}(g(y)) = h(y) \qquad \text{and} \qquad \mathrm{tl}(g(y)) = g(t(y)). \tag{1.2}$$

We formulate now the properties of inductive and coinductive types more abstractly. Let $\mathbf{1}$ be a type with the property that the point x_0 is equivalently[5] given by a map $z \colon \mathbf{1} \to X$. If the types are sets, then $\mathbf{1}$ is a set with just one element, say $* : \mathbf{1}$, and we would define $z(*) = x_0$. Similarly, we require that $0 : \mathrm{N}$ is also given as a map $\mathrm{zero} \colon \mathbf{1} \to \mathrm{N}$. These assumptions allow us to express (1.1) more elegantly in terms of composition of maps:

$$f \circ \mathrm{zero} = z \qquad \text{and} \qquad f \circ \mathrm{suc} = s \circ f. \tag{1.3}$$

To express (1.2) in this way, we do not need any further assumptions:

$$\mathrm{hd} \circ g = h \qquad \text{and} \qquad \mathrm{tl} \circ g = g \circ t. \tag{1.4}$$

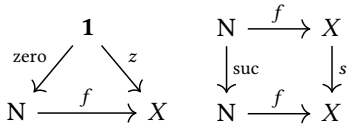
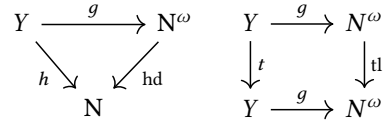
	Natural Numbers (Initial Algebra)	Streams (Final Coalgebra)
Structure	Constructors: zero and successor $\mathrm{zero}\colon \mathbf{1} \to \mathrm{N} \qquad \mathrm{suc}\colon \mathrm{N} \to \mathrm{N}$	Observations: head and tail $\mathrm{hd}\colon \mathrm{N}^\omega \to \mathrm{N} \qquad \mathrm{tl}\colon \mathrm{N}^\omega \to \mathrm{N}^\omega$
UMP	$\dfrac{x_0 : X \qquad s : X \to X}{f : \mathrm{N} \to X}$  	$\dfrac{h : Y \to \mathrm{N} \qquad t : Y \to Y}{g : Y \to \mathrm{N}^\omega}$ 

Table 1.1.: Defining properties of natural numbers and streams

Let us collect the defining properties of natural numbers and streams: natural numbers are determined by *constructors* and a *universal mapping property* (*UMP*) for maps *out* of N, whereas streams are determined by their *observations* and a UMP for maps *into* N^ω. This is summed up in Table 1.1, where we express the existence of the maps f and g as proof rules and the equations (1.3) and (1.4) as commuting diagrams.

To describe in general terms what initial algebras and final coalgebras are, we will work with a *category* $\mathbf{C}$, which is a collection of *objects* and *maps* (also called *morphisms*) between them.[6] We write then $X : \mathbf{C}$, if X is an object of $\mathbf{C}$, and $f\colon X \to Y$, if f is a map from X to Y in $\mathbf{C}$. The integral feature of categories is that we can compose maps of matching type: Given maps $X \xrightarrow{f} Y \xrightarrow{g} Z$ in $\mathbf{C}$, there is a composed map $g \circ f\colon X \to Z$ in $\mathbf{C}$. This composition should also resemble the structure of a monoid, in the sense that it is associative, and that for each object $X : \mathbf{C}$ there is a distinguished map $\mathrm{id}_X\colon X \to X$, such that for all $f\colon X \to Y$, we have $f \circ \mathrm{id}_X = f$ and $\mathrm{id}_Y \circ f = f$. The last bit of lingo that we need is that of a functor. A *functor* is a map $F\colon \mathbf{C} \to \mathbf{D}$ between categories $\mathbf{C}$ and $\mathbf{D}$ that maps objects and maps in $\mathbf{C}$ to objects and maps in $\mathbf{D}$, while preserving the types of maps: If $f\colon X \to Y$ is a map in $\mathbf{C}$, then $F(f)\colon F(X) \to F(Y)$ is a map in $\mathbf{D}$. Moreover, such a functor must preserve identities and composition, analogous to monoid homomorphisms, which means that $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ and $F(g \circ f) = F(g) \circ F(f)$. Using the language of category theory, we can now describe abstractly what inductive and coinductive objects are.

In general, *induction* arises from initial algebras and *coinduction* from final coalgebras. Let $\mathbf{C}$ be a category and $F_1, \ldots, F_n$ be functors with $F_k\colon \mathbf{C} \to \mathbf{C}$. An *algebra* for these functors[7] is a tuple $(X, a_1, \ldots, a_n)$, where $X : \mathbf{C}$ and $a_k\colon F_k(X) \to X$. We say that an algebra $(\Theta, c_1, \ldots, c_n)$ is *initial*, if for each algebra $(X, a_1, \ldots, a_n)$ there is a unique map $f\colon \Theta \to X$, such that for all $k = 1, \ldots, n$ the following diagram commutes.

$$\begin{array}{ccc} F_k(\Theta) & \xrightarrow{F_k(f)} & F_k(X) \\ \downarrow{\scriptstyle c_k} & & \downarrow{\scriptstyle a_k} \\ \Theta & \xrightarrow{\quad f \quad} & X \end{array} \tag{1.5}$$

In this case, we refer to the maps c_k as *constructors*. Dually, a *coalgebra* is a tuple $(Y, t_1, \ldots, t_n)$ with

$Y : \mathbf{C}$ and $t_k \colon Y \to F_k(Y)$, and $(\Omega, d_1, \ldots, d_n)$ is *final*, if for any coalgebra $(Y, t_1, \ldots, t_n)$ there is a unique map $g \colon Y \to \Omega$ that makes the following diagram commute for $k = 1, \ldots, n$.

$$\begin{array}{ccc} Y & \xrightarrow{\quad g \quad} & \Omega \\ \downarrow{\scriptstyle t_k} & & \downarrow{\scriptstyle d_k} \\ F_k(Y) & \xrightarrow{F_k(g)} & F_k(\Omega) \end{array} \tag{1.6}$$

Consequently, we call the maps d_k of a final coalgebra *observations*. Returning to the original terminology, initial algebras are inductive objects, while final coalgebras are coinductive objects. Moreover, it is fairly easy to see that the natural numbers object, as we described it in Table 1.1, forms with zero and suc an initial algebra for the functors $F_1, F_2 : \mathbf{C} \to \mathbf{C}$ with

$$\begin{aligned} F_1(X) &= \mathbf{1} & F_2(X) &= X \\ F_1(f) &= \mathrm{id}_X & F_2(f) &= f, \end{aligned}$$

and that the streams object is final for the functors $G_1, G_2 \colon C \to C$ with

$$\begin{aligned} G_1(X) &= \mathrm{N} & G_2(X) &= X \\ G_1(f) &= \mathrm{id}_{\mathrm{N}} & G_2(f) &= f. \end{aligned}$$

For instance, given an algebra (X, z, s), the equation that arises from (1.5) for the constructor zero of the natural numbers is

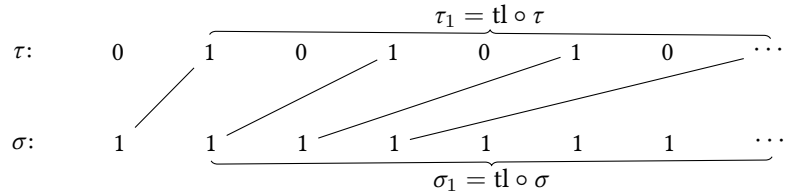
$$f \circ \mathrm{zero} = z \circ F_1(f).$$

Since $F_1(f) = \mathrm{id}_X$ and by the laws of the category $\mathbf{C}$, this equation reduces to $f \circ \mathrm{zero} = z \circ \mathrm{id}_X = z$, which is the identity that we have already encountered in (1.3).

Weaving Together Induction and Coinduction

So far, we have explored inductive and coinductive objects separately. Let us now take a look, again with a computational angle, at an example that crucially combines inductive and coinductive objects.

Suppose we ask ourselves whether a stream σ is contained in another stream τ, that is, whether all the entries in σ occur in order in τ. We say that σ is a *substream* of τ. For instance, a stream that only consists of ones is certainly a substream of a stream that alternates between zero and one. We can display their relation pictorially as follows, where we draw a line between entries in σ and τ to show how they need to be related, in order to prove that σ is a substream of τ.



From this picture, we can extract a process for showing that σ is a substream of τ: The first entry in σ is 1, which we can match with a 1 in τ by skipping the first entry in τ. Thus, we have $\mathrm{hd} \circ \sigma = \mathrm{hd} \circ \tau_1$, where $\tau_1 = \mathrm{tl} \circ \tau$. Having found the first match, we continue with the second entry in σ. Indeed, we can match the second entry by $\mathrm{hd} \circ \sigma_1 = \mathrm{hd} \circ \tau_2$, where $\sigma_1 = \mathrm{tl} \circ \sigma$ and $\tau_2 = \mathrm{tl} \circ \mathrm{tl} \circ \tau_1$. This process can be continued indefinitely, hence it is a coinductive process. However, we note that it is important that finding an entry of σ in τ must not be an infinite process, for otherwise the entry would actually *not* be found. In other words, matching one entry in σ with one in τ is a terminating computation, thus inductive.

We can define the substream relation as follows. Let us, for reasons of brevity, say that σ is a stream in N^ω, if $\sigma : \mathbf{1} \to \mathrm{N}^\omega$ and we write also $\sigma : \mathrm{N}^\omega$. Moreover, let $\mathrm{Rel}(\mathrm{N}^\omega, \mathrm{N}^\omega)$ be the set of binary relations between elements of N^ω. The substream relation, which we will denote by $\leq$ here, will be defined mutually with another relation $\trianglelefteq : \mathrm{Rel}(\mathrm{N}^\omega, \mathrm{N}^\omega)$. These two relations together implement exactly the matching process that we outlined above: The inductive part comes about by saying that $\sigma \trianglelefteq \tau$, if we can obtain a stream τ_1 by dropping a finite prefix of τ, such that $\mathrm{hd} \circ \sigma = \mathrm{hd} \circ \tau_1$. The coinductive relation $\leq$ repeats this process indefinitely. Formally, the substream relation $\leq$ is the *largest* relation $\leq : \mathrm{Rel}(\mathrm{N}^\omega, \mathrm{N}^\omega)$, such that

$$\forall \sigma, \tau : \mathrm{N}^\omega . \, \sigma \leq \tau \to \sigma \trianglelefteq \tau,$$

and $\trianglelefteq : \mathrm{Rel}(\mathrm{N}^\omega, \mathrm{N}^\omega)$ is the *least* relation with

i) $\forall \sigma, \tau : \mathrm{N}^\omega . \, (\mathrm{hd} \circ \sigma = \mathrm{hd} \circ \tau) \wedge (\mathrm{tl} \circ \sigma \leq \mathrm{tl} \circ \tau) \to \sigma \trianglelefteq \tau$ and

ii) $\forall \sigma, \tau : \mathrm{N}^\omega . \, (\sigma \trianglelefteq \mathrm{tl} \circ \tau) \to \sigma \trianglelefteq \tau$.

The clause for $\leq$ says that if σ is a substream of τ, then σ is finitely matched by τ. On the other hand, the clauses of the finite matching relation $\trianglelefteq$ state that $\sigma \trianglelefteq \tau$, either if their heads match and the tail of σ is again a substream of the tail of τ, or if σ finitely matches τ after dropping the first entry from τ. Since $\trianglelefteq$ is the least relation closed under these clauses, it indeed only allows finitely many dropping steps.

There are, of course, some subtleties in this definition and how to use it. Thus, we need a framework for inductive-coinductive definitions and reasoning, which prevents us from running into problems that are caused by those subtleties. Since, as we will see below, there are no mathematical frameworks that support inductive-coinductive reasoning to full extent, we set out in this thesis to develop such frameworks. In the end, we will have the ability to reason about the substream relation formally and work with it in an intuitive way. For instance, if 1^ω is the stream of only ones and alt is the stream alternating between 0 and 1, then an easy application is to show $1^\omega \leq \mathrm{alt}$ by simultaneously showing that $1^\omega \trianglelefteq \mathrm{alt}$ and $1^\omega \trianglelefteq \mathrm{tl} \circ \mathrm{alt}$.

Overall, this is an example that shows how inductive and coinductive reasoning naturally occur together and complement each other. These are the kinds of examples that motivate and drive the developments in this thesis.

Related Work and Origins

Given this motivation, let me sketch the evolution of induction and coinduction, and why we are only at beginning of the evolution of mixed induction-coinduction. This also allows us to show how this thesis fits into the existing literature.

The Orgins of Induction

Induction, as a proof technique on natural numbers, has been around implicitly since ancient times, for instance in a proof by Euclid of the infinitude of the prime numbers, but was only made explicit in the 17th century by Bernoulli. In the early 19th century it was popularised under the name "Mathematical Induction" [Caj18], which is still used to this date, though we will refer to it in this thesis just as induction. The name "Mathematical Induction" was chosen to distinguish it from the argumentation method of "induction", as it is used in the natural sciences. However, the mathematical world had to wait until the end of the 19th century for a truly rigorous formulation of induction. One was given by Dedekind [Ded88] in, what we would call today, second-order logic and the other by Peano [Pea89] as a scheme in first-order logic. From there on, induction on natural numbers became a standard technique in Mathematics, and was further developed into transfinite or well-founded induction. A principal problem of restricting induction to natural numbers is that one has to code other inductive structures, like finite trees, as natural numbers. This process is sometimes referred to as "Gödelisation" because an integral part of Gödel's incompleteness proof relies on coding formulas as natural numbers. With the advent of Computer Science it was realised that functions on inductive objects could be defined directly by "structural recursion" [Pét61] and that properties could be proved by "structural induction" [Bur69]. Throughout this thesis, we will refer to the former here just as *iteration* and the latter as *induction*, no matter what the underlying inductive object is.

The Orgins of Coinduction

Just like induction, also coinduction has gone unnoticed for a long time. Most prominently, the methods for proving the equivalence of states in automata theory or worlds in modal logic featured concepts that are instances of coinduction. But even the function extensionality principle, namely that two functions are equal if and only if they are equal on every argument, is an instance of coinduction, as we will see. An important step towards exposing bisimulations was Milner's work on concurrency [Mil80], where he gives an inductive definition of bisimilarity. The crucial step to exhibit the coinductive nature of bisimilarity was only taken by Park [Par81], who showed how bisimilarity can be constructed as a largest fixed point for a certain monotone function. Also set-theorists became interested in coinductive concepts, in the beginning of the 20th century only implicitly but later more explicitly. The goal there was to overcome the restrictions of the axiom of foundation, which ensures that the membership relation is well-founded. The problem then is to recover the notion of extensionality, namely that two sets shall be considered equal precisely when all their elements are equal. Aczel and Mendler [AM89] and Barwise et al. [BGM71] discovered that extensionality in non-well-founded set theory can be stated through coinduction, see also [BM96]. What really stands out in the work of Aczel [Acz88] is the use of *final coalgebras* and a category theoretical definition of bisimilarity [AM89]. From here on, coalgebras were studied in their own right as a theory of processes or systems, with Rutten [Rut00] laying the ground work for a systematic study of coalgebras and their properties. For insights into the historical development of coinduction, the reader may consult [San09], which focuses on modal logic, non-wellfounded set theory and order theory. The history of coalgebras is further discussed in [Rut00] and in the introductory texts [JR97; JR11; and Jac16].

After coinduction and coalgebras had been developed, people strived to improve the definition

and proof principles associated with them. Two strands, which are the ones important to us here, are the development of *behavioural differential equations* and *up-to techniques*. We have mentioned behavioural differential equations (BDE) already earlier as the starting point of this thesis. They were developed by Rutten [Rut03], building on the idea of input derivatives by Brzozowski [Brz64] and Conway [Con71], which fully specify the behaviour of certain systems. BDEs allowed Rutten to build a coinductive calculus of streams and formal power series, which he then uses to manipulate solutions of some analytic differential equations [Rut05]. This last step is an elegant reformulation of results by Pavlovic and Escardó [PE98]. The idea of using BDEs as process specification language was picked up and further developed, for example, for streams [HKR17; KNR11; Rut05], binary trees [SR10], automata theory [Han+14; KR12], context free languages [WBR11] and final coalgebras that are presented by so-called co-operations [KR08]. Up-to techniques are the other coalgebraic development that is important to us here. These were originally conceived as an improvement to coinduction by Milner [Mil89] in his work on concurrent processes, and further studied by Sangiorgi [San98]. Pous [Pou07] provided later a framework for the composition of up-to techniques, which was presented and studied by Bonchi et al. [Bon+14] in category theoretical form, and further expanded by Rot [Rot15].

Applications and Theory of Inductive-Coinductive Reasoning

The reason we went through the history of induction and coinduction is to show that both of them have been used implicitly for a long time, and only fairly recently have they been made explicit and studied in their own right. In fact, the same is true for mixed induction and coinduction. For instance, the sieve of Eratosthenes itself is an inductive-coinductive process, in that it produces the stream of prime numbers but requires an increasing, but finite, amount of computation steps to compute the next prime number, see [Ber05]. But also in the general theory of coalgebras it was realised that systems often need some further algebraic structure to, for example, model push down automata [GMS14], formulate structural operational semantics [Kli11; TP97] or define operations on coinductive objects [HKR17].

Of these, abstract generalised structural operational semantics [Bar04; TP97] is an interesting case because the studied objects there are *bialgebras*, that is, objects that carry both algebraic and coalgebraic structure. This happens usually because one aims to give operational semantics to a syntactic theory, hence the name of the framework. In this framework, one may find an initial bialgebra, which represents the operational semantics of the pure syntactic theory, and a final bialgebra, which models denotational semantics. Since these two bialgebras generally differ, we note that each of them only admits either induction or coinduction as proof technique, but not both.

There are many more examples, where induction and coinduction crucially occur together, like weak bisimilarity [CUV15; NU10; Rut99; San11], Kőnig's lemma and the fan theorem [NUB11; TvD88], the study of continuous functions [GHP09a; GHP09b], Cauchy sequences, resolution for coinductive logic programs [KL17; KP11], and recursion theory [Bas18b; Cap05]. Surely, there are further examples that need to be uncovered, and I hope that this list inspires the reader to embrace the inductive-coinductive approach.

This list virtually screams for a general account of mixed induction and coinduction. So is it possible that no one has provided some general theory, given that inductive-coinductive reasoning seems to be so prevalent? Indeed, people considered general approaches to some aspects of inductive-coinductive objects. Most prominent are probably the cases of modal logic, programming and to some

extent category and game theory. In the case of modal logic, the desire was to express properties that could refer to more than the (finite) number of steps that are specifiable with plain $\Box\Diamond$-formulas. This lead Pnueli [Pnu77] to add further modal operators, resulting in *linear time logic (LTL)*, that allow one to state properties of finite trace prefixes and indefinitely long trace postfixes. In fact, this allows the expression of restricted inductive-coinductive modal properties. It was later realised that what LTL did could be expressed more generally as least and greatest fixed point formulas, thereby motivating the development of the *modal μ-calculus* [Koz83]. The modal μ-calculus is an example of a language, complemented by semantics [NW96] and proof systems [DHL06; Wal93], that deals with general inductive-coinductive properties, albeit in a restricted setting.

In the case of programming, many people have worked on general calculi that feature inductive-coinductive types. Probably the first to make such structures explicit and present them in a modern form were Hagino [Hag87] and Mendler [Men87; Men91]. Later, Geuvers [Geu92] studied inductive-coinductive types in their own right as well as inside the polymorphic λ-calculus of Girard [Gir71]. Most approaches, including the ones mentioned above, to programming with inductive-coinductive types are based on iteration and coiteration schemes [AMU05; Gre92; How96a; Mat99; UV99b]. However, more recently Abel and Pientka [AP13] and Abel et al. [Abe+13] suggested the use of *copatterns*, which are in a sense dual to the well-known patterns for inductive types. These copatterns enable very elegant specifications of inductive-coinductive processes by using recursive equations, as we will see throughout this thesis. In fact, the original motivation for studying copatterns was the aforementioned problem of extending behavioural differential equations, and copatterns certainly inspired parts of this thesis. Copattern specifications are now part of the Agda language [Agd15], which can serve both as a programming language and a "mathematical assistant" [Bar13]. Towards the end of this thesis, we will demonstrate that reasoning based on equational specifications with copatterns leads to compact proofs for properties of inductive-coinductive objects. Another mathematical assistant that should be mentioned here is Coq [Coq12], since it also provides the possibility to work with general inductive-coinductive objects, including predicates and relations. The problem with Coq is the inherently flawed view that is taken on coinductive types, as we will discuss in detail in Chapter 7. All of these existing calculi for constructing and reasoning about inductive-coinductive objects are, however, either too weak to serve as a general logic or they lack a formal correctness proof.

Then there are also order and category theory, which provide the most abstract accounts of induction and coinduction that there are. In principle, both ordered sets and categories give us the possibility to deal with higher-order recursion, in that we can construct initial and final objects that are themselves monotone functions or functors, respectively [AAG05; Kim10; Par79]. Yet this fact is hardly ever used, and it is one of the central themes in this thesis to take advantage of the construction of higher-order inductive-coinductive objects.

Speaking of category theory, Santocanale [San02b] has used categories, in which certain inductive-coinductive objects are available, to give semantics to parity games. Last but not least, inductive and coinductive objects also arose in categorical logic. For instance, in the work of van den Berg and de Marchi [vdBdM04] the aim is to formulate predicative set theory, that is, set theory without the power set axiom, by using categorical logic. This aim is very close to that of this thesis. Indeed, one of the main motivations, which drives the development of languages for inductive-coinductive reasoning here, is to contribute to foundations that allow us to formalise large parts of Mathematics and Computer Science in a constructive and computer-verifiable way. And once our reasoning

methods have evolved so far to attain this goal, we can free ourselves from Cantor's "paradise".

A Note on Terminology

To simplify the process of relating this thesis to existing work, we need to discuss the use of terminology here and elsewhere. In view of the type theoretic development, we will distinguish between the definition principles for maps on inductive and coinductive objects, and the associated proof principles. For the definition principle on inductive objects, the terms "recursion", "iteration" and "induction" are often used interchangeably. However, I refrain from following this, and reserve "recursion" for general self-reference, "iteration" for the definition principle of maps out of inductive objects and "induction" for the proof principle. This matches also the traditional use of the term in *recursion* (or computability) theory. When it comes to coinductive objects, the situation is that "coinduction" is commonly used to refer to the definition principle (!) of maps into coinductive objects [San11]. The associated proof principle is in the theory of coalgebras called the *coinductive proof principle* [JR11; Rut00] or *bisimulation proof method*, for an appropriate notion of bisimulation, cf. [Sta11]. We shall here, however, dualise the terminology for inductive objects and refer to the definition principle of maps into coinductive objects as *coiteration* and the proof principles, which allows us to show that elements of a coinductive object are equal, as *coinduction*. This streamlines the terminology in [Rut00] and follows, in the case of coinduction, the wording in [HJ97]. For convenience, we list this terminology with its meaning in the following table.

	Inductive Objects	Coinductive Objects
Definition principle	**Iteration**: maps out of inductive objects (also: inductive extension)	**Coiteration**: maps into coinductive objects (also: coinductive extension [Rut00])
Proof principle	**Induction**: uniqueness of iteration, or no proper subalgebras, or minimal structure closed under constructors	**Coinduction**: uniqueness of coiteration, or no proper quotients, or bisimilarity implies equality

Research Aims: How to Weave Induction and Coinduction

So, if induction and coinduction work so nicely together, why is it necessary to write a whole doctoral thesis on this topic? To clarify this, let us discuss the research aims that we will pursue here. First of all, some aspects in our motivating example may strike one at least as odd and as being a burden. For instance, the necessity to represent elements of an object as maps out of $\mathbf{1}$, or even the need to rely on some external set theory to construct the substream relation. Thus, we are certainly in the need of a language that allows us to define and reason about inductive-coinductive objects in an accessible notation and without having to resort to any external theory. This leads us to our first objective, namely to

find and study languages for inductive-coinductive definitions and reasoning.

However, we know that humans make mistakes. Although we can learn from these mistakes, they should be avoided in published results that others rely on. A reasonable way to prevent mistakes

from slipping into accepted knowledge is to have a trusted entity that can check all our definitions and proofs. Traditionally, we entrusted other humans with this task, with the advent of computers it was realised though that proof checking could be mechanised if only sufficiently many details are given. If we had languages, in which propositions and proofs can be implemented with reasonable effort and are automatically verifiable, then this would lead to a shift from technical (proof) details to the actual knowledge contained in publications. Hence, we obtain the requirement that

the studied languages should lend themselves to being automatically verifiable.

Once we have found such languages, we need to ask about the meaning of the objects that can be defined in those languages. In other words, we need to

provide semantics for the studied languages.

This semantics can, and will, be given in terms of category and set theoretical structures and in terms of operational semantics. Relating our languages to existing theories should be seen as a complementary perspective, which exhibits the differences between the theories.

Finally, to make the languages useful, they should

provide an abstraction level similar to category theory but make specifications and proofs easier to write and follow than in the language of categories.

This is somewhat vague, but notice that we used variables in the intuitive description of algebras and coalgebras and the associated equations (1.1) and (1.2). That way, the behaviour of f and g were clear from the outset. Moreover, we had to resort to identifying elements of an object X with maps $\mathbf{1} \to X$. This is not just awkward, but also not correct in general, for example in the case of monoids or presheaves.

Put in general terms, the intention of studying languages for inductive-coinductive definitions and reasoning is to provide a framework that can serve as a logical foundation for category and set theory. Since we will give operational semantics to our languages, we gain some understanding of the languages on their own and we can equip them with meaning independent of other theories. If the languages are now sufficiently rich to accommodate category theory and set theory, then we are in the position to formalise and verify proofs for both of them. This is of course an ambitious goal that will not be fully attained in this thesis, but we will nonetheless contribute to it.

Methodology: The Weaving Tools

Having collected our research aims, let us now discuss how we will approach them. The major viewpoint that runs trough this thesis is type theory. In type theory, one studies *types* and *terms*, where each term crucially has a type. In particular, we will study three type theories. Two of them allow the definition of simple types like natural numbers and streams, while the third is a dependent type theory and thus will admit inductive-coinductive predicates and relations. But we will not restrict ourselves to type theory here. Indeed, we will also use category theory, coalgebraic methods and first-order logic to provide reasoning principles for the type theories.

On the Choice of Type Theory

So, why should we prefer to use type theory as framework, rather than just category theory or set theory? To explain this, we need to understand that category theory and set theory shine in certain areas but are problematic in others. For example, relationships between objects in a wide variety of situations are perfectly captured in categories, but categories in themselves are notoriously difficult to use as logic directly and the arising proofs are not automatically verifiable upfront. We saw the difficulty of using category theory already in the example, in particular in the identification of elements with maps out of **1**. This difficulty will become even more visible in the elaborate example below. Axiomatic set theory, on the other hand, allows us to mechanise proof checking, but forces us to work with explicit encodings, if not enriched with further abstractions (who would really want to write a pair as $\{\{a\},\{a,b\}\}$?). Finally, there are all kinds of philosophical issues with set theory, which I will not dwell on here too much. Let me just say that in mathematical practice, set theory is usually viewed as platonic, that is, one assumes a fixed universe of sets that exists independently of our reality.[8] Moreover, actual infinity is central to contemporary mathematical thinking and practice [Fle07]. For example, whenever we say that $(0,1,2,3,\dots)$ is the stream of natural numbers, we signify that the process indicated by the dots can be completed to an entity that consists of all natural numbers. This is in stark contrast to the idea of coalgebras, which only describe the step-wise behaviour of processes. For instance, the stream of natural numbers would intuitively be specified by saying that its head is zero and that the further entries arise as the successor of the preceding entry. The point of using type theories is now that they allow us to describe infinite objects and processes abstractly, without the need to assume the existence of actual infinity, that is, without having to assume the existence of infinite sets as objects in themselves. In fact, we will see that the type theories allow us to directly express our intuitive understanding of natural numbers: the values (terms in normal form) of the natural numbers type are static representations of numbers, whereas the iteration and induction principle for that type expresses the dynamic character of counting.

Of course, one may object to this view on set theory, both on philosophical and practical grounds. However, I think that type theory is at the moment the approach that reflects, at least in principle, the mathematical practice of abstraction best. The hardest part is only to understand that proofs can be represented as terms, something we will explain in Chapter 6. Thus, the type-theoretic view on inductive-coinductive reasoning is in my opinion a fruitful one to study.

Now, this is a lot of praise for type theories, but we should also discuss their problems and disadvantages in comparison to category and set theory. The first thing that most people would notice, and which often triggers the rejection to use type theories, is the heavy syntactic burden that comes with them. This is certainly a problem, which is not yet solved. However, by using computers as "mathematical assistants" and by bringing type theories closer to mathematical practice, we should eventually overcome this burden. A good example, in my opinion, of a development in the right direction is Agda. Next, one may argue that all the abstraction that we can obtain with type theories can also be reached by using category theory. This is indeed true, but I also do not see category theory and type theory as competitors, they rather complement each other as discussed above and, for example, by Lambek and Scott [LS88]. Concerning (axiomatic) set theory versus type theory, this is mostly a philosophical issue as we saw, and merely a question of whether one accepts impredicative definitions, which were for example rejected by Poincaré; whether one accepts actual infinity, as it is rejected by constructivists and many pre-Cantorian mathematicians; and whether one is willing to combine impredicative definitions and actual infinity with the law of

excluded middle and possibly the axiom of choice, despite the far-reaching consequences like the Banach-Tarski paradox.

Outline and Contributions

After this somewhat philosophical digression, let us get back to the reality of this thesis and outline the scientific contributions that may be found here.

We will go through a variety of languages that present different aspects and expressivity of inductive-coinductive objects. In particular, in Chapter 3 we study two typed calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$ that cover the natural numbers and streams types. By a typed calculus we mean a language that allows the construction of types, which correspond to the category theoretical notion of object, and terms that inhabit these types, which in turn correspond to maps. The reason for studying two languages is that the correctness of the first is easier to justify, whereas the second is easier to use but requires more advanced techniques in its justification. Both calculi have appeared in one or another form already, as we will discuss in Section 3.4. Thus, there is nothing new in the calculi themselves, but rather in the analysis of and reasoning principles for these calculi.

Analysis of and Reasoning Principles for the Calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$

In Chapter 4, we first define observational equivalence for $\lambda\mu\nu$ and $\lambda\mu\nu{=}$ in terms of a modal logic and prove some basic properties of it. This equivalence builds on, but also refines, many existing ideas of program equivalences to make it work in an inductive-coinductive setting and to obtain the desired category theoretical properties. These category theoretical properties are to show that the types in both calculi have the expected unique mapping properties *up to observational equivalence*, which we will express by using 2-categories. Employing 2-categories to study properties of programs is, next to the definition of observational equivalence, the second contribution of Chapter 4. Both observational equivalence and its category theoretical properties, although not described by using 2-categories, appear already in

> [BH16] Henning Basold and Helle Hvid Hansen. 'Well-Definedness and Observational Equivalence for Inductive-Coinductive Programs'. In: *J. Log. Comput.* (2016).

The established 2-categories give us already some first reasoning principles for the calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. However, these principles are fairly difficult to use directly, which leads us to study in Chapter 5 three approaches to expressing and proving properties of programs of the calculi from Chapter 3. The first approach is based on bisimulations, which we prove to be adequate for the modal logic that defines observational equivalence, in the sense that bisimilarity coincides with observational equivalence. This shows that existing notions from the theory of coalgebra can be instantiated even in a mixed inductive-coinductive setting. Furthermore, we show in an extensive example that the substream relation, which we discussed earlier, is transitive. Since this relation is defined as a mixed inductive-coinductive relation, we establish an up-to technique that allows us to combine induction and coinduction in a novel way.

We will also see that it is not easy to use these bisimulations. Therefore, we are led to study another approach, which consists of providing a logic that supports reasoning about inductive and coinductive types, and the corresponding terms, equally well. The main reason for this logic

being easier to use than bisimulations is that it allows proofs to be recursive, that is, it provides a mechanism, the so-called *later modality*, for (controlled) self-references in proofs. Crucially, the later modality enables us to ensure the correctness of recursive proofs locally through inference rules and saves us from having to construct explicit bisimulations. By giving sound semantics to this logic in terms of the earlier studied bisimulations and up-to techniques, we obtain another proof method for observational equivalence. For proofs that involve induction, this method is easier to use than the method based on bisimulations. Moreover, the logic gives rise to automatically verifiable proofs, which is demonstrated in a prototype implementation that accompanies the chapter. To my knowledge, such a logic has not been studied before. However, in

> [Bas18a] Henning Basold. 'Breaking the Loop: Recursive Proofs for Coinductive Predicates in Fibrations'. In: *arXiv* (2018). arXiv: 1802.07143.

I tried to condense the essentials of this logic and present it in the setting of general fibrations. The link between this generalisation and the here presented logic is not made in this thesis.

The last approach to proving observational equivalence we study in Chapter 5 is fully automatic. This allows us to prove or disprove the equivalence of simple terms without human intervention. Both the decision procedure for the language fragment and the accompanying correctness proofs are new, although similar approaches have been studied for non-deterministic automata. We finish Chapter 5 by showing that the automatic approach necessarily has to be restricted to fragments of the languages from Chapter 3. This approach is thus limited in its applicability, compared to the previous proof techniques.

Of these three proof techniques, the bisimulation and the automated method have been presented in [BH16]. The extensive substream example in Section 5.1.3 and the logic in Section 5.2 are unpublished.

Categorical Dependent Inductive-Coinductive Types

Note that in the definition of the substream relation we had to resort to set theory, which is external to the category **C**. Up to Chapter 5, the provided languages only allow the simple types like that of natural numbers or streams, and some basic reasoning about them, but it is not possible to define the substream relation directly in any of these languages. Thus, the next logical step is to provide languages that support also inductive-coinductive predicates and relations. This is the topic of the last two chapters, where we develop a dependent type theory that is purely founded on inductive-coinductive types. Preference was given to dependent type theory rather than the usual syntactic proof systems, as they appeared in Chapter 5, because type theory generalises much better the ideas behind the languages in Chapter 3 to predicates and relations. Moreover, proofs that one writes in a dependent type theory can be readily verified by means of type checking. This motivation behind dependent type theory is further explained in the guide in Section 6.1.

We develop in Chapter 6 requirements on categories that allow us to construct and reason about inductive-coinductive types, which encompass the types from Chapter 3 and inductive-coinductive predicates and relations. Such categories will be called μP-complete categories. Moreover, we show that categories, which admit initial algebras for polynomial functor, are μP-complete under some mild conditions. Both developments stem from

> [Bas15a] Henning Basold. 'Dependent Inductive and Coinductive Types Are Fibrational

Dialgebras'. In: *Proceedings of FICS '15*. Ed. by Ralph Matthes and Matteo Mio. Vol. 191. EPTCS. Open Publishing Association, 2015, pp. 3–17,

but the full proofs were not presented there. We further analyse in Chapter 6 the consequences of the requirements on an μP-complete category, to the effect that we establish induction and coinduction principles inside these categories. Finally, we give a first account of the relation of μP-complete categories to the syntactic calculus we study in Chapter 7.

The basic idea of Chapter 6, to represent inductive-coinductive types as initial and final dialgebras, is an extrapolation of the approach by Hagino [Hag87] to dependent types. Apart from this, all the presented results in this chapter are new.

Syntactic Dependent Inductive-Coinductive Types

The final Chapter 7 provides a calculus, which casts the category theoretical language from Chapter 6 into syntactic form. To show that the resulting calculus can serve as a basis for reasoning about inductive-coinductive objects, we provide a representation of first-order intuitionistic logic in the calculus and show that it is consistent as such. In technical terms we show that the reduction relation for that calculus preserves types and is strongly normalising. Both the calculus and its mentioned properties are a novel development and have been presented in

> [BG16a] Henning Basold and Herman Geuvers. 'Type Theory Based on Dependent Inductive and Coinductive Types'. In: *Proceedings of LICS '16*. Logic In Computer Science. ACM, 2016, pp. 327–336,

where the full proofs may be found in [BG16c]. We finish the chapter, and the thesis, with an application of inductive-coinductive reasoning, which also serves as a running example.

The Substream Relation as Running Example

Throughout this thesis we follow one example that illustrates the stages of development. This example is, how could it be any different, of mixed inductive-coinductive nature. The aim of this example is to define and reason about the *substream relation* that we have seen earlier. In particular, we want to show that this relation is transitive. We will take our first step towards defining the substream relation in Chapter 3 by selecting entries of streams, which is interestingly an inductive-coinductive process. The substream relation is defined in Example 5.1.13 in terms of stream entry selection, which allows us to prove in Section 5.1.3 that the substream relation is transitive. In Chapter 6, we review the definition of the substream relation and obtain a direct description of it as an inductive-coinductive relation, without having to go through the process of selecting from streams. We end Chapter 7 by representing this direct definition and the transitivity proof for the substream relation in a theory of dependent inductive-coinductive types.

Further Contributions Beyond This Thesis

Apart from the above mentioned publication, the author has participated in the following contributions, which, unfortunately, did not make it into this thesis due to size constraints: [Bas+14a; BGvdW17; Bas+14b; Bas+15; Bas+17; BK16; BPR17].

Reading Advice

A quick word on how to read this thesis. Each chapter consists of an introduction, the main content and a discussion of the content, related work, technical contributions and future work. Moreover, the text is often annotated with further thoughts, remarks and some technical points. These annotations appear at the end of each chapter, so as to avoid disrupting the actual text with littering remarks. Let us also mention the dependencies between the content of the chapters. Chapter 3 provides the basic object languages, for which we will provide reasoning techniques in Chapter 4 and Chapter 5. The latter Chapter 5 depends crucially on the notion of program observation that we define in Chapter 4. To read the last two chapters it is not necessary to have read any of the previous ones, but it certainly helps to have done so. Also, it is not strictly necessary to read Chapter 6 before Chapter 7, although the ideas for dependent inductive-coinductive types are mainly developed in Chapter 6. Finally, since we use categories and many other technical notions, Chapter 2 provides for convenience an overview of theory that is frequently used throughout the thesis, and which can be consulted at any time.

Notes

[1] In Œuvres de Henri Poincaré (1956), p. 152.

[2] Sangiorgi [San11] describes coinduction as the study of cyclic processes. I find this view too narrow, as it excludes, for instance, the sampling of physical (stochastic) processes, Brouwer's choice sequences or real numbers that are not computable. Admittedly, in the study of languages for inductive-coinductive reasoning, the only processes we can describe by finite means necessarily have to be cyclic. However, restricting ourselves to only cyclic processes would imply, for example, that there are only countably many real numbers. Let us not get too much into philosophical discussions on Platonism etc. here, but just cherish the idea that coinductive objects appear in a wide range of situations. We will also find that function spaces are coinductive objects and, even though one can see functions as non-cyclic cyclic processes, this stretches the "cyclic processes" perspective quite a bit.

[3] The word "type" has at this point no formal meaning, rather it rather refers to abstract entities, whose only criterion is that they we are able to decide whether something is an element of it, cf. [Wet14]. Note that the use of the notation $x : X$ for stating that x is of type X is consistent with the notation for maps in categories: Given objects A and B in a category $\mathbf{C}$, the notation $f : A \to B$ can be read as "f is of type $A \to B$", where $A \to B$ is the type of maps from A to B in the category $\mathbf{C}$.

[4] The approach of leaving the internal structure of types unspecified is typical for both category theory and type theory. This is also their biggest strength compared to (axiomatic) set theory, where one *does* specify the internal structure of types, here sets, and derives their properties from there.

[5] In general categories this characterisation of natural numbers does, of course, not work. For instance, the correct way to identify the properties of natural numbers in the category of monoids and their homomorphisms is as a free monoid with one generator. Since we are more interested here in foundations for definitions and reasoning, we will not talk explicitly about categories, in which the objects have additional (algebraic) structure. That being said, the dependent type theory, which we

develop towards the end of this thesis, allows us to reason about categories with more structured objects.

[6] It should be noted that in the thesis itself it is assumed that the reader is familiar with category theory, at least for Section 4.2, to some extent Section 5.2.2 and certainly in Chapter 6. Only for the purpose of the introduction do we recall some of the category theoretical terminology.

[7] A reader familiar with algebras and coalgebras will object now that this is not the standard definition. This is indeed true and, in fact, we are dealing rather with *dialgebras* [Hag87] in disguise here. However, for the sake of simplicity, we use the present definition in the introduction, as it allows us to avoid introducing products and coproducts at this stage.

[8] Also I am guilty of this, because in this thesis we will often refer to *the* category of sets, and thereby follow common practice. Such a category does not exist, as the usual Zermelo-Fraenkel axioms have no unique model. Even worse, there are non-standard models that completely break with our intuition. However, moving away from the practice of referring to a category of sets would lead us too far astray and is certainly besides the point of this thesis.

Preliminaries

The purpose of this chapter is to provide a common base of terminology and notation for those topics of term rewriting, category theory and coalgebraic theory that are necessary throughout the thesis. Most of this material can be found in the standard literature on the corresponding topics. Only in Section 2.6.1 do we pick notions of pseudo-adjoints between 2-categories that appear under different names in other places. Thus, that section serves as a name reference for this thesis.

This chapter is structured as follows. We first introduce reduction relations in Section 2.1. In Section 2.2, we settle on some notation for general category theory. The following four sections introduce then more specific category theoretical terminology: presheaves and specific instances thereof are discussed in Section 2.3; fibrations are defined in Section 2.4; Section 2.5 is devoted to algebras, coalgebras, their common generalisation of dialgebras, and a brief overview over coalgebraic predicates; Section 2.6 introduces 2-categories, pseudo-structures, and (co)algebras for pseudo-functors.

2.1. Reduction Relations

In this section, we recall a few standard notions from the theory of term rewriting and equip ourselves with some notation that we will need throughout the whole thesis. A standard reference on term rewriting is, for example, [Klo92].

Generally, we consider an (abstract) *term rewriting system* to be given by a set T and a relation $\longrightarrow \subseteq T \times T$. We refer to T as a set of *terms* and to $\longrightarrow$ as a *reduction relation* on the terms in T. Such a given reduction relation induces further relations on T. The first is the reflexive, transitive closure of $\longrightarrow$, which we usually denote by $\longrightarrow\!\!\!\!\!\rightarrow$. *Convertibility* of terms arises as the symmetric closure of $\longrightarrow\!\!\!\!\!\rightarrow$, that is, as the equivalence closure of $\longrightarrow$. This convertibility relation will be denoted by $\equiv$, and one says that terms s and t in T are *convertible*, if $s \equiv t$.

Terms and reduction relations may feature several properties. Let us introduce the ones that are important to us here. First of all, a term $t \in T$ is in *normal form*, if there is no reduction step from t possible, that is, there is no $s \in T$ with $t \longrightarrow s$. Sometimes, we denote this by $t \not\longrightarrow$. If for $t \in T$ there is a term s in normal form and $t \longrightarrow\!\!\!\!\!\rightarrow s$, then we say that t is *normalising* or that t has a normal form. Should there be no infinite reduction sequence that starts at t, then we say that t is *strongly normalising* and we write $t\downarrow$ in this case. We may generalise this terminology to the reduction relation by saying that $\longrightarrow$ is (strongly) normalising if all terms in T are. Finally, $\longrightarrow$ is said to be *confluent*, if for all $t, s_1, s_2 \in T$ with $s_1 \longleftarrow\!\!\!\!\!\leftarrow t \longrightarrow\!\!\!\!\!\rightarrow s_2$, there is an $s \in T$, such that $s_1 \longrightarrow\!\!\!\!\!\rightarrow s \longleftarrow\!\!\!\!\!\leftarrow s_2$. We note that for confluent reduction relations convertibility is equivalently be given by

$$t_1 \equiv t_2 \quad \text{iff} \quad \exists t_3 . t_1 \longrightarrow\!\!\!\!\!\rightarrow t_3 \longleftarrow\!\!\!\!\!\leftarrow t_2.$$

2.2. General Category Theory

Again, we will merely set up terminology and notations here to have a common ground with the reader in what follows. A general introduction to the necessary category theory can be found in the handbook of Borceux [Bor08], or from a more logic-oriented perspective in [LS88].

In this thesis, we will denote general categories by upper-case, bold-face letters $\mathbf{B}, \mathbf{C}, \ldots$. Special cases are the category **Set** of sets and functions, the (large) category **Cat** of categories and functors, and the final category **1** with one object $*$ and one morphism id_*. Given a category $\mathbf{C}$, we shall denote the collection of objects in $\mathbf{C}$ by $\mathrm{ob}\,\mathbf{C}$, but we also customarily write $A \in \mathbf{C}$ instead of $A \in \mathrm{ob}\,\mathbf{C}$. For objects $A, B \in \mathbf{C}$, the collection of morphisms, also called *hom-set*, between A and B is written as $\mathbf{C}(A, B)$. Note that we talk about sets here, which means that one would usually refer to $\mathbf{C}$ as *locally small*. We shall not worry about size issues here too much and rely on the understanding that $\mathbf{C}(A, B)$ is considered in a larger univerise, if it is not actually a set. This may happen, for example, when we talk about functor categories later. However, if the morphisms between two objects actually form a set, then $\mathbf{C}(A, B)$ can be extended to a functor $\mathbf{C} \colon \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{Set}$ by defining it on morphisms $f \colon A' \to A$ and $g \colon B \to B'$ in $\mathbf{C}$ to be

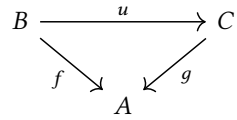
$$\mathbf{C}(f, g)(u) = g \circ u \circ f.$$

From given categories, there are several ways of constructing other categories. The first construction we need is that of *functor categories*: Given categories $\mathbf{C}$ and $\mathbf{D}$, we denote by $[\mathbf{C}, \mathbf{D}]$ the category that has functors $\mathbf{C} \to \mathbf{D}$ as objects and natural transformations between such functors as morphisms. From the hom-functors we get a functor $y : \mathbf{C} \to [\mathbf{C}^{\mathrm{op}}, \mathbf{Set}]$ that embeds any category $\mathbf{C}$ into a functor category, the so-called *Yoneda embedding*.

Another canonical construction of a category from a given category $\mathbf{C}$ is the so-called *arrow category* $\mathbf{C}^{\to}$. This category has as objects morphisms $A \to B$ of $\mathbf{C}$ and as morphisms between $f \colon A \to B$ and $g \colon A' \to B'$ pairs (u, v) of morphisms in $\mathbf{C}$, making the following diagram commute.

$$\begin{array}{ccc} A & \xrightarrow{u} & A' \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ B & \xrightarrow{v} & B' \end{array}$$

For every object $A \in \mathbf{C}$, there is a full subcategory $\mathbf{C}/_A$ of $\mathbf{C}^{\to}$, the *slice category* of $\mathbf{C}$ above A. The objects in this category are again morphisms in $\mathbf{C}$, but this time around with fixed codomain A, and the morphisms are given by commuting triangles of the following form.



Thus $\mathbf{C}/_A$ embeds into $C^{\to}$ by mapping a morphism $u \colon f \to g$ to (u, id_A). We will meet these constructions of arrow and slice categories again when we come to fibrations in Section 2.4.

The last construction on categories that we will need is that of a quotient category. Suppose that $\mathbf{C}$ is a category and $\sim$ an equivalence relation between the morphisms of $\mathbf{C}$ that respects typing and

composition, in the sense that for morphisms f and g in $\mathbf{C}$ with $f \sim g$, we require that

$$\exists A, B.\, f, g \in \mathbf{C}(A, B), \qquad \forall h.\, f \circ h \sim g \circ h, \text{ and} \qquad \forall h.\, h \circ f \sim h \circ g.$$

Under these conditions, the *quotient category* [Mac98, Sec. II.8] $\mathbf{C}/_\sim$ of $\mathbf{C}$ by $\sim$ has the same objects as $\mathbf{C}$ and as morphisms equivalence classes of morphisms in $\mathbf{C}$, that is, we have $(\mathbf{C}/_\sim)(A, B) = \mathbf{C}(A, B)/_\sim$. The above conditions allows us then to define composition on equivalence classes, and to prove the necessary identity and associativity axioms.

2.3. Presheaves

We now discuss a specific kind of functor categories in more detail. These functor categories are categories of so-called presheaves, which are used to represent indexed families of all kinds. An example application are Kripke models for intuitionistic logic that can be represented as presheaves. Another instance of presheaves are the ω^{op}-chains that can be used to construct final coalgebras, see Section 2.5. In this thesis, we will use presheaves to give semantics to a logic, but not with the intention that presheaves model evolving knowledge like in Kripke models for intuitionistic logic. Rather, we interpret the logic over ω^{op}-chains of Boolean values that state whether a formula holds at a stage of the ω^{op}-chain of the approximation of a final coalgebra, see Section 5.2.2 for the details of these semantics. The purpose of this section is to collect some general results on presheaves that are necessary to establish the semantics there.

Definition 2.3.1. Let $\mathbf{C}$ be a small category and $\mathbf{S}$ any category. An $\mathbf{S}$-*valued presheaf* over $\mathbf{C}$ is a functor $\mathbf{C}^{\mathrm{op}} \to \mathbf{S}$.

Since $[\mathbf{C}^{\mathrm{op}}, \mathbf{S}]$ is a functor category, limits and colimits can be computed in this category point-wise. This gives the following standard result, see [Bor08, Sec. 2.15].

Proposition 2.3.2. *Let* $\mathbf{C}$ *be a small category and* $\mathbf{S}$ *any category. If* $\mathbf{S}$ *is (co)complete, then also the functor category* $[\mathbf{C}^{\mathrm{op}}, \mathbf{S}]$ *is (co)complete.*

As already mentioned, the semantics in Section 5.2.2 will be given in terms of ω^{op}-chains of Boolean values. Parts of this semantics is an interpretation of logical implication, which will be given in terms of exponential objects in $[\omega^{\mathrm{op}}, \mathbb{B}]$. As the construction of these exponential objects is a bit delicate, we capture their existence in the following lemma.

Lemma 2.3.3. *Let* $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$ *be the two-valued Boolean algebra and* ω *the poset of natural numbers considered as a category. The presheaf category* $[\omega^{\mathrm{op}}, \mathbb{B}]$ *is an* exponential ideal *in* $[\omega^{\mathrm{op}}, \mathbf{Set}]$*, that is, for every object* $F \in [\omega^{\mathrm{op}}, \mathbf{Set}]$ *and* $\sigma \in [\omega^{\mathrm{op}}, \mathbb{B}]$ *there is an exponential object* $F \Rightarrow \sigma \in [\omega^{\mathrm{op}}, \mathbb{B}]$*. In particular,* $[\omega^{\mathrm{op}}, \mathbb{B}]$ *is Cartesian closed.*

Proof. Note that $[\omega^{\mathrm{op}}, \mathbb{B}]$ is a *reflective subcategory* of $[\omega^{\mathrm{op}}, \mathbf{Set}]$, that is, there is an adjunction $R \dashv I$ as in the following diagram

$$[\omega^{\mathrm{op}}, \mathbb{B}] \;\underset{R}{\overset{I}{\rightleftarrows}}\; [\omega^{\mathrm{op}}, \mathbf{Set}] \qquad (R \dashv I)$$

in which the functor I is fully faithful. The inclusion I is given by

$$I(\sigma)(n) = \begin{cases} \mathbf{1}, & \sigma(n) = \mathrm{tt} \\ \emptyset, & \sigma(n) = \mathrm{ff} \end{cases}$$

and the reflector R by

$$R(F)(n) = \mathrm{tt} \iff F(n) \text{ is inhabited.}$$

Clearly, the induced monad $L = I \circ R$ on $[\omega^{\mathrm{op}}, \mathbf{Set}]$ preserves finite products, from which be obtain by [Joh02, Prop. A4.3.1] that $[\omega^{\mathrm{op}}, \mathbb{B}]$ is an exponential ideal. □

The proof of Lemma 2.3.3 appeals to a rather abstract result. Let us for later reference give an explicit definition of exponential objects in $[\omega^{\mathrm{op}}, \mathbb{B}]$. Usually, see [Awo10, Sec. 8.7], exponential objects in $[\mathbf{C}^{\mathrm{op}}, \mathbf{Set}]$ are defined to be

$$(G^F)(U) = [\mathbf{C}^{\mathrm{op}}, \mathbf{Set}]\,(y(U) \times F, G),$$

that is, $(G^F)(U)$ is given as the set of natural transformations $y(U) \times F \Rightarrow G$. The reason why $[\omega^{\mathrm{op}}, \mathbb{B}]$ has exponential objects is that we can define a map $y_{\mathbb{B}} : \omega \to [\omega^{\mathrm{op}}, \mathbb{B}]$ by

$$y_{\mathbb{B}}(n)(m) \coloneqq \omega(m, n) \text{ is inhabited} = m \leq n.$$

That $y_{\mathbb{B}}$ is indeed a functor follows by transitivity of $\leq$. Since the product in $[\omega^{\mathrm{op}}, \mathbb{B}]$ is given by point-wise conjunction, we also denote it by

$$(\sigma \wedge \tau)(n) \coloneqq \sigma(n) \wedge \tau(n).$$

The exponential object for $\sigma, \tau \in [\omega^{\mathrm{op}}, \mathbb{B}]$ is then given by

$$\begin{aligned} (\sigma \Rightarrow \tau)(n) &= [\omega^{\mathrm{op}}, \mathbb{B}]\,(y_{\mathbb{B}}(n) \wedge \sigma, \tau) \text{ is inhabited} \\ &= \forall m \in \mathbb{N}.\, (y_{\mathbb{B}}(n)(m) \wedge \sigma(m)) \implies \tau(m) \\ &= \forall m \in \mathbb{N}.\, (m \leq n \wedge \sigma(m)) \implies \tau(m). \end{aligned}$$

A reader who has seen Kripke semantics for intuitionistic logic will certainly recognise that this definition resembles the interpretation of implication in a Kripke frame.

Another ingredient of the semantics in Section 5.2.2 is the so-called *later modality* that has been treated category theoretically by Birkedal et al. [Bir+11]. We define it and give some of its properties.

Lemma 2.3.4. *Let* $\mathbf{C}$ *be a category with a final object* $\mathbf{1}$*, then the map* $\triangleright \colon [\omega^{\mathrm{op}}, \mathbf{C}] \to [\omega^{\mathrm{op}}, \mathbf{C}]$ *given on objects by*

$$(\triangleright\sigma)(n) = \begin{cases} \mathbf{1}, & n = 0 \\ \sigma(k), & n = k + 1 \end{cases}$$

$$(\triangleright\sigma)(n \leq m) = \begin{cases} !\colon (\triangleright\sigma)(m) \to \mathbf{1}, & n = 0 \\ \sigma(k \leq k'), & n = k + 1, m = k' + 1 \end{cases}$$

is a functor. Moreover, there is a natural transformation $\mathrm{next} \colon \mathrm{Id} \Rightarrow \triangleright$.

Proof. First, we check that $\triangleright\sigma \in [\omega^{\mathrm{op}}, \mathbf{C}]$, if $\sigma \in [\omega^{\mathrm{op}}, \mathbf{C}]$. Since $(\triangleright\sigma)(0) = \mathbf{1}$ is final in $\mathbf{C}$, there is a unique morphism $!\colon (\triangleright\sigma)(m) \to (\triangleright\sigma)(0)$. For $k \in \mathbb{N}$, we have on the other hand that if $k+1 \leq m$, then $m = k'+1$ with $k' = m-1 \in \mathbb{N}$. Thus it is sufficient to define

$$(\triangleright\sigma)(k+1 \leq k'+1)\colon (\triangleright\sigma)(k'+1) \to (\triangleright\sigma)(k+1).$$

Since $(\triangleright\sigma)(k+1) = \sigma(k)$ and $(\triangleright\sigma)(k'+1) = \sigma(k')$, we can thus put $(\triangleright\sigma)(k+1 \leq k'+1) = \sigma(k \leq k')$. Second, we define $\triangleright$ on morphisms. So let $\sigma, \tau \in [\omega^{\mathrm{op}}, \mathbf{C}]$ and let $f\colon \sigma \to \tau$ be a natural transformation, we define a natural transformation $\triangleright f\colon \triangleright\sigma \to \triangleright\tau$ as follows.

$$(\triangleright f)_n\colon (\triangleright\sigma)(n) \to (\triangleright\tau)(n)$$
$$(\triangleright f)_n = \begin{cases} !\colon \mathbf{1} \to \mathbf{1}, & n = 0 \\ f_k\colon \sigma(k) \to \tau(k), & n = k+1 \end{cases}$$

Naturality of $\triangleright$ and the functor laws follow from finality of $\mathbf{1}$.

Finally, we need to establish the natural transformation $\mathrm{next}\colon \mathrm{Id} \Rightarrow \triangleright$. We define it to be

$$\mathrm{next}_\sigma : \sigma \to \triangleright\sigma$$
$$\mathrm{next}_{\sigma,n} = \begin{cases} !\colon \sigma(0) \to \mathbf{1}, & n = 0 \\ \sigma(k \leq k+1)\colon \sigma(k+1) \to \sigma(k), & n = k+1 \end{cases}$$

That next is natural in σ is again proven by finality of $\mathbf{1}$. □

The last integral result to the semantics is that the later modality gives rise to a fixed point operator in the presheaf category $[\omega^{\mathrm{op}}, \mathbb{B}]$, called *Löb induction* by Birkedal et al. [Bir+11].

Lemma 2.3.5. *Let $\sigma \in [\omega^{\mathrm{op}}, \mathbb{B}]$. There is a morphism $(\triangleright\sigma \Rightarrow \sigma) \to \sigma$ in $[\omega^{\mathrm{op}}, \mathbb{B}]$.*

Proof. A very quick proof goes by using the fact that for any $U \in [\omega^{\mathrm{op}}, \mathbf{Set}]$ there is a morphism $(\triangleright U \Rightarrow U) \to U$ in $[\omega^{\mathrm{op}}, \mathbf{Set}]$, see [Bir+11], and then appealing by to the fact that $[\omega^{\mathrm{op}}, \mathbb{B}]$ is a reflective subcategory of $[\omega^{\mathrm{op}}, \mathbf{Set}]$. For convenience, we give a direct proof here though.

Let $\sigma \in [\omega^{\mathrm{op}}, \mathbb{B}]$ and $n \in \mathbb{N}$. We need to to show that if $(\triangleright\sigma \Rightarrow \sigma)(n)$ holds, then $\sigma(n)$ holds. This is shown by induction on n. Recall that

$$(\triangleright\sigma \Rightarrow \sigma)(n) = \forall m \in \mathbb{N}.\, (m \leq n \wedge (\triangleright\sigma)(m)) \implies \sigma(m).$$

For $n = 0$, we have that $m \leq n$ only if $m = 0$, and that $(\triangleright\sigma)(0) = \mathrm{tt}$. Thus, $(\triangleright\sigma \Rightarrow \sigma)(n) = \sigma(0)$, as required. Suppose now that $n = k+1$. The assumption reads then as

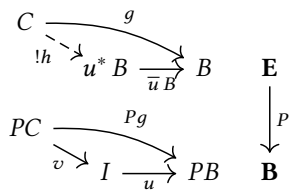
$$(\triangleright\sigma \Rightarrow \sigma)(k+1) = \forall m \in \mathbb{N}.\, (m \leq k+1 \wedge (\triangleright\sigma)(m)) \implies \sigma(m).$$

In particular, we have that $\forall m \in \mathbb{N}.\, (m \leq k \wedge (\triangleright\sigma)(m)) \implies \sigma(m)$, from which be derive by induction that $\sigma(k)$ holds. Using the assumption again with $m = k+1$, we obtain $\sigma(k) \implies \sigma(k+1)$. Thus, we have $\sigma(k+1)$. By this induction, $\sigma(n)$ holds for all $n \in \mathbb{N}$ if $(\triangleright\sigma \Rightarrow \sigma)(n)$ holds. Hence, there is a morphism $(\triangleright\sigma \Rightarrow \sigma) \to \sigma$. □

2.4. Fibrations

The next category theoretical notion we need are fibrations. Fibrations are an elegant way of capturing the fact that in a (higher-order) predicate logic the variables range over some collection of data. These variables are typically assigned some sorts, as the validity of a proposition often depends on the data its variables range over. For instance, let $P(x) :=$ "x has a maximal element". First of all, we need to know what "element of" in this case means, so let us say, for example, that x ranges over lists, and that "element of" means occurrence in a list. Next, what is the meaning of "maximal" here? A possibility is that x ranges over non-empty, finite lists of natural numbers and we can use the usual order on the natural numbers. Then for any such list u, the proposition $P(u)$ has a sensible interpretation and is in fact true. It would not be universally true if we had, for example, x assumed to range over streams of natural numbers, as these have no maximal element. Finally, $P(u)$ would even be nonsensical if u would be a list over some non-ordered structure (leaving things like the well-ordering theorem aside). This way of assigning *types* is exactly what we will use fibrations for in Chapter 6. We will also discuss the intention of using fibrations further there. For now we will just go through the technical background.

Definition 2.4.1. Let $P : \mathbf{E} \to \mathbf{B}$ be a functor, where the $\mathbf{E}$ is called the *total category* and $\mathbf{B}$ the *base category*. We say for $u : I \to J$ in $\mathbf{B}$ that a morphism $f : A \to B$ in $\mathbf{E}$ is *cartesian over* u, provided that i) $Pf = u$, and ii) for all $g : C \to B$ in $\mathbf{E}$ and $v : PC \to I$ with $Pg = u \circ v$ there is a unique $h : C \to A$ such that $f \circ h = g$. For P to be a *fibration*, we require that for every $B \in \mathbf{E}$ and $u : I \to PB$ in $\mathbf{B}$, there is a cartesian morphism $f : A \to B$ over u. Finally, a fibration is *cloven*, if it comes with a unique choice for A and f, in which case we denote A by $u^* B$ and f by $\overline{u}\, B$, as displayed in the following diagram.



On cloven fibrations, we can define for each $u : I \to J$ in $\mathbf{B}$ a functor, the *reindexing along* u, as follows. Let us denote by $\mathbf{P}_I$ the category having objects A with $P(X) = I$ and morphisms $f : A \to B$ with $P(f) = \mathrm{id}_I$. We call $\mathbf{P}_I$ the *fibre above* I. The assignment of $u^* B$ to B for a cloven fibration can then be extended to a functor $u^* : \mathbf{P}_J \to \mathbf{P}_I$. Moreover, one can show that $\mathrm{id}_I^* \cong \mathrm{Id}_{\mathbf{P}_I}$ and $(v \circ u)^* \cong u^* \circ v^*$. In this work, we are often interested in *split fibrations*, which are cloven fibrations such that the above isomorphisms are equalities, that is, $\mathrm{id}_I^* = \mathrm{Id}_{\mathbf{P}_I}$ and $(v \circ u)^* = u^* \circ v^*$.

Example 2.4.2 (See [Jac99])**.** Important examples of fibrations arise from categories with pullbacks. Let $\mathbf{C}$ be a category and $\mathbf{C}^{\to}$ be the arrow category as in Section 2.2. We can then define a functor $\mathrm{cod} : \mathbf{C}^{\to} \to \mathbf{C}$ by $\mathrm{cod}(f : X \to Y) = Y$. This functor turns out to be a fibration, the *codomain fibration*, if $\mathbf{C}$ has pullbacks, where the fibre of cod above an object $A \in \mathbf{C}$ is isomorphic to the slice category $\mathbf{C}/_A$. If we are given a choice of pullbacks, then cod is also cloven. ◀

The split variant of this construction is given by the category of *set-indexed families* over $\mathbf{C}$.

Example 2.4.3. Let $\mathrm{Fam}(\mathbf{C})$ be the category that has families $\{X_i\}_{i\in I}$ of objects X_i in $\mathbf{C}$ indexed by a set I. The morphisms $\{X_i\}_{i\in I} \to \{Y_j\}_{j\in J}$ in $\mathrm{Fam}(\mathbf{C})$ are pairs (u, f) where $u : I \to J$ is a function and f is an I-indexed family of morphisms in $\mathbf{C}$ with $\{f_i : X_i \to Y_{u(i)}\}_{i\in I}$. It is then straightforward to show that the functor $p : \mathrm{Fam}(\mathbf{C}) \to \mathbf{Set}$, given by projecting on the index set, is a split fibration.

Since we will frequently use the fibration $\mathrm{Fam}(\mathbf{C})$ in illustrative examples, it is worthwhile to introduce a special notation for its fibres. Let I be a set and define the the category of I-indexed families by

$$\mathbf{Set}^I = \begin{cases} \text{objects} & X = \{X_i\}_{i\in I} \\ \text{morphisms} & f = \{f_i : X_i \to Y_i\}_{i\in I} \end{cases}.$$

It is fairly clear that each fibre of the family fibration over $I \in \mathbf{Set}$ is isomorphic to $\mathbf{Set}^I$. Note that among the morphisms in $\mathbf{Set}^I$, we also have inclusions. We will use a special notation for these, by adapting that for subset inclusion: $X \sqsubseteq Y \iff \forall i \in I.\, X_i \subseteq Y_i$ ◂

Other examples, that are important to us here, arise as so-called *classifying fibrations* of dependent type theories. We will describe these in the guide to dependent type theory in Section 6.1.

A construction that appear often in category theory is that of products and coproducts. These constructions can be abstractly described in fibrations, and even capture as such universal and existential quantification, cf. [Awo10, Sec. 8.5] and [Jac99, Chap. 4].

Definition 2.4.4 ([Jac99, Def. 1.9.4])**.** Let $P\colon \mathbf{E} \to \mathbf{B}$ be a fibration, $u\colon I \to J$ a morphism in $\mathbf{B}$ and $u^*\colon \mathbf{P}_J \to \mathbf{P}_I$ a reindexing functor along u. We say that P has *products* (resp. *coproducts*) *along* u, if

(i) u^* has a right adjoint $\prod_u$ (resp. a left adjoint $\coprod_u$), and

(ii) the *Beck-Chevalley condition* holds: Given a pullback

$$\begin{array}{ccc} K & \xrightarrow{v} & L \\ {\scriptstyle r}\downarrow & \lrcorner & \downarrow{\scriptstyle s} \\ I & \xrightarrow{u} & J \end{array}$$

the canonical transformation

$$s^* \textstyle\prod_u \Rightarrow \prod_v r^* \qquad \left(\text{resp. } \coprod_v r^* \Rightarrow s^* \coprod_u\right)$$

is an isomorphism. ◂

Products and coproducts are discussed further in Section 6.1, whereas the Beck-Chevalley condition will be treated in more detail in Section 6.5.

2.5. Algebras, Coalgebras and Dialgebras

We now come to some of the most central concepts in this thesis: algebras, coalgebras and their common generalisation to dialgebras. Let us start by briefly recalling the basic definitions of (co)algebras and the unique mapping properties of initial algebras and final coalgebras. For more details, we refer to [Geu92; Hag87; Jac16; JR11; Rut00].

Definition 2.5.1. Let $\mathbf{C}$ and $\mathbf{D}$ be categories and $F, G : \mathbf{C} \to \mathbf{D}$ functors. An (F,G)-*dialgebra* is a morphism $c : FA \to GA$ in $\mathbf{D}$, where A is an object in $\mathbf{C}$. Given dialgebras $c : FA \to GA$ and $d : FB \to GB$, a morphism $h : A \to B$ is said to be a *(dialgebra) homomorphism* from c to d, if $Gh \circ c = d \circ Fh$. We can form a category $\mathrm{DiAlg}(F,G)$ of (F,G)-dialgebras and their homomorphisms. A dialgebra is said to be *initial* (resp. *final*) if it is an initial (resp. final) object in $\mathrm{DiAlg}(F,G)$.

A word about terminology: Given a dialgebra $d : FB \to GB$, we call the unique homomorphism from an initial (F,G)-dialgebra into d the *inductive extension* of d. Dually, the unique homomorphism into a final dialgebra is called the *coinductive extension* of d. ◂

Let us discuss an example of a dialgebra in the category of sets.

Example 2.5.2. Let $F, G : \mathbf{Set} \to \mathbf{Set} \times \mathbf{Set}$ be given by $F = \langle \mathbf{1}, \mathrm{Id} \rangle$ and $G = \langle \mathrm{Id}, \mathrm{Id} \rangle$, that is, F maps a set X to the pair $(\mathbf{1}, X)$ in the product category. Similarly, G, the diagonal functor, maps X to (X,X). Now, let $z : \mathbf{1} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ be the constant zero map and the successor on natural numbers, respectively. It is then easy to see that $(z,s) : F(\mathbb{N}) \to G(\mathbb{N})$ is an initial dialgebra. ◂

We will often need the case of dialgebras, in which F or G is the identity functor.

Definition 2.5.3. Let $H \colon \mathbf{C} \to \mathbf{C}$ be a functor. An *algebra* is a morphism $a \colon HX \to X$ in $\mathbf{C}$ and a *coalgebra* is a morphism $c \colon X \to HX$. The notion of *homomorphism* of algebras and coalgebras is the same as for dialgebras with the choices $G = \mathrm{Id}$ and $F = \mathrm{Id}$, respectively. We denote by $\mathrm{Alg}(H)$ and $\mathrm{CoAlg}(H)$ the categories of algebras and coalgebras with their homomorphisms. These categories may have initial (resp. final) objects, to which we refer as *initial algebra* (resp. *final coalgebra*). ◂

The following is an important result that shows the relation between initial algebras and least fixed points, and final coalgebras and greatest fixed points in ordered structures like lattices.

Lemma 2.5.4 (Lambek)**.** *Let $F \colon \mathbf{C} \to \mathbf{C}$ be a functor. Every initial algebra $\alpha \colon FA \to A$ and final coalgebra $\xi \colon B \to FB$ is an isomorphism. In other words, the objects A and B are fixed points, in the sense that $FA \cong A$ and $FB \cong B$.*

Now that we know that initial algebras and final coalgebras generalise fixed points, one may ask how these can be constructed. The way that is most important to us here are the so-called *initial* and *final chain constructions*.

Definition 2.5.5. Let $F \colon \mathbf{C} \to \mathbf{C}$ be a functor. If $\mathbf{C}$ has an initial object $\mathbf{0}$, then the *initial chain* is the functor $\overrightarrow{F} : \omega \to \mathbf{C}$ from the ordinal ω considered as a poset to $\mathbf{C}$ defined as in the following diagram. Here, ! denotes the unique morphism out of $\mathbf{0}$.

$$\overrightarrow{F} : \qquad \mathbf{0} \xrightarrow{\ !\ } F(\mathbf{0}) \xrightarrow{F(!)} F^2(\mathbf{0}) \xrightarrow{F^2(!)} F^3(\mathbf{0}) \xrightarrow{F^3(!)} \cdots$$

More precisely, this means that $\overrightarrow{F}$ is defined on objects by

$$\overrightarrow{F}(0) = \mathbf{0} \qquad \overrightarrow{F}(k+1) = F\big(\overrightarrow{F}(k)\big)$$

and on morphisms by $\overrightarrow{F}(0 \le n) = ! \colon \mathbf{0} \to F^n(\mathbf{0})$ and $\overrightarrow{F}(k+1 \le n) = F\big(\overrightarrow{F}(k \le n)\big)$. Dually, if $\mathbf{C}$ has a final object $\mathbf{1}$, then there is a functor $\overleftarrow{F} \colon \omega^{\mathrm{op}} \to \mathbf{C}$, the *final chain*, as in the following diagram.

$$\overleftarrow{F} : \qquad \mathbf{1} \xleftarrow{\ !\ } F(\mathbf{0}) \xleftarrow{F(!)} F^2(\mathbf{0}) \xleftarrow{F^2(!)} F^3(\mathbf{0}) \xleftarrow{F^3(!)} \cdots$$

We say that the initial and final chain *stabilise*, if $F\big(\mathrm{colim}\,\overrightarrow{F}\big) \cong \mathrm{colim}\,\overrightarrow{F}$ and $F\big(\lim \overleftarrow{F}\big) \cong \lim \overleftarrow{F}$. ◂

From this construction, one obtains initial algebras and final coalgebras.

Lemma 2.5.6. *If for a functor $F\colon \mathbf{C} \to \mathbf{C}$ the initial chain stabilises, then* $\operatorname{colim} \overrightarrow{F}$ *is the carrier of an initial algebra. Dually,* $\lim \overleftarrow{F}$ *is the carrier of a final coalgebra, if the final chain stabilises.*

These are standard constructions, issues of which are discussed for example by Worrell [Wor05]. Interestingly, this construction amounts to the usual fixed point construction in ordered structures, which originates in the work of Kleene [CC79], a fact we will use in Section 5.2.2.

2.5.1. Coinductive Predicates and Up-To Techniques in Lattices

In this section, we give a brief overview over the theory of coinductive predicates and an enhancement of the resulting proof methods in form of so-called up-to techniques. Since we only need the theory for complete lattices, usually given by the power set of another set, we restrict attention to this simpler case. It should be noted, however, that many of the developments can be generalised to category theoretical settings, see [Bon+14; HJ97; Sta11], but some notions, like that of the companion, are still being developed at that level of generality.

It is assumed that the reader is familiar with the notion of complete lattice. For the sake of brevity, we usually refer to a complete lattice just by the underlying set and leave the order and lattice structure implicit. Given a complete lattice L, the set $\operatorname{Mon}(L, L)$ of monotone maps is again complete lattice with the order and lattice structure given point-wise. Usually, we denote the order on $\operatorname{Mon}(L, L)$ by $\sqsubseteq$ and its join operator by $\bigsqcup$.

Let us begin by introducing what we actually mean by a "coinductive predicate". This is easiest done by taking for a moment a more abstract perspective. Recall that we introduced fibrations as a way to talk abstractly about predicates, relations etc. Now we use this view to define coinductive predicates over a given coalgebra for an arbitrary notion of predicate.

Definition 2.5.7 ([Has+13])**.** Let $P\colon \mathbf{E} \to \mathbf{B}$ be a cloven fibration and $F\colon \mathbf{B} \to \mathbf{B}$ an endofunctor. We say that a functor $G\colon \mathbf{E} \to \mathbf{E}$ is a *lifting* of F, if the following diagram commutes

$$\begin{array}{ccc} \mathbf{E} & \xrightarrow{G} & \mathbf{E} \\ {\scriptstyle P}\downarrow & & \downarrow{\scriptstyle P} \\ \mathbf{B} & \xrightarrow{F} & \mathbf{B} \end{array}$$

and if G preserves Cartesian morphisms. Such a lifting G induces for each $X \in \mathbf{B}$ a functor

$$G_X\colon \mathbf{P}_X \to \mathbf{P}_{FX}.$$

Given a coalgebra $c\colon X \to FX$ in $\mathbf{B}$, a *G-invariant* in c is a $(G_X \circ c^*)$-coalgebra in $\mathbf{P}_X$. Moreover, a *G-coinductive predicate* in c is a final $(G_X \circ c^*)$-coalgebra in $\mathbf{P}_X$. We often denote the carrier of the G-coinductive predicate in c by $\nu(G_X \circ c^*)$. ◂

All the examples of fibrations that we will encounter in this thesis will have the property that each fibre is a complete lattice.

Definition 2.5.8. We say that a fibration $P\colon \mathbf{E} \to \mathbf{B}$ is a *fibre-wise complete lattice*, if for each $X \in \mathbf{B}$, the fibre $\mathbf{P}_X$ over X is a complete lattice. That is to say, there is a complete lattice L, such that $\mathbf{P}_X$ is the associated category. ◂

The importance of this definition comes from the fact that we can find by the Knaster-Tarski theorem for any lifting to a fibre-wise complete lattice the corresponding coinductive predicate.

Proposition 2.5.9. *Let $P\colon \mathbf{E} \to \mathbf{B}$ be a fibration that is a fibre-wise complete lattice. For any functor $F\colon \mathbf{B} \to \mathbf{B}$, any lifting G of F and any coalgebra $c\colon X \to FX$, the G-coinductive predicate in c exists.*

We now restrict attention to a single complete lattice L and a monotone map $\Phi\colon L \to L$, which should be thought of as being of the form $G_X \circ c^*$. The idea of Φ is that it allows us to describe invariants. In particular, we obtain the following standard proof method for coinductive predicates: If $p \leq \Phi(p)$, that is, p is a Φ-invariant, then $p \leq \nu\Phi$. This is a useful proof method that has found many applications. For instance, bisimilarity of transition systems can be described as coinductive predicate, and the above proof method is then the standard bisimulation proof method [San11].

Often, we find ourselves that an invariant has to be chosen quite large, mostly with elements that could be "automatically" added. To overcome this, up-to techniques were introduced in [Mil89] and further developed in [Bon+14; Pou07; PS11; Rot15; San98]. A particular class of up-to techniques, the so-called compatible ones, has emerged in this development. The significance of this class comes from the fact that it is closed under composition and other lattice operations.

Definition 2.5.10. A *Φ-compatible* up-to technique is a monotone map $T\colon L^n \to L^m$ with $T \circ \Phi^{\times n} \sqsubseteq \Phi^{\times m} \circ T$, where $\Phi^{\times n}\colon L^n \to L^n$ is given by point-wise application of Φ to the elements in the product lattice L^n. We say that $p \in L$ is a *Φ-invariant up to T*, if $p^{\times m} \sqsubseteq \Phi(T(p^{\times n}))$.

The following result makes precise the importance of compatible up-to techniques that we mentioned above: They are sound enhancement of the coinductive proof method and they can be composed. This result can be found, for example, in [Bon+14; Pou07; Rot+17].

Lemma 2.5.11.

1. *If $T\colon L \to L$ is Φ-compatible and $p \in L$ is an invariant up to T, then $p \leq \nu\Phi$ (Soundness).*
2. *If $T_1\colon L^n \to L^m$ and $T_2\colon L^m \to L^k$ are Φ-compatible, then so is their composition $T_2 \circ T_1$.*
3. *If $T_1\colon L^n \to L^m$ and $T_2\colon L^n \to L^m$ are Φ-compatible, then so is $T_1 \sqcup T_2$.*
4. *Both the identity $\mathrm{id}\colon L \to L$ and Φ itself are Φ-compatible.*

Proof. For the first item, one defines a monotone map $T^\omega\colon L \to L$ of all finite iterations of T, which is given by $T^\omega = \bigsqcup_{n\in\mathbb{N}} T^n$. It can then be shown that if p is an invariant up to T, then $T^\omega(R)$ is a Φ-invariant. For details see [Bon+14; Rot15]. The second and third item are given by an easy calculation using compatibility and monotonicity:

$$(T_2 \circ T_1) \circ \Phi \sqsubseteq T_2 \circ \Phi \circ T_1 \sqsubseteq \Phi \circ (T_2 \circ T_1).$$

and for $p \in L^n$

$$\begin{aligned}((T_1 \sqcup T_2) \circ \Phi)(p) &= T_1(\Phi(p)) \vee T_2(\Phi(p)) \leq \Phi(T_1(p)) \vee \Phi(T_2(p)) \\ &\leq \Phi(T_1(p) \vee T_2(p)) = (\Phi \circ (T_1 \sqcup T_2))(p).\end{aligned}$$

Finally, the compatibility of id is given by $\mathrm{id} \circ \Phi = \Phi = \Phi \circ \mathrm{id}$, and the compatibility of Φ is immediate by the definition of compatibility. □

The advantage of compatible up-to techniques over other techniques that are merely sound, the first property in Lemma 2.5.11, is that they can be composed. This allows us to break up proofs of soundness into simpler techniques and then combine these techniques in the proof of an invariant. A problem is though that we need to carefully set up the combination of up-to techniques *before* starting the proof. We can resolve this issue by using a universal, compatible up-to technique, the so-called companion [Pou16; PR17].

Definition 2.5.12. The *companion* of Φ is defined by

$$\gamma_\Phi \coloneqq \bigsqcup_{\substack{T \in \mathrm{Mon}(L,L) \\ T \circ \Phi \sqsubseteq \Phi \circ T}} T.$$

The point of the companion is that it is the largest compatible up-to technique for Φ. Thus, we can use in a proof of invariance the companion instead of having to establish explicitly all the up-to techniques that we use in that proof. This is captured by the following lemma.

Lemma 2.5.13. *The companion of Φ is the largest compatible up-to technique and idempotent:*

- *γ_Φ is Φ-compatible*
- *If T is Φ-compatible, then $T \sqsubseteq \gamma_\Phi$.*
- *$\gamma_\Phi \circ \gamma_\Phi \sqsubseteq \gamma_\Phi$*

Proof. That γ_Φ is the largest compatible functions, the first and second property, is given immediately by definition, since the join ranges over all compatible maps. The third property follows from the fact that compatible functions are closed under composition and the other two properties. □

2.6. 2-Categories

A useful tool for describing the difference between the computational behaviour and observable behaviour of programs are 2-categories. The goal of this section is to carefully set up the terminology of 2-category theory, which can be a bit confusing at times. We begin with the basic definition of a (strict) 2-category, which can be found for example in [Bor08] or [Lei04].

Definition 2.6.1. A *(strict) 2-category* $\mathbf{C}$ is a category enriched over the category **Cat** of all categories. This means that for all objects A and B of $\mathbf{C}$ there is a *category* of morphism $\mathbf{C}(A,B)$, and for all objects A, B, C there are composition and unit *functors*

$$c_{ABC} \colon \mathbf{C}(A,B) \times \mathbf{C}(B,C) \to \mathbf{C}(A,C) \qquad \text{and} \qquad u_A \colon \mathbf{1} \to \mathbf{C}(A,A),$$

subject to the associativity and unit axioms in the following two diagrams.

$$\begin{array}{ccc} \mathbf{C}(A,B) \times \mathbf{C}(B,C) \times \mathbf{C}(C,D) & \xrightarrow{\mathrm{Id} \times c_{BCD}} & \mathbf{C}(A,B) \times \mathbf{C}(B,D) \\ \downarrow{\scriptstyle c_{ABC} \times \mathrm{Id}} & & \downarrow{\scriptstyle c_{ABD}} \\ \mathbf{C}(A,C) \times \mathbf{C}(C,D) & \xrightarrow{\quad c_{ACD} \quad} & \mathbf{C}(A,D) \end{array}$$

$$\begin{array}{ccccc} \mathbf{1}\times\mathbf{C}(A,B) & \xleftarrow{\cong} & \mathbf{C}(A,B) & \xrightarrow{\cong} & \mathbf{C}(A,B)\times\mathbf{1} \\ \downarrow{\scriptstyle u_A\times\mathrm{Id}} & & \| & & \downarrow{\scriptstyle \mathrm{Id}\times u_A} \\ \mathbf{C}(A,A)\times\mathbf{C}(A,B) & \xrightarrow{c_{AAB}} & \mathbf{C}(A,B) & \xleftarrow{c_{ABB}} & \mathbf{C}(A,B)\times\mathbf{C}(B,B) \end{array} \quad \blacktriangleleft$$

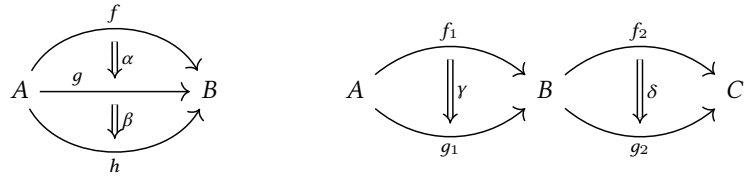
The reason why the 2-categories we defined in Definition 2.6.1 carry the prefix "strict" is that there exist various definitions of a category with 2-morphisms. Rather one is faced with a choice in which form the composition of 1-morphisms is unital and associative. For a strict 2-category we required that these take the form of equalities. However, one can also require instead that there for all f, g, h there merely are invertible 2-morphisms $\alpha\colon h\circ(g\circ f)\Rightarrow(h\circ g)\circ f$, $\gamma_1 : \mathrm{id}\circ f\Rightarrow f$ and $\gamma_2 : f\circ\mathrm{id}\Rightarrow f$, which satisfy some coherence conditions. Such a structure is then called a *bicategory* or *weak 2-category*. For example, given a category $\mathbf{C}$ with pullbacks, there is a bicategory $\mathrm{Span}(\mathbf{C})$ that has as objects the objects of $\mathbf{C}$, spans $A\leftarrow B\rightarrow C$ as morphisms and morphisms of spans as 2-morphisms. Associativity cannot hold strictly in $\mathrm{Span}(\mathbf{C})$ because pullbacks are only unique up-to isomorphism.

A strict 2-category $\mathbf{C}$ can equivalently be described algebraically as follows [Bor08]. First of all, $\mathbf{C}$ consists of the following data:

- a collection of objects (0-cells), denoted by $A, B, \ldots$;
- 1-morphisms (1-cells) between objects, denoted by $f\colon A\rightarrow B$; and
- 2-morphisms (2-cells) between 1-morphisms, denoted by $\alpha\colon f\Rightarrow g$.

Just like we can compose morphisms in ordinary categories, 1- and 2-morphisms can be composed in 2-categories. Clearly, we have different ways of composing these though. So let us go through the different kinds of compositions and their interactions.

Given 1-morphisms $f\colon A\rightarrow B$ and $g\colon B\rightarrow C$ there is a morphism $g\circ f\colon A\rightarrow C$ given by $c_{ABC}(f,g)$, just as for ordinary categories. For 2-morphisms though there are two ways of composing them, as indicated in the following diagrams.



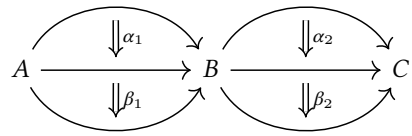
In the left diagram we are able to compose α and β along the indicated 1-morphisms to $\beta\circ_1\alpha\colon f\Rightarrow h$, where the subscript 1 indicates that we compose along 1-morphisms. This is the so-called *vertical composition* of α and β, and is given by composition in $\mathbf{C}(A,B)$. In the diagram on right we compose along objects. This *horizontal composition* of γ and δ results from the action of c_{ABC} morphisms. It is usually denoted by $\delta\circ_0\gamma\colon f_2\circ f_1\Rightarrow g_2\circ g_1$, where the 0 subscript indicates that we compose along objects (0-cells). We often write $\gamma\delta$ instead of $\delta\circ_0\gamma$, which relates the horizontal composition better to the geometry of diagrams.

Finally, all the above compositions are associative, the compositions $\circ$ and $\circ_1$ are unital, and the two compositions of 2-morphisms interact through the *exchange law* as follows.

- For every object A there is an identity morphisms $\mathrm{id}_A = u_A(*)$ that is neutral with respect to composition, that is, $\mathrm{id}_B \circ g = g = g \circ \mathrm{id}_A$ for all morphisms $g\colon A \to B$;
- For all 1-morphisms f, there are 2-morphisms id_f that are neutral for $\circ_1$;
- All the compositions of 1- and 2-morphisms are associative;
- The exchange law for the composition of 2-morphisms holds, that is, we have

  $$(\beta_2 \circ_1 \alpha_2) \circ_0 (\beta_1 \circ_1 \alpha_1) = (\beta_2 \circ_0 \beta_1) \circ_1 (\alpha_2 \circ_0 \alpha_1),$$

  for 2-morphisms as in the following diagram.



Since no other forms of 2-categories, like weak 2-categories or bicategories, do not appear in this thesis, we follow Lack [Lac10] and refer to strict 2-categories just as 2-categories,

Example 2.6.2. We just list a few examples of 2-categories. Further examples, a discussion of 2-categories and more references can be found in [Lac10].

- We can extend the (large) category **Cat** of categories and functors with natural transformations as 2-cells, which we denote by abuse of notation also as **Cat**. This is the prototypical example of a 2-category.
- There is a sub-2-category of **Cat**, which has the same objects and morphisms, but the 2-morphisms are only the natural isomorphisms.
- A similar example is the 2-category **Adj** that has again categories as objects, but now adjunctions as 1-morphisms and morphism between adjunctions, called conjugate pairs, as 2-morphisms.
- There are of course also examples of 2-categories that do not not just have categories as objects. One of these is an extension of the category of topological spaces.

  Let I be the unit interval of the real numbers. A homotopy between two continuous maps $f, g\colon X \to Y$ is a continuous map $h : I \times X \to Y$, such that $h(0,-) = f$ and $h(1,-) = g$. Such homotopies can be composed vertically and horizontally, and there is a homotopy that takes the role of the identity. The only caveat is that associativity and the unit law for composition only hold up-to a homotopy $I \times (I \times X) \to Y$. So we end up with a 2-category that has topological spaces as objects, continuous maps between spaces as 1-morphisms, and homotopy-equivalence classes of homotopies between continuous maps as 2-morphisms. For a few more details see [Bor08, Ex. 7.1.4].

- Another example is the category **PreOrd** that has pre-ordered sets $(X, \leq_X)$ as objects and monotone functions as morphisms. A 2-cell $f \Rightarrow g$ between two maps $f, g : X \to Y$ exists in **PreOrd** if f is point-wise smaller than g. Since morphisms are functions, their composition and identity morphisms are immediate. For 2-cells, note that the vertical composition is transitivity of the point-wise order, and the existence of identities for vertical composition encodes that the point-wise order is reflexive. To define the horizontal composition, suppose we have

$$X \;\underset{f'}{\overset{f}{\Downarrow}}\; Y \;\underset{g'}{\overset{g}{\Downarrow}}\; Z$$

  Then we can easily calculate from monotonicity of g that for all $x \in X$ we have

$$g(f(x)) \leq_Z g(f'(x)) \leq_Z g'(f'(x)),$$

  thus we can compose the inequalities horizontally to obtain $g \circ f \leq g' \circ f'$.

In Section 4.2.2 we will see more concrete examples of 2-categories. There, 2-morphisms will allow us to speak about a program equivalence in categorical terms while still retaining information about computations. ◀

Since the collection of morphisms in a 2-category forms a category, we can talk about isomorphic 1-morphisms.

Definition 2.6.3. Let $f, g \colon A \to B$ be 1-morphisms in a 2-category $\mathbf{C}$. We say that f and g are isomorphic, written $f \cong g$, if they are isomorphic as objects in $\mathrm{Hom}\,(A, B)$, that is, if there are 2-morphisms $\alpha \colon f \Rightarrow g$ and $\beta \colon g \Rightarrow f$ with $\alpha \circ_1 \beta = \mathrm{id}$ and $\beta \circ_1 \alpha = \mathrm{id}$. ◀

Definition 2.6.1 might suggest that 2-categories could be studied just as an instance of enriched category theory. The problem with this view is that it locks us into notions of functors and (co)limits in 2-categories that are usually too strict. For instance, the notion of functor that arises from enriched category theory requires that the composition of 1-morphisms is strictly preserved, that is, $F(g \circ f) = F(g) \circ F(f)$. As it will turn out, we instead need the weaker notion of pseudo-functors, where we only have $F(g \circ f) \cong F(g) \circ F(f)$.

Definition 2.6.4. Let $\mathbf{C}$ and $\mathbf{D}$ be 2-categories. A *pseudo-functor* $F \colon \mathbf{C} \to \mathbf{D}$ maps the 0-, 1- and 2-cells in $\mathbf{C}$ to those in $\mathbf{D}$, such that 1-identities and the 1-composition are preserved up-to isomorphism, and 2-identities and 2-compositions are strictly preserved. More precisely, let $u^{\mathbf{C}}$ and $u^{\mathbf{D}}$ be the unit functors of $\mathbf{C}$ and $\mathbf{D}$, respectively, and $c^{\mathbf{C}}$ and $c^{\mathbf{D}}$ the corresponding composition functors. The pseudo-functor F consists of

- an object $F(A)$ in $\mathbf{D}$ for each object A in $\mathbf{C}$,
- a functor $F_{A,B} \colon \mathbf{C}(A, B) \to \mathbf{C}(FA, FB)$ for all objects A and B,
- a natural isomorphism $F_{\mathrm{id}_A} \colon u^{\mathbf{D}}_{FA} \Rightarrow F_{A,A} \circ u^{\mathbf{C}}_{A}$ for all A,
- a natural isomorphism $F_{ABC} \colon c^{\mathbf{D}}_{FA,FB,FC} \circ (F_{A,B} \times F_{B,C}) \Rightarrow F_{A,C} \circ c^{\mathbf{C}}_{ABC}$ for all A, B and C.

To improve readability, we write F_{id_A} instead of $(F_{\mathrm{id}_A})_*$ for $* \in \mathbf{1}$ and $F(g,f)$ instead of $(F_{ABC})_{(f,g)}$. Finally, the following three coherence diagrams must commute for all suitable f, g and h.

$$\begin{array}{ccc} F(f)\circ \mathrm{id}_{F(A)} & = & F(f) \\ {\scriptstyle \mathrm{id}_{F(f)}F_{\mathrm{id}_A}}\Downarrow & & \| \\ F(f)\circ F(\mathrm{id}_A) & \xRightarrow{F(f,\mathrm{id}_A)} & F(f\circ \mathrm{id}_A) \end{array} \qquad \begin{array}{ccc} \mathrm{id}_{F(B)}\circ F(f) & = & F(f) \\ {\scriptstyle F_{\mathrm{id}_B}\mathrm{id}_{F(f)}}\Downarrow & & \| \\ F(\mathrm{id}_B)\circ F(f) & \xRightarrow{F(\mathrm{id}_B,f)} & F(\mathrm{id}_B\circ f) \end{array}$$

$$\begin{array}{ccc} F(h)\circ F(g)\circ F(f) & \xRightarrow{F(h,g)\mathrm{id}_{F(f)}} & F(h\circ g)\circ F(f) \\ {\scriptstyle \mathrm{id}_{F(h)}F(g,f)}\Downarrow & & \Downarrow{\scriptstyle F(h\circ g,f)} \\ F(h)\circ F(g\circ f) & \xRightarrow{F(h,g\circ f)} & F(h\circ g\circ f) \end{array}$$

We say that F is a *strict 2-functor*, if all the isomorphisms F_{id_A} and $F(g,f)$ are identities.[9] ◀

Just as for any ordinary category $\mathbf{D}$ there is a hom-functor $\mathbf{D}\colon \mathbf{D}^{\mathrm{op}}\times\mathbf{D}\to\mathbf{Set}$, we note that there is the corresponding analogue for 2-categories, see [Bor08, Ex. 7.2.4].

Example 2.6.5. For every 2-category $\mathbf{C}$ there is a strict 2-functor $\mathbf{C}^{\mathrm{op}}\times\mathbf{C}\to\mathbf{Cat}$, where $\mathbf{C}(A,B)$ is the hom-category as in Definition 2.6.1, and for $f\colon A'\to A$ and $g\colon B\to B'$ the functor part is

$$\begin{aligned} &\mathbf{C}(f,g)\colon \mathbf{C}(A,B)\to\mathbf{C}(A',B') \\ &\mathbf{C}(f,g) = c_{A'BB'}(-,g)\circ c_{A'AB}(f,-), \end{aligned}$$

that is, given by pre-composing with f and post-composing with g as for the ordinary hom-functor.

Proof. Since the composition c is a functor, $\mathbf{C}(f,g)$ is a functor. So it remains to prove the preservation of units and composition. It follows easily from the unit laws in Definition 2.6.1 that $\mathbf{C}(\mathrm{id},\mathrm{id}) = \mathrm{id}$. From the associativity law it also follows that $c_{A'BB'}(-,g)\circ c_{A'AB}(f,-) = c_{A'AB'}(f,-)\circ c_{ABB'}(-,g)$, which in turn gives us for all f',g' that $\mathbf{C}(f'\circ f, g\circ g') = \mathbf{C}(f,g)\circ\mathbf{C}(f',g')$. □

2.6.1. Adjunctions, Products, Coproducts and Exponents in 2-Categories

Having introduced 2-categories and a notion of morphism between them (pseudo-functors), it is only natural to study tighter connections between 2-categories, namely that of pseudo-adjunctions. These pseudo-adjunctions will also allow us to introduce suitable generalisations of limits and colimits to 2-categories.

Unfortunately, the machinery to describe suitably weak adjunctions between 2-categories and limits and colimits gets very technical because of the necessary coherence conditions that have to be satisfied. On the positive side though, we have that many of the technicalities are still simpler in comparison to what we would find in, for example, bicategories. We will always give some intuition about how the weakened notions compare to those for ordinary categories. This should give at least an idea to the reader what we are up to, so that the technicalities can be skipped for the most part.

Definition 2.6.6. Let $F,G\colon \mathbf{C}\to\mathbf{D}$ be pseudo-functors. A *pseudo-natural transformation* $\alpha\colon F\Rightarrow G$ is given by the following data:

- a morphism $\alpha_A \colon FA \to GA$ for every object $A \in \mathbf{C}$, and
- for all objects $A, B \in \mathbf{C}$ a natural isomorphism

$$\alpha_{A,B} \colon \mathbf{C}(\alpha_A, \mathrm{id}_{GB}) \circ G_{A,B} \Rightarrow \mathbf{C}(\mathrm{id}_{FA}, \alpha_B) \circ F_{A,B},$$

where $\mathbf{C}$ is the 2-functor obtained in Example 2.6.5.

Moreover, the following two coherence diagrams must commute.

$$\begin{array}{ccccc}
\alpha_A & = & \mathrm{id}_{GA} \circ \alpha_A & \xRightarrow{G_{\mathrm{id}_A} \mathrm{id}_{\alpha_A}} & G(\mathrm{id}_A) \circ \alpha_A \\
\| & & & & \Downarrow{\scriptstyle \alpha_{\mathrm{id}_A}} \\
\alpha_A \circ \mathrm{id}_{FA} & & \xRightarrow{\mathrm{id}_{\alpha_A} F_{\mathrm{id}_A}} & & \alpha_A \circ F(\mathrm{id}_A)
\end{array}$$

$$\begin{array}{ccccc}
Gg \circ Gf \circ \alpha_A & \xRightarrow{\mathrm{id}_{Gg} \alpha_f} & Gg \circ \alpha_A \circ Ff & \xRightarrow{\alpha_g \mathrm{id}_{Gf}} & \alpha_A \circ Fg \circ Ff \\
\Downarrow{\scriptstyle G(g,f)\mathrm{id}_{\alpha_A}} & & & & \Downarrow{\scriptstyle \mathrm{id}_{\alpha_A} F(g,f)} \\
G(g \circ f) \circ \alpha_A & & \xRightarrow{\alpha_{g \circ f}} & & \alpha_A \circ F(g \circ f)
\end{array}$$

As before, we obtain a strict version by requiring that $\alpha_{A,B}$ is the identity. ◂

Definition 2.6.7. A *pseudo-adjunction* between 2-categories $\mathbf{C}$ and $\mathbf{D}$ is a pair of pseudo-functors $F \colon \mathbf{C} \to \mathbf{D}$ and $G \colon \mathbf{D} \to \mathbf{C}$, such that for each $A \in \mathbf{C}$ and $B \in \mathbf{D}$ there is a equivalence of categories

$$\mathbf{D}(FA, B) \simeq \mathbf{C}(A, GB)$$

that is pseudo-natural in both A and B. We say that F is left pseudo-adjoint to G, and denote this by $F \dashv G$. ◂

Definition 2.6.8. We say that a 2-category $\mathbf{C}$ has binary *pseudo-coproducts* and *pseudo-products*, if there is a left pseudo-adjoint $+ \dashv \Delta$, respectively a right pseudo-adjoint $\Delta \dashv \times$, to the diagonal 2-functor $\Delta \colon \mathbf{C} \to \mathbf{C} \times \mathbf{C}$. Moreover, a 2-category with pseudo-products $\mathbf{C}$ is said to have *pseudo-exponents* if for each object $A \in \mathbf{C}$, the one-sided product pseudo-functor $(-) \times A \colon \mathbf{C} \to \mathbf{C}$ has a right pseudo-adjoint $(-) \times A \vdash (-)^A$. Finally, an object $\top$ in $\mathbf{C}$ is said to be *pseudo-final*, if for every object A in $\mathbf{C}$ we have $\mathbf{C}(A, \top) \simeq \mathbf{1}$, that is, the hom-category $\mathbf{C}(A, \top)$ is equivalent to the final category. ◂

Definition 2.6.8 is very compact and generalises directly adjunctions for ordinary categories, but is also very dense at that. So let us unfold what the definition for pseudo-products actually says. Let A and B be objects in $\mathbf{C}$. Then there is an object $A \times B$ with projections $\pi_1 \colon A \times B \to A$ and $\pi_2 \colon A \times B \to B$, such that

- for all $f \colon C \to A$ and $g \colon C \to B$ there is an $h \colon C \to A \times B$ with isomorphism $\pi_1 \circ h \cong f$ and $\pi_2 \circ h \cong g$, and
- for all $h, k \colon C \to A \times B$ and 2-morphisms $\alpha \colon \pi_1 \circ h \Rightarrow \pi_1 \circ k$ and $\beta \colon \pi_2 \circ h \Rightarrow \pi_2 \circ k$, there is a unique $\gamma \colon h \Rightarrow k$ such that $\pi_1 \gamma = \alpha$ and $\pi_2 \gamma = \beta$.

For pseudo-coproducts we have of course the dual situation. So let us instead spell out the definition of pseudo-exponents. Let A be an object in $\mathbf{C}$. We denote the functors that mediate the hom-categories for the exponential objects by

$$\alpha_{B,C} : \mathbf{C}(B \times A, C) \simeq \mathbf{C}(B, C^A) : \beta_{B,C}.$$

The pseudo-naturality of α reads then for $u\colon B' \to B$, $v\colon C \to C'$ and $k\colon B \times A \to C$ as

$$\begin{aligned} \alpha_{B',C'}(v \circ k \circ u \times A) &= \alpha_{B',C'}(\mathbf{C}(u \times A, v)(k)) \\ &\cong \mathbf{C}(u, v^A)(\alpha_{B,C}(k)) \\ &= v^A \circ \alpha(k) \circ u \end{aligned} \tag{2.1}$$

We now have for each object B an evaluation morphism $\mathrm{ev}_B\colon B^A \times A \to B$ that is given by $\mathrm{ev}_B = \beta_{B^A,B}(\mathrm{id}_{B^A})$. Moreover, for every $f\colon B \times A \to C$ we find a morphism $\lambda f\colon A \to C^B$ by putting $\lambda f = \alpha_{B,C}(f)$. This morphism has then the expected property, only weakening the usual one for exponents:

$$\mathrm{ev}_C \circ (\lambda f \times A) \cong f.$$

To see this, we can just give the usual argument, only replacing identities with isomorphisms. First, we have for $g \coloneqq \alpha_{B,C}(\mathrm{ev}_C \circ (\lambda f \times A))$

$$\begin{aligned} g &= \alpha_{B,C}(\mathrm{ev}_C \circ (\lambda f \times A)) \\ &= \alpha_{B,C}(\beta(\mathrm{id}) \circ (\alpha(f) \times A)) && \text{by definition} \\ &\cong \alpha_{B,C}(\beta(\mathrm{id})) \circ \alpha(f) && \text{(2.1) with } v = \mathrm{id}, u = \alpha(f), k = \beta(\mathrm{id}) \\ &\cong \alpha(f) && \alpha, \beta \text{ form an equivalence.} \end{aligned}$$

This gives us now

$$\mathrm{ev}_C \circ (\lambda f \times A) \cong \beta(\alpha(\mathrm{ev}_C \circ (\lambda f \times A))) \cong \beta(g) \cong f,$$

as required. Using similar reasoning, we can now also show that the abstraction is unique up to unique isomorphism, just as we have seen that the pairing for pseudo-products is unique up to unique isomorphism.

2.6.2. Algebras and Coalgebras for Pseudo-Functors

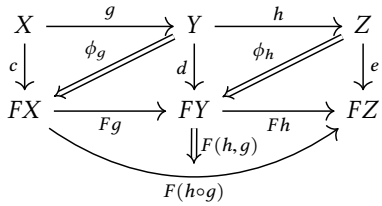
Definition 2.6.9. Given a 2-category $\mathbf{C}$ and a pseudo-functor $F\colon \mathbf{C} \to \mathbf{C}$, we call a morphism $c\colon X \to FX$ an *F-coalgebra*. A *pseudo-homomorphism* of F-coalgebras $c\colon X \to FX$ and $d\colon Y \to FY$ is a morphism $h\colon X \to Y$, such that there is an 2-isomorphism $\phi_h\colon d \circ h \Rightarrow Fh \circ c$. *$F$-algebras* and their homomorphisms are given by duality, with the 2-isomorphism being denoted by θ. ◀

Lemma 2.6.10. *Let $\mathbf{C}$ be a 2-category and $F\colon \mathbf{C} \to \mathbf{C}$ be a pseudo-functor. There are 2-categories $\mathrm{Alg}(F)$ and $\mathrm{CoAlg}(F)$ of algebras and, respectively, coalgebras and their pseudo-homomorphisms.*

Proof. We carry out the proof for $\mathrm{CoAlg}(F)$, that for $\mathrm{Alg}(F)$ follows by duality. The 2-category $\mathrm{CoAlg}(F)$ is given by

$$\mathrm{CoAlg}(F) = \begin{cases} \text{objects:} & \text{pairs } (X, c\colon X \to FX) \text{ of objects and coalgebras in } \mathbf{C} \\ \text{morphisms:} & \text{pseudo-homomorphisms } (X, c) \to (Y, d) \\ \text{2-cells:} & \text{all 2-morphisms of } \mathbf{C} \text{ between pseudo-homomorphisms} \end{cases}$$

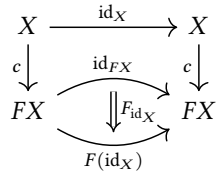
Composition and identities are given as in **C**, so we only need to prove that pseudo-homomorphism (the 1-morphisms of CoAlg(F)) are closed under composition and have identities. So let $g\colon (X,c) \to (Y,d)$ and $h\colon (Y,d) \to (Z,e)$ be pseudo-homomorphisms. Then we have the following situation.



We can paste the given 2-isomorphisms together to obtain

$$F(h,g) \circ_1 (\phi_h \circ_0 \phi_g)\colon e \circ (h \circ g) \Rightarrow F(h \circ g) \circ c,$$

which is again a 2-isomorphism and can thus be chosen for $\phi_{h\circ g}$. Hence, $h \circ g$ is again a pseudo-homomorphism. As for the identity, we have



so that we can choose $\phi_{\mathrm{id}_X} = F_{\mathrm{id}_X}$, which makes the identity a pseudo-homomorphism $c \to c$. □

Definition 2.6.11. Given a pseudo-functor F, a *pseudo-final coalgebra* is a pseudo-final object in CoAlg$F$. Dually, a *pseudo-initial algebra* is a pseudo-initial object in Alg(F).

Note that similar to the more explicit description of pseudo-products we gave above, a pseudo-final coalgebra is given by a coalgebra $\omega\colon \Omega \to F\Omega$, such that for every coalgebra $d\colon X \to FX$ there is a pseudo-homomorphism $f\colon X \to \Omega$ that is unique *up to unique isomorphism.* That is to say, for every pseudo-homomorphism $g\colon X \to \Omega$, there is a unique isomorphism $h \cong g$.

Notes

[9] Sometimes pseudo-functors are also called *weak 2-functors* [Lei04]. Since this terminology is not so commonly used, we opt for "pseudo-functor". Note also that the definition of pseudo-functor can be relaxed further by dropping the requirement that F_{id_A} and $F(g,f)$ are isomorphisms, thus obtaining the notion of *lax functor*, see [Lei04]. We will not need to make use of these in this thesis though.

Inductive-Coinductive Programming

> *We can forgive a man for making a useful thing as long as he does not admire it. The only excuse for making a useless thing is that one admires it intensely.*
>
> — Oscar Wilde, "The Picture of Dorian Gray", 1891.

Though this thesis is about mixed inductive-coinductive reasoning, we first need something to reason *about*. Given that the focus of this thesis lies on proofs that can be checked by computers, a good starting point is the reasoning about programs. So the goal of this chapter is to establish programming calculi that allow us manipulate inductive-coinductive data.

In Section 3.1, we introduce a typed λ-calculus $\lambda\mu\nu$ that has at its heart iteration and coiteration schemes for defining functions out of inductive types and into coinductive types, respectively. This calculus comes with a notion of computation that is defined through a reduction relation, which allows us to execute programs on data. The reduction relation enjoys very good properties like confluence, progression and strong normalisation. However, the calculus itself is difficult to use, especially once we start mixing inductive and coinductive types.

This leads us to consider in Section 3.2 another calculus $\lambda\mu\nu{=}$ in which the iteration and coiteration schemes are replaced by recursive equations. For example, we can define in $\lambda\mu\nu{=}$ the addition of natural numbers simply by case distinction and recursion: $0 + m = m$ and $(\mathrm{suc}\, n) + m = \mathrm{suc}(n + m)$, whereas in $\lambda\mu\nu$ we have to use higher-order iteration. This makes it much easier to specify programs in the calculus, but we lose the property that all computations are terminating, that is, there are programs on which the reduction relation associated with the calculus is not strongly normalising anymore. We come back to this problem in Chapter 4.

In the last Section 3.3, we relate the two calculi by translating the terms of $\lambda\mu\nu$ to terms in $\lambda\mu\nu{=}$. This shows that the iteration and coiteration schemes can be emulated as recursive equations in $\lambda\mu\nu{=}$. Moreover, this also allows us to infer properties, like confluence, of $\lambda\mu\nu$ from those of $\lambda\mu\nu{=}$.

Original Publication The calculus presented in Section 3.1 has not appeared in any of the author's publications but similar calculi can be found throughout the literature. We will discuss this in Section 3.4 at the end of the chapter. The content of Section 3.2 is largely based on Basold and Hansen [BH16], which considers a variation of the copattern calculus given by Abel et al. [Abe+13].

3.1. Programming with Iteration and Coiteration

In this section we give a first introduction to mixed inductive-coinductive programming by devising a simply typed calculus that features both inductive and coinductive data types. The calculus is set up so as to avoid issues related to program termination, something we will deal with in Section 3.2 and Section 4.1. For easier reference, we will refer to the calculus presented in this section as $\lambda\mu\nu$ and that presented in Section 3.2 as $\lambda\mu\nu{=}$.

$$\frac{X \in \Theta}{\Theta \Vdash X : \mathbf{Ty}} \qquad \frac{}{\Theta \Vdash \mathbf{1} : \mathbf{Ty}} \qquad \frac{\Theta \Vdash A : \mathbf{Ty} \qquad \Theta \Vdash B : \mathbf{Ty}}{\Theta \Vdash A + B : \mathbf{Ty}} \qquad \frac{\Theta \Vdash A : \mathbf{Ty} \qquad \Theta \Vdash B : \mathbf{Ty}}{\Theta \Vdash A \times B : \mathbf{Ty}}$$

$$\frac{\Vdash A : \mathbf{Ty} \qquad \Theta \Vdash B : \mathbf{Ty}}{\Theta \Vdash A \to B : \mathbf{Ty}} \qquad \frac{\Theta, X \Vdash A : \mathbf{Ty}}{\Theta \Vdash \mu X.\, A : \mathbf{Ty}} \qquad \frac{\Theta, X \Vdash A : \mathbf{Ty}}{\Theta \Vdash \nu X.\, A : \mathbf{Ty}}$$

Figure 3.1.: Type construction rules

3.1.1. Types and Terms of the Calculus $\lambda\mu\nu$

We first introduce the types with respect to which the calculus will be typed.

Definition 3.1.1. Let TyVar be a countably infinite set of type variables, elements of which we denote by X, Y, Z possibly with subscript indices. We say that A is a *raw type* if it is generated by the following grammar.

$$A, B ::= X \mid \mathbf{1} \mid A + B \mid A \times B \mid A \to B \mid \mu X.\, A \mid \nu X.\, B$$

To avoid ambiguities, we adopt the following common conventions for binding of type operators. First of all, one may always use parentheses to disambiguate. Further, $\times$ binds stronger than $\to$, which binds stronger than $+$; $\times$ and $+$ are left-associative; $\to$ is right-associative; and the binding of fixed point types extends from the dot all the way to the right.

If a raw type A is strictly positive, that is, if type variables never occur left of an arrow, then A is a *type*. More precisely, A is a type if for some sequence Θ of type variables, $\Theta \Vdash A : \mathbf{Ty}$ can be derived inductively from the rules in Figure 3.1. In case Θ is empty, we say that A is *closed type*. The set of all closed types is denoted by Ty. Sometimes we have to be a bit more precise about the variables that are actually used in a type. To this end, we define for a (raw) type A the set $\mathrm{fv}(A)$ of *free variables*, by $\mathrm{fv}(X) = \{X\}$, $\mathrm{fv}(\mathbf{1}) = \emptyset$ and in the other cases in the obvious way. Finally, we call a map $\upsilon\colon \Theta \to \mathrm{Ty}$ a *type substitution* and denote the set of all such substitutions by $\mathrm{TySubst}(\Theta)$. The *empty type substitution* is denoted by $()$. ◂

Let us give a few examples of Definition 3.1.1.

Example 3.1.2. The first example is a type that is supposed to resemble the empty set:

$$\mathbf{0} := \mu X.\, X.$$

We will see that the idea of this type is that any potential inhabitant would have to represent a non-terminating computation. Since we will exclude such computations, the type will have no inhabitants, as required.

Dually to the empty type, we can define a type that corresponds to a singleton set by

$$\mathbf{1}' := \nu X.\, X.$$

In Example 3.1.6 we show that this type has an inhabitant, but it is only in Chapter 4 that we are able to prove that this inhabitant is canonical, thus that $\mathbf{1}'$ is isomorphic to $\mathbf{1}$.

A non-trivial example is the type of natural numbers, which is given by

$$\mathrm{Nat} \coloneqq \mu X.\, \mathbf{1} + X.$$

As usual, the idea is that this type has two constructors, one for the number 0 and one for the successor of a given number. We will see this in Example 3.1.7. Streams, on the other hand, are characterised by the two destructors of taking the head (first element) of a stream and the tail (the remaining stream after the head). So for a given type A, the type of streams over A is given by the following coinductive type.

$$A^{\omega} \coloneqq \nu X.\, A \times X$$

We expose the head and tail destructors in Example 3.1.8. ◂

These are well-known, classical examples of inductive and coinductive types. Let us now come to types that properly mix inductive and coinductive types.

Example 3.1.3. One example are streams over given types A and B in which infinitely many elements of type A occur, called *left-fair streams*. These streams can be defined as type by

$$\mathrm{LFair}\, A\, B \coloneqq \nu X.\, \mu Y.\, A \times X + B \times Y.$$

Note that this type should be read as $\nu X.\, (\mu Y.\, ((A \times X) + (B \times Y)))$ according to the conventions in Definition 3.1.1. Another example that illustrates nested fixed points are A-labelled, finitely-branching, potentially infinite, nonempty trees, which are given by

$$\mathrm{Tr}_A \coloneqq \nu X.\, A \times (\mu Y.\, \mathbf{1} + X \times Y).$$

We will come back to these types in Section 3.2. ◂

Finally, non-examples of types are $\mu X.\, X \to X$ and $\mu X.\, (X \to \mathbf{1}) \to \mathbf{1}$. Both types are forbidden because the type variable X occurs left of an arrow. The latter type is commonly referred to as a (non-strictly) positive type because X occurs left of an even number of arrows. Positive types can be acceptable from a computational perspective [Abe04; Gre92; Mat99; Men91; UV96]. However, allowing non-strictly positive types would make a lot of the development in this thesis much harder, especially that in Section 4.1. Thus, we rule out non-strictly positive types here. Note that the first type would allow us to embed the untyped λ-calculus into the calculus we introduce below, see [BDS13, Sec. 9.3], and thus would lead to the existence of non-normalising terms.

We now introduce the terms of the calculus $\lambda\mu\nu$.

Definition 3.1.4. Let TeVar be a countably infinite set of *term variables*, the elements of which we denote by $x, y, z, x_1, x_2, \ldots$ The *raw terms of* $\lambda\mu\nu$ are generated by the following grammar.

$$\begin{aligned} s, t, u \quad ::= {} & \langle\rangle \mid \kappa_1\, t \mid \kappa_2\, t \mid \alpha\, t \mid \pi_1\, t \mid \pi_2\, t \mid \xi\, t \mid t\, s \mid \langle t, s \rangle \\ & \mid x \mid \lambda x.\, t \mid \{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\, u && x, y \in \mathrm{TeVar} \\ & \mid \mathrm{iter}^{\mu X.\, A}\, (x.\, t)\, s \mid \mathrm{coiter}^{\nu X.\, A}\, (x.\, t)\, s && X \Vdash A : \mathbf{Ty} \end{aligned}$$

We adopt the usual convention that application is left-associative to again allow for disambiguation of raw terms. This applies also to κ_1, π_1 etc.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textbf{(Proj)} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textbf{(1-I)}$$

$$\frac{i \in \{1,2\} \qquad \Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i\, t : A_i}\ (\times\textbf{-E}) \qquad \frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2\rangle : A \times B}\ (\times\textbf{-I})$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B}\ (\to\textbf{-E}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (\to\textbf{-I})$$

$$\frac{i \in \{1,2\} \qquad \Gamma \vdash t : A_i}{\Gamma \vdash \kappa_i\, t : A_1 + A_2}\ (+\textbf{-I})$$

$$\frac{\Gamma, x : A \vdash s : C \qquad \Gamma, y : B \vdash t : C \qquad \Gamma \vdash u : A + B}{\Gamma \vdash \{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\, u : C}\ (+\textbf{-E})$$

$$\frac{\Gamma \vdash t : A[\mu X.\, A/X]}{\Gamma \vdash \alpha\, t : \mu X.\, A}\ (\mu\textbf{-I}) \qquad \frac{\Gamma, x : A[B/X] \vdash t : B \qquad \Gamma \vdash s : \mu X.\, A}{\Gamma \vdash \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\ s : B}\ (\mu\textbf{-E})$$

$$\frac{\Gamma \vdash t : \nu X.\, A}{\Gamma \vdash \xi\, t : A[\nu X.\, A/X]}\ (\nu\textbf{-E}) \qquad \frac{\Gamma, x : B \vdash t : [B/X] \qquad \Gamma \vdash s : B}{\Gamma \vdash \mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\ s : \nu X.\, A}\ (\nu\textbf{-I})$$

Figure 3.2.: Well-Typed Terms of $\lambda\mu\nu$

A *context* Γ is a possibly empty sequence $x_1 : A_1, \ldots, x_n : A_n$ of variables $x_i \in \mathrm{TeVar}$ annotated with types $A_i \in \mathrm{Ty}$. We write $(x : A) \in \Gamma$, if there is an i with $x = x_i$ and $A = A_i$. A raw term t is said to be a *term of* $\lambda\mu\nu$ of type A in context Γ, if $\Gamma \vdash t : A$ can be derived inductively from the rules in Figure 3.2. The set of all $\lambda\mu\nu$-terms is denoted by Λ. ◂

Let us now go through the term constructors given in Definition 3.1.4 and explain their corresponding meaning. In general, for each type we have means to introduce and eliminate terms of the corresponding type, which is indicated in the names of the typing rules in Figure 3.2. Given a term t of type $A_1 \times A_2$, we can access its components $\pi_i\ t : A_i$ by using the projections π_i, and thus a term of type $A_1 \times A_2$ can be given by specifying its components as in $\langle t_1, t_2\rangle$. Dually, terms of type $A_1 + A_2$ are given by applying one of the constructors κ_i, and terms of this type are eliminated by using the case distinction $\{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\, u$. It should be noted that the case distinction binds the variables x and y in s and t, respectively. Next, terms of function type are, as usual, eliminated by application **(→-E)** and introduced by λ-abstraction **(→-I)**. Finally, the recursive types also have a dual set of introduction and elimination principles. Given a type A with $X \Vdash A : \mathbf{Ty}$, the terms of least fixed point type $\mu X.\, A$ are introduced through the constructor α and eliminated by means of iteration $\mathrm{iter}^{\mu X.\, A}\,(x.\, t)\ s$. The term t with $x : A[B/X] \vdash t : B$ in this iteration should thereby be thought of as an algebra for A in the category theoretical sense, so that $\mathrm{iter}^{\mu X.\, A}\,(x.\, t)$ becomes the homomorphism from $\mu X.\, A$ to B. For the greatest fixed point we have the dual concepts, in the sense that we can observe terms of $\nu X.\, A$ by using the destructor ξ, and we can introduce terms by means of coiteration on coalgebras. Since the type superscript of iter and coiter can hinder readability, we leave it out whenever the type is understood from the context.

To further clarify the intention of the term constructors, let us give some example programs in this calculus.

Example 3.1.5. Recall that we have defined the empty type by $\mathbf{0} = \mu X.\, X$. Let us now show how this type resembles the empty set, at least from an abstract perspective, by showing that for any type B there is map from $\mathbf{0}$ to B. To this end, we give a term $E_B^{\mathbf{0}}$ of type $\mathbf{0} \to B$, just like for any set U there is a map $\emptyset \to U$. This term is given by

$$E_B^{\mathbf{0}} \coloneqq \lambda x.\, \mathrm{iter}^{\mathbf{0}}\, (y.\, y)\, x,$$

That $E_B^{\mathbf{0}}$ is well-typed can be seen by the following type derivation, where we use that $X[B/X] = B$.

$$\dfrac{\dfrac{\dfrac{}{x : \mathbf{0}, y : \mathbf{0} \vdash y : \mathbf{0}}\,\textbf{(Proj)} \quad \dfrac{}{x : \mathbf{0} \vdash x : \mathbf{0}}\,\textbf{(Proj)}}{x : \mathbf{0} \vdash \mathrm{iter}^{\mathbf{0}}\, (y.\, y)\, x : B}\,(\mu\textbf{-E})}{\vdash \lambda x.\mathrm{iter}^{\mathbf{0}}\, (y.\, y)\, x : \mathbf{0} \to B}\,(\to\textbf{-I})$$

From a category theoretical perspective, $\mathbf{0}$ thus behaves almost like an initial object, only that $E_B^{\mathbf{0}}$ does not have to be unique, see Lemma 4.2.6. However, it is unique under observational equivalence, which we introduce in Section 4.1. See also Theorem 4.2.14. ◂

Example 3.1.6. Dually to Example 3.1.5, we have that the type $\mathbf{1}' = \nu X.\, X$ corresponds to a singleton set. To show this, we define for every type B a term of type $B \to \mathbf{1}'$ by

$$I_B^{\mathbf{1}'} \coloneqq \lambda x.\, \mathrm{coiter}^{\mathbf{1}'}\, (y.\, y)\, x.$$

This term allows us to introduce terms of type $\mathbf{1}$. In particular, we can apply $I_{(-)}^{\mathbf{1}}$ to the type $T = \mathbf{1}' \to \mathbf{1}'$ and the identity id_T with $\mathrm{id}_T = \lambda x.\, x$ on T to obtain

$$\langle\rangle' \coloneqq I_T^{\mathbf{1}'}\, \mathrm{id}_T,$$

which is then clearly of type $\mathbf{1}'$. So $\mathbf{1}'$ has almost the property of being a final object, only that we need to show that $I_B^{\mathbf{1}'}$ is unique. We come back to that in Section 4.2.2. ◂

Let us now come to some standard examples for functions on natural numbers.

Example 3.1.7. Recall that we defined the type of natural numbers to be $\mathrm{Nat} = \mu X.\, \mathbf{1} + X$. First, note that for each $t : \mathrm{Nat}$ we can define its successor by

$$\mathrm{suc}\, t \coloneqq \alpha\, (\kappa_2\, t).$$

To show that $\mathrm{suc}\, t$ is of type Nat, we use $(\mathbf{1} + X)[\mathrm{Nat}/X] = \mathbf{1} + \mathrm{Nat}$ in the following derivation.

$$\dfrac{\dfrac{\Gamma \vdash t : \mathrm{Nat}}{\Gamma \vdash \kappa_2\, t : \mathbf{1} + \mathrm{Nat}}}{\Gamma \vdash \alpha\, (\kappa_2\, t) : \mathrm{Nat}}$$

We can use suc to represent all natural numbers $n \in \mathbb{N}$ as terms $\underline{n} : \mathrm{Nat}$ inductively as follows.

$$\underline{0} \coloneqq \alpha\, (\kappa_1\, \langle\rangle)$$
$$\underline{n+1} \coloneqq \mathrm{suc}\, \underline{n}$$

Next, we define addition of natural numbers as the term plus : Nat → Nat → Nat by iteration.

$$g_+ \coloneqq \lambda m.\, \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ f \mapsto \mathrm{suc}\ (f\ m)\}\ x$$
$$\mathrm{plus} \coloneqq \lambda n\ m.\, \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ n\ m$$

To see that this definition is type correct, we first show that

$$x : 1 + (\mathrm{Nat} \to \mathrm{Nat}) \vdash g_+ : \mathrm{Nat} \to \mathrm{Nat}$$

by means of the following derivation, where we use $\Gamma = x : 1 + (\mathrm{Nat} \to \mathrm{Nat}), m : \mathrm{Nat}$.

$$\dfrac{\dfrac{}{\Gamma, y : \mathbf{1} \vdash m : \mathrm{Nat}} \quad \dfrac{\dfrac{}{\Gamma, f : \mathrm{Nat} \to \mathrm{Nat} \vdash f\ m : \mathrm{Nat}}}{\Gamma, f : \mathrm{Nat} \to \mathrm{Nat} \vdash \mathrm{suc}\ (f\ m) : \mathrm{Nat}} \quad \dfrac{}{\Gamma \vdash x : 1 + (\mathrm{Nat} \to \mathrm{Nat})}}{\dfrac{\Gamma \vdash \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ f \mapsto \mathrm{suc}\ (f\ m)\}\ x : \mathrm{Nat}}{x : 1 + (\mathrm{Nat} \to \mathrm{Nat}) \vdash \lambda m.\, \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ f \mapsto \mathrm{suc}\ (f\ m)\}\ x : \mathrm{Nat} \to \mathrm{Nat}}}$$

Second, we use the type of g_+ to show that plus is type correct:

$$\dfrac{\dfrac{\dfrac{\dfrac{n : \mathrm{Nat}, m : \mathrm{Nat}, x : 1 + (\mathrm{Nat} \to \mathrm{Nat}) \vdash g_+ : \mathrm{Nat} \to \mathrm{Nat} \qquad n : \mathrm{Nat}, m : \mathrm{Nat} \vdash n : \mathrm{Nat}}{n : \mathrm{Nat}, m : \mathrm{Nat} \vdash \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ n : \mathrm{Nat} \to \mathrm{Nat}}}{n : \mathrm{Nat}, m : \mathrm{Nat} \vdash \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ n\ m : \mathrm{Nat}}}{\vdash \lambda n\ m.\, \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ n\ m : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}}$$

Now that we have convinced ourselves that plus is well-typed, let us understand the idea of this definition. Since the type of plus is Nat → (Nat → Nat), it immediately suggests itself to define plus by iteration on the first argument. In turn, this means that the outcome of the iteration must be a function. This type of iteration over a function space is sometimes referred to as *higher order iteration.*[10] The idea of the iteration itself is that we construction a function by induction on the first argument n that corresponds to the n-fold application of the successor function to its argument m. This is implemented in g_+ as follows: in the base case we just return m itself ($\underline{0} + m = m$), and in the step case we apply the successor to the result of the induction step ($(\mathrm{suc}\, n) + m = \mathrm{suc}\, (n + m)$). This will become clearer once we understand the computational behaviour of g_+ in Example 3.1.12. ◀

Note that in Example 3.1.7 we pulled g_+ out of plus to make its definition easier to understand. We can improve readability further by making use of an equational style of giving definitions, using type annotations, and employing infix notation. The addition can then be represented as:

$$\_ + \_ : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$$
$$n + m = \mathrm{iter}^{\mathrm{Nat}}\ g_+\ n\ m$$
where
$$g_+ : 1 + (\mathrm{Nat} \to \mathrm{Nat}) \to (\mathrm{Nat} \to \mathrm{Nat})$$
$$g_+\ x\ m = \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ f \mapsto \mathrm{suc}\ (f\ m)\}\ x$$

It is clear that this style of defining terms can be reduced to terms in $\lambda\mu\nu$, as long as there are no cyclic references between definitions. In that case we can substitute the term corresponding to a defined symbol for that symbol in all other terms, thus obtaining one big term in $\lambda\mu\nu$. For example,

we can substitute the right-hand side of the definition of g_+ above into the definition of plus, thereby obtaining one term that defines plus.

Let us now come to some standard examples for the coinductive type of streams.

Example 3.1.8. Recall that the type of streams over a type A is given by $A^\omega = \nu X.\, A \times X$. We will use the following common notation for the head and tail of a stream term $s : A^\omega$.

$$\begin{aligned} \mathrm{hd}\, s &= \pi_1(\xi\, s) \\ \mathrm{tl}\, s &= \pi_2(\xi\, s) \end{aligned}$$

Given an $a : A$, we can define a stream term a^ω that is equal to a in every position by coiteration on the singleton state space:

$$\begin{aligned} &a^\omega : A^\omega \\ &a^\omega = \mathrm{coiter}^{\mathrm{Nat}^\omega}\ g_a\ \langle\rangle \\ &\text{where}\ \ g_a\, x = \langle a, x\rangle \end{aligned}$$

Again using coiteration, point-wise addition of streams over natural numbers is given by

$$\begin{aligned} &\_ \oplus \_ : \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \\ &s \oplus t = \mathrm{coiter}^{\mathrm{Nat}^\omega}\ g_\oplus\ (s, t) \\ &\text{where} \\ &\quad g_\oplus : \mathrm{Nat}^\omega \times \mathrm{Nat}^\omega \to \mathrm{Nat} \times (\mathrm{Nat}^\omega \times \mathrm{Nat}^\omega) \\ &\quad g_\oplus\, x = \big\langle \mathrm{hd}\,(\pi_1\, x) + \mathrm{hd}\,(\pi_2\, x), \langle \mathrm{tl}\,(\pi_1\, x), \mathrm{tl}\,(\pi_2\, x)\rangle \big\rangle. \end{aligned}$$

We can now give two different definitions of the stream of natural numbers.

$$\begin{aligned} &\mathrm{nats}_1 : \mathrm{Nat}^\omega \\ &\mathrm{nats}_1 = \mathrm{coiter}^{\mathrm{Nat}^\omega}\ g_1\ \underline{0} \\ &\text{where} \\ &\quad g_1 : \mathrm{Nat} \to \mathrm{Nat} \times \mathrm{Nat} \\ &\quad g_1\, n = \langle n, n + \underline{1}\rangle \end{aligned} \qquad \begin{aligned} &\mathrm{nats}_2 : \mathrm{Nat}^\omega \\ &\mathrm{nats}_2 = \mathrm{coiter}^{\mathrm{Nat}^\omega}\ g_2\ \underline{0}^\omega \\ &\text{where} \\ &\quad g_2 : \mathrm{Nat}^\omega \to \mathrm{Nat} \times \mathrm{Nat}^\omega \\ &\quad g_2\, s = \big\langle \mathrm{hd}\, s, s \oplus \underline{1}^\omega \big\rangle \end{aligned}$$

The definition on the left is quite direct by using an accumulator, whereas nats_2 corresponds to the definition typically found in literature on behavioural differential equations [NR11]. Indeed, in Example 3.1.13, we will see that both definitions have the expected *computational behaviour*. But we will only be able to prove that these two definitions have the same *observable behaviour* in Chapter 5. ◀

3.1.2. Computations in $\lambda\mu\nu$

Since we have mentioned the computational behaviour of terms frequently in the above examples, it is about time we actually say what we mean by that. Computations of terms are given by means of a reduction relation, which we will define now. As preparation, we introduce some short-hand

notation that will make the following development more readable. Let s and t be terms, then we define the following terms, which resemble those commonly used for functions.

$$\begin{aligned} \mathrm{id} &\coloneqq \lambda x.x \\ t \circ s &\coloneqq \lambda x.t\ (s\ x) \\ t + s &\coloneqq \lambda x.\{\kappa_1\ x \mapsto \kappa_1\ (t\ x)\ ;\ \kappa_2\ y \mapsto \kappa_2\ (s\ y)\}\ x \\ t \times s &\coloneqq \lambda x.\langle t\ (\pi_1\ x), s\ (\pi_2\ x)\rangle \end{aligned}$$

The second ingredient we need in order to define a reduction relation on terms of $\lambda\mu\nu$, is an *action of types* C with $X \Vdash C : \mathbf{Ty}$ on terms, analogously to the action of a functor on morphisms. By that we mean that we can derive from the type C and a term $t : A \to B$ a term $C[t] : C[A/X] \to C[B/X]$, such that for any term $s : C[A/X]$ the term $C[t]\ s$ applies t to the parts of s where the type variable X appears in the type of s, but leaves s otherwise intact. The need for this action on terms arises from the fact that the reductions for iteration and coiteration follow the usual homomorphism diagrams for algebras and coalgebras, respectively. For example, recall that the type of streams over A is given by $A^\omega = \nu X.\, A \times X$. The reduction relation on coiteration for streams will essentially implement the diagram for (final) stream coalgebras:

$$\begin{array}{ccc} B & \xrightarrow{\mathrm{coiter}^{A^\omega}\ t} & A^\omega \\ \downarrow{\scriptstyle t} & & \downarrow{\scriptstyle \xi} \\ A \times B & \xrightarrow{\mathrm{id} \times \mathrm{coiter}^{A^\omega}\ t} & A \times A^\omega \end{array}$$

Thus, the action of $A \times X$ on terms will be given by $(A \times X)[t] = \mathrm{id}_A \times t$.

Formally, given a type C with $X \Vdash C : \mathbf{Ty}$, we define for each term t a term $C[t]$, such that the following typing rule is fulfilled.

$$\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash C[t] : C[A/X] \to C[B/X]} \tag{3.1}$$

In Section 4.2 we will see that the action of types on terms is more than a mere notational similarity to functors and that terms given by (co)iteration are indeed homomorphisms under the reduction relation.

Definition 3.1.9. Let C be a type with $X_1, \ldots, X_n \Vdash C : \mathbf{Ty}$, and $\vec{A} = (A_1, \ldots, A_n)$ be a tuple of closed types. We put

$$C\big[\vec{A}\big] \coloneqq C\big[\vec{A}\,/\vec{X}\big].$$

Let $\vec{B} = (B_1, \ldots, B_n)$ be another tuple of closed types and $\vec{t} = (t_1, \ldots, t_n)$ terms with $t_k : A_k \to B_k$, which we denote by ϵ if $n = 0$. We define in Figure 3.3 a term

$$C\big[\vec{t}\big] : C\big[\vec{A}\big] \to C\big[\vec{B}\big]$$

by induction on the construction of C, where we indicate the type of the produced term on the right. Note that in the first case, we drop the arguments because C will never use them.[11] Moreover, in case of the function space, the type C_1 must be closed (due to strict positivity), thus $C_1[\vec{A}] = C_1$ and we only need to post-compose with $C_2[\vec{t}\,]$ in this case. ◂

$$
\begin{aligned}
C\big[\vec{t}\,\big] &= \mathrm{id}_C && : C \to C \text{ if } \mathrm{fv}(C) = \emptyset \\
X_k\big[\vec{t}\,\big] &= t_k && : A_k \to B_k \\
\mathbf{1}\big[\vec{t}\,\big] &= \mathrm{id} && : \mathbf{1} \to \mathbf{1} \\
(C_1 + C_2)\big[\vec{t}\,\big] &= C_1\big[\vec{t}\,\big] + C_2\big[\vec{t}\,\big] && : C_1\big[\vec{A}\big] + C_2\big[\vec{A}\big] \to C_1\big[\vec{B}\big] + C_2\big[\vec{B}\big] \\
(C_1 \times C_2)\big[\vec{t}\,\big] &= C_1\big[\vec{t}\,\big] \times C_2\big[\vec{t}\,\big] && : C_1\big[\vec{A}\big] \times C_2\big[\vec{A}\big] \to C_1\big[\vec{B}\big] \times C_2\big[\vec{B}\big] \\
(C_1 \to C_2)\big[\vec{t}\,\big] &= \lambda f.\, C_2\big[\vec{t}\,\big] \circ f && : \big(C_1 \to C_2\big[\vec{A}\big]\big) \to \big(C_1 \to C_2\big[\vec{B}\big]\big) \\
(\mu Y.\, C)\big[\vec{t}\,\big] &= \lambda x.\, \mathrm{iter}^{(\mu Y.\, C)\big[\vec{A}\big]}\,(y.\, s)\, x && : \big(\mu Y.\, C\big[\vec{A}\big]\big) \to \mu Y.\, C\big[\vec{B}\big] \\
&\quad s = \alpha\,\big(C\big[\vec{t}, \mathrm{id}\big]\, y\big) \\
(\nu Y.\, C)\big[\vec{t}\,\big] &= \lambda x.\, \mathrm{coiter}^{(\nu Y.\, C)\big[\vec{B}\big]}\,(y.\, s)\, x && : \big(\nu Y.\, C\big[\vec{A}\big]\big) \to \nu Y.\, C\big[\vec{B}\big] \\
&\quad s = C\big[\vec{t}, \mathrm{id}\big]\,(\xi\, y)
\end{aligned}
$$

Figure 3.3.: Action of Types on Terms in $\lambda\mu\nu$

Let us show that the action of a type on terms fulfils the judgement advertised in (3.1).

Lemma 3.1.10. *Let C be a type with $X \Vdash C : \mathbf{Ty}$. Then the following judgement holds.*

$$\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash C[t] : C[A] \to C[B]}$$

Proof. One proves the more general statement that for a given type C with $X_1, \ldots, X_n \Vdash C : \mathbf{Ty}$, and for every tuple of terms $\vec{t} = (t_1, \ldots, t_n)$ terms with $t_k : A_k \to B_k$, we have that $C[\vec{t}\,]$ is of type $C[\vec{A}\,] \to C[\vec{B}\,]$. The proof of this is then an easy induction on C, which just uses the type annotations we gave in Definition 3.1.9. □

We now define the reduction relation of $\lambda\mu\nu$. This is done in two steps: First, we define a contraction relation, which takes care of reductions on the outermost term constructors. Second, we use contraction to define reductions in arbitrary positions in terms.

Definition 3.1.11. The *contraction relation* $\succ$ between terms, or just *contraction* of terms, in $\lambda\mu\nu$ is is defined by the following clauses.

$$
\begin{aligned}
\{\kappa_1\, x_1 \mapsto t_1\ ;\ \kappa_2\, x_2 \mapsto t_2\}\ (\kappa_i\, s) &\succ t_i[s/x_i] \\
\pi_i\, \langle t_1, t_2 \rangle &\succ t_i \\
(\lambda x.\, t)\, s &\succ t[s/x] \\
\mathrm{iter}^{\mu X.\, A}\,(x.\, t)\,(\alpha\, s) &\succ t\,\big[A\big[\lambda y.\, \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\, y\big]\, s/x\big] \\
\xi\,\Big(\mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\, s\Big) &\succ A\big[\lambda y.\mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\, y\big]\,(t[s/x])
\end{aligned}
$$

We say that a term t *reduces* to a term s in $\lambda\mu\nu$, if $t \longrightarrow s$ can be derived inductively from the rules in Figure 3.4. The *reduction relation of $\lambda\mu\nu$* $\longrightarrow$ is said to be given as the *compatible closure* of contraction. Finally, *iterated reduction* $\longrightarrow\!\!\!\!\!\rightarrow$ is defined as the reflexive, transitive closure of $\longrightarrow$ and *convertibility* $\equiv$ as the equivalence closure of $\longrightarrow$. ◀

$$\frac{t \succ s}{t \longrightarrow s} \qquad \frac{t \longrightarrow s}{\pi_i\, t \longrightarrow \pi_i\, s} \qquad \frac{t \longrightarrow t'}{\langle t, s\rangle \longrightarrow \langle t', s\rangle} \qquad \frac{s \longrightarrow s'}{\langle t, s\rangle \longrightarrow \langle t, s'\rangle} \qquad \frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s} \qquad \frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

$$\frac{t \longrightarrow s}{\lambda x.\, t \longrightarrow \lambda x.\, s} \qquad \frac{s \longrightarrow s'}{\{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t\}\ u \longrightarrow \{\kappa_1\ x \mapsto s'\ ;\ \kappa_2\ y \mapsto t\}\ u}$$

$$\frac{t \longrightarrow t'}{\kappa_i\, t \longrightarrow \kappa_i\, t'} \qquad \frac{t \longrightarrow t'}{\{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t\}\ u \longrightarrow \{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t'\}\ u}$$

$$\frac{t \longrightarrow t'}{\alpha\, t \longrightarrow \alpha\, t'} \qquad \frac{t \longrightarrow t'}{\xi\, t \longrightarrow \xi\, t'} \qquad \frac{u \longrightarrow u'}{\{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t\}\ u \longrightarrow \{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t\}\ u'}$$

$$\frac{t \longrightarrow t'}{(\mathrm{co})\mathrm{iter}\,(x.\, t)\ s \longrightarrow (\mathrm{co})\mathrm{iter}\,(x.\, t')\ s} \qquad \frac{s \longrightarrow s'}{(\mathrm{co})\mathrm{iter}\,(x.\, t)\ s \longrightarrow (\mathrm{co})\mathrm{iter}\,(x.\, t)\ s'}$$

Figure 3.4.: Compatible Closure of Contraction in $\lambda\mu\nu$

Let us illustrate the reduction relation by means of addition on natural numbers and streams.

Example 3.1.12. We show that addition of natural numbers, defined in Example 3.1.7, has the expected computational behaviour on zero and successors. For $C = \mathbf{1} + X$, we have for any terms u and t the following.

$$\begin{aligned} C[u]\ t &= (\mathbf{1}[u] + X[u])\ t \\ &= (\mathrm{id}_{\mathbf{1}} + u)\ t \\ &= (\lambda x.\, \{\kappa_1\ y \mapsto \kappa_1\ (\mathrm{id}_{\mathbf{1}}\ y)\ ;\ \kappa_2\ z \mapsto \kappa_2\ (u\ z)\}\ x)\ t \\ &\succ \{\kappa_1\ y \mapsto \kappa_1\ (\mathrm{id}_{\mathbf{1}}\ y)\ ;\ \kappa_2\ z \mapsto \kappa_2\ (u\ z)\}\ t \\ &\longrightarrow \{\kappa_1\ y \mapsto \kappa_1\ y\ ;\ \kappa_2\ z \mapsto \kappa_2\ (u\ z)\}\ t \qquad \text{by } \mathrm{id}_{\mathbf{1}}\ y \succ y \end{aligned} \tag{3.2}$$

Recall that g_+ was defined in Example 3.1.7 by $g_+ = \lambda m.\, \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ f \mapsto \mathrm{suc}\ (f\ m)\}\ x$. Using this and (3.2), we get for any term t of type Nat that $\underline{0} + t \longrightarrow\!\!\!\!\!\rightarrow t$ and hence $\underline{0} + t \equiv t$ by the sequence of reduction steps in Figure 3.5, where we use $R \coloneqq \lambda y.\, \mathrm{iter}\,(x.\, g_+)\ y$ as a short-hand notation. Similarly, we get for $s : \mathrm{Nat}$ the reduction sequence in Figure 3.6. Now we note that

$$\mathrm{suc}\ (\mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ s\ t) \longleftarrow \mathrm{suc}\ ((\lambda n\ m.\, \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+)\ n\ m)\ s\ t) = \mathrm{suc}\ (s + t),$$

so that $(\mathrm{suc}\ s) + t \equiv \mathrm{suc}\ (s + t)$ as expected.[12] ◀

Example 3.1.13. Similarly to the last example, we demonstrate that the terms given in Example 3.1.8 have the expected computational behaviour. For the constant streams we have

$$\begin{aligned} \mathrm{hd}\ a^{\omega} = \mathrm{hd}\ (\mathrm{coiter}\ g_a\ \langle\rangle) &\equiv \pi_1\ ((\mathrm{id} \times (\mathrm{coiter}\ g_a))\ (g_a\ \langle\rangle)) \\ &\equiv \pi_1\ ((\mathrm{id} \times (\mathrm{coiter}\ g_a))\ \langle a, \langle\rangle\rangle) \\ &\equiv \pi_1\ \langle a, (\mathrm{coiter}\ g_a\ \langle\rangle)\rangle \\ &\equiv a, \end{aligned}$$

and

$$\mathrm{tl}\ a^{\omega} = \mathrm{tl}\ (\mathrm{coiter}\ g_a\ \langle\rangle) \equiv \cdots \equiv \pi_2\ \langle a, (\mathrm{coiter}\ g_a\ \langle\rangle)\rangle \equiv \mathrm{coiter}\ g_a\ \langle\rangle = a^{\omega}.$$

$$
\begin{array}{ll}
\underline{0} + t & \\
= (\lambda n\, m.\, \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, n\, m)\, \underline{0}\, t & \\
\succ (\lambda m.\, \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, \underline{0}\, m)\, t & \\
\succ \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, \underline{0}\, t & \\
= \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, (\alpha\, (\kappa_1\, \langle\rangle))\, t & \\
\longrightarrow g_+\, [C[R]\, (\kappa_1\, \langle\rangle)/x]\, t & \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, (\alpha\, (\kappa_1\, \langle\rangle)) \succ \ldots \\
\longrightarrow g_+[\{\kappa_1\, y \mapsto \kappa_1\, y\ ;\ \kappa_2\, z \mapsto \kappa_2\, (R\, z)\}\, (\kappa_1\, \langle\rangle)/x]\, t & \text{by (3.2)} \\
\longrightarrow g_+[\kappa_1\, \langle\rangle/x]\, t & \{\kappa_1\, x \mapsto \kappa_1\, x\ ;\ \cdots\}\, (\kappa_1\, \langle\rangle) \succ \kappa_1\, \langle\rangle \\
= \big(\lambda m.\, \{\kappa_1\, y \mapsto m\ ;\ \kappa_2\, f \mapsto \mathrm{suc}\, (f\, m)\}\, (\kappa_1\, \langle\rangle)\big)\, t & \\
\succ \{\kappa_1\, y \mapsto t\ ;\ \kappa_2\, f \mapsto \mathrm{suc}\, (f\, t)\}\, (\kappa_1\, \langle\rangle) & \\
\succ t &
\end{array}
$$

Figure 3.5.: Reduction Sequence for $\underline{0} + t$ in $\lambda\mu\nu$

$$
\begin{array}{ll}
(\mathrm{suc}\, s) + t = (\lambda n\, m.\, \mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, n\, m)\, (\alpha\, (\kappa_2\, s))\, t & \\
\quad = \cdots & \\
\quad \longrightarrow g_+[\{\kappa_1\, y \mapsto \kappa_1\, y\ ;\ \kappa_2\, z \mapsto \kappa_2\, (\mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, z)\}\, (\kappa_2\, s)/x]\, t & \\
\quad \longrightarrow g_+\, [\kappa_2\, (\mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, s)/x]\, t & \\
\quad = \big(\lambda m.\, \{\kappa_1\, y \mapsto m\ ;\ \kappa_2\, f \mapsto \mathrm{suc}\, (f\, m)\}\, (\kappa_2\, (\mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, s))\big)\, t & \\
\quad \succ \{\kappa_1\, y \mapsto t\ ;\ \kappa_2\, f \mapsto \mathrm{suc}\, (f\, t)\}\, (\kappa_2\, (\mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, s)) & \\
\quad \succ \mathrm{suc}\, (\mathrm{iter}^{\mathrm{Nat}}\, (x.\, g_+)\, s\, t) &
\end{array}
$$

Figure 3.6.: Reduction Sequence for $(\mathrm{suc}\, s) + t$ in $\lambda\mu\nu$

An easy calculation also shows that

$$\mathrm{hd}\, (s \oplus t) \equiv \mathrm{hd}\, s + \mathrm{hd}\, t \qquad \text{and} \qquad \mathrm{tl}\, (s \oplus t) \equiv \mathrm{tl}\, s \oplus \mathrm{tl}\, t.$$

We can use these equivalences now to check the computational behaviour of the second definition of the stream of natural numbers:

$$
\begin{aligned}
\mathrm{hd}\, \mathrm{nats}_2 = \mathrm{hd}\, (\mathrm{coiter}\, g_2\, \underline{0}^\omega) &\equiv \pi_1\, ((\mathrm{id} \times \mathrm{coiter}\, g_2)(g_2\, \underline{0}^\omega)) \\
&\equiv \pi_1\, ((\mathrm{id} \times \mathrm{coiter}\, g_2)(\mathrm{hd}\, \underline{0}^\omega, \underline{0}^\omega \oplus \underline{1}^\omega)) \equiv \underline{0}.
\end{aligned}
$$

Note that we can continue to explicitly check that the n-th position of nats_2, given by $\mathrm{hd}\, (\mathrm{tl}^n\, \mathrm{nats}_2)$, is indeed $\underline{n}$. We introduce in Chapter 5 the necessary machinery to prove this for all n. ◀

Let us now show that the reduction relation preserves types of terms, that is, if $\Gamma \vdash t : A$ and $t \longrightarrow s$, then $\Gamma \vdash s : A$. We say then that *subject reduction* holds for $\longrightarrow$. The first step towards this was Lemma 3.1.10, where we proved that the type action is type correct, which allows us to prove subject reduction for the contraction relation introduced in Definition 3.1.11. This, in turn, gives immediately that the reduction relation preserves types by its definition as compatible closure of contraction.

Theorem 3.1.14. *The reduction relation of $\lambda\mu\nu$ preserves types.*

Proof. We first show that the contraction relation preserves types. The cases for sum types, product and function types are readily proved by a standard argument. For the contraction on recursive types, we use Lemma 3.1.10 as follows. Suppose we have $\Gamma, x : A[B/X] \vdash t : B$ and $\Gamma \vdash s : A[\mu X.\, A/X]$, so that $\Gamma \vdash \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\,(\alpha\, s) : B$. The following derivation then shows that the right-hand side of the contraction for iteration also has type B.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \lambda y.\, \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\; y : (\mu X.\, A) \to B}{\Gamma \vdash A\big[\lambda y.\, \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\; y\big] : A[\mu X.\, A] \to A[B]} \qquad \Gamma \vdash s : A[\mu X.\, A/X]}{\Gamma \vdash A\big[\lambda y.\, \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\; y\big]\; s : A[B]}}{\Gamma \vdash t\; [A\big[\lambda y.\, \mathrm{iter}^{\mu X.\, A}\,(x.\, t)\; y\big]\; s/x] : B}$$

Similarly, for $t : A[B/X]$ and $s : B$, we have that in the case of ν-types the type of the right side of the contraction matches that of the left side. This is demonstrated by the following type derivation.

$$\dfrac{\dfrac{\Gamma \vdash \lambda y.\, \mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\; y : B \to \nu X.\, A}{\Gamma \vdash A\big[\lambda y.\, \mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\; y\big] : A[B] \to A[\nu X.\, A]} \qquad \dfrac{\Gamma \vdash s : B}{\Gamma \vdash t[s/x] : A[B]}}{\Gamma \vdash A\big[\lambda y.\, \mathrm{coiter}^{\nu X.\, A}\,(x.\, t)\; y\big]\; (t[s/x]) : A[\nu X.\, A]}$$

Second, since the contraction relation preserves types and the reduction relation is the compatible closure of contraction, type preservation of reductions is immediate. □

This concludes our introduction into simple inductive-coinductive programming, and we move on to a more practical language.[13]

3.2. Programming with Equations

In the previous section, we have introduced the simply typed calculus $\lambda\mu\nu$ for programming with mixed inductive-coinductive types. It was fairly straightforward to set up the calculus and, as noted in Note 13, all terms of that calculus are strongly normalising. However, since the calculus is based on iteration and coiteration schemes, programming can become quite complicated. We have seen this already in Example 3.1.7, but the complications become even more prevalent when iteration and coiteration are mixed.

For example, suppose we want to define a map that projects a left-fair stream to its A-elements. In $\lambda\mu\nu$ we can define such a map as follows.

Example 3.2.1. To define the projection $\mathrm{proj} : \mathrm{LFair}\, A\, B \to A^\omega$, we put $U = A \times (\mathrm{LFair}\, A\, B) + B \times Y$ and let $L = \mu Y.\, U$ be the first unfolding of LFair. Then we define

$$\begin{aligned}
\mathrm{proj} &= \lambda x.\, \mathrm{coiter}^{A^\omega}\; f\; x \\
&\quad f : \mathrm{LFair}\, A\, B \to A \times \mathrm{LFair}\, A\, B \\
&\quad f\; x = \mathrm{iter}^{L}\; g\; (\xi\; x) \\
&\quad g : U[A \times \mathrm{LFair}\, A\, B/Y] \to A \times \mathrm{LFair}\, A\, B \\
&\quad g\; x = \{\kappa_1\; x \mapsto x\;;\; \kappa_2\; y \mapsto \pi_2\; y\}\; x
\end{aligned}$$

This is a short definition, but coming up with it, let alone understanding it, is rather difficult.

This situation can be improved by moving away from (co)iteration schemes and instead use equational specifications, which allow recursive references to function symbols. This is essentially how one writes programs in a functional language like Haskell. We will introduce in this section a calculus, denoted by $\lambda\mu\nu{=}$, that implements this idea by replacing the (co)iteration schemes of $\lambda\mu\nu$ by recursive equations, patterns and so-called copatterns. This calculus is a variation of the copattern calculus given by Abel et al. [Abe+13] . We discuss the differences and rationale for these changes in Section 3.4. The calculus $\lambda\mu\nu{=}$ has been studied in [BH16]. Note, however, that in loc. cit. the syntax is slightly different, in that the authors use a syntax similar to that of $\lambda\mu\nu$ for destructors of products and ν-types. The choice of syntax in this section brings the calculus $\lambda\mu\nu{=}$ closer to behavioural differential equations [Rut03], object oriented programming and the original syntax in [Abe+13]. It also emphasises that function space types are coinductive, in that its destructor, the function application, is also written in post-fix notation.

3.2.1. Types and Terms of the Calculus $\lambda\mu\nu{=}$

The types for the calculus $\lambda\mu\nu{=}$ are exactly the ones we gave in Definition 3.1.1 for $\lambda\mu\nu$. As for the terms, the introduction rules for coproducts and μ-types are the same, whereas the elimination rules for coinductive types (products, functions and ν-types) are now all written in post-fix notation. Also, the elimination rules for inductive types (sums and μ-types) and introduction rules for coinductive types will be replaced by patterns and copatterns, respectively, combined with recursive equations. Such recursive equations can thereby be given via a binding construct $\mathbf{rlet}\ \Sigma\ \mathbf{in}\ t$, where Σ contains symbols and their definitions, and t is a term. The calculus is formally given by the following definition.

Definition 3.2.2. Let TeVar and SigVar be countably infinite, disjoint sets of term variables $x, y, z, \ldots$ and signature variables $f, g, h, \ldots$, respectively. The *raw terms* s, t, *patterns* p, *copatterns* q, *declaration bodies* D and *declaration blocks* Σ of $\lambda\mu\nu{=}$ are generated by the following grammar.

$$
\begin{aligned}
s, t \quad &::= x \in \mathrm{TeVar} \mid \langle\rangle \mid \kappa_1\ t \mid \kappa_2\ t \mid \alpha\ t \mid t.\mathrm{pr}_1 \mid t.\mathrm{pr}_2 \mid t.\mathrm{out} \mid t\ s \\
&\quad \mid f \in \mathrm{SigVar} \mid \lambda D \mid \mathbf{rlet}\ \Sigma\ \mathbf{in}\ t \\
p \ &::= x \in \mathrm{TeVar} \mid \kappa_i\ p \mid \alpha\ p \\
q \ &::= \cdot \mid q.\mathrm{pr}_1 \mid q.\mathrm{pr}_2 \mid q.\mathrm{out} \mid q\ p \\
D \ &::= \{q_1 \mapsto t_1\ ;\ \ldots\ ;\ q_n \mapsto t_n\} \\
\Sigma \ &::= f_1 : A_1 = D_1, \ldots, f_n : A_n = D_n
\end{aligned}
$$

◀

To resolve ambiguities, To increase readability, we adopt here the same conventions as for $\lambda\mu\nu$ (application is left-associative and variables do not need parentheses) plus that application of destructors is also left-associative. For example, $t.\mathrm{out}.\mathrm{pr}_1$ is to be read as $(t.\mathrm{out}).\mathrm{pr}_1$. The analogous conventions also hold for patterns and copatterns.

We now define what the (well-typed) terms of $\lambda\mu\nu{=}$ are.

Definition 3.2.3. A *context* Γ is a possibly empty sequence $x_1 : A_1, \ldots, x_n : A_n$ of variables $x_i \in \mathrm{TeVar}$ annotated with types A_i. A raw term t is said to be a *term of* $\lambda\mu\nu{=}$ of type A in context

Γ using the declarations in Σ, if $\Gamma; \Sigma \vdash t : A$ can be derived inductively from the rules in Fig. 3.7. We denote the set of all well-typed terms by $\mathrm{Terms}_{\lambda\mu\nu=}$. The judgement for well-typed terms involves the following judgements, which are defined in Fig. 3.7 as well.

- $\Gamma; \Sigma \vdash_{\mathrm{bdy}} D : A$, states that D is a *declaration body* of type A in the variable context Γ using the declarations in Σ;
- $\Sigma_1 \vdash_{\mathrm{dec}} \Sigma_2$, states that Σ_2 is a well-formed *declaration block*, using declarations in Σ_1;
- $\Gamma \vdash_{\mathrm{pat}} p : A$, states that p is a *pattern* on the type A that binds variables in Γ; and
- $\Gamma \vdash_{\mathrm{cop}} q : A \Rightarrow B$, states that q is a *copattern*, binding variables in Γ, such that for all evaluation contexts e that match q, applying e to a term of type A results in a term of type B. ◀

$$\frac{(x : A) \in \Gamma}{\Gamma; \Sigma \vdash x : A}\ \textbf{(Proj)} \qquad \frac{(f : A = D) \in \Sigma}{\Gamma; \Sigma \vdash f : A}\ \textbf{(ProjSig)} \qquad \frac{}{\Gamma; \Sigma \vdash \langle\rangle : \mathbf{1}}\ \textbf{(1-I)}$$

$$(i = 1, 2)\ \frac{\Gamma; \Sigma \vdash t : A_i}{\Gamma; \Sigma \vdash \kappa_i\, t : A_1 + A_2}\ (+\textbf{-I}) \qquad \frac{\Gamma; \Sigma \vdash t : A[\mu X.\, A/X]}{\Gamma; \Sigma \vdash \alpha\, t : \mu X.\, A}\ (\mu\textbf{-I})$$

$$\frac{\Gamma; \Sigma \vdash t : A_1 \times A_2}{\Gamma; \Sigma \vdash t.\mathrm{pr}_1 : A_1}\ (\times\textbf{-E}_1) \qquad \frac{\Gamma; \Sigma \vdash t : A_1 \times A_2}{\Gamma; \Sigma \vdash t.\mathrm{pr}_2 : A_2}\ (\times\textbf{-E}_2)$$

$$\frac{\Gamma; \Sigma \vdash t : \nu X.\, A}{\Gamma; \Sigma \vdash t.\mathrm{out} : A[\nu X.\, A/X]}\ (\nu\textbf{-E}) \qquad \frac{\Gamma; \Sigma \vdash t_1 : A \to B \qquad \Gamma; \Sigma \vdash t_2 : A}{\Gamma; \Sigma \vdash t_1\ t_2 : B}\ (\to\textbf{-E})$$

$$\frac{\Gamma; \Sigma \vdash_{\mathrm{bdy}} D : A}{\Gamma; \Sigma \vdash \lambda D : A}\ \textbf{(Abs)} \qquad \frac{\Sigma_1 \vdash_{\mathrm{dec}} \Sigma_2 \qquad \Gamma; \Sigma_1, \Sigma_2 \vdash t : A}{\Gamma; \Sigma_1 \vdash \textbf{rlet}\ \Sigma_2\ \textbf{in}\ t : A}\ \textbf{(Rlet)}$$

$$\frac{\Gamma_i \vdash_{\mathrm{cop}} q_i : B \Rightarrow A_i \quad \Gamma, \Gamma_i; \Sigma \vdash t_i : A_i \quad \text{for all } 1 \le i \le n}{\Gamma; \Sigma \vdash_{\mathrm{bdy}} \{q_1 \mapsto t_1; \ldots; q_n \mapsto t_n\} : B}$$

$$\frac{\emptyset; \Sigma_1, \Sigma_2 \vdash_{\mathrm{bdy}} D : A \quad \text{for all } (f : A = D) \in \Sigma_2}{\Sigma_1 \vdash_{\mathrm{dec}} \Sigma_2}$$

$$\frac{x \in \mathrm{Var}}{x : D \vdash_{\mathrm{pat}} x : D} \qquad \frac{}{\vdash_{\mathrm{pat}} \langle\rangle : \mathbf{1}} \qquad \frac{x : D \vdash_{\mathrm{pat}} p : A_i}{x : D \vdash_{\mathrm{pat}} \kappa_i\, p : A_1 + A_2} \qquad \frac{x : D \vdash_{\mathrm{pat}} p : A[\mu X.\, A/X]}{x : D \vdash_{\mathrm{pat}} \alpha\, p : \mu X.\, A}$$

$$\frac{}{\emptyset \vdash_{\mathrm{cop}} \cdot : C \Rightarrow C} \qquad \frac{\Gamma \vdash_{\mathrm{cop}} q : C \Rightarrow A_1 \times A_2}{\Gamma \vdash_{\mathrm{cop}} q.\mathrm{pr}_1 : C \Rightarrow A_1} \qquad \frac{\Gamma \vdash_{\mathrm{cop}} q : C \Rightarrow A_1 \times A_2}{\Gamma \vdash_{\mathrm{cop}} q.\mathrm{pr}_2 : C \Rightarrow A_2}$$

$$\frac{\Gamma \vdash_{\mathrm{cop}} q : C \Rightarrow \nu X.\, A}{\Gamma \vdash_{\mathrm{cop}} q.\mathrm{out} : C \Rightarrow A[\nu X.\, A/X]} \qquad \frac{\Gamma \vdash_{\mathrm{cop}} q : C \Rightarrow (A \to B) \quad x : D \vdash_{\mathrm{pat}} p : A \quad x \notin \Gamma}{\Gamma, x : D \vdash_{\mathrm{cop}} q\, p : C \Rightarrow B}$$

Figure 3.7.: Rules for forming terms, declaration bodies and blocks, and (co)patterns

Let us explain the typing rules in Definition 3.2.3. The main difference with the calculus $\lambda\mu\nu$ is that we omit the iteration and coiteration schemes in favour of **rlet**-blocks. Such blocks allow us to construct terms with recursive occurrences of symbols, that is, in a term **rlet** Σ_2 **in** t, the symbols defined in the declaration block Σ_2 can be (directly or indirectly) self-referential. This can be seen from the single rule for $\Sigma_1 \vdash_{\mathrm{dec}} \Sigma_2$ in Figure 3.7, where every declaration body in Σ_2 is checked with all declarations of Σ_2 in scope. Note that there is no restriction on the recursive occurrence of symbols, which means that it is possible to write non-terminating programs in $\lambda\mu\nu{=}$, see Example 3.2.21. On the other hand, this also allows us to write programs that are well-defined but not obeying any syntactic restriction like guardedness [Gim95], see Example 4.1.6.

The declaration bodies form the heart of the calculus, as they allow us to define objects of coinductive type by means of copatterns and functions out of inductive types by means of patterns. For instance, to define a function of type $A + B \to C$, we may use pattern matching to distinguish whether we get an element from A or B as input. Given terms s and t of type C, we can form the declaration body D with $D = \{\cdot\ (\kappa_1\ x) \mapsto s\ ;\ \cdot\ (\kappa_2\ y) \mapsto t\}$, which covers the two cases of the sum type. We may then use the abstraction rule to form the term λD of type $A + B \to C$ that allows us to carry out the case distinction, see Example 3.2.4. Concerning coinductive types, we instead specify the outcome of observations that can be made on a term through the use of destructors. For example, given terms s and t respectively of type A and B, we can obtain the pair of these terms by abstraction of the declaration body D defined by $D = \{\cdot\ .\mathrm{pr}_1 \mapsto s\ ;\ \cdot\ .\mathrm{pr}_2 \mapsto t\}$, which gives us $\lambda D : A \times B$. Pairing for products is further explained in Example 3.2.5. The last example that we discuss at this point are streams. To increase readability, we introduce some short-hand notations for *head* and *tail* projections on streams:

$$.\mathrm{hd} \coloneqq .\mathrm{out}.\mathrm{pr}_1 \qquad \text{and} \qquad .\mathrm{tl} \coloneqq .\mathrm{out}.\mathrm{pr}_2. \tag{3.3}$$

As one would expect, a stream is given by providing the terms for the head and tail, say $s : A$ and $t : A^\omega$, and bundling them in the body

$$\{\cdot\ .\mathrm{hd} \mapsto s\ ;\ \cdot\ .\mathrm{tl} \mapsto t\} : A \times A^\omega.$$

We will see in Example 3.2.9, Example 3.2.10 and Example 3.2.11 further specifications of streams.

Let us now explain patterns and copatterns in more broad terms. As we said above, copatterns allow us to specify the outcome of observations on a term, while patterns allow us to analyse arguments of inductive type. This idea is reflected in the interplay of the typing rules for declaration bodies, patterns and copatterns. Generally, the judgement $\Gamma \vdash_{\mathrm{cop}} q : C \Rightarrow B$ says that if we want to form a declaration body of the form $\{q \mapsto t\}$ of type C, then t must be of type B and the abstraction $\lambda\{q \mapsto t\}$ binds all the variables in Γ. Consider, for instance, the declaration body $D = \{\cdot\ (\alpha\ x) \mapsto t\}$. First of all, we expect that λD is a function that takes an argument of type $\mu X.\, A$ for some type A and returns something of type B. Second, λD should bind the variable x in t. Both requirements are included in the typing of the copattern $\cdot\ (\alpha x)$, which in turn is obtained by the following derivation.

$$\dfrac{\emptyset \vdash_{\mathrm{cop}} \cdot : \big((\mu X.\, A) \to B\big) \Rightarrow \big((\mu X.\, A) \to B\big) \qquad \dfrac{x : A[\mu X.\, A/X] \vdash_{\mathrm{pat}} x : A[\mu X.\, A/X]}{x : A[\mu X.\, A/X] \vdash_{\mathrm{pat}} \alpha\ x : \mu X.\, A}}{x : A[\mu X.\, A/X] \vdash_{\mathrm{cop}} \cdot\ (\alpha\ x) : \big((\mu X.\, A) \to B\big) \Rightarrow B}$$

We can now read off from this that t can have a free variable of type $A[\mu X.\, A/X]$, which is bound in λD, that t must be of type B, and that the type of λD is then $(\mu X.\, A) \to B$. In the course of

the examples below, we will familiarise ourselves more with this interplay between the rules for copatterns and declaration bodies.

There are two aspects of declarations that are worthwhile to discuss. First, we mentioned already that the variables contained in copatterns are bound through declaration bodies. To avoid dependencies between patterns in a body and the problems that spring from such dependencies, one usually requires patterns of a declaration to be *linear*, that is, variables may occur at most once in them. We ensure this here in the rule for copatterns of function application by checking, upon forming the copattern $q\,p$, that the variable that appears in the argument pattern p does not occur already in the copattern q. This guarantees that any variable occurs at most once in a copattern, which rules out copatterns like "$\cdot\,(\kappa_1\,x)\,x$". Second, the reader might have noticed that neither the definition of raw terms nor that of well-typed terms ensures that a declaration body has to cover all cases. Consider for instance the term f given by $f = \lambda\{\cdot\,(\kappa_1\,x) \mapsto t\}$, which does not cover the case that an argument may be of the form $\kappa_2\,s$. Thus, if we tried to evaluate $f\,(\kappa_2\,s)$, then the computation would get stuck. So far, there is nothing in the calculus that ensures that a declaration body covers all possible cases for (co)patterns. It even might happen that there are several cases for the same copattern, which would make computations non-deterministic. We will discuss this further in Example 3.2.23 and then address these issues when proving confluence in Proposition 3.2.32.

Let us now go through the remaining rules in Figure 3.7. We have already seen that **(Rlet)** allows us to construct terms with recursively defined symbols. The rule **(ProjSig)** complements this, in that it allows us to use any declared symbol as term. Sometimes, we would like to give a term in the form of a declaration body without having to introduce a new symbol, see Example 3.2.5 below. This can be done by using the rule **(Abs)**. The other rules to construct terms correspond exactly to those in $\lambda\mu\nu$, except that the destructors appear in post-fix position.

We illustrate $\lambda\mu\nu{=}$ on some non-recursive examples, in which we recover notations from $\lambda\mu\nu$. In Section 3.3, we will properly study the relation between these two calculi.

Example 3.2.4. We have almost the same notation for case distinction on sums:

$$\{\kappa_1\,x \mapsto s\;;\;\kappa_2\,y \mapsto t\}\,u \coloneqq \lambda\{\cdot\,(\kappa_1\,x) \mapsto s\;;\;\cdot\,(\kappa_2\,y) \mapsto t\}\,u$$

Let us check that this definition is well-typed. The first step is to assign types to the two copatterns in the term on the right. This is done by the following two derivations.

$$\dfrac{\dfrac{}{\vdash_{\mathrm{cop}} \cdot : (A+B \to C) \Rightarrow (A+B \to C)} \qquad \dfrac{\dfrac{}{x : A \vdash_{\mathrm{pat}} x : A}}{x : A \vdash_{\mathrm{pat}} \kappa_1\,x : A+B}}{x : A \vdash_{\mathrm{cop}} \cdot\,(\kappa_1\,x) : (A+B \to C) \Rightarrow C}$$

$$\dfrac{\dfrac{}{\vdash_{\mathrm{cop}} \cdot : (A+B \to C) \Rightarrow (A+B \to C)} \qquad \dfrac{\dfrac{}{y : B \vdash_{\mathrm{pat}} y : B}}{y : B \vdash_{\mathrm{pat}} \kappa_2\,y : A+B}}{y : B \vdash_{\mathrm{cop}} \cdot\,(\kappa_2\,y) : (A+B \to C) \Rightarrow C}$$

Next, we check the type of the abstraction. For this, suppose that we have $\Gamma, x : A \vdash s : C$ and $\Gamma, y : B \vdash t : C$ and $\Gamma; \Sigma \vdash u : A+B$. Using the short-hand notation $U = A+B \to C$, we can make

the following derivation.

$$\dfrac{\dfrac{\begin{array}{l} x : A \vdash_{\mathrm{cop}} \cdot\,(\kappa_1\, x) : U \Rightarrow C \\ y : B \vdash_{\mathrm{cop}} \cdot\,(\kappa_2\, y) : U \Rightarrow C \end{array} \qquad \Gamma, x : A \vdash s : C \qquad \Gamma, y : B \vdash t : C}{\Gamma; \Sigma \vdash_{\mathrm{bdy}} \{\cdot\,(\kappa_1\, x) \mapsto s\ ;\ \cdot\,(\kappa_2\, y) \mapsto t\} : A + B \to C}}{\Gamma; \Sigma \vdash \lambda\{\cdot\,(\kappa_1\, x) \mapsto s\ ;\ \cdot\,(\kappa_2\, y) \mapsto t\} : A + B \to C}$$

Finally, we check the application to u:

$$\dfrac{\Gamma; \Sigma \vdash \lambda\{\cdot\,(\kappa_1\, x) \mapsto s\ ;\ \cdot\,(\kappa_2\, y) \mapsto t\} : A + B \to C \qquad \Gamma; \Sigma \vdash u : A + B}{\Gamma; \Sigma \vdash \lambda\{\cdot\,(\kappa_1\, x) \mapsto s\ ;\ \cdot\,(\kappa_2\, y) \mapsto t\}\, u : C}$$

Thus the definition of case distinction we gave above is well-typed. ◂

Next, we can also recover the notation for product types that we used in $\lambda\mu\nu$.

Example 3.2.5. The projections on product types can be defined for $u : A \times B$ by

$$\pi_1\, u \coloneqq u.\mathrm{pr}_1 \qquad \text{and} \qquad \pi_2\, u \coloneqq u.\mathrm{pr}_2,$$

so that the pairing constructor is given by

$$\langle s, t\rangle \coloneqq \lambda\{\cdot\,.\mathrm{pr}_1 \mapsto s\ ;\ \cdot\,.\mathrm{pr}_2 \mapsto t\}.$$

Let us again check that these definitions are well-typed. This is clear for the projections, so we check just the pairing. Assume that $\Gamma; \Sigma \vdash s : A$ and $\Gamma; \Sigma \vdash t : B$. Then we have

$$\dfrac{\dfrac{\dfrac{\vdash_{\mathrm{cop}} \cdot : A \times B \Rightarrow A \times B}{\vdash_{\mathrm{cop}} \cdot\,.\mathrm{pr}_1 : A \times B \Rightarrow A} \qquad \dfrac{\vdash_{\mathrm{cop}} \cdot : A \times B \Rightarrow A \times B}{\vdash_{\mathrm{cop}} \cdot\,.\mathrm{pr}_2 : A \times B \Rightarrow B}}{\Gamma; \Sigma \vdash_{\mathrm{bdy}} \{\cdot\,.\mathrm{pr}_1 \mapsto s\ ;\ \cdot\,.\mathrm{pr}_2 \mapsto t\} : A \times B}}{\Gamma; \Sigma \vdash \lambda\{\cdot\,.\mathrm{pr}_1 \mapsto s\ ;\ \cdot\,.\mathrm{pr}_2 \mapsto t\} : A \times B}$$

as required. ◂

Finally, we can also recover the usual notation for function spaces.

Example 3.2.6. Note that application of terms of function type is written in $\lambda\mu\nu{=}$ just as in $\lambda\mu\nu$. However, λ-abstraction is slightly more complicated:

$$\lambda x.\, t \coloneqq \lambda\{\cdot\, x \mapsto t\}.$$

Assuming that $\Gamma, x : A; \Sigma \vdash t : B$, we can indeed derive the following.

$$\dfrac{\dfrac{\dfrac{\vdash_{\mathrm{cop}} \cdot : (A \to B) \Rightarrow (A \to B)}{\vdash_{\mathrm{cop}} \cdot\, x : (A \to B) \Rightarrow B} \qquad \Gamma, x : A; \Sigma \vdash t : B}{\Gamma; \Sigma \vdash_{\mathrm{bdy}} \{\cdot\, x \mapsto t\} : A \to B}}{\Gamma; \Sigma \vdash \lambda\{\cdot\, x \mapsto t\} : A \to B}$$

So the λ-abstraction is well-typed with the expected type. ◂

Recall from Example 3.1.6 that we could represent in $\lambda\mu\nu$ a one-element set by the fixed point type $\nu X.\, X$. This is again true in $\lambda\mu\nu{=}$, except that the inhabitant $\langle\rangle'$ is easier to establish.

Example 3.2.7. Recall from Example 3.1.6 that we used the coiteration scheme of $\lambda\mu\nu$ on a function type to get an inhabitant of the type $\mathbf{1}'$. In $\lambda\mu\nu{=}$ this inhabitant can be given much easier by

$$\langle\rangle' \coloneqq \mathbf{rlet}\ f : \mathbf{1}' = \{\cdot\,.\mathrm{out} \mapsto f\}\ \mathbf{in}\ f.$$

Let us use the typing rules to show that $\langle\rangle' : \mathbf{1}'$ indeed holds.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\mathrm{cop}} \cdot : \mathbf{1}' \Rightarrow \mathbf{1}'}{\vdash_{\mathrm{cop}} \cdot\,.\mathrm{out} : \mathbf{1}' \Rightarrow X[\mathbf{1}'/X]} \qquad f : \mathbf{1}' \vdash f : \mathbf{1}'}{f : \mathbf{1}' \vdash_{\mathrm{bdy}} \{\cdot\,.\mathrm{out} \mapsto f\} : \mathbf{1}'}}{\vdash_{\mathrm{dec}} f : \mathbf{1}' = \{\cdot\,.\mathrm{out} \mapsto f\}} \qquad f : \mathbf{1}' \vdash f : \mathbf{1}'}{\vdash \langle\rangle' : \mathbf{1}'}$$

In Section 4.1, it will turn out that also this inhabitant of $\mathbf{1}'$ is canonical. ◀

We have advertised the present calculus as being easier to program with. The next example demonstrates this on the projection out of the left-fair streams. Since these streams are given by a mixed inductive-coinductive type, the example combines patterns and copatterns.

Example 3.2.8. Recall the type $\mathrm{LFair}\, A\, B = \nu X.\, \mu Y.\, (A \times X + B \times Y)$, consisting of streams over A and B, such that elements of A occur infinitely often. We define a map that projects a left-fair stream onto a stream over A as in Example 3.2.1. This can be done by defining maps p_A and erase_B by mutual recursion as follows.

$$\begin{aligned}
&\mathrm{proj}_A \coloneqq \mathbf{rlet}\\
&\quad p_A : \mathrm{LFair}\, A\, B \to A^\omega = \big\{(\cdot\, x).\mathrm{out} \mapsto \mathrm{erase}_B\ (x.\mathrm{out})\big\}\\
&\quad \mathrm{erase}_B : \mu Y.\, (A \times \mathrm{LFair}\, A\, B) + B \times Y) \to A \times A^\omega = \{\\
&\qquad \cdot\ (\alpha\ (\kappa_1\ u)) \mapsto \langle u.\mathrm{pr}_1,\ p_A\ (u.\mathrm{pr}_2)\rangle\ ;\\
&\qquad \cdot\ (\alpha\ (\kappa_2\ u)) \mapsto \mathrm{erase}_B\ (u.\mathrm{pr}_2)\\
&\quad \}\\
&\mathbf{in}\ p_A
\end{aligned}$$

Note that p_A is defined coinductively, whereas erase_B is defined by induction.

We now derive the type of proj_A. For brevity, let us agree on the following short-hand notation.

$$\begin{aligned}
F &= \mathrm{LFair}\, A\, B & L &= \mu Y.\, (A \times \mathrm{LFair} A\, B) + B \times Y)\\
V &= F \to A^\omega & W &= L \to A \times A^\omega\\
\Sigma &= \{p_A : V, \mathrm{erase}_B : W\}
\end{aligned}$$

First, we find that the type of the copattern used in p_A is $V \Rightarrow A \times A^\omega$:

$$\dfrac{\dfrac{\vdash_{\mathrm{cop}} \cdot : V \Rightarrow V \qquad u : F \vdash_{\mathrm{pat}} u : F}{u : F \vdash_{\mathrm{cop}} \cdot\, u : V \Rightarrow A^\omega}}{u : F \vdash_{\mathrm{cop}} (\cdot\, u).\mathrm{out} : V \Rightarrow A \times A^\omega} \tag{3.4}$$

We can use this information then to check that the body of p_A is type-correct:

$$\dfrac{\dfrac{\dfrac{}{u : F; \Sigma \vdash u : F}}{u : F; \Sigma \vdash \mathrm{erase}_B : W \qquad u : F; \Sigma \vdash u.\mathrm{out} : L}}{\dfrac{(3.4) \qquad u : F; \Sigma \vdash \mathrm{erase}_B(u.\mathrm{out}) : A \times A^\omega}{\Sigma \vdash_{\mathrm{bdy}} \left\{(\cdot\, u).\mathrm{out} \mapsto \mathrm{erase}_B\ (u.\mathrm{out})\right\} : V}} \tag{3.4}$$

Similarly, we can type the body of erase_B. Putting these two derivations together, we can check that Σ is well-formed.

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Sigma \vdash_{\mathrm{bdy}} \left\{(\cdot\, u).\mathrm{out} \mapsto \ldots\right\} : V & \Sigma \vdash_{\mathrm{bdy}} \left\{\cdot\ (\alpha(\kappa_1\ v)) \mapsto \ldots\right\} : W\end{array}}{\vdash_{\mathrm{dec}} \Sigma} \tag{3.5}$$

Finally, we check the **rlet**-declaration that defines proj_A:

$$\dfrac{(3.5) \qquad \Sigma \vdash p_A : V}{\vdash \mathrm{proj}_A : \mathrm{LFair}\ A\ B \to A^\omega}$$

Thus proj_A is a well-formed term.

Note that proj_A is defined by a single **rlet**-block, which declares a fresh symbol p_A that defines proj_A. Since this situation occurs quite frequently, we will usually leave out this **rlet** and present its content as a set of equations, just like one would write programs in Haskell or how behavioural differential equations are given. The above program is then given as follows.

$$\begin{aligned}
&\mathrm{proj}_A : \mathrm{LFair}\ A\ B \to A^\omega \\
&(\mathrm{proj}_A\ x).\mathrm{out} = \mathrm{erase}_B\ (x.\mathrm{out}) \\
&\mathrm{erase}_B : \mu Y.\ (A \times \mathrm{LFair}\ A\ B) + B \times Y) \to A \times A^\omega \\
&\mathrm{erase}_B\ (\alpha\ (\kappa_1\ y)) = \langle y.\mathrm{pr}_1,\ \mathrm{proj}_A\ (y.\mathrm{pr}_2)\rangle \\
&\mathrm{erase}_B\ (\alpha\ (\kappa_2\ y)) = \mathrm{erase}_B\ (y.\mathrm{pr}_2)
\end{aligned}$$

This lifts some notational burden and makes programs easier to read. ◀

The next example demonstrates the use of mixed induction-coinduction on streams over natural numbers. Note that, in contrast to left-fair streams, this is a type in which least and greatest fixed point types are separable but the function still mixes induction and coinduction.

Example 3.2.9. We define a function H of type $\mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega$ that maps streams s and t to a stream r with $r(n) = t(\sum_{i=0}^{n} s(i))$ by mixing patterns and copatterns as in the following program. We display the program in two forms: on the left in the formally correct notation of $\lambda\mu\nu=$ and on the right in the more readable, Haskell-like notation that we introduced in Example 3.2.8.

Formal notation ($\lambda\mu\nu{=}$ term):

$$
\begin{aligned}
&H \coloneqq \textbf{rlet}\\
&\quad h : \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega = \{ \cdot\, s\, t \mapsto f\ (s.\mathrm{hd})\ s\ t \,\}\\
&\quad f : \mathrm{Nat} \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega = \{\\
&\qquad (\cdot\ 0 \quad\ s\ t).\mathrm{hd} \quad \mapsto t.\mathrm{hd}\\
&\qquad (\cdot\ 0 \quad\ s\ t).\mathrm{tl} \quad \mapsto h\ (s.\mathrm{tl})\ t\\
&\qquad \cdot\ (n+1)\ s\ t \quad\ \mapsto f\ n\ s\ (t.\mathrm{tl}) \quad \}\\
&\textbf{in}\ h
\end{aligned}
$$

Haskell-like notation:

$$
\begin{aligned}
&H : \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega\\
&H\ s\ t = f\ (s.\mathrm{hd})\ s\ t\\
&f : \mathrm{Nat} \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega\\
&(f\ 0 \quad\ s\ t).\mathrm{hd} \quad = t.\mathrm{hd}\\
&(f\ 0 \quad\ s\ t).\mathrm{tl} \quad = H\ (s.\mathrm{tl})\ t\\
&f\ (n+1)\ s\ t \quad\ = f\ n\ s\ (t.\mathrm{tl})
\end{aligned}
$$

The combination of patterns and copatterns is demonstrated in the declaration of f, which uses pattern matching on the first argument, followed by specifying head and tail of $f\,0\,s\,t$. ◂

The following example defines a simple stream, which we will use later again.

Example 3.2.10. The stream of alternating bits can be given as follows, where we present again on the left the formal term of $\lambda\mu\nu{=}$ and the more readable version on the right.

Formal notation ($\lambda\mu\nu{=}$ term):

$$
\begin{aligned}
&\mathrm{alt} \coloneqq \textbf{rlet}\\
&\quad s_{\mathrm{alt}} : \mathrm{Nat}^\omega = \{\\
&\qquad s_{\mathrm{alt}}.\mathrm{hd} \quad\ \mapsto \underline{0}\\
&\qquad s_{\mathrm{alt}}.\mathrm{tl}.\mathrm{hd} \quad \mapsto \underline{1}\\
&\qquad s_{\mathrm{alt}}.\mathrm{tl}.\mathrm{tl} \quad \mapsto s_{\mathrm{alt}} \quad \}\\
&\textbf{in}\ s_{\mathrm{alt}}
\end{aligned}
$$

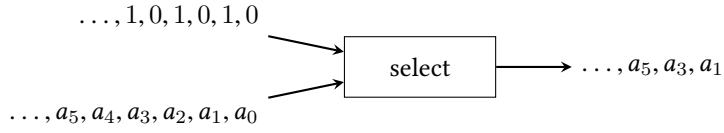
Haskell-like notation:

$$
\begin{aligned}
&\mathrm{alt} : \mathrm{Nat}^\omega\\
&\mathrm{alt}.\mathrm{hd} \quad\ = \underline{0}\\
&\mathrm{alt}.\mathrm{tl}.\mathrm{hd} \quad = \underline{1}\\
&\mathrm{alt}.\mathrm{tl}.\mathrm{tl} \quad = \mathrm{alt}
\end{aligned}
$$

The reader will notice that we specified the first two entries of the stream alt directly. This is not directly possible in other specification formats for streams, like stream differential equations (SDE) [HKR17], because it cannot always be guaranteed that there exists a solution to SDE with such deeper specifications, cf. ibid. We discuss this issue in Section 4.1. ◂

In the final example of this section we select entries from streams, which crucially uses a mixed inductive-coinductive type. The definitions in the example are also a rich source of further examples later on.

Example 3.2.11. The goal of this example is to define a function select that selects entries from a stream. Its intended behaviour is visualised in the following diagram.



In this diagram, we see that select has two inputs: a stream of 0s and 1, and the stream from which we select entries. A 1 in the first stream marks thereby a position in the second stream that

should be kept, whereas 0s mark positions that shall be dropped. However, there is a problem if the first stream consists only of 0s, as we would have to drop all entries of the second input and thus would not be able to compute an output stream.

This problem is solved, if we only accept streams that contain infinitely many 1s or, equivalently, in which all consecutive sequences of 0s are finite. We can guarantee this property by using *stream selectors* instead of 0-1-streams, which are elements of the following type F.

$$\begin{aligned} F &\coloneqq \nu X.\, \mu Y.\, X + Y \\ F_\mu &\coloneqq \mu Y.\, F + Y \end{aligned}$$

Let us give some names to the constructors of F_μ to improve readability. For patterns $\Gamma_1 \vdash_{\text{pat}} p_1 : F$ and $\Gamma_2 \vdash_{\text{pat}} p_2 : F_\mu$, we define the following patterns for the type F_μ.

$$\begin{aligned} \text{pres}\, p_1 &\coloneqq \alpha\, (\kappa_1\, p_1) \\ \text{drop}\, p_2 &\coloneqq \alpha\, (\kappa_2\, p_2) \end{aligned}$$

The first pattern signifies that an entry should be preserved, which corresponds to a 1 in the intuitive explanation of selectors above. On the other hand, we mark positions that shall be dropped by the second pattern, the 0 in the bit sequence view. Using these notations, we can implement the select function, mutually with an auxiliary function on F_μ, as follows.

$$\begin{aligned} &\text{select} \colon F \to A^\omega \to A^\omega \\ &\text{select}\, x = \text{select}_\mu\, (x.\text{out}) \\ &\text{select}_\mu \colon F_\mu \to A^\omega \to A^\omega \\ &(\text{select}_\mu\, (\text{pres}\, x)\, s).\text{hd} \quad &&= s.\text{hd} \\ &(\text{select}_\mu\, (\text{pres}\, x)\, s).\text{tl} \quad &&= \text{select}\, x\, (s.\text{tl}) \\ &\text{select}_\mu\, (\text{drop}\, u)\, s \quad &&= \text{select}_\mu\, u\, (s.\text{tl}) \end{aligned}$$

The functions select and select_μ are typical examples of a mixed inductive-coinductive definition: We define select_μ by induction on the first argument, and by coinduction in the base case $\text{pres}\, x$ of this induction. The role of select is to unfold the selector one step, so that we can calculate the next element of the output stream.

Let us now also use the same notation for constructing elements of F_μ as we did for the patterns. That is, given an element x of type F, we write $\text{pres}\, x \coloneqq \alpha\, (\kappa_1\, x) : F_\mu$ and similar for drop. This allows us to concisely give a selector for the odd positions of a stream as follows.

$$\begin{aligned} &\text{oddF} : F \\ &\text{oddF.out} = \text{drop}\, (\text{pres oddF}) \end{aligned}$$

Intuitively, this selector corresponds to the stream $(0, 1, 0, 1, 0, 1 \dots)$ in the original picture. It allows us to define the following function that drops all the even positions of streams over A.

$$\begin{aligned} &\text{odd} \colon A^\omega \to A^\omega \\ &\text{odd} = \text{select oddF} \end{aligned}$$

We will see in Example 3.2.19 the computational behaviour of select. In Chapter 5 we will be able to show that select indeed produces an output for any selector, and in the same chapter we prove some further properties of select. Finally, in Chapter 7 we apply selectors and the select function in the proof of a proposition. ◂

Let us discuss some interesting aspects of Example 3.2.11. First, Bertot [Ber05] uses selection from streams by means of predicates to implement the sieve of Eratosthenes in Coq. Suppose that A is a type, P a predicate[14] on A and $s : A^\omega$ a stream on A. Bertot then aims to define a stream select $P\, s$ that arises from s by keeping exactly those entries that fulfil the predicate P. However, it is clear that such a stream does not exist for all combinations of predicates and streams, but only if the predicate P holds on infinitely many entries of the stream s. If we say in this situation that P is *fair on* s, then we can alternatively describe this fairness property as inductive-coinductive property: P is fair on s if P *eventually* holds on some element in s and P is fair on the stream after that position. We will make this definition precise in Section 7.5.3. Let us just say that this fairness of P on s corresponds to the formula $\Box\Diamond P$ (read: always, eventually P) in linear temporal logic. In contrast to Bertot's development, we have selected entries from streams in Example 3.2.11 by specifying the *positions* that should be kept/dropped from a stream, so that a selector is independent of the stream from which we select. However, these two approaches of selecting can be seen to be equivalent.

The second interesting aspect of entry selection is its relation to (uniform) continuity. Let us write $s \approx_n t$ for $n \in \mathbb{N}$ and stream terms $s, t : A^\omega$, if s and t agree on the first n entries. More precisely,

$$s \approx_n t \coloneqq \forall k < n.\, (s.\mathrm{tl}^k.\mathrm{hd} \equiv t.\mathrm{tl}^k.\mathrm{hd}),$$

where $.\mathrm{tl}^k$ is the k-fold application of .tl. This gives us a metric on stream terms [Smy92, Chap. 6]:

$$d(s,t) = \inf\{2^{-n} \mid n \in \mathbb{N}, s \approx_n t\}.$$

A term $f : A^\omega \to A^\omega$ is *uniformly continuous* with *modulus of continuity* $M : \mathbb{N} \to \mathbb{N}$, if for all $n \in \mathbb{N}$

$$\forall s, t : A^\omega.\, d(s,t) \leq 2^{-M(n)} \implies d(f\, s, f\, t) \leq 2^{-n}.$$

Note now that for a stream s the n-th element of odd s is the $(2n+1)$-th element of the input stream s. Thus, we claim that odd is uniformly continuous with modulus $M(n) = 2n + 1$. Indeed, we can define a map $\mathrm{mod} : F \to \mathrm{Nat} \to \mathrm{Nat}$ that computes for $x : F$ the modulus of continuity of select x:

$$\begin{aligned}
&\mathrm{mod}\, x = \mathrm{mod}_\mu\, (x.\mathrm{out}) \\
&\mathrm{mod}_\mu : F_\mu \to \mathrm{Nat} \to \mathrm{Nat} \\
&\mathrm{mod}_\mu\, (\mathrm{pres}\, x)\, 0 &&= \underline{0} \\
&\mathrm{mod}_\mu\, (\mathrm{pres}\, x)\, (\mathrm{suc}\, n) &&= \mathrm{suc}\, (\mathrm{mod}\, x\, n) \\
&\mathrm{mod}_\mu\, (\mathrm{drop}\, x)\, n &&= \mathrm{suc}\, (\mathrm{mod}_\mu\, x\, n)
\end{aligned}$$

In Example 3.2.20, we show that mod oddF $\underline{n}$ computes $\underline{2n+1}$. This will allow us to show that the *look-ahead* of odd, that is, the number of entries that are read from an input to produce an output, is bounded by mod oddF. In other words, odd is uniformly continuous with modulus mod oddF.[15]

3.2.2. Computations in $\lambda\mu\nu{=}$

We now give a reduction relation on terms in $\lambda\mu\nu{=}$. This reduction relation has at its heart again a contraction relation, which essentially applies the defining equations as rewrite rules. For instance, we will have

$$\begin{aligned}\langle\rangle'.\mathrm{out} &= (\mathbf{rlet}\ f : \mathbf{1}' = \{\cdot\ .\mathrm{out} \mapsto f\}\ \mathbf{in}\ f).\mathrm{out}\\ &\longrightarrow \mathbf{rlet}\ f : \mathbf{1}' = \{\cdot\ .\mathrm{out} \mapsto f\}\ \mathbf{in}\ (f.\mathrm{out})\\ &\longrightarrow \mathbf{rlet}\ f : \mathbf{1}' = \{\cdot\ .\mathrm{out} \mapsto f\}\ \mathbf{in}\ f\\ &= \langle\rangle'.\end{aligned}$$

Turning copattern equations into rewrite rules also ensures that the definitions in Example 3.2.4, 3.2.5 and 3.2.6 yield terms whose reductions coincide with their counterparts in $\lambda\mu\nu$.

Definition 3.2.12. A *substitution* is a function $\sigma\colon \mathrm{TeVar} \to \mathrm{Terms}_{\lambda\mu\nu=}$ that maps variables to terms. We denote by $\mathrm{dom}(\sigma)$ the *domain* of σ, which is given by $\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$. Given a term $t \in \mathrm{Terms}_{\lambda\mu\nu=}$, the application $t[\sigma]$ of σ to t is the term that results from replacing all variables $x \in \mathrm{dom}(\sigma)$ that are unbound in t by $\sigma(x)$ while renaming bound variables in t (α-renaming) to avoid binding of variables in $\sigma(x)$. For a context Γ, we denote by $\mathrm{dom}(\Gamma)$ the set of variables that occur in Γ. We then say that a substitution σ is a Γ-*substitution*, if $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\sigma)$. ◀

Definition 3.2.13. Given a copattern q with $\Gamma \vdash_{\mathrm{cop}} q : A \Rightarrow B$ and a Γ-substitution σ, we call $e = q[\sigma]$ an *evaluation context on type A with result in B* and we say that q *matches* e. For terms t, we denote by $e[t/\cdot]$ the term obtained by replacing the hole $\cdot$ in $q[\sigma]$ by t. ◀

Note that in Definition 3.2.13, if there is a context Γ', such that $\Gamma' \vdash t : A$ and $\Gamma' \vdash \sigma(x) : \Gamma(x)$ for all $x \in \mathrm{dom}(\Gamma)$, where $\Gamma(x)$ is the type of the variable x in Γ, we have $\Gamma' \vdash e[t/\cdot] : B$.

We use matching to define contraction and reductions of terms.

Definition 3.2.14. Given a term $t : A$ and an evaluation context e, we say that the term $e[t/\cdot]$ *contracts to t' using declarations in* Σ, if $e[t/\cdot] \succ_\Sigma t'$ can be derived using the following rules:

$$\frac{q_i \mapsto t_i \in D}{q_i[\sigma][\lambda D/\cdot] \succ_\Sigma t_i[\sigma]} \qquad \frac{e[\lambda D/\cdot] \succ_\Sigma t' \qquad f : A = D \in \Sigma}{e[f/\cdot] \succ_\Sigma t'}$$

For each declaration block Σ, we define the *reduction relation of $\lambda\mu\nu{=}$* $\longrightarrow_\Sigma$ on terms as the compatible closure of contraction and the following rule, which makes declarations available in reductions.

$$\frac{t \longrightarrow_{\Sigma_1,\Sigma_2} t'}{\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t \longrightarrow_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t'}$$

Finally, we need that **rlet**-bindings interact well with the rest of the calculus. Thus, we have two more rules for each evaluation context e and constructor $c \in \{\kappa_1, \kappa_2, \alpha\}$.

$$\overline{e[\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t/\cdot] \longrightarrow_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ e[t/\cdot]} \qquad \overline{\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (c\ t) \longrightarrow_{\Sigma_1} c\ (\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t)}$$

We denote the relation $\longrightarrow_\emptyset$ by $\longrightarrow$. As before, we write $\twoheadrightarrow$ for the reflexive, transitive closure of $\longrightarrow$ and $\equiv$ for the equivalence closure of $\longrightarrow$ (*convertibility*). ◀

Let us now show that we can recover the reductions of $\lambda\mu\nu$ on non-recursive types.

Example 3.2.15. Recall that we introduced in Example 3.2.4, 3.2.5 and 3.2.6 notation that resembled the structure of non-recursive types of $\lambda\mu\nu$ in $\lambda\mu\nu{=}$. We now show how in each of the three cases Definition 3.2.14 applies and that the result of a reduction is the same as in $\lambda\mu\nu$. Let $\Gamma, x : A \vdash s : C$, $\Gamma, y : B \vdash t : C$ and $\Gamma; \Sigma \vdash a : A$, so that we can form

$$\{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\ (\kappa_1\, a) = \lambda\big\{\cdot\ (\kappa_1\, x) \mapsto s\ ;\ \cdot\ (\kappa_2\, y) \mapsto t\big\}\ (\kappa_1\, a)$$

in the notation of Example 3.2.4. If we now put $q_1 = \cdot\ (\kappa_1\, x)$ and $\sigma(x) = a$, then

$$\{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\ (\kappa_1\, a) = q_1[\sigma][\lambda\big\{\cdot\ (\kappa_1\, x) \mapsto s\ ;\ \cdot\ (\kappa_2\, y) \mapsto t\big\}/\cdot].$$

Thus, by the first rule in Definition 3.2.14 we have

$$\{\kappa_1\, x \mapsto s\ ;\ \kappa_2\, y \mapsto t\}\ (\kappa_1\, a) \succ_\Sigma s[\sigma] = s[a/x],$$

which is exactly how we defined contraction for $\lambda\mu\nu$ on terms of sum type in Definition 3.1.11.

Similarly, we have for $q_1 = \cdot\ .\mathrm{pr}_1$ and the empty substitution (everywhere undefined function) σ

$$\begin{aligned}\pi_1\ \langle s, t\rangle &= \lambda\big\{\cdot\ .\mathrm{pr}_1 \mapsto s\ ;\ \cdot\ .\mathrm{pr}_2 \mapsto t\big\}.\mathrm{pr}_1\\ &= q_1[\sigma][\lambda\big\{\cdot\ .\mathrm{pr}_1 \mapsto s\ ;\ \cdot\ .\mathrm{pr}_2 \mapsto t\big\}/\cdot]\\ &\succ_\Sigma s[\sigma]\\ &= s,\end{aligned}$$

so that the projections on product types also work as expected.

Finally, also β-reduction on functions is recovered in $\lambda\mu\nu{=}$:

$$(\lambda x.\, t)\ s = (\cdot\ x)[s/x][\lambda\{\cdot\ x \mapsto t\}/\cdot] \succ_\Sigma t[s/x].$$ ◀

Let us now give an example of computations on recursive type.

Example 3.2.16. Recall that we have defined in Example 3.1.8 constant streams by using coiter in $\lambda\mu\nu$. The definition of these constant streams for $a : A$ in $\lambda\mu\nu{=}$ is even easier:

$$a^\omega \coloneqq \mathbf{rlet}\ c_a : A^\omega = \{\cdot.\mathrm{hd} \mapsto a\ ;\ \cdot.\mathrm{tl} \mapsto c_a\}\ \mathbf{in}\ c_a$$

If we put $\Sigma = \{c_a : A^\omega = \{\cdot.\mathrm{hd} \mapsto a\ ;\ \cdot.\mathrm{tl} \mapsto c_a\}\}$, then we have

$$a^\omega.\mathrm{hd} = (\mathbf{rlet}\ \Sigma\ \mathbf{in}\ c_a).\mathrm{hd} = (\cdot\ .\mathrm{hd})[\mathbf{rlet}\ \Sigma\ \mathbf{in}\ c_a/\cdot] \longrightarrow \mathbf{rlet}\ \Sigma\ \mathbf{in}\ ((\cdot\ .\mathrm{hd})[c_a/\cdot]).$$

To proceed further, we use the first contract rule to obtain

$$((\cdot\ .\mathrm{hd})[\lambda\{\cdot.\mathrm{hd} \mapsto a\ ;\ \cdot.\mathrm{tl} \mapsto c_a\}/\cdot]) \succ_\Sigma a,$$

from which we get by the second contraction rule that

$$((\cdot\ .\mathrm{hd})[c_a/\cdot]) \succ_\Sigma a.$$

Thus, by the reduction rule for **rlet**-blocks, we can continue the reduction by

$$\mathbf{rlet}\ \Sigma\ \mathbf{in}\ ((\cdot\ .\mathrm{hd})[c_a/\cdot]) \longrightarrow \mathbf{rlet}\ \Sigma\ \mathbf{in}\ a.$$

It is now crucial to note that the last two reduction rules in Definition 3.2.14 give us that this resulting term behaves exactly like a itself despite it sitting under a **rlet**-binding. For instance, we can reduce the term further if $a = \underline{0}$:

$$\mathbf{rlet}\ \Sigma\ \mathbf{in}\ \underline{0} = \mathbf{rlet}\ \Sigma\ \mathbf{in}\ \alpha\ (\kappa_1\langle\rangle) \longrightarrow \alpha\ (\mathbf{rlet}\ \Sigma\ \mathbf{in}\ (\kappa_1\langle\rangle)) \longrightarrow \alpha\ (\kappa_1(\mathbf{rlet}\ \Sigma\ \mathbf{in}\ \langle\rangle)) \longrightarrow \alpha\ (\kappa_1\ \langle\rangle).$$

This means that we are not able to distinguish **rlet** Σ **in** $\underline{0}$ from $\underline{0}$ inside $\lambda\mu\nu{=}$. We make this precise in Section 4.1. ◂

Convention 3.2.17. Since we were hiding outermost **rlet**-bindings in the Haskell-like notation, see Example 3.2.8 and 3.2.9, it is convenient to do the same for terms like **rlet** Σ **in** $\underline{0}$ in Example 3.2.16 above. Hence, we will usually denote **rlet** Σ **in** $\underline{0}$ by $\underline{0}$ instead.

We now use this convention to demonstrate the computational behaviour of H from Example 3.2.9 on the alternating bit stream.

Example 3.2.18. Using the computation $\underline{0}^\omega.\mathrm{hd} \longrightarrow \underline{0}$ from Example 3.2.16, the definition of alt in Example 3.2.10 and of H, we obtain the following computation.

$$(H\ \underline{0}^\omega\ \mathrm{alt}).\mathrm{hd} \longrightarrow (f\ (\underline{0}^\omega.\mathrm{hd})\ \underline{0}^\omega\ \mathrm{alt}).\mathrm{hd} \longrightarrow (f\ \underline{0}\ \underline{0}^\omega\ \mathrm{alt}).\mathrm{hd} \longrightarrow \mathrm{alt.hd} \longrightarrow \underline{0}$$

Thus, we have $(H\ \underline{0}^\omega\ \mathrm{alt}).\mathrm{hd} \twoheadrightarrow \underline{0}$ and $(H\ \underline{0}^\omega\ \mathrm{alt}).\mathrm{hd} \equiv \underline{0}$, see Definition 3.2.14. For simplicity, we will usually just write $\equiv$ whenever we carry out computation steps, as this allows us to abstract away from the exact number of intermediate computation steps. We use this convention in the following computation of the head of $H\ 1^\omega\ \mathrm{alt}$.

$$(H\ 1^\omega\ \mathrm{alt}).\mathrm{hd} \equiv (f\ (1^\omega.\mathrm{hd})\ 1^\omega\ \mathrm{alt}).\mathrm{hd} \equiv (f\ 1\ 1^\omega\ \mathrm{alt}).\mathrm{hd} \equiv (f\ 0\ 1^\omega\ (\mathrm{alt.tl})).\mathrm{hd} \equiv \mathrm{alt.tl.hd} \equiv \underline{1}$$ ◂

Let us briefly come back to the selector example (3.2.11).

Example 3.2.19. Recall that we have defined $\mathrm{odd}\colon A^\omega \to A^\omega$ to be odd $=$ select oddF. We are going to use odd with $A = \mathrm{Nat}$ and apply it to alt from Example 3.2.10. Since the intuition of odd is that it only keeps the odd positions of a stream, counting from 0, we expect that odd alt is equal to $\underline{1}$ in all positions. Indeed, the computational behaviour of odd alt is given as follows. First, we have

$$\begin{aligned}\mathrm{odd\ alt} &= \mathrm{select\ oddF\ alt}\\ &\longrightarrow \mathrm{select}_\mu\ (\mathrm{oddF.out})\ \mathrm{alt}\\ &\longrightarrow \mathrm{select}_\mu\ (\mathrm{drop}\ (\mathrm{pres\ oddF}))\ \mathrm{alt}\\ &\longrightarrow \mathrm{select}_\mu\ (\mathrm{pres\ oddF})\ (\mathrm{alt.tl}),\end{aligned}$$

which can only be reduced further if we request the head or tail of odd alt, see the definition of select_μ. For the head we get then the following reduction to $\underline{1}$, which is the element in the first odd position of alt.

$$(\mathrm{select}_\mu\ (\mathrm{pres\ oddF})\ (\mathrm{alt.tl})).\mathrm{hd} \longrightarrow \mathrm{alt.tl.hd} \longrightarrow \underline{1}$$

Upon requesting the tail of odd alt, we obtain

$$(\mathrm{select}_\mu\ (\mathrm{pres\ oddF})\ (\mathrm{alt.tl})).\mathrm{tl} \longrightarrow \mathrm{select\ oddF}\ (\mathrm{alt.tl.tl}) \longrightarrow \mathrm{select\ oddF\ alt} = \mathrm{odd\ alt}.$$

Thus, if we read further positions by applying the tail repeatedly as in $(\mathrm{odd\ alt}).\mathrm{tl}^n.\mathrm{hd}$, we would always get $\underline{1}$ as output. So, as expected, odd alt is the constant stream $\underline{1}^\omega$. We will develop in Section 4.1 a notion of program equivalence that allows us to formulate this statement precisely by saying that odd alt and $\underline{1}^\omega$ are *observationally equivalent*. ◂

Example 3.2.20. Recall that we have defined after Example 3.2.11 a map $\mathrm{mod}\colon F \to \mathrm{Nat} \to \mathrm{Nat}$. Let us first show that $\mathrm{mod\ oddF}\ \underline{n} \equiv \underline{2n+1}$ by induction on n. In the base case we have

$$\begin{aligned}\mathrm{mod\ oddF}\ \underline{0} &\equiv m_\mu(\mathrm{drop}\ (\mathrm{pres\ oddF}))\ \underline{0}\\ &\equiv \mathrm{suc}\ (m_\mu\ (\mathrm{pres\ oddF})\ \underline{0})\\ &\equiv \mathrm{suc}\ \underline{0}\\ &= \underline{2\cdot 0+1}\end{aligned}$$

For the induction step, we assume that $\mathrm{mod\ oddF}\ \underline{n} \equiv \underline{2n+1}$ holds and proceed by

$$\begin{aligned}\mathrm{mod\ oddF}\ \underline{n+1} &\equiv m_\mu(\mathrm{drop}\ (\mathrm{pres\ oddF}))\ \underline{n+1}\\ &\equiv \mathrm{suc}\ (m_\mu\ (\mathrm{pres\ oddF})\ (\mathrm{suc}\ \underline{n}))\\ &\equiv \mathrm{suc}\ (\mathrm{suc}\ (\mathrm{mod\ oddF}\ \underline{n}))\\ &\equiv \mathrm{suc}\ (\mathrm{suc}\ \underline{2n+1})\\ &= \underline{2n+1+2}\\ &= \underline{2(n+1)+1},\end{aligned}$$

which proves the induction step. Note that the last two steps are arithmetic identities on natural numbers and are *not* given by conversion of terms.

It is now easy to see that mod oddF is the modulus of continuity of odd $=$ select oddF, that is, for all $n \in \mathbb{N}$ and $s, t : A^\omega$ we have

$$d(s,t) \le 2^{-(\mathrm{mod\ oddF}\ n)} \implies d(\mathrm{odd}\ s, \mathrm{odd}\ t) \le 2^{-n}.$$

More generally, select is continuous on the domain $F \times A^\omega$, see [BH16]. ◂

So far, we have only seen examples of computations that eventually end. Since $\lambda\mu\nu{=}$ is fairly liberal in how equational specifications can be used, we can write non-terminating programs in $\lambda\mu\nu{=}$, that is, terms from which infinite reduction sequences originate.

Example 3.2.21. The following term Ω is a generic term for any type A that has no normal form, which means that any reduction sequence from Ω is infinite.

$$\Omega_A \coloneqq \mathbf{rlet}\ \omega : A = \{\cdot \mapsto \omega\}\ \mathbf{in}\ \omega$$

With $\Sigma = \{\omega : A = \{\cdot \mapsto \omega\}\}$, we have

$$\omega = \cdot[\omega/\cdot] \succ_\Sigma \omega,$$

thus $\Omega_A \longrightarrow \Omega_A$, which gives immediately the desired infinite reduction sequence.

The same mechanism also allows us to construct a fixed point of an arbitrary term $f : A \to A$:

$$\mathrm{fix}\, f \coloneqq \mathbf{rlet}\, \mathrm{r} : A = \{\cdot \mapsto f\ r\}\ \mathbf{in}\ r.$$

Depending on the evaluation strategy and f, this term might have a normal form. We will not go into this further here though. ◀

We will now prove some results about the reduction relation of $\lambda\mu\nu{=}$, which are going to be important in what follows. Let us first show that computations do not alter types of terms.

Theorem 3.2.22. *The reduction relation $\longrightarrow_\Sigma$ preserves types, that is, if $\Gamma; \Sigma \vdash t : A$ and $t \longrightarrow_\Sigma s$ for some term s, then $\Gamma; \Sigma \vdash s : A$. In particular, $\longrightarrow$ preserves types of closed terms.*

Proof. The proof that $\longrightarrow_\Sigma$ preserves types on the rules that do not involve **rlet**-blocks is immediate by type preservation of the contraction relation $\succ_\Sigma$, see [Abe+13]. Type preservation of reductions that are given by the three rules involving **rlet**-blocks follows immediately from the fact that the type of $\mathbf{rlet}\, \Sigma_1\ \mathbf{in}\ t$ is given by the type of t. □

We have discussed after Definition 3.2.3 that nothing prevents us in $\lambda\mu\nu{=}$ from giving declaration bodies that are not exhaustive or may even overlap in some case. The following example demonstrates the computations on such terms.

Example 3.2.23. First, we note that there are terms from which no reduction is possible because their behaviour is underspecified. An example is the term $\lambda\{\cdot.\mathrm{pr}_1 \mapsto t\}.\mathrm{pr}_2$, since the body $\{\cdot.\mathrm{pr}_1 \mapsto t\}$ has no case for the second projection and thus there is no contraction possible. Second, from the term $\lambda\{\cdot.\mathrm{pr}_1 \mapsto t\ ;\ \cdot.\mathrm{pr}_1 \mapsto s\}.\mathrm{pr}_1$ there are two reductions possible: either to t or to s. If t and s cannot be reduced to a common term, then the reduction is non-deterministic in this case. ◀

This example demonstrates that we need to ensure that a collection of copatterns covers all possible cases (in which case we call it *exhaustive*) and does not lead to non-deterministic reductions (in which case we call it *non-overlapping*). The former is important because we might otherwise get stuck on a term of type Nat that cannot be reduced further and is neither a successor or zero. Non-determinism, on the other hand, causes confluence of the reduction relation to fail. Since confluence becomes important in Section 4.1, we need to rule out non-deterministic declaration bodies as well. Collections of copatterns that are exhaustive and non-overlapping are said to be *covering*. Formally, covering is given with respect to a type as in the following definition, which we adapt from [Abe+13].

Definition 3.2.24. A set of *annotated copatterns* is a set $\mathcal{Q}$ of triples $(\Gamma; q; B)$, where Γ is a context, q a copattern and B a type. We write $\mathcal{Q}^\downarrow$ for the underlying set of copatterns:

$$\mathcal{Q}^\downarrow \coloneqq \{q \mid (\Gamma; q; B) \in \mathcal{Q}\}.$$

A set Q of copatterns *covers* a type A, if there is a set $\mathcal{Q}$ of annotated copatterns such that $Q = \mathcal{Q}^\downarrow$

and $A \lhd| \mathcal{Q}$ can be derived inductively from the following rules.

$$\frac{}{A \lhd| \{(\emptyset; \cdot; A)\}} \qquad \frac{A \lhd| \mathcal{Q} \cup \{(\Gamma, x : \mathbf{1}; q; C)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma; q[\langle\rangle/x]; C)\}} \qquad \frac{A \lhd| \mathcal{Q} \cup \{(\Gamma; q; B \to C)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma, x : B; q\, x; C)\}}$$

$$\frac{A \lhd| \mathcal{Q} \cup \{(\Gamma; q; B \times C)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma; q.\mathrm{pr}_1; B), (\Gamma; q.\mathrm{pr}_2; C)\}} \qquad \frac{A \lhd| \mathcal{Q} \cup \{(\Gamma; q; \nu X.\, B)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma; q.\mathrm{out}; B[\nu X.\, B/X])\}}$$

$$\frac{A \lhd| \mathcal{Q} \cup \{(\Gamma, x : B_1 + B_2; q; C)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma, x_i : B_i; q[\kappa_i\, x_i/x]; C) \mid i = 1, 2\}} \qquad \frac{A \lhd| \mathcal{Q} \cup \{(\Gamma, x : \mu X.\, B; q; C)\}}{A \lhd| \mathcal{Q} \cup \{(\Gamma, y : B[\mu X.\, B/X]; q[\alpha\, y/x]; C)\}} \quad \blacktriangleleft$$

Note that a set of covering copatterns is constructed in Definition 3.2.24 by extending copatterns on coinductive types and by refining patterns on inductive types. This can be seen by relating the types used in the rules of the covering relation to the typing of copatterns:

Lemma 3.2.25. *If $A \lhd| \mathcal{Q}$, then for all $(\Gamma; q; B) \in \mathcal{Q}$ we have $\Gamma \vdash_{\mathfrak{cop}} q : A \Rightarrow B$.*

Proof. This follows easily by induction on the rules applied to obtain $A \lhd| \mathcal{Q}$. □

Let us give some example for covering and non-covering sets of copatterns.

Example 3.2.26. Let $U = F_\mu \to B^\omega \to B^\omega$ and

$$Q = \{(\cdot\ (\mathrm{pres}\ x)\ s).\mathrm{hd}, (\cdot\ (\mathrm{pres}\ x)\ s).\mathrm{tl}, \cdot\ (\mathrm{drop}\ u)\ s\},$$

which are the three copatterns used in the definition of $\mathrm{select}_\mu : U$ in Example 3.2.11. We define two contexts $\Gamma_1 = x : F, s : A^\omega$ and $\Gamma_2 = u : F_\mu, s : A^\omega$, and use these to check that there is a set of annotated copatterns that corresponds to Q and covers U:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{U \lhd| \{(\emptyset; \cdot; F_\mu \to B^\omega \to B^\omega)\}}}{U \lhd| \{(y : F_\mu; \cdot\ y; A^\omega \to A^\omega)\}}}{U \lhd| \{(y : F_\mu, s : A^\omega; \cdot\ y\ s; A^\omega)\}}}{U \lhd| \{(z : F + F_\mu, s : A^\omega; \cdot\ (\alpha\ z)\ s; A^\omega)\}}}{U \lhd| \{(x : F, s : A^\omega; \cdot\ (\mathrm{pres}\ x)\ s; A^\omega), (u : F_\mu, s : A^\omega; \cdot\ (\mathrm{drop}\ u)\ s; A^\omega)\}}}{U \lhd| \{(\Gamma_1; (\cdot\ (\mathrm{pres}\ x)\ s).\mathrm{out}; A \times A^\omega), (\Gamma_2; \cdot\ (\mathrm{drop}\ u)\ s; A^\omega)\}}}{U \lhd| \{(\Gamma_1; (\cdot\ (\mathrm{pres}\ x)\ s).\mathrm{hd}; A), (\Gamma_1; (\cdot\ (\mathrm{pres}\ x)\ s).\mathrm{tl}; A^\omega), (\Gamma_2; \cdot\ (\mathrm{drop}\ u)\ s; A^\omega)\}}$$

Thus Q covers the type $U = F_\mu \to B^\omega \to B^\omega$. Non-examples are the sets $\{\cdot.\mathrm{hd}\ ;\ \cdot.\mathrm{tl}\ ;\ \cdot.\mathrm{hd}\}$, as these copatterns are overlapping, and $\{(\cdot\ 0).\mathrm{hd}\ ;\ (\cdot\ 0).\mathrm{tl}\}$, as this set is not exhaustive. ◀

We can now use the definition of covering to define a class of terms that can be reduced to certain normal forms (called values, see Definition 3.2.30) and on which the reduction relation is confluent.

Definition 3.2.27. A declaration body $D = \{q_1 \mapsto t_1; \ldots; q_n \mapsto t_n\}$ with $\Gamma; \Sigma \vdash_{\mathrm{bdy}} D : A$ is said to be *well-covering*, if $\{q_i \mid i = 1, \ldots n\}$ covers A. We then call terms t and declaration block Σ *well-covering*, if every declaration body in t and Σ is well-covering. ◀

Example 3.2.28. By Example 3.2.26, we have that select_μ is well-covering. ◂

Let us now put the covering relation to use. The class of terms that we have been advertising, on which the reduction relation is well-behaved, is given by the following definition.

Definition 3.2.29. For a declaration block Σ and a type A, we define several sets of terms:

$$\begin{aligned} \Lambda^{=}_{\Sigma}(A) &:= \{t \mid \emptyset; \Sigma \vdash t : A \text{ and } t \text{ is well-covering}\} \\ \Lambda^{=}_{\Sigma} &:= \bigcup_{A \in \mathrm{Ty}} \Lambda^{=}_{\Sigma}(A) \\ \Lambda^{=}(A) &:= \Lambda^{=}_{\emptyset}(A) \\ \Lambda^{=} &:= \Lambda^{=}_{\emptyset} \end{aligned}$$

First, we show that every well-covering term is either a value or can be further reduced. The notion of value is given in the following definition.

Definition 3.2.30. A term t of type A is a *value* if (i) A is an inductive type and $t = c\ s$ for a constructor c and a value s; (ii) A is coinductive; or (iii) $t = x$ for some variable x. ◂

Lemma 3.2.31 (Progress)**.** *Let A be a type and Σ a well-covering declaration block. For every term $t \in \Lambda^{=}_{\Sigma}(A)$, we have that either t is a value or that there is a t' with $t \longrightarrow_\Sigma t'$.*

Proof. The proof of this fact is easily adopted from [Abe+13, Sec. 5], where one crucially has to appeal to the two rules that distribute evaluation contexts and constructors over **rlet**-bindings. □

On the same class of terms, we now also prove confluence. The details of the proof can be found in Appendix A.

Proposition 3.2.32. *For any type A and any well-covering Σ, $\longrightarrow_\Sigma$ is confluent on $\Lambda^{=}_{\Sigma}(A)$ and, in particular, $\longrightarrow$ is confluent on $\Lambda^{=}$.*

Let us briefly discuss what the relevance of the progress result and confluence is in the context of this thesis. First of all, confluence allows us to simplify the definition of convertibility to $t_1 \equiv t_2 \iff \exists t_3.\, t_1 \longrightarrow\!\!\!\!\!\rightarrow t_3 \longleftarrow\!\!\!\!\!\leftarrow t_2$, see Section 2.1. More importantly, it ensures that constructors in head position of terms are unique.

Lemma 3.2.33. *If $t \in \Lambda^{=}(A_1 + A_2)$ and $t \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\ t'$ for some $i \in \{1, 2\}$, then for all t'' with $t \longrightarrow\!\!\!\!\!\rightarrow \kappa_j\ t''$ we must have that $i = j$.*

Proof. By confluence, there must be a term s with $\kappa_i\ t' \longrightarrow\!\!\!\!\!\rightarrow s \longleftarrow\!\!\!\!\!\leftarrow \kappa_j\ t''$. Since reduction steps do not change or remove constructors, it follows that $s = \kappa_i\ s'$ with $i = j$. □

In combination with the progress lemma, we can find for every term of inductive type a *weak head normal form (WHNF)* with unique constructors as follows. We denote the set family of all strongly normalising terms by $\mathbf{SN} = \{\mathbf{SN}_A\}_{A \in \mathrm{Ty}}$, whereby $\mathbf{SN}_A = \{t \in \Lambda^{=}(A) \mid t\!\downarrow\}$, see Section 2.1.

Lemma 3.2.34 (Weak Head Normal Forms)**.** *Let $t \in \mathbf{SN}_A$ be a strongly normalising term. If $A = \mu X.\, B$, then there is a $t' \in \mathbf{SN}_{B[\mu X.\, B/X]}$ with $t \equiv_\alpha t'$. Otherwise, if $A = B_1 + B_2$, then there is $i \in \{1, 2\}$ and a $t' \in \mathbf{SN}_{B_i}$ with $t \equiv \kappa_i\ t'$, and for every s with $t \equiv \kappa_j\ s$ we have $i = j$.*

Proof. Since t is strongly normalising, there is a normal form u with $t \longrightarrow\!\!\!\!\!\rightarrow u$. The claim follows by applying Lemma 3.2.31 to u. For sums, uniqueness of i is the content of Lemma 3.2.33. □

3.3. Relation Between $\lambda\mu\nu$ and $\lambda\mu\nu{=}$

Let us comment on the relation between the two presented calculi. As we have seen, they embody two different styles of recursion We can, however, emulate the recursion style of $\lambda\mu\nu$ in $\lambda\mu\nu{=}$.

Definition 3.3.1. We simultaneously define by induction on the type A the following three term constructors in $\lambda\mu\nu{=}$.

1. Given a type A and a term t with $X \Vdash A : \mathbf{Ty}$ and $\Gamma; \Sigma \vdash t : B \to C$, we define a term $\Gamma; \Sigma \vdash A[t] : A[B] \to A[C]$;
2. given a term r with $\Gamma; \Sigma \vdash r : A[B/X] \to B$, we define an iterator $\Gamma; \Sigma \vdash R_\mu\, r : \mu X.\, A \to B$;
3. given a term r with $\Gamma; \Sigma \vdash r : B \to A[B/X]$, we define a coiterator $\Gamma; \Sigma \vdash R_\nu\, r : B \to \nu X.\, A$.

The term $A[t]$ is given literally as in $\lambda\mu\nu$ (Definition 3.1.9), only with iter replaced by R_μ and coiter by R_ν. For the iterator, we define

$$R_\mu\, r \coloneqq \mathbf{rlet}\ f : \mu X.\, A \to B = \left\{\cdot\ (\alpha\ x) \mapsto r\ (A[f]\ x)\right\}\ \mathbf{in}\ f,$$

and the coiterator, on the other hand, we define by

$$R_\nu\, r \coloneqq \mathbf{rlet}\ f : B \to \nu X.\, A = \left\{(\cdot\ x).\mathrm{out} \mapsto A[f]\ (r\ x)\right\}\ \mathbf{in}\ f. \qquad \blacktriangleleft$$

We then obtain for the terms in Definition 3.3.1 the same typing rules as in $\lambda\mu\nu$.

Lemma 3.3.2. *The following typing rules hold in $\lambda\mu\nu{=}$ for the terms defined in Definition 3.3.1.*

$$\frac{\Gamma; \Sigma \vdash t : B \to C}{\Gamma; \Sigma \vdash A[t] : A[B/X] \to A[C/X]} \qquad \frac{\Gamma; \Sigma \vdash r : A[B/X] \to B}{\Gamma; \Sigma \vdash R_\mu\, r\, s : \mu X.\, A \to B} \qquad \frac{\Gamma; \Sigma \vdash r : B \to A[B/X]}{\Gamma; \Sigma \vdash R_\nu\, r\, s : B \to \nu X.\, A}$$

These terms are, moreover, well-covering if r, s and t are.

Proof. The proof for the first rule is again carried out by induction on A, just like in Lemma 3.1.10. That the typing rules of the iterator and coiterator hold is seen as follows. First, we show that the bodies for the coiteration and iteration are well-typed, where we use the typing rule for $A[t]$. For the case of R_ν, we let $D_\nu = \left\{(\cdot\ x).\mathrm{out} \mapsto A[f]\ (r\ x)\right\}$ and $\Sigma' = \Sigma, (f : B \to \nu X.\, A = D_\nu)$. Then we can derive that D is well-typed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Sigma' \vdash f : B \to \nu X.\, A}{\Gamma; \Sigma' \vdash A[f] : A[B] \to A[\nu X.\, A]} \qquad \dfrac{\Gamma; \Sigma' \vdash r : B \to A[B] \qquad \Gamma; \Sigma' \vdash x : B}{\Gamma, x : B; \Sigma' \vdash r\ x : A[B]}}{\Gamma, x : B; \Sigma' \vdash A[f]\ (r\ x) : A[\nu X.\, A]}}{\Gamma; \Sigma' \vdash_{\mathrm{bdy}} \left\{(\cdot\ x).\mathrm{out} \mapsto A[f]\ (r\ x)\right\} : B \to \nu X.\, A}$$

Similarly for the body of R_μ, we put $D_\mu = \left\{\cdot\ (\alpha\ x) \mapsto r\ (A[f]\ x)\right\}$ and $\Sigma' = \Sigma, (f : B \to \nu X.\, A = D_\mu)$, allowing us to carry out the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Sigma' \vdash f : \mu X.\, A \to B}{\Gamma; \Sigma' \vdash A[f] : A[\mu X.\, A] \to A[B]} \qquad \Gamma, x : A[\mu X.\, A]; \Sigma' \vdash x : A[\mu X.\, A]}{\Gamma, x : A[\mu X.\, A]; \Sigma' \vdash A[f]\ x : A[B]}}{\Gamma, x : A[\mu X.\, A]; \Sigma' \vdash r\ (A[f]\ x) : B}}{\Gamma; \Sigma' \vdash_{\mathrm{bdy}} \left\{\cdot\ (\alpha\ x) \mapsto r\ (A[f]\ x)\right\} : \mu X.\, A \to B}$$

Since the bodies are well-typed, it is clear that $R_\mu\ r$ and $R_\nu\ r$ have the required types.

Finally, well-covering follows also by a straightforward induction on the type A. We check, as an example, that $R_\nu\ r$ is well-covering. This requires us to construct an annotated set of copatterns that corresponds to $\{(\cdot\ x).\mathrm{out}\}$, which can be constructed by the following derivation.

$$\dfrac{\dfrac{\dfrac{}{B \to \nu X.\, A \lhd\mid \{(\emptyset; \cdot; B \to \nu X.\, A)\}}}{B \to \nu X.\, A \lhd\mid \{(x : B; \cdot\ x; \nu X.\, A)\}}}{B \to \nu X.\, A \lhd\mid \{(x : B; (\cdot\ x).\mathrm{out}; A[\nu X.\, A/X])\}}$$

Thus, $\{(\cdot\ x).\mathrm{out}\}$ covers $B \to \nu X.\, A$, as required, and $R_\nu\ r$ is well-covering. □

This definition allows us to translate terms of $\lambda\mu\nu$ to terms in $\lambda\mu\nu{=}$.

Definition 3.3.3. We define a function $\ulcorner(-)\urcorner\colon \Lambda \to \Lambda^{=}$ by induction on the terms of $\lambda\mu\nu$:

$$\begin{aligned}
\ulcorner\langle\rangle\urcorner &= \langle\rangle & \ulcorner s\ t\urcorner &= \ulcorner s\urcorner\ \ulcorner t\urcorner \\
\ulcorner\kappa_i\ t\urcorner &= \kappa_i\ \ulcorner t\urcorner & \ulcorner x\urcorner &= x \\
\ulcorner\alpha\ t\urcorner &= \alpha\ \ulcorner t\urcorner & \ulcorner\lambda x.\, t\urcorner &= \lambda\{\cdot\ x \mapsto \ulcorner t\urcorner\} \\
\ulcorner\pi_i\ t\urcorner &= \ulcorner t\urcorner.\mathrm{pr}_i & \ulcorner\langle s, t\rangle\urcorner &= \lambda\{\cdot\ .\mathrm{pr}_1 \mapsto \ulcorner t\urcorner\ ;\ \cdot\ .\mathrm{pr}_2 \mapsto \ulcorner s\urcorner\} \\
\ulcorner\xi\ t\urcorner &= \ulcorner t\urcorner.\mathrm{out} & \ulcorner\mathrm{iter}^{\mu X.\, A}\ (x.\, t)\ s\urcorner &= R_\mu\ (\lambda x.\ulcorner t\urcorner)\ \ulcorner s\urcorner \\
& & \ulcorner\mathrm{coiter}^{\nu X.\, A}\ t\ s\urcorner &= R_\nu\ (\lambda x.\ulcorner t\urcorner)\ \ulcorner s\urcorner \\
& & \ulcorner\{\kappa_1\ x \mapsto s\ ;\ \kappa_2\ y \mapsto t\}\ u\urcorner &= \lambda\{\kappa_1\ x \mapsto \ulcorner s\urcorner\ ;\ \kappa_2\ y \mapsto \ulcorner t\urcorner\}\ \ulcorner u\urcorner
\end{aligned}$$

Theorem 3.3.4. *Definition 3.3.3 gives rise to a map $\ulcorner(-)\urcorner\colon \Lambda \to \Lambda^{=}$, in the sense that terms of $\lambda\mu\nu$ are mapped to well-covering terms of $\lambda\mu\nu{=}$. Moreover, it preserves types:*

$$\Gamma \vdash t : A \implies \Gamma \vdash \ulcorner t\urcorner : A.$$

Proof. Both well-covering and type preservation are proved by induction on terms. All cases, except that for iter and coiter, are thereby immediate. That the translation of iter and coiter is well-covering and type preserving has been proved in Lemma 3.3.2. □

3.4. Conclusion and Related Work

The purpose of this chapter was to give some understanding of how to program with inductive-coinductive data. We chose here two calculi that employ different ways of dealing with inductive and coinductive data, and compared their corresponding strengths and weaknesses: The first calculus $\lambda\mu\nu$ is hard to use, but all terms are strongly normalising. On the other hand, $\lambda\mu\nu{=}$ is fairly easy to use, but we can write non-terminating programs in it. Note that we have not proved strong normalisation of $\lambda\mu\nu$ in this chapter, this will be rectified in Chapter 7. We tackle the problem of non-termination in $\lambda\mu\nu{=}$ in the next chapter.

In Chapters 6 and 7, we will establish category theoretical and type theoretic extensions of the simple calculus $\lambda\mu\nu$ to the setting of dependent types. This will allow us to intertwine programming and reasoning. Also, we will give there further applications of inductive-coinductive programming.

The calculi presented in this chapter have been discussed in one form or another in other places, so let us clarify the relation to existing work. Possibly the earliest calculi with mixed inductive-coinductive types were given by Mendler [Men87; Men91] and Hagino [Hag87]. Mendler combines thereby the polymorphic λ-calculus with inductive and coinductive types, whereas Hagino bases his calculus on the notion of dialgebra, something we will come back to in Chapter 7. Hagino's iteration and coiteration schemes match quite clearly with the ones we used in Section 3.1. Mendler [Men91], on the other hand, models the iteration principle using constants

$$R^{\mu X.\, A} \colon \forall Y.(\forall Z.(Z \to Y) \to A \to Y) \to \mu X.\, A \to Y.$$

If we note that the type $\forall Z.(Z \to Y) \to A \to Y$ corresponds with maps $A[Y/X] \to Y$, then we can read the type of $R^{\mu X.\, A}$ as "for every type Y and every A-algebra there is a map $\mu X.\, A \to Y$". Thus, also Mendler's style of introducing iteration gives rise to the same iteration principle that we were using in $\lambda\mu\nu$, cf. Section 3.3 and [UV96]. Similarly, also his coiteration principle matches that of $\lambda\mu\nu$. Thus both calculi agree on the inductive and coinductive types, but Mendler's calculus also features (impredicative) polymorphism. This allows Mendler to obtain iteration and coiteration principles without having to introduce explicitly the action of types on terms, see [UV96; UV02] for a detailed discussion and [UV99a; Ven00] for a discussion from a category theoretical perspective. To ensure strong normalisation, Mendler, of course, allows fixed points only of positive types. In this respect, Mendler's calculus [Men87; Men91] goes beyond $\lambda\mu\nu$, as we only allow strictly positive types. In principle, we could also extend $\lambda\mu\nu$ and $\lambda\mu\nu{=}$ to arbitrary positive types, see [Mat99] for useful examples, but this would make the later developments in this thesis much harder. Similarly, Greiner [Gre92] considered a language with positive inductive and coinductive types as well, but restricts to top-level polymorphism. Finally, Howard [How92; How96a] proved confluence and strong normalisation for a calculus that is essentially an extension of $\lambda\mu\nu$ to positive types, and implemented this calculus in the Lemon language [How95]. Interestingly, Howard [How96a] extends his calculus also with an operation force$\colon \nu X.\, A \to \mu X.\, A$ for, what he calls, pointed types. This operation introduces full recursion into the calculus for pointed types, essentially by turning the category associated to the calculus into an algebraically compact category à la Freyd [Fre90].

Besides calculi with mixed inductive-coinductive types, one can find in the literature many calculi with just inductive types [BDS13; Con97; Gim95; Mat99; Wer94], only coinductive types [Møg14] and calculi with one-layer inductive or coinductive types [Geu92]. Of these, most are based on iteration/coiteration schemes to avoid termination issues. Since programming with equational definitions is much more practical, as we have seen in Section 3.2, also calculi that support programming with recursive equations have been proposed. The calculus in Section 3.2 is, as mentioned, based on the work by Abel et al. [Abe+13]. The termination issue for such calculi is being dealt with by, for example, Abel and Pientka [AP13], Atkey and McBride [AM13], Barthe et al. [Bar+04], Coquand [Coq93], Giménez [Gim95], Sacchini [Sac13] and Xi [Xi01].

Finally, syntactic specification formats for elements of final coalgebras have been proposed in several forms. The formats that are closest to the specifications that can be given in $\lambda\mu\nu{=}$ are the behavioural differential equations (BDE) described, for example, by Hansen et al. [HKR17], Kupke and Rutten [KR08] and Rutten [Rut03]. To guarantee (unique) solutions to BDEs, Kupke and Rutten [KR08] define a notion of complete sets of equations. These correspond to covering sets of copatterns, which we used to ensure that normalising terms have weak head normal forms with unique constructors in head position (Lemma 3.2.34). An important difference between the systems

of equations developed in [KR08] and copattern equations is that specifications in $\lambda\mu\nu{=}$ can only be given on fixed point types, which correspond to final coalgebras. In contrast, the equations in [KR08] can be given on so-called observational coalgebras, which are coalgebras for which the map into the final coalgebra is injective. It would be interesting to study a type theory that allows the specification of sub-types of largest fixed point types by specifying additional properties that elements of that type must satisfy, see [Bas15b] for an example. Another approach to specifications that do not force the use of iteration and coiteration schemes are cyclic proof systems and games. These have, for example, been investigated by Cockett [Coc01] and Santocanale [San02a; San02b].

Notes

[10] Note that the the addition of natural numbers can also be defined without iteration on functions by using

$$g_+ \coloneqq \lambda m.\, \{\kappa_1\ y \mapsto m\ ;\ \kappa_2\ k \mapsto \mathrm{suc}\ k\}\ x$$
$$\mathrm{plus} \coloneqq \lambda n\ m.\, \mathrm{iter}^{\mathrm{Nat}}\ (x.\, g_+\ m)\ n$$

instead. However, it is illustrative to see how the function space can be exploited in iteration. An common example of a function that can only be defined with higher-order iteration is the Ackermann function.

[11] The extra case for types that do not use any variables in the definition of the action of types on terms is needed in a very subtle way. Not only does this give us better computation rules, but it is actually crucial in the proof of Proposition 4.1.9. An elegant way around this extra case would be to use externally given monotonicity witnesses, as in [Mat99; UV02]. We will discuss other approaches in the conclusion of this chapter.

[12] It should be noted that programming just with iteration and coiteration schemes can be very challenging. For example, the predecessor on natural numbers or the construction of a stream from its head and tail are surprisingly difficult to define. In the setting of initial algebras and final coalgebras their definitions follow from Lambek's lemma (Lemma 2.5.4), but in a syntactic calculus it takes a bit of ingenuity to come up with their definitions. For the predecessor function this even lead to a nice story about Kleene having an epiphany at the dentist of how to implement the predecessor in pure λ-calculus [Cro75]. This problem can be remedied by generalising the concept of iteration and coiteration that we used here, to that of primitive recursion and primitive corecursion, respectively, see [BDS13; Geu92; Lei89; UV99c; Ven00; VU98]. We will not discuss this further though, as it not relevant to us here, and since these problems do not occur in the calculus introduced in Section 3.2 below. Also the dependent type theory in Chapter 7 immediately admits primitive recursion.

[13] One could continue to prove relevant properties about this calculus like *subject reduction* (types of terms are preserved under reduction steps), *confluence* (any two reduction sequences originating at the same term can be joined), *progress* (every term can be reduced to a value, where a *value* is a term that has been constructed only by introduction rules, projection and weakening) and *strong normalisation* (no term has an infinite reduction sequence). However, we refrain from doing so, as (variations of) the calculus $\lambda\mu\nu$ have been extensively studied. Proofs of strong normalisation can, for example, be found in [AA99; How92; Mat99], whereby confluence is also proven by Howard

[How92]. We just note here that $\lambda\mu\nu$ can be encoded into other calculi presented in this thesis, while preserving reduction steps. More specifically, $\lambda\mu\nu$ is a (strongly normalising) fragment of the calculus considered in Section 3.2, from which we can obtain subject reduction, progress and confluence, see Section 3.3. $\lambda\mu\nu$ is also a non-dependent fragment of the dependent type theory studied in Chapter 7, from which we can obtain strong normalisation. Hence, we will not study the properties of the present calculus in isolation.

[14] We do not have a precise notion of predicate at this point, but we just assume an intuitive understanding that we can check whether a predicate holds for a given element of A.

[15] We note that one can also define by iteration a function $\_@\_ : A^\omega \to \mathrm{Nat} \to A$ that, given a stream s and an index $n : \mathrm{Nat}$, returns the element $s \mathbin{@} n$ at position n in s. Moreover, there is a stream $\mathrm{nats} : \mathrm{Nat}^\omega$ that enumerates the natural numbers. It is then fairly easy to see that $(\mathrm{select}\ x\ s) \mathbin{@} \underline{n} \equiv s \mathbin{@} (\mathrm{select}\ x\ \mathrm{nats} \mathbin{@} \underline{n})$ for every $n \in \mathbb{N}$. From these considerations, it follows that $\mathrm{mod}\ x\ \underline{n} \equiv \mathrm{select}\ x\ \mathrm{nats} \mathbin{@} \underline{n}$, which gives us a more direct way to express mod and a direct link with select. To ease calculations, we stick here to the explicit definition of mod though.

Observations

> *In this sense, meanings control us, inculcate obedience to the discipline inscribed in them. And this is by no means purely institutional or confined to the educational process. [...] The right word in a new situation does not always readily present itself. Language sometimes seems to lead a life of its own. Words are unruly "They've a temper, some of them", Humpty Dumpty goes on to observe.*
>
> — Catherine Belsey, "Poststructuralism", 2002.

In the last chapter, we defined the two calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. The aim of this chapter is to make some first steps towards reasoning principles for the programs of these calculi.

One of the most important ingredients in reasoning about programs is the ability to compare their behaviour. So far, we are only able to compare programs on the basis of their *computational behaviour*, which is given by the reduction relations on the terms of the two calculi. In particular, these reduction relations give rise to convertibility as their equivalence closure. Convertibility can readily be used to compare the computational behaviour of programs. However, this comparison is often too fine-grained to be useful. For instance, the terms $\underline{1}^\omega$ and odd alt are not convertible to each other. But we have

$$\underline{1}^\omega.\mathrm{hd} \equiv \underline{1} \qquad \underline{1}^\omega.\mathrm{tl} \equiv \underline{1}^\omega \qquad \text{(Example 3.2.16)}$$

$$\mathrm{odd\ alt}\ .\mathrm{hd} \equiv \underline{1} \qquad \mathrm{odd\ alt}\ .\mathrm{tl} \equiv \mathrm{odd\ alt}, \qquad \text{(Example 3.2.19)}$$

from which we would like to infer that the programs $\underline{1}^\omega$ and odd alt produce the same stream entries but arrive at them through different computations. The question is now how we can formally characterise that they have "the same entries"?

An important notion that pervades reasoning about programs is that of *observations*, which determine in turn the *observational behaviour* of programs. For example, on streams the two fundamental observations are taking the head and the tail of a stream. Functions, on the other hand, have as many observations as they have possible arguments, since we can observe the outcome of the application of a function to any argument. In a mixed inductive-coinductive setting, general observations on programs are more complex. For instance, what is the observational behaviour of the term $\underline{1}$, or of the selector oddF from Example 3.2.11? In the former case, $\underline{1}$ is as a value, see Definition 3.2.30, completely determined by its computational behaviour. However, for oddF we have oddF.out ≡ drop (pres oddF), thus we should clearly consider the constructors drop and pres as part of the observational behaviour of oddF. This raises the question of how to characterise the observational behaviour for general inductive-coinductive programs. We will resolve this in Section 4.1.2 by introducing a type-driven notion of observation in the form of a modal logic of program tests, which will formally define what the observable behaviour of program is. Having defined the observational behaviour of programs, we can also say that programs are *observationally equivalent*, given that they show the same observational behaviour.

Despite the fact that comparing the observational behaviour of programs is often more useful, also computational behaviour plays a role. For example, the coiteration on streams over A (Section 3.3)

has the property that

$$(R_\nu\ r\ s).\mathrm{hd} \equiv (r\ s).\mathrm{pr}_1,$$

where $r\colon B \to A \times B$ and $s : B$. Since conversions that arise from iteration and coiteration can be very complex and tedious to carry out by hand, it lifts a great burden from us if we can pass these to a machine. This, however, requires that the conversion relation is computable, which would fail if we had, for example, that $R_\nu \equiv h$ for all homomorphisms $h\colon B \to A^\omega$ that fulfil $h.\mathrm{hd} \equiv (r\ s).\mathrm{pr}_1$ and $h.\mathrm{tl} \equiv h\,((r\ s).\mathrm{pr}_2)$. Rather, we will say that $R_\nu\ r$ and all such homomorphisms h are observationally equivalent, which we write as $R_\nu\ r \equiv_{\mathrm{obs}} h$.

Our goal in Section 4.2 is to analyse and compare both computational and observational behaviour in a common framework, towards which we proceed in two steps. The first step in understanding the difference between the two behaviours is the use of categories in which objects are types and morphisms are equivalence classes of terms up-to convertibility. Thus, we first take only the computational behaviour into account. In this setup, we can already identify some concepts that resemble category theoretical constructions. For instance, we will find that product types are weak products, in other words, they have projections and maps can be paired, but this pairing is not unique. Next, we will extend these categories to 2-categories, where the objects and morphisms are as before but 2-cells relate observationally equivalent terms. Setting up 2-categories in this way has two effects. First of all, we can identify more precisely the structure that the types come with. For example, we have that the product types give rise to pseudo-products, which in this case means that the pairing is unique up-to observational equivalence. Secondly, we are able to distinguish the computational from the observational behaviour, in the sense that computational equivalence is given as equality of morphisms, while observational equivalence is situated at the level of 2-cells. This has the effect that, for instance, the identity $(R_\nu\ r\ s).\mathrm{hd} = (r\ s).\mathrm{pr}_1$ holds in the 2-category, but in general we only have an isomorphism $R_\nu\ r \cong h$, see Definition 2.6.3, between the coiteration and a homomorphism h. Thus, this 2-category theoretical setup allows to identify the properties and differences of computational and observable behaviour of programs in $\lambda\mu\nu$ and $\lambda\mu\nu{=}$.

The purpose of this chapter is to develop the fundamental ability to compare the behaviour of programs, specifically programs of inductive-coinductive types. Towards the end of Section 4.1, we are thus able to express interesting properties of programs in $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. In Chapter 5, we will then provide techniques to prove such properties. Also the category theoretical description of the types and terms will reappear in Chapter 6, although there we will deal with dependent types. Finally, in Chapter 7, we will be able to compare programs in the there provided dependent type theory, without having to resort to an external logic or set theory.

Original Publication Section 4.1 has been presented in [BH16]. In the same paper, a category theoretical account of the calculus from Section 3.2 has also been given. However, the analysis we gave there was not very precise, as we did not make any distinction between computations and observations. This is remedied in Section 4.2.

4.1. Observational Equivalence and Normalisation

We have already come across the notion of observational behaviour of programs a couple of times. The purpose of this section is to make this notion precise. In what follows, we introduce *test formulas*

that allow us to carry out observations on terms of $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. We then use these test formulas to define *observational equivalence*, which compares terms on the basis of their observational rather than their computational behaviour. Moreover, we use test formulas to single out the *observationally normalising* terms of $\lambda\mu\nu{=}$, which are those terms that respond to any observation with a strongly normalising term.

Test formulas are formulas of a multi-modal logic, where the modalities capture the observations that can be carried out on terms of a given type. Thus, test formulas are themselves typed. These formulas are closely related to the testing logic considered in [San11, Sec. 6.2], and the many-sorted coalgebraic modal logics for polynomial functors studied, for example, by Jacobs [Jac01] and Kupke [Kup06, Sec. 2.1.2]. Considering how the calculi in Chapter 3 were set up, the reader might have guessed by now that the essential observations on terms of product- and ν-type are given by the corresponding destructors π_1, π_2 and ξ, respectively. What needs clarification are observations on terms of inductive type and on functions.

The distinguishing feature of inductive types are the constructors. This is reflected both in the case distinction for sums and the recursion principle in $\lambda\mu\nu$, and the covering of patterns in $\lambda\mu\nu{=}$. In both calculi, we construct programs on sum and μ-types from programs that cover all possible constructor cases. These considerations lead us to use modalities in the testing logic that inspect the constructor in the head position of terms, see Definition 4.1.19.

For tests on terms of function type we would expect that an observation amounts to function application, thus there should be a modality for each possible argument. However, there is a caveat. If we allow any argument, then we could inspect the computational behaviour of terms, which should not be an observable property. Hence, it should also not be taken into account for observational equivalence. Consider, for example, the following two terms $f_1, f_2 \colon \mathrm{Nat} \to \mathrm{Nat}$ in $\lambda\mu\nu{=}$.

$$f_1 \coloneqq \lambda\{\cdot\, 0 \mapsto 0 \;;\; \cdot\, (\mathrm{suc}\, n) \mapsto 1\}$$
$$f_2 \coloneqq \lambda\{\cdot\, 0 \mapsto 0 \;;\; \cdot\, (\mathrm{suc}\, 0) \mapsto 1 \;;\; \cdot\, (\mathrm{suc}\, (\mathrm{suc}\, n)) \mapsto 1\}$$

Then for the non-terminating program Ω from Example 3.2.21, we have $f_1\, (\mathrm{suc}\, \Omega) \longrightarrow 1$, whereas $f_2\, (\mathrm{suc}\, \Omega)$ does not have a weak head normal form. Thus, we can distinguish f_1 and f_2 by applying them to a term that does not have a normal form. If we restrict attention to strongly normalising terms t though, then $f_1\, t \equiv f_2\, t$. For this reason, the modalities on function types will range only over the observationally normalising terms of $\lambda\mu\nu{=}$, see Definition 4.1.3.

Observationally normalising terms subsume both terminating computations of inductive type and, what is usually called, *productive* programs of coinductive type. In the case of streams, the requirement that a term $t : \mathrm{Nat}^\omega$ is observationally normalising will be defined as follows. First, the head of t is a strongly normalising term of type Nat, that is, $t.\mathrm{hd} \equiv \underline{n}$ for some $n \in \mathbb{N}$. Second, its tail $t.\mathrm{tl}$ is strongly normalising and again observationally normalising. What this gives us, is that we are able to interpret t as a function $t^\dagger \colon \mathbb{N} \to \mathbb{N}$ in the set-theoretic sense by iteratively extracting the entries of t, in other words we put $t^\dagger(0) = n$ and $t^\dagger(k+1) = (t.\mathrm{tl})^\dagger(k)$. In general, the restriction to observationally normalising terms allows us to interpret these as set-theoretic (total) functions.

4.1.1. Observational Normalisation

As we discussed in the introduction, we want to test terms of function type only on terms that normalise under any observation. Thus, to be able to define tests for $\lambda\mu\nu{=}$, our first task is to define

the corresponding class of, what we call here, observationally normalising terms. The intention is to capture the largest class of terms that can serve as an interpretation for types, which separates inductive and coinductive types. In particular, this means that all terms in that class must be strongly normalising, that is, correspond to terminating computations, and the class must be closed under observations. For instance, given observationally normalising terms t and s with $t\colon A \to B$ and $s : A$, also the application $t\ a$ should be observationally normalising. Since we are interested in the *largest* class of terms, the iterators and coiterators, which we introduced in Definition 3.3.1, should also be in that class. The definition of observational normalisation and the proof that the iterators and coiterators are observationally normalising are our concern in the remainder of the present subsection. Closure under observations in proved in Theorem 4.1.17 in the next subsection.

Before we come to the definition of observational normalisation, we need to introduce some notation. First of all, to define observational normalisation on open terms, we need to generalise the notion of substitution that we used before, see Definition 3.2.12. Recall that a substitution is a map σ that assigns to variables in TeVar terms in $\lambda\mu\nu$ or $\lambda\mu\nu{=}$. Moreover, we denoted the type of x in a context Γ by $\Gamma(x)$.

Definition 4.1.1. Let $U = \{U_A\}_{A\in \mathrm{Ty}}$ be a family of sets indexed by types. For a context Γ, we say that a substitution σ is U-Γ-*closing*, if $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\sigma)$ and for every $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) \in U_{\Gamma(x)}$. We denote by $\mathrm{Subst}(U;\Gamma)$ the set of all U-Γ-closing substitutions.

Secondly, we will define observationally normalising terms on open types, whose variables will be interpreted as sets of terms. This leads us to the following definition.

Definition 4.1.2. Let Θ be a type variable context and υ a substitution of closed types for the variables in Θ, see Definition 3.1.1. We denote by $I_{\Theta,\upsilon}$ be set of valuation functions V that map variables X in Θ to subsets of $\Lambda^{=}_{\upsilon(X)}$, that is,

$$I_{\Theta,\upsilon} = \left\{V\colon \Theta \to \mathcal{P}(\Lambda^{=}) \;\middle|\; V(X) \subseteq \Lambda^{=}_{\upsilon(X)}\right\}.$$

Moreover, let us denote by ε the empty valuation. Finally, if $V \in I_{\Theta,\upsilon}$, $A \in \mathrm{Ty}$ and $U \subseteq \Lambda^{=}_{A}$, then we write $V[X \mapsto U]$ for the updated valuation in $I_{(\Theta,X),\upsilon[A/X]}$. ◀

Now we are in the position to define the set of observationally normalising terms. We do this in three steps: First, we define by iteration on types an operator **WD** ("well-defined") that takes an open type and a valuation for the variables of that type as input. This first step is required to define observationally normalising terms on fixed point types. Next, the set **ON** of observationally normalising terms is given by evaluating **WD** on closed types. Finally, observational normalisation on open terms is defined in terms **ON** and **ON**-closing substitutions. These three steps are captured in the following definitions.

Definition 4.1.3. Let Θ be a type variable context, $\upsilon \in \mathrm{TySubst}(\Theta)$, and $V \in I_{\Theta,\upsilon}$. Given a type A

with $\Theta \Vdash A : \mathbf{Ty}$, we define $\mathbf{WD}_A^{\upsilon,V} \subseteq \Lambda^{=}_{A[\upsilon]}$ by iteration on A as follows.

$$
\begin{aligned}
\mathbf{WD}_{\mathbf{1}}^{\upsilon,V} &= \mathbf{SN}_{\mathbf{1}} \\
\mathbf{WD}_{X}^{\upsilon,V} &= V(X) \\
\mathbf{WD}_{A_1+A_2}^{\upsilon,V} &= \bigcup_{i=1,2} \left\{ s \in \mathbf{SN}_{(A_1+A_2)[\upsilon]} \;\middle|\; (\exists s'.\, s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, s') \wedge \left(\forall s'.\, (s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, s') \implies s' \in \mathbf{WD}_{A_i}^{\upsilon,V} \right) \right\} \\
\mathbf{WD}_{A_1\times A_2}^{\upsilon,V} &= \left\{ s \in \mathbf{SN}_{(A_1\times A_2)[\upsilon]} \;\middle|\; \textstyle\bigwedge_{i=1,2} s.\mathrm{pr}_i \in \mathbf{WD}_{A_i}^{\upsilon,V} \right\} \\
\mathbf{WD}_{A\to B}^{\upsilon,V} &= \left\{ s \in \mathbf{SN}_{A\to B[\upsilon]} \;\middle|\; \forall t \in \mathbf{WD}_A^{(),\varepsilon}.\, s\, t \in \mathbf{WD}_B^{\upsilon,V} \right\} \\
\mathbf{WD}_{\mu X.\, A}^{\upsilon,V} &= \mu U. \left\{ s \in \mathbf{SN}_{(\mu X.\, A)[\upsilon]} \;\middle|\; (\exists s'.\, s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \right. \\
&\qquad\qquad \left. \wedge \left(\forall s'.\, (s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \implies s' \in \mathbf{WD}_A^{\upsilon[\mu X.\, A/X], V[X\mapsto U]} \right) \right\} \\
\mathbf{WD}_{\nu X.\, A}^{\upsilon,V} &= \nu U. \left\{ s \in \mathbf{SN}_{(\nu X.\, A)[\upsilon]} \;\middle|\; s.\mathrm{out} \in \mathbf{WD}_A^{\upsilon[\nu X.\, A/X], V[X\mapsto U]} \right\},
\end{aligned}
$$

In this definitions, the fixed points are taken in the complete lattice of subsets of $\Lambda^{=}$, which exist because $\mathbf{WD}_A^{\upsilon,V}$ is clearly monotone in V in each case. For a closed type A, that is, if Θ is empty, we define the set family $\mathbf{ON}_A$ of *observationally normalising terms* of type A by[16]

$$\mathbf{ON}_A \coloneqq \mathbf{WD}_A^{(),\varepsilon}.$$

Moreover, given a context Γ, we define $\mathbf{ON}_A^{\Gamma}$ to be the set of all terms t such that $t \in \mathbf{SN}$ and $t[\sigma] \in \mathbf{ON}_A$ for any $\mathbf{ON}$-Γ-closing substitution σ, see Definition 4.1.1. Finally, given a type A with $\Theta, X \Vdash A : \mathbf{Ty}$, we define a monotone function

$$
\begin{aligned}
&\Psi_A^{\upsilon,V} : \mathcal{P}\left(\Lambda^{=}_{A[\mu X.\, A/X][\upsilon]}\right) \to \mathcal{P}\left(\Lambda^{=}_{A[\mu X.\, A/X][\upsilon]}\right) \\
&\Psi_A^{\upsilon,V}(U) = \left\{ s \;\middle|\; (\exists s'.\, s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \wedge \left(\forall s'.\, (s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \implies s' \in \mathbf{WD}_A^{\upsilon[\mu X.\, A/X], V[X\mapsto U]} \right) \right\} \cap \mathbf{SN},
\end{aligned}
$$

so that $\mathbf{WD}_{\mu X.\, A}^{\upsilon,V} = \mu U.\, \Psi_A^{\upsilon,V}(U)$. ◂

Before we continue, let us remark on the use of the least fixed point the definition of $\mathbf{WD}$ in the case of μ-types. This least fixed point ensures that a term can only be finitely often unfolded, which is crucial in ensuring that the iterator that we introduced in Definition 3.3.1 is observationally normalising. This, in turn, will allow us to prove in Section 4.2 that in the category of observationally normalising terms, all μ-types are initial algebras. Interestingly, in the calculus $\lambda\mu\nu{=}$ strong normalisation and admitting only finitely many unfolding steps do not coincide, as the following example shows.[17]

Example 4.1.4. Given a closed type A, we can define the type A^* of lists over A to be $\mu X.\, \mathbf{1} + A \times X$. Suppose that we are given a term a of type A, then the following is a program in $\lambda\mu\nu{=}$.

$$
\begin{aligned}
&p : A \times A^* \\
&p.\mathrm{pr}_1 = a \\
&p.\mathrm{pr}_2 = \alpha\, (\kappa_1\, p)
\end{aligned}
$$

Note that $p.\mathrm{pr}_2$ denotes an infinite list of a's, but $p.\mathrm{pr}_2 \in \mathbf{SN}_{A^*}$. However, one can show $p.\mathrm{pr}_2 \notin \mathbf{ON}_{A^*}$ by induction on a hypothetical proof of $p.\mathrm{pr}_2 \in \mathbf{ON}_{A^*}$. ◀

As intended, strong normalisation is implied by observational normalisation.

Lemma 4.1.5. *There is an inclusion* $\mathbf{ON} \sqsubseteq \mathbf{SN}$ *of set families, that is,* $\mathbf{ON}_A \subseteq \mathbf{SN}_A$ *for all* $A \in \mathrm{Ty}$.

Proof. To prove that $\mathbf{ON} \sqsubseteq \mathbf{SN}$ holds, one shows for all types A with $\Theta \Vdash A : \mathbf{Ty}$, substitutions $\upsilon \in \mathrm{TySubst}(\Theta)$ and valuations $V \in I_{\Theta,\upsilon}$ with $V(X) \subseteq \mathbf{SN}_{\upsilon(X)}$ that $\mathbf{WD}_A^{\upsilon,V} \subseteq \mathbf{SN}_{A[\upsilon]}$ by induction on A. This induction is then almost trivial, except for the cases of μ- and ν-types, in which we have to use the fixed point properties. The result follows for closed types by definition of $\mathbf{ON}$. □

In following example we illustrate that the definition of observational normalisation includes terms that do not necessarily correspond to guarded recursive equations [Bar04] or guarded recursive schemes [Gim95]. This example is intended for readers who are familiar with *stream differential equations* (SDE) [HKR17] with the goal being an interpretation of some SDEs in $\lambda\mu\nu{=}$.

Example 4.1.6. To simplify the development in this example, we restrict attention to SDEs over a signature with two symbols, say f_1 and f_2, each of arity 2. Given a type A, let us denote the product $A \times A$ by A^2. We can represent a signature with two symbols of arity 2 by the type

$$F \coloneqq X^2 + X^2,$$

in the sense that the type F^* of terms over this signature with variables in Y is given by

$$F^* \coloneqq \mu X.\, F + Y$$

with $Y \Vdash F^* : \mathbf{Ty}$. Let us now fix a type V of four elements that we will use as stream variables:

$$V \coloneqq (\mathbf{1} + \mathbf{1}) + (\mathbf{1} + \mathbf{1}).$$

We denote these variables by

$$\begin{aligned} \mathsf{x} &\coloneqq \kappa_1\,(\kappa_1\,\langle\rangle) & \mathsf{y} &\coloneqq \kappa_1\,(\kappa_2\,\langle\rangle) \\ \mathsf{x}' &\coloneqq \kappa_2\,(\kappa_1\,\langle\rangle) & \mathsf{y}' &\coloneqq \kappa_2\,(\kappa_2\,\langle\rangle). \end{aligned}$$

The variables represent two streams x and y, and their tails x' and y' in the syntax of SDEs.

Let $A \in \mathrm{Ty}$. A system of SDEs over F is given by four observationally normalising terms

$$\begin{aligned} h_i &\colon A^2 \to A, \qquad i = 1, 2 \\ d_i &\colon A^2 \to F^*[V], \quad i = 1, 2, \end{aligned}$$

We interpret these terms as the following system of stream differential equations, see [HKR17].

$$\begin{aligned} f_i(s,t).\mathrm{hd} &= h_i(s.\mathrm{hd}, t.\mathrm{hd}) \\ f_i(s,t).\mathrm{tl} &= d_i(s.\mathrm{hd}, t.\mathrm{hd})[s/\mathsf{x}, t/\mathsf{y}, (s.\mathrm{tl})/\mathsf{x}', (t.\mathrm{tl})/\mathsf{y}'] \end{aligned} \tag{4.1}$$

These equations say that the head and tail of $f_i(s,t)$ are given by evaluating at the head of s and t the maps h_i and d_i, respectively. In the case of the tail, we also need to substitute the input

streams s and t and their tails into the expression that results from d_i. The goal of this example is to implement the substitution and the above system of SDEs in $\lambda\mu\nu{=}$.

We first deal with substitution. Let $I \coloneqq V \to A^\omega$, which is the type of interpretations of the variables in V over streams. The substitution operation we need is a map that, given an interpretation $u : I$, replaces every variable $v : V$ in a term in $F^*[V]$ by $u\,v$, thus obtaining a term in $F^*[A^\omega]$. This is in fact just given by $F^*[u] \colon F^*[V] \to F^*[A^\omega]$, see Definition 3.3.1. To simplify notation, we introduce a term $\mathrm{mkI} \colon A^\omega \times A^\omega \to I$ that allows us to construct the substitution in (4.1):

$$\begin{aligned} \mathrm{mkI}\,u\,\mathsf{x} &= u.\mathrm{pr}_1 & \mathrm{mkI}\,u\,\mathsf{y} &= u.\mathrm{pr}_2 \\ \mathrm{mkI}\,u\,\mathsf{x}' &= u.\mathrm{pr}_1.\mathrm{tl} & \mathrm{mkI}\,u\,\mathsf{y}' &= u.\mathrm{pr}_2.\mathrm{tl} \end{aligned}$$

The substitution in (4.1) is then given for stream terms s and t by $\mathrm{mkI}\,\langle s, t\rangle$. We can now interpret (4.1) in $\lambda\mu\nu{=}$ by mutual recursion:

$$\begin{aligned} &\lfloor - \rfloor \colon F[A^\omega] \to A^\omega \\ &\lfloor \kappa_i\,u \rfloor.\mathrm{hd} = h_i\,((\mathrm{hd} \times \mathrm{hd})\,u) && i = 1, 2 \\ &\lfloor \kappa_i\,u \rfloor.\mathrm{tl} = \left\lfloor F^*[\mathrm{mkI}\,u]\,(d_i\,((\mathrm{hd} \times \mathrm{hd})\,u)) \right\rfloor_\mu && i = 1, 2 \\ &\lfloor - \rfloor_\mu \colon F^*[A^\omega] \to A^\omega \\ &\lfloor \alpha\,(\kappa_1\,t) \rfloor_\mu = \lfloor F[\lfloor - \rfloor_\mu]\,t \rfloor \\ &\lfloor \alpha\,(\kappa_2\,s) \rfloor_\mu = s \end{aligned}$$

The way $\lfloor - \rfloor$ corresponds to (4.1) is as follows. First of all, recall that the SDE term "$f_i(s,t)$" is given by the term $\kappa_i\,\langle s, t\rangle$ of type $F[A^\omega]$. Thus, the cases for head and tail in the definition of $\lfloor \kappa_i\,u \rfloor$ correspond to the head and tail cases in (4.1). This also readily explains the definition of the head of $\lfloor - \rfloor$. However, the tail case needs a few more words of explanation. In (4.1), a substitution and the complex SDE term d_i are used. The substitution is carried out in the definition of $\lfloor - \rfloor$ by applying $F^*[\mathrm{mkI\,u}]$ to d_i. Afterwards, we use $\lfloor - \rfloor_\mu$ to give an interpretation of the SDE term, in which variables have been replaced by elements of A and streams over A. Interpreting such a term proceeds then by iteration, where each function symbol, denoted by t in the definition of $\lfloor - \rfloor_\mu$, is again interpreted by appealing to $\lfloor - \rfloor$, and each stream variable is interpreted by its assigned stream.

Note that in the tail-case of $\lfloor - \rfloor$ the outermost function is not $\lfloor - \rfloor$ itself. Thus, the above definition is not syntactically guarded and it is thus not immediately clear that it is well-defined. We claim, however, that $\lfloor - \rfloor \in \mathbf{ON}$. Since we first need that the action $F^*[-]$ on terms preserves observational normalisation, we defer the proof.

It should also be noted that $\lfloor - \rfloor$ can be implemented in Agda by using *sized types*. Sized types allow the check for well-definedness of recursive equations through the use of a typing mechanism, see for example [AP13]. We will not discuss this here further, but see [Bas16] for an implementation of $\lfloor - \rfloor$ over a general signature that uses sized types in Agda. ◂

Throughout the remainder of this section we show that the iterator and coiterator from Definition 3.3.1 are observationally normalising. This is what one would expect, as the encoding of the calculus $\lambda\mu\nu$ into $\lambda\mu\nu{=}$ from Section 3.3 should restrict to **ON** because all the terms in $\lambda\mu\nu$ are strongly normalising. However, the proof is fairly involved, since the iterator and coiterator are defined simultaneously with the action of open types on terms. This forces us to prove also observational normalisation the iterators, coiterators and type actions simultaneously, see Proposition 4.1.9 below.

In order to ease the following development, let us introduce a notation for the construction of sets of terms of function type. This construction can be recognised as one that is used in, for example, strong normalisation proofs to give an interpretation to function spaces.

Definition 4.1.7. Let $A, B \in \mathrm{Ty}$, $S \subseteq \Lambda^{=}_{A}$ and $T \subseteq \Lambda^{=}_{B}$. We define a set $S \Rightarrow T \subseteq \Lambda^{=}_{A\to B}$ by

$$S \Rightarrow T \coloneqq \{t \in \Lambda^{=}_{A\to B} \mid \forall u \in S.\, t\, u \in T\}. \qquad \blacktriangleleft$$

Note that with this notation we have that

$$\mathbf{WD}^{v,V}_{A\to B} = \left(\mathbf{ON}_A \Rightarrow \mathbf{WD}^{v,V}_{B}\right) \cap \mathbf{SN}_{A\to B[v]}.$$

Throughout the proof of observational normalisation, we will need to maintain that valuations are backwards closed under reductions. This is a technical, but unfortunately necessary, condition that we cast into the following definition.

Definition 4.1.8. We say that a predicate $S \subseteq \Lambda^{=}_{A}$ *backwards closed under reductions*, if for all s, t with $s \in \mathbf{SN}$, $s \longrightarrow\!\!\!\!\!\rightarrow t$ and $t \in S$, we also have $s \in S$. A valuation is backwards closed, if it is point-wise backwards closed under reductions. $\blacktriangleleft$

We are now in the position to formulate the main result of this section: the iterators and coiterators from Definition 3.3.1 are observationally normalising, and the action of types on terms preserves observational normalisation. Since we prove the result by induction on open types, we need to formulate it more generally in terms of **WD**. The three parts of the following Proposition 4.1.9 are then proved by mutual induction.

Proposition 4.1.9. *Let Θ be a type variable context and $v, \phi \in \mathrm{TySubst}(\Theta)$. Moreover, let $V \in I_{\Theta, v}$ $W \in I_{\Theta,\phi}$ be backwards closed valuations with $V(X) \subseteq \mathbf{ON}_{v(X)}$ and $W(X) \subseteq \mathbf{ON}_{\phi(X)}$ for all $X \in \Theta$.*

(i) If C is a type with $\Theta \Vdash C : \mathbf{Ty}$ and $\overrightarrow{t} : \Theta \to \Lambda^{=}$ with $\overrightarrow{t}(X) \in V(X) \Rightarrow W(X)$, then

$$C\left[\overrightarrow{t}\right] \in \mathbf{WD}^{v,V}_{C} \Rightarrow \mathbf{WD}^{\phi,W}_{C}.$$

(ii) If C, D are types with $\Theta, Y \Vdash C : \mathbf{Ty}$ and $\Theta \Vdash D : \mathbf{Ty}$, then

$$R_\mu \in \left(\mathbf{WD}^{v[D[\phi]/X], V\left[X\mapsto \mathbf{WD}^{\phi,W}_{D}\right]}_{C} \Rightarrow \mathbf{WD}^{\phi,W}_{D}\right) \Rightarrow \mathbf{WD}^{v,V}_{\mu Y.\, C} \Rightarrow \mathbf{WD}^{\phi,W}_{D}$$

(iii) If C, D are types with $\Theta, Y \Vdash C : \mathbf{Ty}$ and $\Theta \Vdash D : \mathbf{Ty}$, then

$$R_\nu \in \left(\mathbf{WD}^{v,V}_{D} \Rightarrow \mathbf{WD}^{\phi[D[v]/X], W\left[X\mapsto \mathbf{WD}^{v,V}_{D}\right]}_{C}\right) \Rightarrow \mathbf{WD}^{v,V}_{D} \Rightarrow \mathbf{WD}^{\phi,W}_{\nu Y.\, C}$$

In particular, we have

(ii') If A is a type with $X \Vdash A : \mathbf{Ty}$, $B \in \mathrm{Ty}$ and $r \in \mathbf{ON}_{A[B/X]\to B}$, then $R_\mu\, r \in \mathbf{ON}_{(\mu X.\, A)\to B}$.

(iii') If A is a type with $X \Vdash A : \mathbf{Ty}$, $B \in \mathrm{Ty}$ and $r \in \mathbf{ON}_{B\to A[B/X]}$, then $R_\nu\, r \in \mathbf{ON}_{B\to \nu X.\, A}$. $\blacktriangleleft$

In what follows, we prepare the proof of Proposition 4.1.9. First of all, we prove that **WD** is backwards closed. This will be necessary, since we will interpret variables at fixed point types by appealing to to **WD**.

Lemma 4.1.10. *Suppose A is a type with $\Theta \Vdash A : \mathbf{Ty}$, υ a substitution and $V \in I_{\Theta,\upsilon}$ a backwards closed valuation. Then also $\mathbf{WD}_A^{\upsilon,V}$ is backwards closed. In particular, for all $B \in \mathrm{Ty}$, $t \in \mathbf{ON}_B$ and $s \in \mathbf{SN}$ with $s \longrightarrow\!\!\!\!\!\rightarrow t$, we have $s \in \mathbf{ON}_B$.*

Proof. Let A, υ and V be as assumed in the lemma. We show that $\mathbf{WD}_A^{\upsilon,V}$ is backwards closed by induction on A.

- Note that $\mathbf{WD}_{\mathbf{1}}^{\upsilon,V}$ is backwards closed by definition, and that $\mathbf{WD}_X^{\upsilon,V}$ is backwards closed by the assumption that V is backwards closed.
- In case of the sum types, suppose that $t \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V}$ and that $s \longrightarrow\!\!\!\!\!\rightarrow t$. To prove that $s \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V}$, we have to provide an $i \in \{1,2\}$, such that there is an s' with $s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, s'$ and such that for any such s', we have $s' \in \mathbf{WD}_{A_i}^{\upsilon,V}$. Since $t \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t'$ for some $i \in \{1,2\}$ and t', we also have a weak head reduction $s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t$. Thus, we can use $s' = t'$. Suppose now that $s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, s'$ for some s'. Then, by confluence, there is a t' with $t \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t' \mathrel{\reflectbox{$\longrightarrow\!\!\!\!\!\rightarrow$}} \kappa_i\, s'$. Hence, we also have $s' \longrightarrow\!\!\!\!\!\rightarrow t'$. Since $t \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V}$, we obtain $t' \in t \in \mathbf{WD}_{A_i}^{\upsilon,V}$ and, by the induction hypothesis, that $s' \in \mathbf{WD}_{A_i}^{\upsilon,V}$. Putting everything together, we have $s \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V}$.
- Similarly, the case of μ-types is proven by defining

$$S \coloneqq \left\{ t \in \mathbf{WD}_{\mu X.\, A}^{\upsilon,V} \;\middle|\; \forall s.\, (s \longrightarrow\!\!\!\!\!\rightarrow t) \implies s \in \mathbf{WD}_{\mu X.\, A}^{\upsilon,V} \right\}$$

  and then showing that $\mathbf{WD}_{\mu X.\, A}^{\upsilon,V} \subseteq S$ by appealing to the definition of $\mathbf{WD}_{\mu X.\, A}^{\upsilon,V}$ as a least fixed point, confluence and the induction hypothesis.
- For function types, we have to show for given $t \in \mathbf{WD}_{A\to B}^{\upsilon,V}$ and $s \longrightarrow\!\!\!\!\!\rightarrow t$, that $s \in \mathbf{SN}$ and that for all $u \in \mathbf{ON}A$ that $s\, u \in \mathbf{WD}_B^{\upsilon,V}$. The former is given by assumption, while the latter follows from the induction hypothesis from the fact that $t\, u \in \mathbf{WD}_B^{\upsilon,V}$ and because $s \longrightarrow\!\!\!\!\!\rightarrow t$ induces $s\, u \longrightarrow\!\!\!\!\!\rightarrow t\, u$.
- The case of products follows by a similar argument as in the case of function types.
- To prove the case of ν-types, one easily shows, by using the induction hypothesis and the fact that $\mathbf{WD}_{\nu X.\, A}^{\upsilon,V}$ defined as a largest fixed point, that the set S given by

$$S \coloneqq \{ s \in \mathbf{SN}_{\nu X.\, A[\upsilon]} \mid s \longrightarrow\!\!\!\!\!\rightarrow t \}$$

  is contained in $\mathbf{WD}_{\nu X.\, A}^{\upsilon,V}$. The claim of the lemma then immediately follows. □

Next, we show that that **WD** is closed under identity maps, composition and the formation of pattern matching and pairs. These results give us directly the proof of Proposition 4.1.9.(i) in the case of function, sum and product types.

Lemma 4.1.11. *Let A, A_1, A_2 be types with $\Theta_1 \Vdash A, A_1, A_2 : \mathbf{Ty}$, B, B_1, B_2 with $\Theta_2 \Vdash B, B_1, B_2 : \mathbf{Ty}$, C a type with $\Theta_3 \Vdash C : \mathbf{Ty}$, υ, ϕ and ω substitutions for, respectively, Θ_1, Θ_2 and Θ_3, $V \in I_{\Theta_1,\upsilon}$, $W \in I_{\Theta_2,\upsilon}$, and $U \in I_{\Theta_3,\omega}$. Then the following holds.*

(i) $\mathrm{id} \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_A^{\upsilon,V}$

(ii) If $s \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_B^{\phi,W}$ *and* $t \in \mathbf{WD}_B^{\upsilon,V} \Rightarrow \mathbf{WD}_C^{\phi,U}$, *then* $t \circ s \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_C^{\omega,U}$.

(iii) If $t_i \in \mathbf{WD}_{A_i}^{\upsilon,V} \Rightarrow \mathbf{WD}_B^{\phi,W}$ *for all* $i \in \{1,2\}$ *then*

$$\lambda\{\kappa_1\, x \mapsto t_1\, x\ ;\ \kappa_2\, y \mapsto t_2\, y\} \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V} \Rightarrow \mathbf{WD}_B^{\phi,W}.$$

(iv) If $t_i \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_{B_i}^{\phi,W}$ *for all* $i \in \{1,2\}$ *then*

$$\lambda x.\, \langle t_1\, x, t_2\, x\rangle \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_{B_1\times B_2}^{\phi,W}.$$

(v) If $t_i \in \mathbf{WD}_{A_i}^{\upsilon,V} \Rightarrow \mathbf{WD}_{B_i}^{\phi,W}$ *for all* $i \in \{1,2\}$ *then* $t_1 + t_2 \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V} \Rightarrow \mathbf{WD}_{B_1+B_2}^{\phi,W}$ *and* $t_1 \times t_2 \in \mathbf{WD}_{A_1\times A_2}^{\upsilon,V} \Rightarrow \mathbf{WD}_{B_1\times B_2}^{\phi,W}$.

Proof. (i) Since $\mathbf{WD}_A^{\upsilon,V}$ is backwards closed by Lemma 4.1.10, $\mathrm{id} \in \mathbf{WD}_A^{\upsilon,V} \Rightarrow \mathbf{WD}_A^{\upsilon,V}$ immediately follows from $\mathrm{id}\, u \longrightarrow\!\!\!\!\!\rightarrow u$ for $u \in \mathbf{WD}_A^{\upsilon,V}$.

(ii) Again by using Lemma 4.1.10, closure of **WD** under composition follows directly from the definition of **WD** and $\Rightarrow$.

(iii) Closure under pattern matching is given as follows. To show $\lambda\{\kappa_1\, x \mapsto t_1\, x\ ;\ \kappa_2\, y \mapsto t_2\, y\} \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V} \Rightarrow \mathbf{WD}_B^{\phi,W}$, suppose $s \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V}$. Then there is an $i \in \{1,2\}$ and an $s' \in \mathbf{WD}_{A_i}^{\upsilon,V}$, such that $s \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, s'$. This gives us $\lambda\{\kappa_1\, x \mapsto t_1\, x\ ;\ \kappa_2\, y \mapsto t_2\, y\}\ s \longrightarrow\!\!\!\!\!\rightarrow t_i\, s'$ and, by assumption, that $t_i\, s' \in \mathbf{WD}_B^{\phi,V}$. Hence, by Lemma 4.1.10, $\lambda\{\kappa_1\, x \mapsto t_1\, x\ ;\ \kappa_2\, y \mapsto t_2\, y\}\ s \in \mathbf{WD}_B^{\phi,V}$, and thus $\lambda\{\kappa_1\, x \mapsto t_1\, x\ ;\ \kappa_2\, y \mapsto t_2\, y\} \in \mathbf{WD}_{A_1+A_2}^{\upsilon,V} \Rightarrow \mathbf{WD}_B^{\phi,W}$.

(iv) The case for pairing is analogous to that for pattern matching.

(v) This follows immediately from closure under composition, and from closure under pattern matching and pairing, respectively. □

After all this preliminary setup, we can finally proceed to prove the main result of this section.

Proof of Proposition 4.1.9. We prove the three parts of the proposition mutually by induction on C.

(i) Let C be a type with $\Theta \Vdash C : \mathbf{Ty}$ and $\vec{t}$ with $\vec{t}\,(X) \in V(X) \Rightarrow W(X)$, as in the proposition. To prove $C\big[\vec{t}\,\big] \in \mathbf{WD}_C^{\upsilon,V} \Rightarrow \mathbf{WD}_C^{\phi,W}$, we proceed by distinguishing the cases for C.

- If C is closed or $C = \mathbf{1}$, then $C\big[\vec{t}\,\big] = \mathrm{id}_C$. The proof is then immediate from Lemma 4.1.11.(i).

- For variables, we have $X[\vec{t}] = \vec{t}(X)$. By assumption, we have $\vec{t}(X) \in V(X) \Rightarrow W(X)$ and thus $\vec{t}(X) \in \mathbf{WD}_X^{v,V} \Rightarrow \mathbf{WD}_X^{\phi,W}$.
- The cases for sum, product and function types are given immediately by Lemma 4.1.11 and the induction hypothesis.
- For the case of least fixed point types, recall that

$$(\mu Y.\, C)\big[\vec{t}\,\big] = R_\mu\ \big(\alpha \circ C\big[\vec{t}, \mathrm{id}\big]\big),$$

which means we have to prove that

$$R_\mu\ \big(\alpha \circ C\big[\vec{t}, \mathrm{id}\big]\big) \in \Big(\mathbf{WD}_{\mu Y.\, C}^{v,V} \Rightarrow \mathbf{WD}_{\mu Y.\, C}^{\phi,W}\Big). \tag{4.2}$$

We define new substitutions $v' = v[\mu Y.\, C[\phi]/X]$ and $\phi' = \phi[\mu Y.\, C[\phi]/X]$, and valuations $V' = V\Big[X \mapsto \mathbf{WD}_{\mu Y.\, C}^{\phi,W}\Big]$ and $W' = W\Big[X \mapsto \mathbf{WD}_{\mu Y.\, C}^{\phi,W}\Big]$. Note that V' and W' are backwards closed by Lemma 4.1.10, and that $\mathrm{id} \in V'(X) \Rightarrow W'(X)$ by Lemma 4.1.11. By the induction hypothesis we have thus obtain

$$C\big[\vec{t}, \mathrm{id}\big] \in \mathbf{WD}_C^{v',V'} \Rightarrow \mathbf{WD}_C^{\phi',W'}.$$

Since composition with α preserves **WD** by definition, we also have

$$\alpha \circ C\big[\vec{t}, \mathrm{id}\big] \in \mathbf{WD}_C^{v',V'} \Rightarrow \mathbf{WD}_{\mu Y.\, C}^{\phi,W}.$$

Now we can apply (ii) by using $D = \mu Y.\, C$ to obtain (4.2).

- The case for ν-types is analogous to that of μ-types, only that we use here the induction hypothesis (iii).

(ii) Let C, D be types with $\Theta, Y \Vdash C : \mathbf{Ty}$ and $\Theta \Vdash D : \mathbf{Ty}$, as in the proposition. Put $\phi' = v[D[\phi]/Y]$ and $W' = V\Big[Y \mapsto \mathbf{WD}_D^{\phi,W}\Big]$. We have to show for all terms $r \in \mathbf{WD}_C^{\phi',W'} \Rightarrow \mathbf{WD}_D^{\phi,W}$ and $s \in \mathbf{WD}_{\mu Y.\, C}^{v,V}$ that $R_\mu\ r\ s \in \mathbf{WD}_D^{\phi,W}$. To do so, we define $S \subseteq \Lambda^{=}_{\mu Y.\, C}$

$$S \coloneqq \Big\{ s \in \mathbf{WD}_{\mu Y.\, C}^{v,V} \;\Big|\; R_\mu\ r\ s \in \mathbf{WD}_D^{\phi,W} \Big\}$$

and prove that $\Psi_C^{v,V}(S) \subseteq S$. This gives us by induction that $\mathbf{WD}_{\mu Y.\, C}^{v,V} \subseteq S$ and therefore the desired property.

Let $s \in \Psi_C^{v,V}(S)$. We have to show that $s \in S$, that is, $s \in \mathbf{WD}_{\mu Y.\, C}^{v,V}$ and $R_\mu\ r\ s \in \mathbf{WD}_D^{\phi,W}$. First, we obtain $s \in \mathbf{WD}_{\mu Y.\, C}^{v,V}$ immediately from $\Psi_C^{v,V}(S) \subseteq \Psi_C^{v,V}\big(\mathbf{WD}_{\mu Y.\, C}^{v,V}\big) = \mathbf{WD}_{\mu Y.\, C}^{v,V}$. Moreover, we get a term s' with $s \longrightarrow\!\!\!\!\!\rightarrow \alpha\ s'$ and $s' \in \mathbf{WD}_C^{v[\mu Y.\, C[v]/Y], V[Y \mapsto S]}$. This gives us a reduction

$$R_\mu\ r\ s \longrightarrow\!\!\!\!\!\rightarrow r\ (C[v][R_\mu\ r]\ s').$$

By Lemma 4.1.10, it suffices now to prove that $r\ (C[v][R_\mu\ r]\ s') \in \mathbf{WD}_D^{\phi,W}$. Using the assumption that $r \in \mathbf{WD}_C^{\phi',W'} \Rightarrow \mathbf{WD}_D^{\phi,W}$, we need to show that $C[v][R_\mu\ r]\ s' \in \mathbf{WD}_C^{\phi',W'}$. If we put $v' = v[\mu Y.\, C[v]/Y]$ and $V' = V[Y \mapsto S]$, then this last step follows from (i) by the following four observations:

- $V'(Y) \subseteq \mathbf{ON}_{v'(Y)}$ and $W'(Y) \subseteq \mathbf{ON}_{\phi'(Y)}$;
- $C[v][R_\mu\ r] = C[\overrightarrow{\mathrm{id}}, R_\mu\ r]$ because $v(X)$ is closed for all $X \in \Theta$ and by the first case in the definition of the action of types on terms in Figure 3.3 on page 45;
- $\mathrm{id} \in V'(X) \Rightarrow W'(X)$ for all $X \in \Theta$ by the backwards closure of W; and
- $R_\mu\ r \in V'(Y) \Rightarrow W'(Y)$ by definition of S.

Thus, (i) applies and we obtain $C[v][R_\mu\ r] \in \mathbf{WD}_C^{v',V'} \Rightarrow \mathbf{WD}_C^{\phi',W'}$. Hence, $R_\mu\ r\ s \in \mathbf{WD}_D^{\phi,W}$ and we have $\Psi_C^{v,V}(S) \subseteq S$, as desired.

(iii) The proof of this part uses, dually to the proof of Proposition 4.1.9.(ii), the fact that $\mathbf{WD}_{\nu Y.\,C}^{\phi,W}$ is defined as a largest fixed point and the predicate

$$S \coloneqq \left\{ R_\nu\ r\ s \in \mathbf{SN}_{(\nu Y.\,C)[\phi]} \;\middle|\; s \in \mathbf{WD}_D^{v,V} \right\}. \qquad \square$$

This concludes our study of observational normalisation for now, and we can continue with the discussion of program tests and the induced program equivalence.

4.1.2. Tests and Observational Equivalence

Since we will use tests for both calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$, we first give a definition of tests that is parameterised in the arguments that can be used for observations on functions.

Definition 4.1.12. Let $U = \{U_A\}_{A \in \mathrm{Ty}}$ be any family of sets indexed by types. For a type A, we say that φ is a *test (formula) on* A (over U), if $\varphi :\downarrow A$ can be inductively derived from the following rules.

$$\frac{}{\top :\downarrow A} \qquad \frac{}{\bot :\downarrow A} \qquad \frac{\varphi_1 :\downarrow A_1 \qquad \varphi_2 :\downarrow A_2}{[\varphi_1, \varphi_2] :\downarrow A_1 + A_2} \qquad \frac{\varphi :\downarrow A[\mu X.\,A/X]}{\left[\alpha^{-1}\right] \varphi :\downarrow \mu X.\,A}$$

$$\frac{\varphi :\downarrow A[\nu X.\,A/X]}{[\xi]\,\varphi :\downarrow \nu X.\,XA} \qquad \frac{\varphi :\downarrow A_i}{[\pi_i]\,\varphi :\downarrow A_1 \times A_2} \qquad \frac{\varphi :\downarrow B \qquad v \in U_A}{[v]\,\varphi :\downarrow A \to B}$$

The set of all test formulas on A over U is denoted by

$$\mathrm{Tests}(U)_A \coloneqq \{\varphi \mid \varphi :\downarrow A\},$$

and the family of all tests over U consequently by $\mathrm{Tests}(U)$. ◀

The tests for the calculus $\lambda\mu\nu$ are easy to give, as termination is not an issue in $\lambda\mu\nu$. Thus, we can use any term as function argument, that is, any well-formed term in Λ (Definition 3.1.4).

Definition 4.1.13. The tests for $\lambda\mu\nu$ are given by $\mathrm{Tests}(\lambda\mu\nu) \coloneqq \mathrm{Tests}(\Lambda)$.

Defining tests on terms of the calculus $\lambda\mu\nu{=}$ is a bit more complicated, as we need to restrict the arguments used in tests on function types to observationally normalising terms. Thus, we need to define observationally normalising terms. The first step towards this is to interpret tests as terms in $\lambda\mu\nu{=}$. Note that there is a type of Boolean values, which is given by

$$\mathrm{Bool} \coloneqq \mathbf{1} + \mathbf{1}.$$

$$
\begin{aligned}
&[\![\top]\!]_A && = \lambda x.\mathrm{true} && \text{for all } A \in \mathrm{Ty}\\
&[\![\bot]\!]_A && = \lambda x.\mathrm{false} && \text{for all } A \in \mathrm{Ty}\\
&[\![[\varphi_1, \varphi_2]]\!]_{A_1 + A_2} && = \lambda\{\kappa_1\ x \mapsto [\![\varphi_1]\!]_{A_1}\ x\ ;\ \kappa_2\ x \mapsto [\![\varphi_2]\!]_{A_2}\ x\}\\
&[\![\left[\alpha^{-1}\right]\varphi]\!]_{\mu X.\,A} && = \lambda\{\alpha\ x \mapsto [\![\varphi]\!]_{A[\mu X.\,A/X]}\ x\}\\
&[\![[\xi]\,\varphi]\!]_{\nu X.\,A} && = \lambda x.[\![\varphi]\!]_{A[\nu X.\,A/X]}(\xi\ x)\\
&[\![[\pi_i]\,\varphi]\!]_{A_1 \times A_2} && = \lambda x.[\![\varphi]\!]_{A_i}(\pi_i\ x)\\
&[\![[v]\,\varphi]\!]_{A \to B} && = \lambda f.[\![\varphi]\!]_B(f\ v)
\end{aligned}
$$

Figure 4.1.: Interpretation of tests as terms

This type has the expected truth constants (considered as terms in $\lambda\mu\nu{=}$):

$$\mathrm{true} \coloneqq \kappa_1\ \langle\rangle \qquad \text{and} \qquad \mathrm{false} \coloneqq \kappa_2\ \langle\rangle.$$

The interpretation of tests as well-covering (Definition 3.2.29) terms in $\lambda\mu\nu{=}$ is then given as follows.

Definition 4.1.14. Let U be a set family indexed by types, such that $U \sqsubseteq \Lambda^{=}$, that is, $U_A \subseteq \Lambda^{=}(A)$ for all $A \in \mathrm{Ty}$.[18] The *interpretation of tests*[19] on a type A as terms in $\lambda\mu\nu{=}$ is given by the map

$$[\![-]\!]_A \colon \mathrm{Tests}(U)_A \to \Lambda^{=}(A \to \mathrm{Bool}),$$

which is defined inductively in Figure 4.1. ◂

Having defined observationally normalising terms, we are now in the position to define tests for the copattern calculus.

Definition 4.1.15 (Tests for $\lambda\mu\nu{=}$)**.** The collection of tests for $\lambda\mu\nu{=}$ is given by

$$\mathrm{Tests}(\lambda\mu\nu{=}) \coloneqq \mathrm{Tests}(\mathbf{ON})$$ ◂

What makes observationally normalising interesting is that any observation on them is strongly normalising. This is made precise in Theorem 4.1.17 below. To prove that theorem, we need the following lemma.

Lemma 4.1.16. *Let A be a type with $\Theta \Vdash A : \mathbf{Ty}$, v a substitution and $V \in I_{\Theta,v}$ a valuation with $V(X) \subseteq \mathbf{ON}_{v(X)}$. Then $\mathbf{WD}_A^{v,V} \subseteq \mathbf{ON}_{A[v]}$.*

Proof. Straightforward by induction on A. □

Theorem 4.1.17. *Let $A \in \mathrm{Ty}$ and $t \in \mathbf{ON}_A$. Then for any test φ on A, $[\![\varphi]\!]\ t$ is strongly normalising.*

Proof. Towards the proof of this lemma, we need to generalise the statement, similar to Lemma 4.1.5, as follows. Let A be a type with $\Theta \Vdash A : \mathbf{Ty}$, v a substitution and $V \in I_{\Theta,v}$ a valuation with $V(X) \subseteq \mathbf{ON}_{v(X)}$. We now prove that if $t \in \mathbf{WD}_A^{v,V}$, then for all $\varphi :\downarrow A[v]$ we have $([\![\varphi]\!]\ t) \in \mathbf{SN}_{\mathrm{Bool}}$ by induction on φ. The result follows for closed types by definition of **ON**.

To prove this statement, we observe that for any test φ, the interpretation $[\![\varphi]\!]$ is a neutral term, that is, there is no reduction originating at $[\![\varphi]\!]$. This has the consequence that if $[\![\varphi]\!]\ t \longrightarrow s$, then

either $s = [\![\varphi]\!]\, t'$ with $t \longrightarrow t'$, or $[\![\varphi]\!]\, t \succ s$. Since $t \in \mathbf{SN}$ by Lemma 4.1.5, we thus have that any reduction sequence of $[\![\varphi]\!]\, t$ towards a normal form s is of the form

$$[\![\varphi]\!]\, t \longrightarrow\!\!\!\!\!\rightarrow [\![\varphi]\!]\, t' \succ s' \longrightarrow\!\!\!\!\!\rightarrow s.$$

Using this property, all cases of the induction are straightforward, except for the case of least fixed point types. Thus, we focus on this case: Suppose we are given a test $\left[\alpha^{-1}\right] \varphi$ with $\varphi :\downarrow A[\mu X.\, A/X]$. We define $S \subseteq \Lambda^{=}_{\mu X.\, A}$ by

$$S = \left\{ t \in \mathbf{WD}^{v,V}_{\mu X.\, A} \;\middle|\; [\![\left[\alpha^{-1}\right] \varphi]\!]\, t \in \mathbf{SN} \right\},$$

and show $\mathbf{WD}^{v,V}_{\mu X.\, A} \subseteq S$. To prove this, we use the fixed point property of $\mathbf{WD}^{v,V}_{\mu X.\, A}$. Thus, we have to show $\Psi^{v,V}_{A}(S) \subseteq S$. To this end, let $t \in \Psi^{v,V}_{A}(S)$, leaving us with having to prove that $[\![\left[\alpha^{-1}\right] \varphi]\!]\, t \in \mathbf{SN}$. By the property we mentioned in the beginning, we have

$$[\![\left[\alpha^{-1}\right] \varphi]\!]\, t \longrightarrow\!\!\!\!\!\rightarrow [\![\left[\alpha^{-1}\right] \varphi]\!]\, (\alpha\, t') \succ [\![\varphi]\!]\, t'. \tag{4.3}$$

Recall that

$$\Psi^{v,V}_{A}(S) = \left\{ s \;\middle|\; (\exists s'.\, s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \wedge \left(\forall s'.\, (s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \implies s' \in \mathbf{WD}^{v[\mu X.\, A/X], V[X \mapsto S]}_{A} \right) \right\}.$$

From this and (4.3), we obtain $t' \in \mathbf{WD}^{v[\mu X.\, A/X], V[X \mapsto S]}_{A}$. By Lemma 4.1.16, we can apply the induction hypothesis and get $[\![\varphi]\!]\, t' \in \mathbf{SN}$. Thus, $[\![\left[\alpha^{-1}\right] \varphi]\!]\, t \in \mathbf{SN}$ by the above-mentioned property. Since this holds for any $t \in S$, we have $\Psi^{v,V}_{A}(S) \subseteq S$. □

Let us illustrate how to use Theorem 4.1.17 to show that a certain stream term is not observationally normalising. This shows that we cannot compute a stream from that term, thereby showing that the stream term is not productive.

Example 4.1.18. Let x and even be given by

$$\begin{array}{ll} x : \mathrm{Nat}^{\omega} & \mathrm{even} : \mathrm{Nat}^{\omega} \to \mathrm{Nat}^{\omega} \\ x.\mathrm{hd} = 1 & (\mathrm{even}\, s).\mathrm{hd} = \mathrm{hd}\, s \\ x.\mathrm{tl} = \mathrm{even}\, x & (\mathrm{even}\, s).\mathrm{tl} = \mathrm{even}\, (s.\mathrm{tl}.\mathrm{tl}) \end{array} \tag{4.4}$$

To see that x is not observationally normalising, we apply the evaluation context $e = \cdot.\mathrm{tl}.\mathrm{tl}.\mathrm{hd}$ to x and find that

$$e[x] = x.\mathrm{tl}.\mathrm{tl}.\mathrm{hd} \longrightarrow (\mathrm{even}\, x).\mathrm{tl}.\mathrm{hd} \longrightarrow (\mathrm{even}\, (x.\mathrm{tl}.\mathrm{tl})).\mathrm{hd} \longrightarrow x.\mathrm{tl}.\mathrm{tl}.\mathrm{hd} = e[x].$$

Hence, there is a diverging reduction sequence starting at $e[x]$, from which we get by Theorem 4.1.17 that x is not in **ON** by using the test [tl] [tl] [hd] $\top$, where the modalities [hd] and [tl] are defined in the obvious way.

Recall that we defined the interpretation of stream terms as functions $x^{\dagger} : \mathbb{N} \to \mathbb{N}$ iteratively by $x^{\dagger}(0) = n$ for $x.\mathrm{hd} \equiv \underline{n}$ and $x^{\dagger}(k+1) = (x.\mathrm{tl})^{\dagger}(k)$. Note that the divergence of $e[x]$ implies that there is no natural number $n \in \mathbb{N}$, such that $e[x] \equiv \underline{n}$. This in turn means that we cannot produce the value of $x^{\dagger}(2)$, hence we cannot construct $x^{\dagger}$ by successively computing head and tail of x. We say that x is not *productive*, see [End+10; EH11]. If we interpret (4.4) as a system of stream differential equations, then there is no unique solution for the variable x in (4.4), cf. [HKR17, Sec. 8.4]. ◀

Let us now evaluate tests on terms. We do this by introducing a satisfaction relation analogous to the evaluation of formulas of modal logics over Kripke models, see [BRV01; BvB07].

Definition 4.1.19. Let t be a term in $\Lambda(A)$ or $\Lambda^{=}(A)$ and $\varphi \in \mathrm{Tests}(U)_A$ be a test on the type $A \in \mathrm{Ty}$ with $U = \Lambda$ or $U = \mathbf{ON}$, respectively. We define $t \vDash_A \varphi$, which is to be read as t *satisfies* φ, by induction on φ as follows. Here, we write tt and ff for the Boolean constants for true and false, so that $t \vDash_A \varphi$ holds iff $t \vDash_A \varphi = \mathrm{tt}$ by the definition below. In the case of $\lambda\mu\nu{=}$, we use the notations $\pi_1\, t \coloneqq t.\mathrm{pr}_1$, $\pi_2\, t \coloneqq t.\mathrm{pr}_2$ and $\xi\, t \coloneqq t.\mathrm{out}$, see Section 3.3.

$$
\begin{array}{ll}
t \vDash_A \top & \coloneqq \mathrm{tt} \\
t \vDash_A \bot & \coloneqq \mathrm{ff} \\
t \vDash_{A_1 + A_2} [\varphi_1, \varphi_2] & \coloneqq \exists t'.\ t \equiv \kappa_i\, t' \text{ and } t' \vDash_{A_i} \varphi_i \\
t \vDash_{\mu X.\, B} \left[\alpha^{-1}\right] \varphi & \coloneqq \exists t'.\ t \equiv \alpha\, t' \text{ and } t' \vDash_{B[\mu X.\, B/X]} \varphi \\
t \vDash_{\nu X.\, B} [\xi]\, \varphi & \coloneqq \xi\, t \vDash_{B[\nu X.\, B/X]} \varphi \\
t \vDash_{A_1 \times A_2} [\pi_i]\, \varphi & \coloneqq \pi_i\, t \vDash_{A_i} \varphi \\
t \vDash_{B \to C} [v]\, \varphi & \coloneqq t\, v \vDash_C \varphi.
\end{array}
$$

Two terms t_1, t_2 in $\Lambda(A)$ or $\Lambda^{=}(A)$ are *observationally equivalent*, written $t_1 \equiv^A_{\mathrm{obs}} t_2$, if they satisfy the same tests:

$$t_1 \equiv^A_{\mathrm{obs}} t_2 \text{ iff } \forall \varphi :\downarrow A.\, t_1 \vDash_A \varphi \Leftrightarrow t_2 \vDash_A \varphi.$$

Let U be either Λ or $\mathbf{ON}$. We say that two *open* terms t_1, t_2 in $\lambda\mu\nu$ or, respectively, in $\lambda\mu\nu{=}$ with $\Gamma \vdash t_1, t_2 : A$ are observationally equivalent, if for all U-Γ-closing σ we have that $t_1[\sigma] \equiv^A_{\mathrm{obs}} t_2[\sigma]$ and denote this by $\Gamma \vdash t_1 \equiv^A_{\mathrm{obs}} t_2$. ◂

In what follows, we will frequently omit the type sub- and superscripts and simply write $\vDash$, $[\![-]\!]$ and $\equiv_{\mathrm{obs}}$ whenever the typing can be inferred from the context.

Let us now demonstrate how tests can be used to distinguish the behaviour of terms.

Example 4.1.20. An example of a pair of terms that are, rightfully, distinguished by observational equivalence is $H\, 0^\omega : \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega$ and $H\, 1^\omega : \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega$, see Example 3.2.9. To see this, we use tests $\varphi_{=n} :\downarrow \mathrm{Nat}$, for testing equality to $n \in \mathbb{N}$, that are given inductively by

$$
\begin{aligned}
\varphi_{=0} &\coloneqq \left[\alpha^{-1}\right] [\top, \bot] && :\downarrow \mathrm{Nat} \\
\varphi_{=n+1} &\coloneqq \left[\alpha^{-1}\right] [\bot, \varphi_{=n}] && :\downarrow \mathrm{Nat}.
\end{aligned}
$$

We can then use the formula

$$\psi \coloneqq [\mathrm{alt}]\, [\mathrm{hd}]\, \varphi_{=1} \quad :\downarrow \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega$$

to distinguish $H\, 0^\omega$ and $H\, 1^\omega$, where alt is the alternating bit stream defined in Example 3.2.18. Recall also from the same example that $(H\, 0^\omega\ \mathrm{alt}).\mathrm{hd} \equiv \underline{0}$ and $(H\, 1^\omega\ \mathrm{alt}).\mathrm{hd} \equiv \underline{1}$. This allows us to

show that $H\,0^\omega$ does not satisfy ψ, while $H\,1^\omega$ does satisfy ψ, as we have

$$\begin{aligned}
H\,0^\omega \vDash [\mathrm{alt}]\,[\mathrm{hd}]\,\varphi_{=1} &\iff H\,0^\omega\,\mathrm{alt} \vDash [\mathrm{hd}]\,\varphi_{=1} \\
&\iff (H\,0^\omega\,\mathrm{alt}).\mathrm{hd} \vDash \varphi_{=1} \\
&\iff \underline{0} \vDash \left[\alpha^{-1}\right]\,[\bot, \varphi_{=0}] && \text{by } (H\,0^\omega\,\mathrm{alt}).\mathrm{hd} \equiv \underline{0} \\
&\iff \kappa_1\,\langle\rangle \vDash [\bot, \varphi_{=0}] \\
&\iff \langle\rangle \vDash \bot
\end{aligned}$$

and

$$\begin{aligned}
H\,1^\omega \vDash [\mathrm{alt}]\,[\mathrm{hd}]\,\varphi_{=1} &\iff \cdots \iff \underline{1} \vDash \varphi_{=1} \iff \cdots && \text{by } (H\,1^\omega\,\mathrm{alt}).\mathrm{hd} \equiv \underline{1} \\
&\iff \langle\rangle \vDash \top.
\end{aligned}$$

So the terms $H\,0^\omega$ and $H\,1^\omega$ are distinguished by the test ψ. ◂

After having seen an example of how to distinguish terms, let us now consider an example where we actually prove that two terms are observationally equivalent. This is typically done by induction on tests.

Example 4.1.21. Recall that we gave in Example 3.2.7 another unit type $\mathbf{1}' = \nu X.\, X$ with a canonical inhabitant $\langle\rangle'$. We now show that this term is actually unique up to observational equivalence. This requires us to show for all terms $t : \mathbf{1}'$ that $t \equiv_{\mathrm{obs}} \langle\rangle'$, which means by definition of observational equivalence that we have to show for all tests $\varphi \in \mathrm{Tests}(\lambda\mu\nu{=})_{\mathbf{1}'}$ that $t \vDash \varphi$ iff $\langle\rangle' \vDash \varphi$.

We proceed by induction on φ to prove for all terms $t : \mathbf{1}'$ that $t \vDash \varphi$ iff $\langle\rangle' \vDash \varphi$. Note that the term t is universally quantified in the induction hypothesis. Let $t : \mathbf{1}'$ be a term. In the base case, where either $\varphi = \top$ or $\varphi = \bot$, we trivially have $t \vDash \varphi \iff \langle\rangle' \vDash \varphi$. For the induction step, φ is of the form $\varphi = [\xi]\,\psi$ with $\psi \in \mathrm{Tests}(\lambda\mu\nu{=})_{\mathbf{1}'}$. Since $\langle\rangle'.\mathrm{out} \equiv \langle\rangle'$, see the opening discussion of Section 3.2.2, $\langle\rangle' \vDash [\xi]\,\psi \iff \langle\rangle' \vDash \psi$. Applying the induction hypothesis to ψ, we have $t.\mathrm{out} \vDash \psi \iff \langle\rangle' \vDash \psi$. Thus, by definition of the satisfaction relation, we have $t \vDash [\xi]\,\psi \iff \langle\rangle' \vDash [\xi]\,\psi$, hence $t \vDash \varphi \iff \langle\rangle' \vDash \varphi$. So by induction, t and $\langle\rangle'$ satisfy the same tests φ, thus are observationally equivalent. ◂

At this point, a reader versed in the area of transition systems, coalgebras and such might wonder about the relation of observational equivalence to bisimilarity. Indeed, it would be natural to view $\langle\rangle'$ as a single state transition system and prove that any other term of type $\mathbf{1}'$ is bisimilar to that system. The hope would then be that such a proof corresponds to showing that the term is observationally equivalent to $\langle\rangle'$. In Section 5.1, we will show that one can define a general notion of bisimilarity on terms of all inductive-coinductive types, which coincides with observational equivalence.

Another point of interest might be whether it is possible to automatically prove that two terms are observationally equivalent. In fact, if there are no terms of function type involved, like in Example 4.1.21, then observational equivalence is decidable. One might call the function-free fragment of $\lambda\mu\nu{=}$ the *data fragment*, as we cannot carry out any computations, thus only construct data, in that fragment. However, if we include functions, then observational equivalence becomes undecidable. We discuss this in detail in Section 5.3.

The following example illustrates a, perhaps surprising, effect of the restriction to observationally normalising function arguments in the definition of observational normalisation itself: There is a well-defined function from the empty type $\mathbf{0}$ to any type A.

Example 4.1.22. Recall the empty type from Example 3.1.2 was given by $\mathbf{0} = \mu X.\, X$. We can give for each type A a function $E^{\mathbf{0}}_A \colon \mathbf{0} \to A$ from $\mathbf{0}$ into A by[20]

$$\begin{aligned} E^{\mathbf{0}}_A \coloneqq\ & \mathbf{rlet} \\ & f : \mathbf{0} \to A = \left\{ \cdot\, (\alpha\, x) \mapsto f\, x \right\} \\ & \mathbf{in}\ f. \end{aligned}$$

Note that there is no observationally normalising inhabitant of $\mathbf{0}$. This can be proved either by contradiction to Lemma 4.1.5, or directly by induction. Since the latter is more elegant, let us prove that $\mathbf{ON_0} = \emptyset$ by induction. To this end, recall that $\mathbf{ON_0} = \mu U.\, \Psi_X^{(),\varepsilon}(U)$. Thus, we can prove $\mathbf{ON_0} = \emptyset$ by showing $\Psi_X^{(),\varepsilon}(\emptyset) \subseteq \emptyset$. Suppose now that $s \in \Psi_X^{(),\varepsilon}(\emptyset)$, thus there is an s' with $s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s'$ and $s' \in \emptyset$. This means, however, that there is no such s' and thus also no such s. Hence, $\Psi_X^{(),\varepsilon}(\emptyset)$ and $\mathbf{ON_0}$ are empty, as claimed. Since $\mathbf{ON_0}$ is empty and there are no reduction possible on $E^{\mathbf{0}}_A$, we have that $E^{\mathbf{0}}_A \in \mathbf{ON}_{\mathbf{0}\to A}$. ◂

Recall that we have defined an interpretation $[\![-]\!]$ of tests as terms of type $A \to \mathrm{Bool}$ in $\lambda\mu\nu{=}$. One might reasonably expect that this interpretation coincides with the satisfaction relation, in the sense that for all tests and terms $[\![\varphi]\!] \equiv \mathrm{true}$ iff $t \vDash \varphi$. However, there is a subtle difference on non-terminating terms: Let A be a type and put $\varphi = \left[\alpha^{-1}\right] \top$, so that φ is a test on $\mu X.\, A$. Recall that we defined in Example 3.2.21 a term $\Omega_{\mu X.\, A}$ that only admits reduction steps of the form $\Omega_{\mu X.\, A} \longrightarrow \Omega_{\mu X.\, A}$. This means then that neither $[\![\varphi]\!](\Omega_{\mu X.\, A}) \equiv \mathrm{true}$ nor $[\![\varphi]\!](\Omega_{\mu X.\, A}) \equiv \mathrm{false}$. On the other hand, we have $\Omega_{\mu X.\, A} \vDash \varphi \iff \exists t'.\, \Omega_{\mu X.\, A} \alpha\, t'$ and $t' \vDash \top$. Since no such t' exists, we conclude that $\Omega_{\mu X.\, A} \nvDash \varphi$. So the difference between the evaluation of $[\![\varphi]\!](t)$ and $t \vDash \varphi$ lies in the fact that the former requires that there must be a way to reduce $[\![\varphi]\!](t)$ to true or false. We capture this situation in the following proposition.

Proposition 4.1.23. *If $t \in \Lambda^{=}(A)$ and $\varphi :\downarrow A$, such that $[\![\varphi]\!](t)$ has a normal form, then*

$$[\![\varphi]\!](t) \equiv \mathrm{true} \quad \textit{iff} \quad t \vDash \varphi.$$

Proof. This follows by an easy induction on φ. □

To put the above discussion into broader terms, if we accept the principle of excluded middle, then for all tests $\varphi :\downarrow A$

$$\forall t \in \Lambda^{=}(A).\, t \vDash \varphi \vee t \nvDash \varphi.$$

The example above shows however that in general $\neg(\forall t \in \Lambda^{=}(A).\, [\![\varphi]\!](t) \equiv \mathrm{true} \vee [\![\varphi]\!](t) \equiv \mathrm{false})$. More generally, we can evaluate tests by using $[\![-]\!]$ only on observationally normalising terms, whereas on non-normalising terms $[\![-]\!]$ cannot assign a definite truth value to a test.

We finish this section by collecting some properties of observational equivalence. First, we show that the definition of observational equivalence of open terms makes sense.

Lemma 4.1.24. *For all $x : A \vdash t_i : B$ with $i = 1, 2$, we have both in $\lambda\mu\nu$ and $\lambda\mu\nu{=}$*

$$x : A \vdash t_1 \equiv^{B}_{\mathrm{obs}} t_2 \Leftrightarrow \lambda x.t_1 \equiv^{A\to B}_{\mathrm{obs}} \lambda x.t_2 \tag{4.5}$$

Proof. Since we have a common notation for function application and abstraction in both calculi, we give the proof independent of the calculus at hand. Let us write $r_i = \lambda x.t_i$.

'⇒' We need to show that r_1 and r_2 satisfy the same tests φ. If $\varphi = \top$ or $\varphi = \bot$, this is trivial. If $\varphi = [v]\,\psi$, we have by assumption

$$r_1 \vDash \varphi \iff t_1[v/x] \vDash \psi \iff t_2[v/x] \vDash \psi \iff r_2 \vDash \varphi$$

'⇐' Let U be either Λ or **ON**. To show that $t_1 \equiv_{\mathrm{obs}} t_2$, we need to show that $t_1[\sigma] \equiv^B_{\mathrm{obs}} t_2[\sigma]$ for every U-$(x : A)$-closing substitution σ. That is to say we need to show $t_1[v/x] \equiv^B_{\mathrm{obs}} t_2[v/x]$ for every $v \in U$. But this follows immediately, as for every test ψ on B, we have

$$t_1[v/x] \vDash \psi \iff r_1 \vDash [v]\,\psi \iff r_2 \vDash [v]\,\psi \iff t_2[v/x] \vDash \psi,$$

just as for the other direction of the lemma. □

The next lemma states a number of properties of that we will need frequently.

Lemma 4.1.25. *Observational equivalence $\equiv_{\mathrm{obs}}$ has the following properties.*

(i) Substitutivity*: Given terms $r \in \mathbf{ON}^{x:A}_B$ and $t_1, t_2 \in \mathbf{ON}_A$ such that $t_1 \equiv^A_{\mathrm{obs}} t_2$, we have that $r[t_1/x] \equiv^B_{\mathrm{obs}} r[t_2/x]$.*

(ii) $\equiv_{\mathrm{obs}}$ *is a* congruence *on* **ON***, that is, $\equiv_{\mathrm{obs}}$ is an equivalence relation on* **ON** *and fulfils substitutivity as in (i), see [Pit04, Def. 7.5.1].*

(iii) If $t_1 \equiv^A_{\mathrm{obs}} t_2$, then for all f with $\vdash f : A \to B$ we have $f\,t_1 \equiv^B_{\mathrm{obs}} f\,t_2$.

(iv) $\equiv_{\mathrm{obs}}$ *strictly contains convertibility (i.e., $\equiv \subset \equiv_{\mathrm{obs}}$).*

(v) $\equiv_{\mathrm{obs}}$ *is extensional for terms of function type: if $t_1, t_2 : A \to B$ and $t_1\,v \equiv_{\mathrm{obs}} t_2\,v$ for all $v : A$ in* **ON***, then $t_1 \equiv_{\mathrm{obs}} t_2$.*

(vi) $\equiv_{\mathrm{obs}}$ *contains η-equivalence: $\lambda x.tx \equiv_{\mathrm{obs}} t$, $x \notin \mathrm{fv}(t)$.*

Most of these properties are straightforward to prove, only substitutivity (i) requires a bit more work. The proof uses the following technical lemma.

Lemma 4.1.26. *Let $s \in \mathbf{ON}^{x:A}_B$ and let $t \in \mathbf{ON}_A$. For all tests $\varphi :\downarrow B$ with $s[t/x] \vDash \varphi$ there are tests $\psi_1, \ldots, \psi_n$ on A such that*

(i) $t \vDash \psi_i$ for all $1 \le i \le n$ and

(ii) for all $t' \in \mathbf{ON}_A$, if $t' \vDash \psi_i$ for all $1 \le i \le n$, then $s[t'/x] \vDash \varphi$.

This lemma states that we can find for each computation that s can make, formulas that characterise inputs to s that lead to the same output as t. One can read this as a continuity property of s, in the sense that finite observations on $s[t/x]$ only depend on finite observations on t. We will not go into this further here, but it is discussed in [BH16].

Proof of Lemma 4.1.26. To prove the statement of the lemma, we need to generalise it a bit. This generalisation becomes easier, if we use *conjunctive tests*, that is, tests that may contain conjunctions. Given tests $\varphi_1, \varphi_2 : A$, their conjunction $\varphi_1 \wedge \varphi_2$ is satisfied by $t : A$ if $t \vDash \varphi_i$ for both $i = 1$ and $i = 2$. Conjunctive tests allow us to simplify the formulation of Lemma 4.1.26 by replacing $\psi_1, \dots, \psi_n$ by $\psi = \psi_1 \wedge \cdots \wedge \psi_n$.

More generally, we set out to prove the following. Let $f \in \mathbf{ON}_A^\Gamma$ and let $\tau \in \mathrm{Subst}(\mathbf{ON}, \Gamma)$. For all tests $\varphi :\downarrow A$ with $f[\tau] \vDash \varphi$ there are conjunctive tests $\{\psi_x\}_{x \in \mathrm{dom}(\Gamma)}$ such that

(i) $\tau(x) \vDash \psi_x$ for all $x \in \mathrm{dom}(\Gamma)$ and

(ii) for all $\sigma \in \mathrm{Subst}(\mathbf{ON}, \Gamma)$, if $\sigma(x) \vDash \psi_x$ for all $x \in \mathrm{dom}(\Gamma)$, then $f[\sigma] \vDash \varphi$.

We only sketch the proof of this generalised property. Since f is in $\mathbf{ON}_A^\Gamma$, $[\![\varphi]\!](f[\tau])$ is strongly normalising, thus there is a finite reduction sequence $[\![\varphi]\!](f[\tau]) \longrightarrow\!\!\!\!\!\rightarrow N$ to a normal form. We obtain the family $\{\psi_x\}$ by induction on this reduction sequence. In this induction, two cases have to be distinguished: either $\tau(x)$ is contracted within an evaluation context e, in which case ψ_x is extended by the modalities given by e, or $\tau(x)$ is used as a function argument in a contraction. In the latter case, we reduce the corresponding function to λD or a symbol g, and extend ψ_x by the pattern of D or g that matched $\tau(x)$.[21] □

Proof of Lemma 4.1.25. (i) To show that $r[t_1/x] \equiv_{\mathrm{obs}} r[t_2/x]$, we show that both terms simultaneously satisfy any test $\varphi : B$.

First, assume that $r[t_1/x] \vDash \varphi$. Lemma 4.1.26 gives us that for $\tau = [t_1/x]$ there are tests $\psi_1, \dots, \psi_n$ on A, such that $\forall 1 \le i \le n.\, t_1 \vDash \psi_i$ and if $\forall 1 \le i \le n.\, t_2 \vDash \psi_i$, then $r[t_2/x] \vDash \varphi$. Since $t_1 \equiv_{\mathrm{obs}} t_2$, both terms simultaneously satisfy all the ψ_i, thus $r[t_2/x] \vDash \varphi$.

Conversely, $r[t_2/x] \vDash \varphi$ implies, in the same way, that $r[t_1/x] \vDash \varphi$. Summarising, the terms $r[t_1/x]$ and $r[t_2/x]$ satisfy the same tests, hence are observationally equivalent.

(ii) This follows immediately from $\equiv$ being an equivalence relation and from (i).

(iii) This is just the combination of (i) and Lemma 4.1.24.

(iv) The inclusion is proved by induction on tests. It is strict by appealing to item (v): For instance, the terms $\lambda x.\, x$ and $\lambda\{\cdot\ (\alpha\ x) \mapsto x\}$ are not convertible but they are observationally equivalent by extensionality.

(v) Trivially by the shape of tests on $A \to B$.

(vi) Let $t\colon A \to B$ be a term with $x \notin \mathrm{fv}(t)$. For every $v \in \mathbf{ON}_A$, we have $(\lambda x.(t\,x))\,v \equiv t\,v$ and hence by (iv) and (v) the equivalence $\lambda x.(t\,x) \equiv_{\mathrm{obs}} t$ follows. □

The last property of $\equiv_{\mathrm{obs}}$ we record is that the equational theory $\equiv_{\mathrm{obs}}$ is consistent.

Definition 4.1.27 (cf. [Bar85, Def. 2.1.30])**.** An *equational theory* for a family $\{U_A\}_{A \in \mathrm{Ty}}$ is a family $\approx$ of equivalence relations $\approx_A \subseteq U_A \times U_A$. The equational theory is said to be *consistent*, if there is $A \in \mathrm{Ty}$ and $s, t \in U_A$ with $s \not\approx_A t$.

Proposition 4.1.28. *The equational theory $\equiv_{\mathrm{obs}}$ is consistent both for Λ and $\Lambda^=$.*

Proof. The terms $\mathrm{true}, \mathrm{false} : \mathrm{Bool}$ are not observationally equivalent, as they are distinguishable by the test $[\top, \bot]$. Hence $\equiv_{\mathrm{obs}}$ is consistent. □

4.2. Category Theoretical Properties of $\lambda\mu\nu$ and $\lambda\mu\nu=$

We have now seen two different calculi for programming with mixed inductive-coinductive types, and we introduced a notion of program equivalence that is based on program observations for these calculi. Moreover, we defined what it means for a program in the calculus $\lambda\mu\nu=$ to be observationally normalising, using again the same notion of program observation. Basing program equivalence and observational normalisation on observations is an intuitive approach, but we need to show how these notions relate to properties that occur elsewhere in the description of program behaviour. Specifically, we want to relate here to the usual category theoretical definition of inductive-coinductive objects in terms of unique mapping properties. We proceed in two steps to clarify the relation. First, we introduce in Section 4.2.1 the so-called classifying category for each of the calculi. Then we show that the classifying categories indeed admit to some extent the categorical structures that are suggested by the types. For instance, greatest fixed point types are weakly final coalgebras.

The problem with classifying categories is that we are trying to force the calculi into a cloak that does not really fit them. Just to name one major issue: the action of types on terms, given in Definition 3.1.9, looks like the definition of a functor on the classifying categories but it really is not. Thus we are not even able to, for instance, form a category of coalgebras for open types.

We overcome these issues in a second step, where we refine the classifying categories by (1) restricting the attention to only observationally normalising terms in the classifying category for $\lambda\mu\nu=$, and (2) augmenting the classifying categories with observational equivalence as 2-categorical structure. The first step is important to properly classify inductive types (sums and least fixed points), since non-termination leads to inhabitants of inductive types on which no observation is possible, see the discussion before Proposition 4.1.23. Through the second step we obtain a precise categorical classification of the types, in the sense that the action of a type on terms turns out to be a pseudo-functor and largest fixed point types are pseudo-final coalgebras.

One might now raise the objection to the 2-categorical approach that we could just take a quotient of the classifying 2-categories, which allows us to drop the prefix "pseudo" everywhere. However, besides violating the abstract "principle of equivalence" [nLa16], we also lose some information by forming a quotient category: we cannot distinguish anymore between term equivalences arising from computations and those stemming from observational equivalence.

4.2.1. Simple Classifying Categories

The goal of this section is to describe the categorical structure that arises from the type structure of the calculi $\lambda\mu\nu$ and $\lambda\mu\nu=$. We begin by identifying properties of calculi for which we are able to give classifying categories[22], see Definition 4.2.2.

Definition 4.2.1. Let $\mathcal{U}$ be a set. We assume that there is a judgement of the form

$$x : A \vdash t : B,$$

which is a relation between contexts $x : A$ and elements $t \in \mathcal{U}$, where A and B are types as in Definition 3.1.1. The set $\mathcal{U}$ together with the relation $\vdash$ is called a *classifiable calculus*, if

1. Projection holds: TeVar $\subseteq \mathcal{U}$, and for all $x \in$ TeVar, the judgement $x : A \vdash x : A$ is valid;
2. Substitution exists: for every $t, s \in \mathcal{U}$ and $x \in$ TeVar, there is an element $t[s/x] \in \mathcal{U}$;

3. Substitution preserves typing: if $x : B \vdash t : C$ and $x : A \vdash s : B$, then $x : A \vdash t[s/x] : C$;
4. Substitution preserves identities: For all $t, s \in \mathcal{U}$ and $x \in \mathrm{TeVar}$, $x[t/x] = t$ and $t[x/x] = t$;
5. Substitution lemma holds: For all $t, s \in \mathcal{U}$ and $x \in \mathrm{TeVar}$,

$$t[s[r/x]/x] = t[s/x][r/x]. \tag{4.6}$$

Given a classifiable calculus, we will refer to the elements of $\mathcal{U}$ as *terms*. ◀

A classifiable calculus allows us, as the name suggests, to construct its classifying category.[23]

Definition 4.2.2. Let $\mathcal{U}$ be a classifiable calculus. We denote by $\mathcal{C}\ell_s(\mathcal{U})$ the *simple classifying category* of $\mathcal{U}$, given as follows.

$$\mathcal{C}\ell_s(\mathcal{U}) = \begin{cases} \text{objects:} & \text{Types } A \\ \text{morphisms:} & t : A \to B \text{ is a term } t \in \mathcal{U} \text{ with } x : A \vdash t : B \end{cases}$$

An identity morphism $A \to A$ is given by $x : A \vdash x : A$ and composition is defined by substitution: $t \circ s = t[s/x]$. That the composition is well-defined follows from type preservation. The identity law is immediate by $t[x/x] = t$ and $x[t/x] = t$, whereas associativity is given by (4.6). ◀

Lemma 4.2.3. *Both Λ and $\Lambda^=$ are classifiable, thus their respective simple classifying categories $\mathcal{C}\ell_s(\Lambda)$ and $\mathcal{C}\ell_s(\Lambda^=)$ exist.*

Proof. The judgement $\Gamma, x : A \vdash x : A$ is immediate by projection in both calculi. Equation (4.6) is proved by a simple but tedious induction on terms, just like the other conditions. □

This description of the calculi allows us to state Theorem 3.3.4 more precisely.

Proposition 4.2.4. *The map $\ulcorner(-)\urcorner : \Lambda \to \Lambda^=$ extends to a functor $\mathcal{C}\ell_s(\Lambda) \to \mathcal{C}\ell_s(\Lambda^=)$.*

Proof. We define $\ulcorner(-)\urcorner$ on types to be the identity. Then, by the type preservation in Theorem 3.3.4, we have that for a morphism (a term) $t : A \to B$ that $\ulcorner t \urcorner : \ulcorner A \urcorner \to \ulcorner B \urcorner$. So it remains to prove that the encoding preserves identities and composition. The former is immediate, since $\ulcorner x \urcorner = x$. That composition is preserved means that $\ulcorner s[t/x] \urcorner = \ulcorner s \urcorner [\ulcorner t \urcorner / x]$, which is proved by induction on s. □

To establish further categorical structure on the simple classifying categories, we need to take some term equivalences into account.

Definition 4.2.5. Suppose $\mathcal{U}$ is a classifiable calculus with an equivalence relation $\equiv$ on terms that respects substitution: if $t \equiv t'$ and $s \equiv s'$, then $t[s/x] \equiv t'[s'/x]$. Suppose further that $\equiv$ is *type correct*, that is, for all $\Gamma \vdash t : A$ and s, if $t \equiv s$ then $\Gamma \vdash s : A$. Then the simple *classifying category modulo computations* $\mathcal{C}\ell_s^{\equiv}(\mathcal{U})$ is the quotient category $\mathcal{C}\ell_s(\mathcal{U})/\!\equiv$, see Section 2.2. ◀

Many categorical constructions are described through *universal mapping property* (UMP). For instance, the Cartesian product of two objects A_1 and A_2 is given by an object $A_1 \times A_2$ and two projections $\pi_i : A_1 \times A_2 \to A_i$, such that for all $f_1 : C \to A_1$ and $f_2 : C \to A_2$ there *exists a unique* $g : C \to A_1 \times A_2$ with $\pi_i \circ g = f_i$. In the context of type theory one tries to model these UMPs

as much as possible inside the syntactic theory. However, since the equivalence between terms is usually restricted to α-equivalence and reductions, the uniqueness usually fails. This situation is captured in the usual notion of *weak product*, which is an object $A_1 \times A_2$ with the projections π_i and for which only the existence of g is ensured, but not its uniqueness. In general, weak limits and weak colimits satisfy the existence property of limits and colimits, but the uniqueness property does not necessarily hold.

Lemma 4.2.6. *Both $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ and $\mathcal{C}\ell_s^{\equiv}(\Lambda^{=})$ have finite weak products and weak coproducts. Moreover, for all u, v, f and g the following diagram commutes in $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ and $\mathcal{C}\ell_s^{\equiv}(\Lambda^{=})$.*

$$\begin{array}{ccc} A & \xrightarrow{\langle u,v\rangle} & B_1 \times B_2 \\ & \searrow^{\langle f\circ u, g\circ v\rangle} & \downarrow{\scriptstyle f\times g} \\ & & C_1 \times C_2 \end{array}$$

Proof. We start with $\lambda\mu\nu$. Clearly, the type $\mathbf{1}$ is a weakly final object, since for each type A, we have $x : A \vdash \langle\rangle : \mathbf{1}$. The weak binary product of two types A_1 and A_2 is given by the product type. Its projections are $x : A_1 \times A_2 \vdash \pi_i\, x : A_i$, and pairing of terms gives the categorical pairing: Let $x : C \vdash t_i : A_i$, then $x : C \vdash \langle t_1, t_2\rangle : A_1 \times A_2$. Finally, we have

$$\pi_i \circ \langle t_1, t_2\rangle = (\pi_i\, x)[\langle t_1, t_2\rangle/x] = \pi_i\, \langle t_1, t_2\rangle \equiv t_i.$$

Moreover, the distribution diagram is again given by reduction:

$$(f \times g) \circ \langle u, v\rangle = \langle f[\pi_1\langle u, v\rangle/x], g[\pi_1\langle u, v\rangle/x]\rangle \equiv \langle f[u/x], g[v/x]\rangle = \langle f \circ u, g \circ v\rangle.$$

It is similarly easy to show that binary weak coproducts in $\lambda\mu\nu$ are given by sum types. Lastly, the weakly initial object of $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ is given by $\mathbf{0}$ and its elimination principle $E_A^{\mathbf{0}}$ as in Example 3.1.5.

For $\lambda\mu\nu{=}$ we have identified the same term constructors for sum and product types in the Examples 3.2.4 and 3.2.5, respectively. Moreover, the reduction rules are precisely the same for these terms. The unit type $\mathbf{1}$ is exactly the same as in $\lambda\mu\nu$, and in Example 4.1.22 we have also identified the empty type with its elimination principle. Thus, $\lambda\mu\nu{=}$ admits finite weak products and coproducts. □

We can also give an abstract account of function types. This is a bit more complex though, as with a naive approach to weakening the usual notion of exponential object is bound to run into the problem that one requires η-equivalence for products, that is, we would need to have $\langle \pi_1\, t, \pi_2\, t\rangle \equiv t$. Fortunately, there is an alternative, which has been proposed by Hayashi [Hay85, Thm. 2.3]. Note that the terminology used there is that the binary product is *semi-right-adjoint* to the diagonal. We refrain from using this terminology here, since it breaks down for coproducts.

Definition 4.2.7. A category $\mathbf{C}$ with (weak) binary products has *weak exponential objectss*, if for all objects A and B, there is an object B^A and a morphism $\mathrm{ev}_B \colon B^A \times A \to B$, subject to the following condition. For all $h \colon B \times A \to C$ there is a morphism $\lambda h \colon B \to C^A$, such that for all $u \colon D \to B$ the following two diagrams commute.[24]

$$\begin{array}{ccc} D & & \\ {\scriptstyle \lambda h\times \mathrm{id}}\downarrow & \searrow^{h} & \\ C^A \times A & \xrightarrow{\mathrm{ev}_C} & C \end{array} \qquad \begin{array}{ccc} D & & \\ {\scriptstyle u}\downarrow & \searrow^{\lambda(h\circ(u\times \mathrm{id}))} & \\ B & \xrightarrow{\lambda h} & C^A \end{array}$$

Lemma 4.2.8. *Both $\mathcal{CL}_s^{\equiv}(\Lambda)$ and $\mathcal{CL}_s^{\equiv}(\Lambda^{=})$ have weak exponential objects.*

Proof. For $\mathcal{CL}_s^{\equiv}(\Lambda)$ the exponential object B^A for types A and B is given by the function space $A \to B$ and the evaluation map ev_B by

$$x : B^A \times A \vdash (\pi_1\ x)\ (\pi_2\ x) : B.$$

Let $t\colon B \times A \to C$, that is, $x : B \times A \vdash t : C$. We define $\lambda t\colon B \to C^A$ to be the term

$$x : B \vdash \lambda y.\ t[\langle x, y\rangle/x] : A \to C.$$

For terms $x : D \vdash u : B$ and $x : D \vdash v : A$ we now have

$$\begin{aligned}
\mathrm{ev}_C \circ \langle \lambda t \circ u, v\rangle &= ((\pi_1\ x)\ (\pi_2\ x))[\langle \lambda t \circ u, v\rangle/x] \\
&\equiv (\lambda t \circ u)\ v \\
&= ((\lambda y.\ t[\langle x, y\rangle)/x])[u/x])\ v \\
&= (\lambda y.\ t[\langle u, y\rangle/x])\ v \\
&\equiv t[\langle u, v\rangle/x] \\
&= t \circ \langle u, v\rangle
\end{aligned}$$

and

$$\begin{aligned}
\lambda(t \circ (u \times \mathrm{id})) &\equiv \lambda(t \circ \langle u \circ \pi_1, \pi_2\rangle) \\
&= \lambda(t[\langle u[\pi_1\ x/x], \pi_2\ x\rangle/x]) \\
&= \lambda y.\ (t[\langle u[\pi_1\ x/x], \pi_2\ x\rangle/x])[(x, y)/x] \\
&\equiv \lambda y.\ (t[\langle u[x/x], y\rangle/x]) \\
&= \lambda y.\ (t[\langle u, y\rangle/x]) \\
&= (\lambda y.\ (t[\langle x, y\rangle/x]))[u/x] \\
&= \lambda t \circ u.
\end{aligned}$$

Thus $\mathcal{CL}_s^{\equiv}(\Lambda)$ has weak exponential objects. That $\mathcal{CL}_s^{\equiv}(\Lambda^{=})$ also has weak exponentials follows by the same reasoning and using the notations introduced in Example 3.2.6. □

For the following proposition we need to relax our understanding of what algebras and coalgebras are. Recall from Section 2.5 that an initial algebra for a functor $F\colon \mathbf{C} \to \mathbf{C}$ is a morphism $FA \to A$ for some object A, such that for all algebras $FB \to B$ there is a unique homomorphism $A \to B$. The concept of a final coalgebra is defined dually. We would like to understand fixed point types in these terms. Just as in Lemma 4.2.6, we can drop the uniqueness constraint for the homomorphism to obtain a notion of weakly initial algebras and weakly final coalgebras. However, already the very usage of functors in the definition of (co)algebras poses a problem because the action of types on terms that we gave in Definition 3.1.9 does not satisfy the laws of a functor. Thus, fixed point types do not have a (co)algebra structure for a functor F in the usual sense, rather we can only require F to be a map on objects and morphisms.

Definition 4.2.9. A *pre-functor* $F\colon \mathbf{C} \to \mathbf{D}$ between categories $\mathbf{C}$ and $\mathbf{D}$ is a map that sends every object $X \in \mathrm{ob}\,\mathbf{D}$ to an object $F(X) \in \mathrm{ob}\,\mathbf{D}$, and every morphism $f\colon X \to Y$ in $\mathbf{C}$ to a morphism $F(f)\colon F(X) \to F(Y)$ in $\mathbf{D}$.

Pre-functors are the weakest possible mapping between categories. However, since they neither preserve identities nor composition, there is no good general theory for pre-functors. The reason why we are introducing them here nevertheless is to have a language for organising the properties of fixed point types. By abuse of language, we will also talk about algebras, coalgebras and their homomorphisms, even if we are only given a pre-functor instead of a functor, cf. Definition 2.5.3.

Definition 4.2.10. Let $F\colon \mathbf{C} \to \mathbf{C}$ be a pre-functor. A *weakly final coalgebra* for F is a morphism $c\colon X \to FX$ in $\mathbf{C}$, such that for every coalgebra $d\colon Y \to FY$ there is a coalgebra homomorphism $f\colon Y \to X$ from d to c. *Weakly initial algebras* for F are defined dually.

Lemma 4.2.11. *Let $X \Vdash T : \mathbf{Ty}$, that is, let T be a type with a free type variable X, see Definition 3.1.1. Then T gives rise to pre-functors $\mathcal{C}\ell_s^{\equiv}(\Lambda) \to \mathcal{C}\ell_s^{\equiv}(\Lambda)$ and $\mathcal{C}\ell_s^{\equiv}(\Lambda^{=}) \to \mathcal{C}\ell_s^{\equiv}(\Lambda^{=})$ by putting*

$$T(A) \coloneqq T[A] = T[A/X] \qquad \text{and} \qquad T(t) \coloneqq T[\lambda x.t]\, x,$$

using $T[-]$ as defined in Definition 3.1.9. Moreover, there is a weakly initial algebra $\delta\colon T(\mu X.\, T) \to \mu X.\, T$ and a weakly final coalgebra $\omega\colon \nu X.\, T \to T(\nu X.\, T)$ both in $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ and $\mathcal{C}\ell_s^{\equiv}(\Lambda^{=})$. This means that for any morphisms $a\colon T(A) \to A$ and $c\colon A \to T(A)$ there are (not necessarily unique) morphisms $\overline{a}\colon \mu X.\, T \to A$ and $\widetilde{c}\colon A \to \nu X.\, T$, such that the following two diagrams commute.

$$\begin{array}{ccc} T(\mu X.\, T) & \xrightarrow{T(\overline{a})} & T(A) \\ \downarrow{\scriptstyle \delta} & & \downarrow{\scriptstyle a} \\ \mu X.\, T & \xrightarrow{\overline{a}} & A \end{array} \qquad\qquad \begin{array}{ccc} A & \xrightarrow{\widetilde{c}} & \nu X.\, T \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle \omega} \\ T(A) & \xrightarrow{T(\widetilde{c})} & T(\nu X.\, T) \end{array}$$

Proof. To show that T gives rise to a map $\mathcal{C}\ell_s^{\equiv}(\Lambda) \to \mathcal{C}\ell_s^{\equiv}(\Lambda)$ requires us to show that $t \equiv s$ implies $T(t) \equiv T(s)$. This, in turn, follows from $\equiv$ being a congruence and by a simple induction on T. Recall that $T(A) = T[A/X]$, so that

$$x : T(\mu X.\, T) \vdash \alpha\, x : \mu X.\, T \qquad \text{and} \qquad x : \nu X.\, T \vdash \xi\, x : T(\nu X.\, T),$$

and the algebra and coalgebra structures are thus given by $\delta \coloneqq \alpha\, x$ and $\omega \coloneqq \xi\, x$. Next, given $y : T(A) \vdash a : A$ and $y : A \vdash c : T(A)$, as in the assumption, we can define the *inductive extension* of a by $\overline{a} \coloneqq \mathrm{iter}^{\mu X.\, T}\, (y.\, a)\, y$ and the *coinductive extension* of c by $\widetilde{c} \coloneqq \mathrm{coiter}^{\nu X.\, T}\, (y.\, c)\, y$, thus obtaining

$$y : \mu X.\, T \vdash \overline{a} : A \qquad \text{and} \qquad y : A \vdash \widetilde{c} : \nu X.\, T.$$

Before we continue, let us briefly spell out the diagram for weakly final coalgebras in $\mathcal{C}\ell_s^{\equiv}(\Lambda)$. Since morphisms in $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ are (equivalence classes) of terms with a free variable for the domain, we can picture the homomorphism diagram for $\widetilde{c}$ as follows.

$$\begin{array}{ccc} A & \xrightarrow{\quad y : A \vdash \mathrm{coiter}^{\nu X.\, T}\, (y.\, c)\, y : \nu X.\, T \quad} & \nu X.\, T \\ {\scriptstyle x : A \vdash c : T(A)}\downarrow & \equiv & \downarrow{\scriptstyle x : \nu X.\, T \vdash \xi\, x : T(\nu X.\, T)} \\ T(A) & \xrightarrow[\quad x : T(A) \vdash T(\mathrm{coiter}^{\nu X.\, T}\, (y.\, c)\, y) : T(\nu X.\, T) \quad]{} & T(\nu X.\, T) \end{array}$$

That this homomorphism diagram commutes follows directly from the definition of conversion:

$$\begin{aligned}
\omega \circ \widetilde{c} &= (\xi\ x)[\mathrm{coiter}^{\nu X.\,T}\ (y.\,c)\ y/x] \\
&= \xi\ (\mathrm{coiter}^{\nu X.\,T}\ (y.\,c)\ y) \\
&\equiv T[\lambda y.(\mathrm{coiter}^{\nu X.\,T}\ (y.\,c)\ y)](c[y/y]) \\
&= T[\lambda y.(\mathrm{coiter}^{\nu X.\,T}\ (y.\,c)\ y)]\ c \\
&= T\Big(\mathrm{coiter}^{\nu X.\,T}\ (\lambda y.c)\ y\Big)[c/x] \\
&= T(\widetilde{c}) \circ c,
\end{aligned}$$

and analogously for $\mu X.\,T$.

In $\mathcal{C}\ell_s(\Lambda^=)$ we find back exactly the same structure by putting

$$\delta \coloneqq \alpha\ x \quad \text{and} \quad \omega \coloneqq x.\mathrm{out},$$

and

$$\begin{aligned}
\overline{a} &\coloneqq \mathbf{rlet}\ f : \mu X.\,T \to A = \{\cdot\ (\alpha\ x) \mapsto a[T(f\ x)/x]\}\ \mathbf{in}\ f\ x \\
\widetilde{c} &\coloneqq \mathbf{rlet}\ f : A \to \nu X.\,T = \{(\cdot\ x).\mathrm{out} \mapsto T(f\ x)[c/x]\}\ \mathbf{in}\ f\ x.
\end{aligned}$$

That these are well-typed is checked as in Theorem 3.3.4. Note that the inductive extensions $\overline{a}$ and the coinductive extensions $\widetilde{c}$ resemble the rewriting steps as given by contraction for iter and coiter in Definition 3.1.11. Hence that these are homomorphisms just follows as for $\lambda\mu\nu$. □

4.2.2. Classifying 2-Categories

On the one hand, we have given in Section 4.2.1 a category theoretical account of the computational behaviour of terms in the two calculi $\lambda\mu\nu$ and $\lambda\mu\nu=$. On the other hand, we have described in Section 4.1 the observational behaviour of terms. The purpose of the present section is to merge these two views. This is achieved by augmenting the classifying categories $\mathcal{C}\ell_s^{\equiv}(\Lambda)$ and $\mathcal{C}\ell_s^{\equiv}(\Lambda^=)$ with a 2-categorical structure that encodes the information about observations. This allows us to contrast the equality between morphisms in these categories, which relates terms with the same *computational behaviour*, and 2-cells between the morphisms, which relates terms with the same *observational behaviour*.

The 2-categories, which we obtain by adding 2-cells to the simple classifying categories, have "pseudo" instead of "weak" categorical structure. For instance, the product types do not just have projections π_i and pairing, subject to the equations $\pi_i \circ \langle f_1, f_2\rangle = f_i$, but the pairing is additionally unique up to isomorphism, see Definition 2.6.8. This stronger property allows us to show that the action of types on terms gives rise to a pseudo-functor. Thus, we can form 2-categories of algebras and coalgebras, whose pseudo-initial and, respectively, pseudo-final objects are given by fixed point types.

When enhancing the results of Section 4.2.1, we will find that we get an even stronger result than just pseudo-structures. Note that in the definition of pseudo-products, Definition 2.6.8, we have that for all $f_i \colon C \to A_i$ there is an $h \colon C \to A_1 \times A_2$ with $\pi_i \circ h \cong f_i$. So there only needs to be an *isomorphism* between $\pi_i \circ h$ and f_i. However, we have shown in Lemma 4.2.6 that in the

classifying categories there is a (not necessarily unique) h, such that $\pi \circ h = f_i$. We will call such a structures pseudo-products with strict choice. These characterise precisely the interplay of the reduction relation of the calculi and observational equivalence.

We begin by casting this last discussion into definitions, thereby refining the definitions given in Definition 2.6.8.

Definition 4.2.12. Let $\mathbf{C}$ be a 2-category. We say that $\mathbf{C}$ has finite *pseudo-products with strict choice*, if it has finite pseudo-products, see Definition 2.6.8, such that for all A_1, A_2 and $f_i \colon C \to A_i$ there is an $h \colon C \to A_1 \times A_2$ with $\pi_i \circ h = f_i$, where $\pi_i \colon A_1 \times A_2 \to A_i$ are the projections of the pseudo-product. We denote this h by $\langle f_1, f_2 \rangle$. Moreover, we have for all appropriate u, v, f, g that $(f \times g) \circ \langle u, v \rangle = \langle f \circ u, g \circ v \rangle$, cf. Lemma 4.2.6. Dually, $\mathbf{C}$ is said to have *pseudo-coproducts with strict choice*, if it has pseudo-coproducts, with injections $\kappa_i \colon A_i \to A_1 + A_2$, such that for all morphisms $f_i \colon A_i \to C$, there is a morphism $[f_1, f_2] : A_1 + A_2 \to C$ with $[f_1, f_2] \circ \kappa_i = f_i$. Finally, *pseudo-exponents with strict choice* in $\mathbf{C}$ are pseudo-exponents, such that the two identities in Definition 4.2.7 are fulfilled.[25] ◀

Let us now identify the concepts from Definition 4.2.12 in the calculi $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. To do that, we first need to enrich their classifying categories with a 2-categorical structure that represents observational equivalence.

Lemma 4.2.13. *There are 2-categories* $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ *and* $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$*, where*[26]

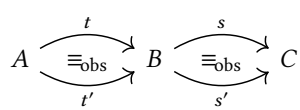
$$\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda) \coloneqq \begin{cases} \textit{objects:} & \textit{types } A \\ \textit{morphisms:} & [t]_{\equiv} \colon A \to B \textit{ is the convertibility equivalence class of a term} \\ & t \in \Lambda \textit{ with } x : A \vdash t : B \\ \textit{2-cells:} & t \Rightarrow s \textit{ exists iff } t \equiv_{\mathrm{obs}} s. \end{cases}$$

and

$$\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON}) \coloneqq \begin{cases} \textit{objects:} & \textit{types } A \\ \textit{morphisms:} & [t]_{\equiv} \colon A \to B \textit{ is the convertibility equivalence class of a term } t \in \mathbf{ON}_B^{x:A} \\ \textit{2-cells:} & t \Rightarrow s \textit{ exists iff } t \equiv_{\mathrm{obs}} s. \end{cases}$$

Proof. The vertical composition of 2-cells in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ is just given by transitivity of observational equivalence: If $\gamma : r \Rightarrow s$ and $\rho \colon s \Rightarrow t$, then $r \equiv_{\mathrm{obs}} s$ and $s \equiv_{\mathrm{obs}} t$. By transitivity we have $r \equiv_{\mathrm{obs}} t$, and thus there is a $\delta \colon r \Rightarrow t$. Since 2-cells are unique in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$, we can define $\rho \circ_1 \gamma = \delta$.

Horizontal composition arises in a situation like the following.



By (i) of Lemma 4.1.25 and the definition of observational equivalence on open terms we have

$$s[t/x] \equiv_{\mathrm{obs}} s[t'/x] \equiv_{\mathrm{obs}} s'[t'/x],$$

so that we can compose these two equivalences horizontally. Finally, note that since there is at most one 2-cell between two terms, the exchange law is automatically validated.

For $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ we note that observationally normalising terms are closed under composition, so that restricting to **ON**-terms gives at least a category. The 2-categorical structure is then given just as for $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$. □

Note that, since 2-cells in the classifying 2-categories are given by observational equivalence, two morphisms $[s]_{\equiv}$ and $[t]_{\equiv}$ are isomorphic, see Definition 2.6.3, if only if s and t are observationally equivalent. Since by Lemma 4.1.25.(iv) the choice of the representatives s and t does not matter when reasoning about observational equivalence, we will identify equivalence classes like $[s]_{\equiv}$ with the representing term s, and consider s as a morphism in the corresponding classifying 2-category.

Let us extend the results about the classifying categories in the last section to take the observational behaviour of terms into account. More precisely, we show that the categorical structures that we identified as weak in $\mathcal{CL}^{\equiv}_{s}(\Lambda)$ and $\mathcal{CL}^{\equiv}_{s}(\Lambda^{=})$ now become *pseudo*-structures *with strict choice* in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$. Spelling this out for, for instance, for product types, we have for $t_1 : C \to A_1$, $t_2 : C \to A_2$ and $h : C \to A_1 \times A_2$ in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ with $\pi_i \circ h \cong t_i$ that $h \cong \langle t_1, t_2\rangle$, that is, $\pi_i \circ h \equiv_{\mathrm{obs}} t_i$ implies $h \equiv_{\mathrm{obs}} \langle t_1, t_2\rangle$. Moreover, the choice of $\langle t_1, t_2\rangle$ is strict because the computation rules imply the identity $\pi_i \circ \langle t_1, t_2\rangle = t_i$ of equivalence classes.

Theorem 4.2.14. *Both the 2-categories $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ have finite pseudo-products, finite pseudo-coproducts and pseudo-exponents with strict choice.*

Proof. We have seen in the Lemmas 4.2.6 and 4.2.8 that the classifying categories have the necessary structures. That is, they have weak product, coproduct and exponent objects, projections, injections and application, and pairing, copairing and abstraction, which fulfil the necessary identities. Note that in the case of $\lambda\mu\nu=$, $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ is closed under pairing, copairing and abstraction by Lemma 4.1.11.

Thus it remains to prove that pairing, copairing and abstraction are unique up to isomorphism. We prove this in full detail in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$, the case of $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ is essentially the same. Recall that the pairing of $t_1 : C \to A_1$ and $t_2 : C \to A_2$ was given by $x : C \vdash \langle t_1, t_2\rangle : A_1 \times A_2$. Now let $h : C \to A_1 \times A_2$ with $\pi_i \circ h \cong t_i$, that is, $\pi_i\, h \equiv_{\mathrm{obs}} t_i$.[27] We have for $i = 1, 2$

$$\pi_i\, h \equiv_{\mathrm{obs}} t_i \equiv \pi_i\, \langle t_1, t_2\rangle,$$

which can easily seen to imply $h \equiv_{\mathrm{obs}} \langle t_1, t_2\rangle$ and so $h \cong \langle t_1, t_2\rangle$. The analogous reasoning also works for the copairing $[t_1, t_2] = \{\kappa_1\, x \mapsto t_1 \;;\; \kappa_2\, x \mapsto t_2\}\, x$ and the abstraction $\lambda t = \lambda y.t[\langle x, y\rangle/x]$. □

Similarly, we can identify the observational behaviour of algebras and coalgebras.

Definition 4.2.15. A *pseudo-final coalgebra with strict choice* is a pseudo-final coalgebra $c : X \to FX$, see Definition 2.6.9, such that for all $d : Y \to FY$ there is a (not necessarily unique) homomorphism $\widetilde{d} : Y \to X$ for which the 2-cell $\phi_{\widetilde{d}} : c \circ \widetilde{d} \Rightarrow F\widetilde{d} \circ d$ is the identity. Dually, a pseudo-initial algebra $a : FA \to A$, see Definition 2.6.9, has a strict choice, if for every algebra $b : FB \to B$ there is a pseudo-homomorphism $\overline{b} : A \to B$, such that the mediating 2-cell $\theta_{\overline{b}} : \overline{b} \circ a \Rightarrow b \circ F\overline{b}$ is the identity. ◂

Next, we prove that for each type A the action of A on terms, see Definition 3.1.9, gives rise to a pseudo-functor on the classifying categories. Moreover, we show that fixed point types are pseudo-initial algebras or pseudo-final coalgebras with strict choice for these pseudo-functors. These two

results are proved my mutual induction, since pseudo-initial algebras and pseudo-final coalgebras are used to define functors from fixed point types, and conversely, pseudo-functoriality is needed to obtain pseudo-initial algebras and pseudo-final coalgebras for smaller types.

We start the mutual induction by showing that the type action indeed gives a pseudo-functor.

Lemma 4.2.16. *Let T be a type with $X_1, \dots, X_n \Vdash T : \mathbf{Ty}$.[28] If all fixed point types occurring in T are pseudo-initial algebras and pseudo-final coalgebras, then the maps of categories associated with T in Lemma 4.2.11 can be extended to pseudo-functors of arity n: $T(-)\colon \big(\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)\big)^n \to \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and $T(-)\colon \big(\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})\big)^n \to \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$.*

Proof. To show that that $T(-)$ is a pseudo-functor, we need to extend $T(-)$ to 2-cells and prove that $T(-)$ preserves identity morphisms and composition up to isomorphism, that is, up to observational equivalence. First, recall that for a tuple $\vec{t}$ of terms we defined $T(\vec{t}) = T[\overrightarrow{\lambda x.\, t}]\, x$ in Lemma 4.2.11. By the definition of the type action in Definition 3.1.9, we have that $T[-]$ substitutes $\vec{t}$ into an observationally normalising context. Thus, if $\vec{t} \equiv_{\mathrm{obs}} \vec{t'}$, we have by Lemma 4.1.25(i) that also $T(\vec{t}) \equiv_{\mathrm{obs}} T(\vec{t'})$. This in turn means, that we can send a 2-cell in $\big(\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)\big)^n$, which says that all the terms in $\vec{t}$ and $\vec{t'}$ are observationally equivalent, to the unique 2-cell in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ that expresses that $T(\vec{t})$ and $T(\vec{t'})$ are observationally equivalent. So T also has a well-defined action on 2-cells.

It remains to prove that T fulfils the law of a pseudo-functor. We proceed by induction on T. In the base case $T = X_i$, we note that $X_i[-]$ is the ith projection from the product category, hence a pseudo-functor.

To prove the induction step, we need to show that for compound types T, $T(-)$ is a pseudo-functor, provided that its components are pseudo-functors. Concretely, we need to show that the pseudo-functor laws hold for the compound types $T_1 + T_2$, $T_1 \times T_2$ and $T_1 \to T_2$, provided the laws hold for T_1 and T_2. This is a routine calculation by using Theorem 4.2.14.

The case for fixed point types is essentially an adoption of the standard proof that the point-wise existence of final coalgebras gives rise to a functor [Kim10], but we have to replace equality between morphisms by observational equivalence. We give the details for $\nu X.\, T$, the case for least fixed point types is analogous. Consider the fixed point type $\nu X.\, T$. The induction hypothesis is in this case for $X_1, \dots, X_n, X \Vdash T : \mathbf{Ty}$ that $T(-)\colon \big(\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)\big)^{n+1} \to \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ is a pseudo-functor. Now let $\vec{t}\colon \vec{A} \to \vec{B}$ be a morphism in $\big(\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)\big)^n$. By Lem. 4.2.18 below, the pseudo-functors $T(\vec{A}, -)$ and $T(\vec{B}, -)$ of type $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda) \to \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ have as pseudo-final coalgebras $\xi_{\vec{A}}$ and $\xi_{\vec{B}}$, and we take $(\nu X.\, T)(\vec{t})$ to be the coinductive extension (Lemma 4.2.11) of

$$\nu T(\vec{A}, -) \xrightarrow{\xi_{\vec{A}}} T(\vec{A}, \nu T(\vec{A}, -)) \xrightarrow{T(\vec{A}, \mathrm{id})} T(\vec{B}, \nu T(\vec{A}, -)),$$

cf. Definition 3.1.9. The pseudo-functor laws follow from uniqueness up to isomorphism of coinductive extensions on pseudo-final coalgebras, see Section 2.6.2.

In case of $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$, one proceeds in exactly the same way, only that we have to use Proposition 4.1.9.(i) to ensure that the functorial action of T is a morphism in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$. □

The following technical lemma is used in the proof of Lemma 4.2.18. It allows us to construct for each coalgebra c and each test φ on the unfolding of largest fixed point types, a "universal" term t

that cannot be distinguished from any homomorphism from c into the largest fixed point up to the observation depth given by φ.

Lemma 4.2.17. *Let T be a type with $X \Vdash T : \mathbf{Ty}$.*

1. *For every coalgebra $c\colon A \to T[A]$ in one of the classifying 2-categories, every (observationally normalising) term $u : A$ and every test $\varphi :\downarrow T[\nu X.T]$, there is a term t with $x : A \vdash t : T[\nu X.T]$, such that for any T-coalgebra pseudo-homomorphism $h\colon A \to \nu X.T$ from c to $\omega = \xi\, x$, we have*

$$t[u/x] \vDash \varphi \iff (\xi\, h)[u/x] \vDash \varphi.$$

2. *For every algebra $a\colon T[A] \to A$ in one of the classifying 2-categories, every (observationally normalising) term $u : T[A]$ and every test $\varphi :\downarrow A$, there is $x : T[\mu X.T] \vdash t : A$, such that for any pseudo-homomorphism $h\colon \mu X.T \to A$ from $\delta = \alpha\, x$ to a, we have*

$$t[u/x] \vDash \varphi \iff h[\alpha\, u/x] \vDash \varphi.$$

Proof. We only sketch the proof of 1, the proof of 2 is given by duality. Since h is a pseudo-homomorphism, we have $(\xi\, h)[u/x] \vDash \varphi \iff (T[h]\, c)[u/x] \vDash \varphi$ hence it suffices to prove the existence of a t with $t[u/x] \vDash \varphi \iff (T[h]\, c)[u/x] \vDash \varphi$. We do this by proving the following.

For any sub-expression S of T,[29] any $x : A \vdash c' : T[A]$, any $u' \in \mathbf{ON}_A$ and any test $\psi :\downarrow S[\nu X.T]$ there is $x : A \vdash t : S[\nu X.T]$, such that for any T-coalgebra pseudo-homomorphism f from c' to $\omega = \xi\, x$ the equivalence $t[u'/x] \vDash \psi \iff (S[f]\, c')[u'/x] \vDash \psi$ holds. The result then follows by taking $c' = c$ and $\psi = \varphi$. The claim itself is proved by induction on ψ. □

The following lemma is needed in the induction step for fixed points in the proof of Lem. 4.2.16.

Lemma 4.2.18. *For any type T with $X \Vdash T : \mathbf{Ty}$, if the associated map of classifying categories defined in Lemma 4.2.11 is a pseudo-functor, then $\delta\colon T(\mu X.T) \to \mu X.T$ is a pseudo-initial algebra and $\omega\colon \nu X.T \to T(\nu X.T)$ a pseudo-final coalgebra with strict choice.*

Proof. We already proved in Lemma 4.2.11 that δ and ω are weakly initial and final, respectively. Thus we already have a choice of strict (co)inductive extensions, which are also morphisms in the corresponding classifying categories by Proposition 4.1.9. It only remains to prove that these extensions are unique up to isomorphism.

The argument for $\mu X.T$ is completely analogous to that for $\nu X.T$, so we only prove that the latter is pseudo-final. So let $c\colon A \to T(A)$ be a coalgebra and $h\colon A \to \nu X.T$ be a pseudo-homomorphism into $\omega\colon \nu X.T \to T(\nu X.T)$, that is to say, with $\xi\, h \equiv_{\mathrm{obs}} T[h]\, c$. We have to show $h \equiv_{\mathrm{obs}} \widetilde{c}$.

Since h and $\widetilde{c}$ are open terms, we need to show that for any (observationally normalising) term $u : A$ and every test $\psi :\downarrow \nu X.T$, $h[u/x] \vDash \psi \iff \widetilde{c}[u/u] \vDash \psi$. If ψ is $\top$ or $\bot$, then this is clear. So suppose $\psi = [\xi]\,\varphi$ for some test $\varphi :\downarrow T(\nu X.T)$. By Lemma 4.2.17 there is a t, such that, $x : A \vdash t : T(\nu X.T)$ and for which we have

$$\begin{aligned}
h[u/x] \vDash \psi &\iff (\xi\, h)[u/x] \vDash \varphi && \text{Def. } \vDash \\
&\iff t[u/x] \vDash \varphi && \text{Lem. 4.2.17} \\
&\iff (\xi\, \widetilde{c})[u/x] \vDash \varphi \\
&\iff \widetilde{c}[u/u] \vDash \psi.
\end{aligned}$$

Since this holds for any u and φ, we have that $h \equiv_{\mathrm{obs}} \widetilde{c}$, as required. Thus ω is a pseudo-final coalgebra with strict choice. □

From Lem. 4.2.16 and Lem. 4.2.18 the main result of this section follows.

Theorem 4.2.19. *For all types T with $X \Vdash T : \mathbf{Ty}$, the pseudo-functor T has a pseudo-initial algebra with strict choice on the carrier $\mu X.T$, and a pseudo-final coalgebra with strict choice on the carrier $\nu X.T$.*

All the statements and proofs in this section were given both for the classifying 2-category of $\lambda\mu\nu$ and $\lambda\mu\nu{=}$. To do so, we have often carried out proofs just for $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and treated $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ by analogy. This can be made more precise by defining observational normalisation and observational equivalence for certain 2-categories, where the 2-cells are intended to be rewriting steps. We would then extend the simple classifying categories $\mathcal{C}\ell_s(\Lambda)$ and $\mathcal{C}\ell_s(\Lambda^{=})$ with the structure of a 2-category, where a 2-cell $t \Rightarrow s$ exists if $t \longrightarrow\!\!\!\!\!\rightarrow s$. One could then characterise strongly normalising terms, define tests and observational normalisation, and carry out the rest of the development in this section in these categories. The properties of these 2-categories that are needed to do that are basically the content of the Lemmas 4.2.6, 4.2.8 and 4.2.11. The only difference is that computation steps in these categories are given by 2-cells instead of equality, since in the above-mentioned lemmas we used quotients of $\mathcal{C}\ell_s(\Lambda)$ and $\mathcal{C}\ell_s(\Lambda^{=})$. The problem with this approach is that, while being very general, it is also very technical. So we will not pursue this further here.

4.3. Conclusion and Related Work

Summary

In the present chapter we have set the scene for the remainder of this thesis by defining a notion of observation for programs, and we gave some first reasoning principles for programs.

In the course of this, we identified two important aspects of programs: observational normalisation and observational equivalence. Observational normalisation is only relevant in the case of the calculus $\lambda\mu\nu{=}$, as in the calculus $\lambda\mu\nu$ all terms are strongly normalising to begin with. The reason for singling out observationally normalising terms is that we needed a class of terms that normalise under all observations to give a reasonable notion of tests on terms of function types. Further, observationally normalising terms were precisely the class of terms that allowed us to construct a 2-category of terms that admits initial algebras and final coalgebras on μ- and ν-types.

The second notion we studied was observational equivalence, which is induced by observation on programs. First, we defined observations for programs of mixed inductive-coinductive type uniformly through program tests. This logic is interesting because it makes the usual slogan that "inductive types are defined by their constructors and coinductive types are defined by their destructors" precise. Moreover, this definition of observation opens up the possibility to take a coalgebraic view on observational equivalence. We discuss will this in Section 5.1.2.

The reasoning principles for programs come about as mapping principles of pseudo-products etc. In particular, we can show that programs are observationally equivalent by using induction and coinduction, that is, by establishing that programs are homomorphisms, respectively, out of an inductive or into a coinductive type. Moreover, the established 2-categories allowed us to distinguish computational and observational behaviour of programs, in the sense that convertible terms are equal,

whereas observationally equivalent terms are merely isomorphic. This is captured by showing that the pseudo-structure of the 2-categories $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ come with a choice of a strict morphism.

In the next chapter, we will devise more reasoning methods for observational normalisation and observational equivalence. This improves on the category theoretical reasoning about programs in three ways. First, observational normalisation cannot be proven for general terms by means of the (2-)categories we have given. Second, proving equivalences in $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ is limited to terms that arise through recursion and corecursion, which takes away the usefulness of the copattern calculus $\lambda\mu\nu{=}$. Third, reasoning in the basic language of (2-)categories is very hard. This is why one usually associates to a category an equational theory or a logic, which captures the essential properties of that category, see [Gol84; LS88].

Related Work

There is an enormous body of work on program equivalences, and the category theoretical view on programs and their properties. So let us clarify the relation to existing work and novelties in each section.

Observational Equivalence and Normalisation Productivity and well-definedness of programs have also been studied in other settings than type theory. For instance, conditions for productivity of stream programs have been provided in [EGH08; End+10; HKR14]. For more general programs, abstract formats for ensuring productivity can be obtained from the framework of (abstract) GSOS [Kli11; TP97], and from the guardedness condition of [AMV11]. Some steps towards less syntactic, more general conditions for ensuring productivity have recently been made in [EH11] by using infinitary rewriting techniques.

Interestingly, the definition of observational normalisation is very similar to the interpretation of types as saturated sets in proofs of strong normalisation, see for instance [AA99]. Also the proofs that, for instance, the iterator and coiterator are strongly normalising have the same flavour. However, a major difference to [AA99] is that in the present setting we use impredicative definitions of fixed points. It would be interesting to study a predicative definition as in loc. cit., but we shall leave that for the future.

Another possibility to study inductive and coinductive types in type theory is via an encoding into the (impredicative) polymorphic λ-calculus, see e.g. Geuvers [Geu92]. Using this approach, Plotkin and Abadi [PA93] and Hasegawa [Has94] have shown that parametricity schemes induce an equivalence which yields initial algebras and final coalgebras. However, since these parametricity results are external to the polymorphic λ-calculus it is hard to devise induction and coinduction schemes *inside* a calculus based on the impredicative encoding. This issue becomes even more important in dependently typed languages, see Chapter 7. There has been some recent progress by Ghani et al. [GFO16] in making parametricity available inside the theory, but this naturally becomes highly complicated. Thus we avoid such an encoding into an impredicative language entirely.

Program equivalence in a dependent type theory is formalised in Altenkirch et al. [AMS07] via a propositional equality type called *observational equality*. This equality type is defined inductively over types in a manner similar to ours, but they do not need to require (an analogue of) observational normalisation as their term language contains only restricted forms of λ-abstraction and recursion which ensures that all terms are strongly normalising. In particular, their system has no coinductive

types. In contrast, our approach has a more flexible term language that allows definitions via systems of copattern equations.

The notion of observational equivalence is similar to the *contextual equivalence* of Plotkin [Plo77] and Milner [Mil77], since test formulae can be interpreted as certain program contexts. Similar in spirit is also Zeilbergers's observational equivalence [Zei09], as he considers programs to be equivalent if they yield the same, what he calls, result in all environments. Interestingly, he starts by considering only the designated result that a program diverges, and then observes that, in order to get a useful equivalence, he needs to have a second possible result. This is similar to the fact that our test language has two basic tests, namely the everywhere true and the everywhere false test. Without these, we would get a trivial equivalence.

The *observational congruence* of Pitts [Pit00] characterises contextual equivalence for an extension of PCF as the greatest relation with certain properties. This is similar to the characterisation of observational equivalence as bisimilarity that we will give in Section 5.1.2. We can show that observational equivalence is contained in Pitts' observational congruence when suitably adopted to our setting. However, the precise connection is not clear.

For inductive types and abstractions, observational equivalence is also closely related to the notion of applicative bisimilarity in the lazy λ-calculus by Abramsky [Abr90]. Applicative bisimilarity compares terms by reducing to a weak head normal form (with a λ in head position) and comparing the results of function application. It is possible to extend applicative bisimilarity in a natural way to all our terms, but the resulting notion will differ from observational equivalence. In particular, applicative bisimilarity is weaker on diverging terms, as also discussed in [Gor95].

Our work is closest in spirit to the work by Howe [How89; How96b] and Gordon [Gor95]. Howe [How89; How96b] studies coinductive proof principles for reasoning about equivalence in a general class of lazy languages with binding, and discusses applications in intuitionistic type theory and the Nuprl theorem prover. Unique mapping properties are not considered. Gordon [Gor95] gives a bisimulation characterisation of contextual equivalence in the language FPC, and studies up-to techniques. Again, coinductive types are not part of FPC, and unique mapping properties are not investigated.

Category Theoretical Properties of $\lambda\mu\nu$ and $\lambda\mu\nu=$ Category theoretical properties of type theories have been discussed in various forms. A rigorous account has been given by Lambek and Scott [LS88], where simply typed calculi with a natural numbers object are treated. Similar to that, Jacobs [Jac99] describes several type theories without recursive types and studies their properties through the use of category theory. Also calculi with non-nested inductive and coinductive types have been related to categorical concepts, see for example [Geu92; GP07; How96a; UV99a; Ven00]. However, to the best of my knowledge, the 2-categorical view in Section 4.2.2, which gives a precise distinction between computational and observable behaviour, has not been investigated anywhere else. All category theoretical descriptions of calculi use either similar structures as in Section 4.2.1 or quotients of the term sets, which makes the differences between observations and computations invisible.

Contributions

Having discussed the related work, let us clarify the contributions made in this chapter. First of all, the uniform definition of a program equivalence for typed, recursive programs through a

testing logic has not been studied before. The testing logic builds thereby on the definition of observationally normalising terms, which are terms on which all observations are computable, that is, any observation results in a strongly normalising term and least fixed point types only admit finitely many consecutive unfolding steps. To my knowledge, such a notion has not been studied on calculi that allow non-terminating programs, although the definition of observational normalisation is closely related to the saturated sets semantics of types in strong normalisation proofs, see e.g. [AA99] and Chapter 7. More interesting, however, are the proof principles for observational equivalence that we will establish in Chapter 5.

The second contribution in this chapter is the study of observational equivalence through the eye of 2-categories. More specifically, we constructed 2-categories, whose objects are types, morphisms are equivalence classes of terms of $\lambda\mu\nu$ or $\lambda\mu\nu{=}$ under conversion, and 2-cells are given by the observational equivalence of terms. This setup allowed us to characterise, for instance, product types as pseudo-products with strict choice. In particular, this means that on the product type there are projections and one can pair morphisms. The crucial property of the paired morphism is that its composition with the projections is *strictly equal* to the components, while the pairing itself is unique only *up to isomorphism*. Since strict equality corresponds to conversion and isomorphisms correspond to observational equivalence, the notion of pseudo-product with strict choice captures and distinguishes precisely the computational and observational behaviour of product types. Thus, the main contribution in Section 4.2 is this 2-category theoretical characterisation of computational and observational behaviour of all types in the two calculi.

Future Work

There is always room for improvements, but let us highlight one particular possibility for extending the 2-category view in Section 4.2.2. Recall that we defined a 2-category $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ that has only observationally normalising terms of $\lambda\mu\nu{=}$ as morphisms. We can drop this restriction and obtain a 2-category $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$ that has all $\lambda\mu\nu{=}$-terms as morphisms. It seems that all proofs in Section 4.2.2 can be adapted to this 2-category, so that also $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$ has pseudo-products, pseudo-coproducts etc. This raises of course the question what the relation is between $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$ and $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$. Clearly, there is an inclusion 2-functor $I\colon \mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON}) \hookrightarrow \mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$. Now note that $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ only consists of total functions, whereas $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$ also contains partial functions. Thus, we would expect there to be a relation like the adjunction between the categories of sets with total functions and sets with partial functions. In this classical case, a partial function $f\colon X \rightharpoonup Y$ is sent to a total function $g\colon X + \mathbf{1} \to Y + \mathbf{1}$, and the adjunction property expresses that a partial function $f\colon X \rightharpoonup Y$ corresponds to exactly one total function $g\colon X \to Y + \mathbf{1}$:

$$\frac{X \longrightarrow Y \qquad \text{in } \mathbf{Set}_{\mathrm{par}}}{X \longrightarrow Y + \mathbf{1} \qquad \text{in } \mathbf{Set}}$$

However, the definition of g from f requires the law of excluded middle, because one defines $g(x) = \kappa_1(f(x))$ if $x \in \mathrm{dom}(f)$ and $g(x) = \kappa_2(*)$ otherwise. Thus, it is not possible to write g as a program, since termination is not decidable in general.

Fortunately, an alternative to represent partial functions is offered by the so-called *type of partial elements*, which was first introduced by Capretta [Cap05]. For any type $A \in \mathrm{Ty}$ the type of partial

A-elements is given in our type system by $A^\nu \coloneqq \nu X.\, \overline{A}$, where we obtain $\overline{A}$ by replacing every type variable Y in A by $X + Y$. The idea is that every recursion step can now be captured through the type A^ν. One needs to show that $(-)^\nu$ can be extended to a functor $\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=}) \to \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$, which is moreover right-adjoint to the inclusion. I conjecture that this can be done by instrumenting a program so that every computation step is recorded in A^ν. For instance, recall from Example 4.1.22 that the type $\mathbf{0} = \mu Y.\, Y$ has no normalising inhabitant. However, we can define the term $B = \alpha\, B$ of type $\mathbf{0}$ in $\lambda\mu\nu{=}$. This term can be transformed into $\check{B} : \mathbf{0}^\nu$ with $\mathbf{0}^\nu = \nu X.\, \mu Y.\, X + Y$ by putting

$$\check{B}.\mathrm{out} = \alpha\, (\kappa_1\, B).$$

The aim is thus to assign to each t with $x : A \vdash t : B$ an observationally normalising term $\check{t}$ with $x : A \vdash \check{t} : B^\nu$ by means of such an instrumentation. Conversely, we can erase the partiality annotation from $x : A \vdash s : B^\nu$ by defining a term called extract of type $B^\nu \to B$ by induction on B. We can use extract to define a term $\hat{s}$ with $x : A \vdash \hat{s} : B$ by putting $\hat{s} \coloneqq \mathrm{extract}\, s$. These two constructions should give us then a correspondence

$$\frac{A \longrightarrow B \qquad \text{in } \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})}{A \longrightarrow B^\nu \qquad \text{in } \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})}$$

which is one-to-one up to observational equivalence. In other words, I conjecture that there is a pseudo-adjunction as in the following diagram.

$$\mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON}) \;\underset{(-)^\nu}{\overset{I}{\perp}}\; \mathcal{CL}^{\equiv}_{s,\mathrm{obs}}(\Lambda^{=})$$

The difficulty is the definition of the instrumented term $\check{t}$. I claim that $\check{t}$ is computable, since it seems to be possible to define the instrumentation by induction on the term structure, cf. [Cap05, Sec. 4]. This in contrast to the classical approach, which requires us to decide termination. But as the instrumentation is also a tedious process, I leave this for the future. One final note on this instrumentation: One might be tempted to use directly the delay monad from [Cap05]. This, however, would not allow us to define the above advertised adjunction. For instance, if we use $\mathbf{0}^\nu = \nu X.\, \mathbf{0} + X$ instead, then the only way to define $\check{B}$ is by $\check{B}.\mathrm{out} = \kappa_2\, \check{B}$, from which we cannot recover a term that is observationally equivalent to B. This is the reason for the more complicated instrumentation, which allows us to preserve the observational behaviour of terms.

Following the program of Lambek and Scott [LS88], it would be natural to show that the classifying 2-category of the calculus $\lambda\mu\nu$ is initial among all 2-categories that have the same closure properties, that is, which have all finite pseudo-products etc. This should not be a too difficult task, but it is left open for now. More intricate is to show that there is also a functor from the classifying 2-category of the observationally normalising terms in $\lambda\mu\nu{=}$ to the classifying 2-category of $\lambda\mu\nu$. The difficulty lies in the fact that systems of equations need to be reduced to iteration and coiteration. We also need to leave this reduction open for now.

Notes

[16] We could have called terms in **ON** *persistently strongly normalising*, in analogy with the similar notion of the typed λ-calculus [BDS13, Sec. 17.2]. However, we favour the name observationally normalising since it emphasises that terms in **ON** must be strongly normalising under all possible observations.

[17] The fact that strong normalisation and the non-existence of infinite unfoldings on least fixed point types do not coincide is actually an oversight in [BH16]. I would like to thank Andreas Abel for pointing this problem out to me, and preventing it therefore from slipping into my thesis.

[18] See Example 2.4.3 for notation regarding families of sets.

[19] The interpretation in Definition 4.1.14 of tests on A as terms of type $A \to \mathsf{Bool}$ is similar to the notion of *observation* from the λ-calculus [BDS13, Sec. 3.5]. However, there observations on function types allow *any* term as argument, whereas we restrict to observationally normalising terms. This has the effect that we identify terms of function type on the basis of their observational rather than their computational behaviour, see the discussion in the introduction of this chapter. This connection with the λ-calculus is one reason for the name "observational equivalence", another is that our notion is an instance of the coalgebraic notion of observable behaviour, as we will see in Section 5.1.

[20] The notation $E^{\mathbf{0}}_A$ is used here because under the propositions-as-types interpretation, see Section 6.1, $\mathbf{0}$ can be seen as the formula denoting falsum. Under this interpretation, the term $E^{\mathbf{0}}_A$ is the elimination principle for $\mathbf{0}$.

[21] Note that λD and g are similar to what Abramsky [Abr90] calls principal weak head normal forms.

[22] Usually, classifying categories are given for terms in arbitrary contexts. In that case, a classifiable calculus would also have to satisfy weakening and exchange. However, the remaining development in this section does not require a general notion of classifying categories, and in fact is simpler in the way it is presented. Note also that one could directly assume a calculus to be given in form of a classifying category. But since the existence of such a category has to be proved in one way or another, we prefer the given presentation, which allows us to give a general existence proof.

[23] Note that the notion of classifying category that we use here is a simplification of the classifying categories in [Jac99].

[24] Hayashi [Hay85] uses a slightly more complicated definition. The version that we use here is equivalent to the original one under the assumption of the distributivity law for products that we established in Lemma 4.2.6. This simplification was suggested to me by Andreas Abel.

[25] It should be noted that pseudo-products, pseudo-coproducts, and pseudo-exponents with strict choice are stronger than their weak counterparts. The reason is that we can alternatively define, for example, pseudo-products with strict choice as weak products, such that the pairing is unique up to a unique 2-isomorphism.

[26] In the proof of Lemma 4.2.13 we see that the coherence between vertical and horizontal composition are automatically fulfilled, since in the classifying 2-categories there is at most one 2-cell between

morphisms. In other words, the classifying 2-categories $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda)$ and $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\mathbf{ON})$ are order-enriched (or even more specific: enriched over equivalence relations), cf. Definition 2.6.1. Such 2-categories are also called "locally posetal" by Lack [Lac10]. This property of the classifying 2-categories is going to very useful in the following, as we always just have to prove that two terms are observationally equivalent but we can save ourselves from proving any coherence conditions.

[27] Note that $\pi_i \circ h$ is a statement in the 2-category $\mathcal{C}\ell^{\equiv}_{s,\mathrm{obs}}(\Lambda)$. Recall that $\pi_i \colon A_1 \times A_2 \to A_i$ was defined, by abuse of notation, to be the term $\pi_i\, x$. Thus $\pi_i \circ h = (\pi_i\, x)[h/x] = \pi_i\, h$.

[28] Definition 3.1.9 and Lemma 4.2.16 can both be given more generally for types T in which variables X_i occur in either negative position or in positive position. However, since we restrict attention to strictly positive types, the current formulation suffices.

[29] Sub-expressions of syntactic (Kripke-)polynomial functors are called *ingredients* by, for example, Kupke [Kup06].

Inductive-Coinductive Reasoning

Here comes the argument
Here comes the argument
Here comes the argument
Folderol

— Fugazi, "Argument", 2001.

In the last chapter we have constructed two languages that allow us to program with mixed inductive-coinductive types. Moreover, we have established a common notion of observational equivalence on terms and, for the second language, singled out a class of observationally normalising terms. The goal of this chapter is to establish methods of reasoning about inductive-coinductive programs, in particular methods for proving that programs are observationally equivalent. We will study three quite different proof methods, each having its own advantages and disadvantages, as we will see.

In the remainder of this introduction we will discuss further the different proof methods that we advertised above. In Section 5.1, we obtain a proof technique for observational equivalence from a transition system that represents the observations that can be made on programs. We show that observational equivalence coincides with the canonical notion of bisimilarity that can be obtained by applying coalgebraic methods to the transition system. As such, the proof technique is readily usable but often requires the construction of annoyingly complicated relations. This can be improved drastically through the use of up-to techniques, which we will demonstrate as well. As an illustration of the bisimulation proof method, we show that the substream relation, which we defined in Example 5.1.13, is transitive. Lastly, we also discuss the possibility of characterising observational normalisation as a coinductive predicate.

The proof techniques we have established up to here are fairly easy to prove correct and can be used immediately. However, since they are cast in naive set theory, it becomes difficult to automatically show, for instance, that a relation is a bisimulation because the description of that relation can be arbitrarily complex. Moreover, already the need to come up with an invariant that describes a bisimulation is a huge burden and is unnecessary. In fact, the so-called *bisimulation game* allows one to prove or disprove bisimilarity of processes interactively. The only problem with plays of a bisimulation game is that their correctness is difficult to ensure in general because of global parity conditions that have to be checked. All three issues, automatic verification, upfront guessing and global correctness conditions, will be remedied by the syntactic proof systems for observational equivalence in Section 5.2.

The syntactic proof system in Section 5.2 allows us to prove formulas of a first-order predicate logic with an internal equality that models observational equivalence. Since the aim of the proof system is to allow for easy discovery of proof steps along the way, in contrast to having to guess a bisimulation relation upfront, we opt for a recursive proof system. Such a proof system allows us to refer back to proof steps that we encountered before. To ensure that recursive references give rise to well-defined proofs, we use the so-called later modality [Nak00], which allows us to control

where recursion steps can occur. In contrast to cyclic proof systems, our proof system allows proofs to be checked locally at every proof step. We describe this in detail in Section 5.2.

The two proof methods for observational equivalence that we discussed up to this point require some ingenuity in constructing proofs of program equivalences. In Section 5.3 we show that this is unavoidable, since observational equivalence is undecidable, as one would expect. More positively, we also establish there a fragment of the languages, on which observational equivalence is decidable.

Original Publication For the most part, the content of Section 5.1 has been presented in [BH16], but without the extensive example and the up-to technique for carrying out induction in a bisimulation proof in Section 5.1.3. In the same paper, also the results in Section 5.3 have been published. The logic $\mathbf{FOL}_{\blacktriangleright}$ in Section 5.2 is, on the other hand, completely new. However, in [Bas18a] a general approach for constructing a logic for coinductive predicates, which is based on the later modality, from a logic given in form of a fibration is presented. This subsumes, in principle, the development in Section 5.2, see the discussion [Bas18a].

5.1. Program Properties as Coinductive Predicates

Since observational equivalence is defined in terms of tests, the only way to prove that two programs are equivalent is so far by induction on tests. In this section we develop a set-based proof technique for observational equivalence, which is simpler and more intuitively usable than induction on tests. This proof technique is centred around the idea that observations on programs give rise to a labelled transition system on programs. We introduce this transition system in Section 5.1.1. Associated to such a transition system is the usual notion of bisimilarity. In Section 5.1.2 we show that observational equivalence coincides with this notion of bisimilarity. This allows us to prove that programs are observationally equivalent by establishing a bisimulation relation containing these terms, instead of having to proceed by induction on tests. We demonstrate the use of this proof method on an extensive example in Section 5.1.3.

The description of observational equivalence as coinductive predicate gives us a powerful proof technique. However, this proof technique is also very tedious to use in its basic form, which leads us to consider up-to techniques that can drastically reduce the complexity of equivalence proofs. We give in Section 5.1.2 several general up-to techniques, that can be used in any proof of observational equivalence. In the example in Section 5.1.3, we also develop an up-to technique that allows us to use induction inside bisimulation proofs.

5.1.1. Terms as Transition System

Towards coinductive proof principles for observational equivalence and observational normalisation, the first step is to establish a labelled transition system (LTS), in which the states are terms of $\lambda\mu\nu{=}$. The labels in this transition system are the observations that are given by the modalities of the testing logic in Definition 4.1.12. Given such a label, the successors of a term t are all those terms that can be reached after making the observation given by that label.

The transition system on terms is defined by composing two relations. One relation, denoted by $\longrightarrow\!\triangleright$, represents the observations that can be made. For example, a possible observation on a term t of type $A \times B$ is the first projection, which leads us to have a transition $t \xrightarrow{\pi_1}\!\triangleright\ \pi_1\, t$ in the transition

system. On an inductive type like $A_1 + A_2$ we can observe a constructor on terms of the form $\kappa_i\ s$ for some term s of type A_i, in which case there is a transition $\kappa_1\ s \xrightarrow{\kappa_i} s$. Note that such a transition is only possible on terms in weak head normal form (Lemma 3.2.34). Thus, to avoid that the transition system gets stuck, we also need to take reductions into account. We can achieve this by adding transitions to all terms that are reachable by reductions after making an observation on a term of coinductive type and before observing terms of inductive type. The transition relation we obtain from combining $\longrightarrow\!\triangleright$ and reductions is denoted by $\longrightarrow\!\triangleright\!\triangleright$.

The observation relation $\xrightarrow{l}\!\triangleright_A \subseteq \Lambda^=(A) \times \Lambda^=(A)$ is given by the following rules. Note that the relation is annotated with the type of the term on the left-hand side. This is necessary to decide when to make a reduction step in the definition of $\longrightarrow\!\triangleright\!\triangleright$ below.

$$\frac{t \in \Lambda^=(A_1 \times A_2) \qquad i \in \{1,2\}}{t \xrightarrow{\pi_i}\!\triangleright_{A_1 \times A_2}\ t.\mathrm{pr}_i} \qquad \frac{t \in \Lambda^=(\nu X.\, XA)}{t \xrightarrow{\xi}\!\triangleright_{\nu X.\, A}\ t.\mathrm{out}} \qquad \frac{t \in \Lambda^=(A \to B) \qquad u \in \mathbf{ON}_A}{t \xrightarrow{u}\!\triangleright_{A \to B}\ t\ u}$$

$$\frac{t \in \Lambda^=(A_i)}{\kappa_i\ t \xrightarrow{\kappa_i}\!\triangleright_{A_1 + A_2}\ t} \qquad \frac{t \in \Lambda^=(A[\mu X.\, A/X])}{\alpha\ t \xrightarrow{\alpha}\!\triangleright_{\mu X.\, A}\ t}$$

The labelled transition relation $\xrightarrow{l}\!\triangleright\!\triangleright_A \subseteq \Lambda^=(A) \times \Lambda^=(A)$ is then defined by composing the labelled observation relation $\longrightarrow\!\triangleright$ with the reduction relation, as follows. Here ; denotes relation composition and $\longrightarrow\!\!\!\!\rightarrow$ is the reflexive, transitive closure of the reduction relation $\longrightarrow$, see Definition 3.2.14.[30]

$$\xrightarrow{l}\!\triangleright\!\triangleright_A = \begin{cases} \longrightarrow\!\!\!\!\rightarrow\ ;\ \xrightarrow{l}\!\triangleright_A & \text{if } A \text{ is inductive} \\ \xrightarrow{l}\!\triangleright_A\ ;\ \longrightarrow\!\!\!\!\rightarrow & \text{if } A \text{ is coinductive} \end{cases}$$

If the type A is clear from the context, we shall drop the subscript of this relation.

Let us give some examples of transitions that can be taken in the above LTS. The first example illustrates the need for reduction steps in between observations.

Example 5.1.1. Let A, B, C be types, and s and t be terms with $s : A$ and $t : C$, so that we can form $\langle \kappa_1\ s, t \rangle : (A + B) \times C$. There is now an observation step

$$\langle \kappa_1\ s, t \rangle \xrightarrow{\pi_1}\!\triangleright\ \langle \kappa_1\ s, t \rangle.\mathrm{pr}_1$$

to a term of type $A + B$. However, we cannot make any further observations, since the term on the right-hand side has no constructor in head position. So the observation relation gets stuck. This leads us to use the reduction relation to obtain

$$\langle \kappa_1\ s, t \rangle.\mathrm{pr}_1 \longrightarrow\!\!\!\!\rightarrow \kappa_1\ s,$$

from where we can again take an observation step

$$\kappa_1\ s \xrightarrow{\kappa_1}\!\triangleright\ s.$$

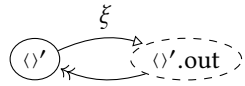
Thus the following sequence of steps is possible in the transition system.

$$\langle \kappa_1\ s, t \rangle \xrightarrow{\pi_1}\!\triangleright\!\triangleright\ \kappa_1\ s \xrightarrow{\kappa_1}\!\triangleright\!\triangleright\ s. \qquad \blacktriangleleft$$

Let us give some further and more concrete examples of possible transitions.

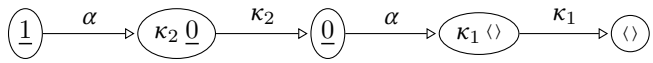
Example 5.1.2. We give several snapshots of the transition system, both on inductive and coinductive types. A full circle represents hereby a state of the transition system, whereas a dashed circle represents intermediate states.

1. The term $\langle\rangle'$ of type $\mathbf{1}'$ was defined in Example 3.2.7 so that $\langle\rangle'.\mathrm{out} \longrightarrow\!\!\!\!\rightarrow \langle\rangle'$. This leads to the loop displayed in the following diagram.



   It is noteworthy that we cannot extract any further information about $\langle\rangle'$, since ξ is the only observation we can make on $\langle\rangle'$. In fact, this is what allowed us to show that $\mathbf{1}'$ is a pseudo-terminal object in Theorem 4.2.14.

2. We display some transitions that start in the encoding $\underline{1} : \mathrm{Nat}$ of 1. Recall that $\mathrm{Nat} = \mu X.\, \mathbf{1} + X$, and that the encoding was defined in Example 3.1.7 by $\underline{0} = \alpha\,(\kappa_1\,\langle\rangle)$ and $\underline{1} = \alpha\,(\kappa_2\,\underline{0})$.



3. Of course, there are also terms in $\lambda\mu\nu{=}$ that have no outgoing transition because they do not have a WHNF. For example, recall from Example 3.2.21 that we can define for any type A a term $\Omega_A = \mathbf{rlet}\ f : \mathrm{Nat} = \{\cdot \mapsto f\}\ \mathbf{in}\ f$, which has no normalising reductions. If we now instantiate this term for an inductive type, like for instance Nat, then Ω_{Nat} has no WHNF. This is to say that there is no term $t : \mathbf{1} + \mathrm{Nat}$ with $\Omega_{\mathrm{Nat}} \longrightarrow\!\!\!\!\rightarrow \alpha\, t$. Thus there is no transition originating from Ω_{Nat} in the LTS.

4. The final example is concerned with terms of coinductive types. Recall that we have defined in Example 3.2.9 a function term $H\colon \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega$. So for any observationally normalising term u of type Nat^ω there is a step $H \xrightarrow{\underline{u}}\!\!\!\!\triangleright\ H\,u$ possible. In particular, we can use a term $\mathrm{nats} \in \mathbf{ON}_{\mathrm{Nat}^\omega}$ that represents the stream $(0, 1, 2, \ldots)$ of natural numbers, see Example 3.1.8. We show in the diagram in Figure 5.1 a part of the transition system, starting at H. It is important to note that we only display a fraction of the possible transitions, as there is, for example, a transition originating from H for every term in $u \in \mathbf{ON}_{\mathrm{Nat}^\omega}$ and to every term t with $H\,u \longrightarrow\!\!\!\!\rightarrow t$. ◀

We now give a description of the labelled transition relation $(\Lambda^{=}, \longrightarrow\!\!\!\!\triangleright\!\!\triangleright)$ as a coalgebra on terms. This will allow us to apply coalgebraic techniques to define bisimilarity on this labelled transition system. An integral part of the transition relation $\longrightarrow\!\!\!\!\triangleright\!\!\triangleright$ on terms is that it is type-driven, which in turn enforces that only terms of the same type can be related by a bisimulation relation. We implement this typing constraint in a coalgebra, which represents the transition system, by using the set family of terms indexed by their types. Thus, we consider $\Lambda^{=}$ as an element in $\mathbf{Set}^{\mathrm{Ty}}$, the category of set families indexed by types, see Example 2.4.3, and the transition system as a coalgebra for a functor $F\colon \mathbf{Set}^{\mathrm{Ty}} \to \mathbf{Set}^{\mathrm{Ty}}$. This functor describes the branching type of the transition system: for $X \in \mathbf{Set}^{\mathrm{Ty}}$ and $C \in \mathrm{Ty}$, $F(X)_C$ is the set that can be inspected by tests on type C, see the rules for $\longrightarrow\!\!\!\!\triangleright$ above. Formally F and the coalgebra representing the transition system are given as follows.

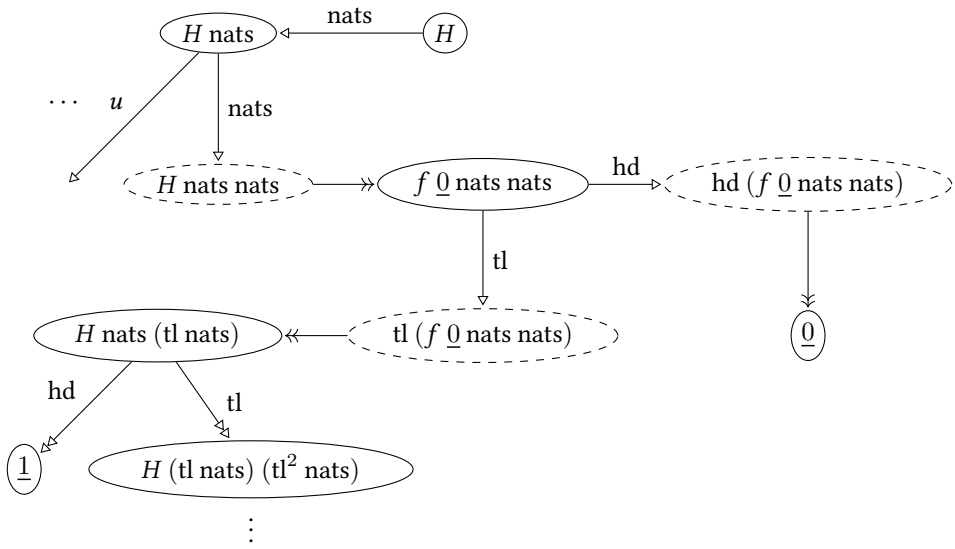


Figure 5.1.: A few transitions in the LTS on terms originating at H

Definition 5.1.3. Let $F\colon \mathbf{Set}^{\mathrm{Ty}} \to \mathbf{Set}^{\mathrm{Ty}}$ be the functor given by

$$F(X)_C = \begin{cases} \mathbf{1}, & C = \mathbf{1} \\ \coprod_{i\in\{1,2\}} \mathcal{P}(X_{A_i}) + \mathbf{1}, & C = A_1 + A_2 \\ \mathcal{P}(X_{A[\mu X.\, A/X]}) + \mathbf{1}, & C = \mu X.\, A \\ \prod_{i\in\{1,2\}} \mathcal{P}(X_{A_i}), & C = A_1 \times A_2 \\ \mathcal{P}(X_{A[\nu X.\, A/X]}), & C = \nu X.\, A \\ \mathcal{P}(X_B)^{\mathbf{ON}_A}, & C = A \to B \end{cases}$$

where $\mathcal{P}(-)$ is the covariant powerset functor, and for a set U, $(-)^U$ is the function space functor. F acts on morphisms in the obvious way. We define the coalgebra on terms $\delta : \Lambda^{=} \to F(\Lambda^{=})$ by

$$\begin{aligned} \delta_{\mathbf{1}}(t) &= * \\ \delta_{A_1+A_2}(t) &= \begin{cases} \iota_i(\{t' : A_i \mid t \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t'\}), & \exists t'.\, t \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t' \\ *, & \text{otherwise} \end{cases} \\ \delta_{\mu X.\, A}(t) &= \begin{cases} \{t' : A \mid t \longrightarrow\!\!\!\!\!\rightarrow \alpha\, t'\}, & \exists t'.\, t \longrightarrow\!\!\!\!\!\rightarrow \alpha\, t' \\ *, & \text{otherwise} \end{cases} \\ \delta_{A_1\times A_2}(t)(i) &= \{t' : A_i \mid \pi_i\, t \longrightarrow\!\!\!\!\!\rightarrow t'\} \\ \delta_{\nu X.\, A}(t) &= \{t' : A \mid \xi\, t \longrightarrow\!\!\!\!\!\rightarrow t'\} \\ \delta_{A\to B}(t)(u) &= \{t' : C \mid t\, u \longrightarrow\!\!\!\!\!\rightarrow t'\} \end{aligned}$$

Note that uniqueness of WHNFs, Lemma 3.2.33, ensures that δ is well-defined on sum types.[31] ◀

Of course, we would expect the transition system $\longrightarrow\!\!\!\!\triangleright\!\!\triangleright$ and the coalgebra δ to be equivalent. To show this, let us introduce another functor $G\colon \mathbf{Set}^{\mathrm{Ty}} \to \mathbf{Set}^{\mathrm{Ty}}$. In the definition of G, we use the constructors and projections as labels in the index of the corresponding coproducts and products. The reason for using F rather than G in the course of this chapter is that G is complicated. However, it simplifies the construction of a coalgebra from the transition system.

$$G(X)_C = \begin{cases} \coprod_{\kappa_i \in \{\kappa_1, \kappa_2\}} \mathcal{P}(X_{A_i}) + \mathbf{1}, & C = A_1 + A_2 \\ \mathbf{1}, & C = \mathbf{1} \\ \coprod_{l \in \{\alpha\}} \mathcal{P}(X_{A[\mu X.\, A/X]}) + \mathbf{1}, & C = \mu X.\, A \\ \prod_{\pi_i \in \{\pi_1, \pi_2\}} \mathcal{P}(X_{A_i}), & C = A_1 \times A_2 \\ \prod_{l \in \{\xi\}} \mathcal{P}(X_{A[\nu X.\, A/X]}), & C = \nu X.\, A \\ \prod_{u \in \mathbf{ON}_A} \mathcal{P}(X_B), & C = A \to B \end{cases}$$

Note that for fixed point types there is only one label, thus $G(X)_C$ is in these cases isomorphic to $F(X)_C$. Similarly, we can rename the labels κ_i in the sum case to 1 and 2 and obtain thereby an isomorphism $G(X)_{A_1+A_2} \cong F(X)_{A_1+A_2}$. Analogously, $G(X)_C$ and $F(X)_C$ are also isomorphic for all other types C. Since these isomorphisms are natural in X, we obtain an isomorphism $F \cong G$. We can now associate a G-coalgebra $(\Lambda^=, d)$ with the rule-based transition system:

$$\begin{aligned} &d\colon \Lambda^= \to G(\Lambda^=) \\ &d_A(t) = \iota_\ell\big(\{t' \mid t \xrightarrow{\ell}\!\!\!\!\triangleright\!\!\triangleright_A t'\}\big), && A \text{ inductive and } \exists t'.\, t \xrightarrow{\ell}\!\!\!\!\triangleright\!\!\triangleright_A t' \\ &d_A(t) = *, && A \text{ inductive and } \neg(\exists t'.\, t \xrightarrow{\ell}\!\!\!\!\triangleright\!\!\triangleright_A t') \\ &d_A(t)(\ell) = \{t' \mid t \xrightarrow{\ell}\!\!\!\!\triangleright\!\!\triangleright_A t'\}, && A \text{ coinductive} \end{aligned}$$

Note that d is again well-defined on inductive types due to uniqueness of WHNFs. It is hopefully clear that d is a G-coalgebraic representation of the relation $\longrightarrow\!\!\!\!\triangleright\!\!\triangleright$. We now show that the F-coalgebra δ and the G-coalgebra d essentially define the same transition system.

Proposition 5.1.4. *The transition systems defined by the F-coalgebra δ and by the G-coalgebra d are equivalent in the sense that the following diagram commutes.*

$$\begin{array}{ccc} & \overset{\delta}{\nearrow} & F(\Lambda^=) \\ \Lambda^= & & \uparrow \cong \\ & \underset{d}{\searrow} & G(\Lambda^=) \end{array}$$

Proof. Let f be the isomorphism $G(\Lambda^=) \to F(\Lambda^=)$. We proceed by case distinction in $A \in \mathrm{Ty}$ to show that $\delta_A = f_A \circ d_A$.

- For sum types $A_1 + A_2$, we have for every $t : A_1 + A_2$ that either there is $i \in \{1, 2\}$, such that

$$\begin{aligned}
f_{A_1+A_2}(d_{A_1+A_2}(t)) &= f_{A_1+A_2}(\iota_{\kappa_i}\{t' \mid t \xrightarrow{\kappa_i}\!\!\triangleright\!\!\triangleright_{A_1+A_2} t'\}) && \text{by } \exists t'.\, t \xrightarrow{\kappa_i}\!\!\triangleright\!\!\triangleright_{A_1+A_2} t' \\
&= \iota_i\{t' \mid t \xrightarrow{\kappa_i}\!\!\triangleright\!\!\triangleright_{A_1+A_2} t'\} && \text{by definition of } f \\
&= \iota_i\{t' \mid t \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t' \xrightarrow{\kappa_i}\!\!\triangleright_{A_1+A_2} t'\} && \text{by } \exists t'.\, t \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t' \xrightarrow{\kappa_i}\!\!\triangleright_{A_1+A_2} t' \\
&= \iota_i\{t' \mid t \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t'\} && \text{by } \exists t'.\, t \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t' \\
&= \delta_{A_1+A_2}(t),
\end{aligned}$$

  or $f_{A_1+A_2}(d_{A_1+A_2}(t)) = f_{A_1+A_2}(*) = * = \delta_{A_1+A_2}(t)$.

- For function types $A \to B$ we have for every $u \in \mathbf{ON}_A$

$$\begin{aligned}
f_{A\to B}(d_{A\to B}(t))(u) &= d_{A\to B}(t)(u) \\
&= \{t' \mid t \xrightarrow{u}\!\!\triangleright\!\!\triangleright_{A\to B} t'\} \\
&= \{t' \mid t \xrightarrow{u}\!\!\triangleright_{A\to B} t\, u \longrightarrow\!\!\!\!\rightarrow t'\} \\
&= \{t' \mid t\, u \longrightarrow\!\!\!\!\rightarrow t'\} \\
&= \delta_{A\to B}(t)(u).
\end{aligned}$$

All other cases are treated analogously. Thus $\delta = f \circ d$, as required. □

5.1.2. Observational Equivalence as Bisimilarity

We will now establish that observational equivalence coincides with the usual notion of bisimilarity on the F-coalgebra δ. Since observational equivalence is defined as logical equivalence by the testing logic given in Section 4.1, this means that the testing logic is *adequate* (or *expressive*) for bisimilarity on the term coalgebra δ.

We define bisimulations on δ vial a relation lifting of the functor F (Definition 2.5.7). We now introduce the necessary terminology.

Definition 5.1.5. Given an index set I, the *category of relations* over families indexed by I is defined as follows. Here, $\sqsubseteq$ is the index-wise set inclusion, see Example 2.4.3.

$$\mathrm{Rel}^I = \begin{cases} \text{objects:} & (X, R) \text{ with } X, R \in \mathbf{Set}^I \text{ and } R \sqsubseteq X \times X \\ \text{morphisms:} & f\colon (X, R) \to (Y, S) \text{ is given by } f\colon X \to Y, \text{ s.t. } (f \times f)(R) \sqsubseteq S \end{cases}$$

This category is the total category of a fibration $P^I\colon \mathrm{Rel}^I \to \mathbf{Set}^I$ induced by $P^I(X, R) = X$. Each fibre of P^I is given by

$$\mathrm{Rel}^I_X = \begin{cases} \text{objects:} & R \text{ with } R \sqsubseteq X \times X \\ \text{morphisms:} & R \to S \iff R \sqsubseteq S \end{cases},$$

which forms a complete lattice, see Definition 2.5.8. Reindexing on Rel^I along $f\colon X \to Y$ is described in terms of taking preimages:

$$\begin{aligned} &f^*\colon \mathrm{Rel}^I_Y \to \mathrm{Rel}^I_X \\ &f^*(R) = \left\{(f_i \times f_i)^{-1}(R_i)\right\}_{i\in I} \end{aligned}$$

We will omit I in the superscripts of Rel and P, if it is understood from the context. ◀

Recall from Definition 2.5.7 that a lifting H to Rel^I of a functor $L\colon \mathbf{Set}^I \to \mathbf{Set}^I$ restricts for each $X \in \mathbf{Set}^I$ to a functor $H_X\colon \mathrm{Rel}^I_X \to \mathrm{Rel}^I_{LX}$ on fibres, and that we can model for a given coalgebra $c\colon X \to LX$ relational properties of c as final coalgebras for the functor $H_c\colon \mathrm{Rel}^I_X \to \mathrm{Rel}^I_X$, where

$$H_c \coloneqq c^* \circ H_X.$$

In particular, it is well-known that bisimulations for $\delta\colon \Lambda^= \to F(\Lambda^=)$ are exactly the post-fixed points of $\overline{F}_\delta$, where $\overline{F}$ is the canonical relation lifting of F, see [Sta11].

Definition 5.1.6. The *canonical relation lifting* $\overline{F}\colon \mathrm{Rel}^{\mathrm{Ty}} \to \mathrm{Rel}^{\mathrm{Ty}}$ of F is given by

$$\overline{F}(X,R) = (FX, \langle F(\pi_1), F(\pi_2)\rangle(FR)).$$

Let $\Phi\colon \mathrm{Rel}^{\mathrm{Ty}}_{\Lambda^=} \to \mathrm{Rel}^{\mathrm{Ty}}_{\Lambda^=}$ be the monotone map

$$\Phi \coloneqq \overline{F}_\delta = \delta^* \circ F_X.$$

An *observational bisimulation* is a Φ-invariant (or F-bisimulation on δ), see Definition 2.5.7, that is, a relation $R \in \mathrm{Rel}^{\mathrm{Ty}}_{\Lambda^=}$ on terms, such that

$$R \sqsubseteq \Phi(R).$$ ◀

Notation 5.1.7. In the remainder of this chapter, we will only use type-indexed relations. So to keep the notation simple, we omit the superscript in $\mathrm{Rel}^{\mathrm{Ty}}$ and just write Rel. ◀

Let us describe the canonical lifting of F and the monotone operator Φ, more concretely. Suppose, we are given $R \in \mathrm{Rel}_{\Lambda^=}$. Then for all $(U_1, U_2) \in F(\Lambda^=)_A \times F(\Lambda^=)_A$, we have

$$(U_1, U_2) \in \overline{F}(R)_A \iff \exists U \in F(R)_A.\, U_1 = (F\pi_1)_A(U) \text{ and } U_2 = (F\pi_2)_A(U).$$

Thus we have

$$\Phi(R)_A = \{(t_1, t_2) \in \Lambda^=(A)^2 \mid (\delta_A(t_1), \delta_A(t_2)) \in \overline{F}(R)_A\}.$$

Let us now give an explicit description of the properties that a relation has to fulfil in order to be an observational bisimulation. The easiest case to spell out the requirements on an observational bisimulation is if we do not have to take explicit reduction steps into account. This is captured in the following lemma, for which we recall that $\equiv$ denotes the convertibility relation, Definition 3.2.14.

Lemma 5.1.8. *Let $R \in \mathrm{Rel}_{\Lambda^=}$ and assume that R is $\equiv$-closed, that is, for all $(s,t) \in R_A$, $s \equiv u$ and $t \equiv v$, the pair (u,v) is also in R_A. Then R is an observational bisimulation if and only if the following conditions are fulfilled.*

- *If $(s,t) \in R_{A_1+A_2}$, then either neither of s and t has a WHNF, or there is an $i \in \{1,2\}$ with $s \equiv \kappa_i\, s'$, $t \equiv \kappa_i\, t'$ and $(s',t') \in R_{A_i}$.*
- *If $(s,t) \in R_{\mu X.\, A}$, then either neither of s and t has a WHNF, or $s \equiv \alpha\, s'$, $t \equiv \alpha\, t'$ and $(s',t') \in R_{A[\mu X.\, A/X]}$.*

- *If* $(s,t) \in R_{A_1 \times A_2}$, *then* $(s.\mathrm{pr}_1, t.\mathrm{pr}_1) \in R_{A_1}$ *and* $(s.\mathrm{pr}_2, t.\mathrm{pr}_2) \in R_{A_2}$.
- *If* $(s,t) \in R_{A \to B}$, *then for all* $u \in \mathbf{ON}_A$ *we must have* $(s\ u, t\ u) \in R_B$.
- *If* $(s,t) \in R_{\nu X.\, A}$, *then* $(s.\mathrm{out}, t.\mathrm{out}) \in R_{A[\nu X.\, A/X]}$.

Proof. Let $R \in \mathrm{Rel}_{\Lambda^=}$ be $\equiv$-closed. We need to show that $R \sqsubseteq \Phi(R)$ if and only if the above conditions hold. So let A be a type and $(s,t) \in R_A$. That $(s,t) \in \Phi(R)_A$ is equivalent to the corresponding condition for each type $A \in \mathrm{Ty}$ is shown by case distinction on A. We only demonstrate these equivalences on function types and least fixed point types, since the other cases are dealt with analogously.

- Suppose that $(s,t) \in \Phi(R)_{A \to B}$. We need to show that this is equivalent to saying that $\forall u \in \mathbf{ON}_A.\, (s\ u, t\ u) \in R_B$. First, we can rewrite the condition $(s,t) \in \Phi(R)_{A \to B}$ to clauses that are familiar from labelled transition systems without using the fact that R is $\equiv$-closed:

$$
\begin{aligned}
& (s,t) \in \Phi(R)_{A \to B} \\
\iff & (\delta_{A \to B}(s), \delta_{A \to B}(t)) \in \overline{F}(R)_{A \to B} \\
\iff & \exists U \in \mathcal{P}(R_B)^{\mathbf{ON}_A}.\, \delta_{A \to B}(s) = (F\pi_1)(U) \wedge \delta_{A \to B}(t) = (F\pi_2)(U) \\
\iff & \exists U \in \mathcal{P}(R_B)^{\mathbf{ON}_A}.\, \forall u \in \mathbf{ON}_A. \\
& \quad \forall s' \in \delta_{A \to B}(s)(u).\, \exists t'.\, (s',t') \in U(u) \wedge \forall (s',t') \in U(u).\, s' \in \delta_{A \to B}(s)(u) \\
& \quad \wedge\, \forall t' \in \delta_{A \to B}(t)(u).\, \exists s'.\, (s',t') \in U(u) \wedge \forall (s',t') \in U(u).\, t' \in \delta_{A \to B}(t)(u) \\
\iff & \forall u \in \mathbf{ON}_A.\, \forall s' \in \delta_{A \to B}(s)(u).\, \exists t' \in \delta_{A \to B}(t)(u).\, (s',t') \in R_B \\
& \qquad \wedge \forall t' \in \delta_{A \to B}(t)(u).\, \exists s' \in \delta_{A \to B}(s)(u).\, (s',t') \in R_B \\
\iff & \forall u \in \mathbf{ON}_A.\, \forall s'.\, (s\ u \longrightarrow\!\!\!\!\!\rightarrow s') \implies (\exists t'.\, t\ u \longrightarrow\!\!\!\!\!\rightarrow t' \wedge (s',t') \in R_B) \\
& \qquad \wedge \forall t'.\, (t\ u \longrightarrow\!\!\!\!\!\rightarrow t') \implies (\exists s'.\, s\ u \longrightarrow\!\!\!\!\!\rightarrow s' \wedge (s',t') \in R_B)
\end{aligned}
$$

  Since R is closed under conversions, we obtain then the desired property:

$$
\begin{aligned}
& (s,t) \in \Phi(R)_{A \to B} \\
\iff & \forall u \in \mathbf{ON}_A.\, \forall s'.\, (s\ u \longrightarrow\!\!\!\!\!\rightarrow s') \implies (\exists t'.\, t\ u \longrightarrow\!\!\!\!\!\rightarrow t' \wedge (s',t') \in R_B) && \text{see above} \\
& \qquad \wedge \forall t'.\, (t\ u \longrightarrow\!\!\!\!\!\rightarrow t') \implies (\exists s'.\, s\ u \longrightarrow\!\!\!\!\!\rightarrow s' \wedge (s',t') \in R_B) \\
\iff & \forall u \in \mathbf{ON}_A.\, \forall s'.\, (s\ u \longrightarrow\!\!\!\!\!\rightarrow s') \implies (s',t\ u) \in R_B && R \text{ } \equiv\text{-closed} \\
& \qquad \wedge \forall t'.\, (t\ u \longrightarrow\!\!\!\!\!\rightarrow t') \implies (s\ u,t') \in R_B \\
\iff & \forall u \in \mathbf{ON}_A.\, (s\ u, t\ u) \in R_B \wedge (s\ u, t\ u) \in R_B && R \text{ } \equiv\text{-closed} \\
\iff & \forall u \in \mathbf{ON}_A.\, (s\ u, t\ u) \in R_B.
\end{aligned}
$$

  Therefore, we have that $(s,t) \in \Phi(R)_{A \to B} \iff \forall u \in \mathbf{ON}_A.\, (s\ u, t\ u) \in R_B$, and thus, on function types, $R \subseteq \Phi(R)_{A \to B}$ is equivalent to

$$\forall (s,t).\, (s,t) \in R \implies \forall u \in \mathbf{ON}_A.\, (s\ u, t\ u) \in R_B.$$

- For the case of least fixed point types we proceed similarly:

$$\begin{aligned}
&(s,t) \in \Phi(R)_{\mu X.\,A}\\
\iff &(\delta_{\mu X.\,A}(s), \delta_{\mu X.\,A}(t)) \in \overline{F}(R)_{\mu X.\,A}\\
\iff &\delta_{\mu X.\,A}(s) = * = \delta_{\mu X.\,A}(t) \vee\\
&\big(\forall s' \in \delta_{\mu X.\,A}(s).\, \exists t' \in \delta_{\mu X.\,A}(t).\, (s',t') \in R_{A[\mu X.\,A/X]}\\
&\wedge\, \forall t' \in \delta_{\mu X.\,A}(t).\, \exists s' \in \delta_{\mu X.\,A}(s).\, (s',t') \in R_{A[\mu X.\,A/X]}\big)\\
\iff &\text{neither } s \text{ nor } t \text{ has a WHNF} \vee\\
&\big(\forall s'.\, (s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s') \implies (\exists t'.\, t \longrightarrow\!\!\!\!\!\rightarrow \alpha\, t' \wedge (s',t') \in R_{A[\mu X.\,A/X]})\\
&\wedge\, \forall t'.\, (t \longrightarrow\!\!\!\!\!\rightarrow \alpha\, t') \implies (\exists s'.\, s \longrightarrow\!\!\!\!\!\rightarrow \alpha\, s' \wedge (s',t') \in R_{A[\mu X.\,A/X]})\big)
\end{aligned}$$

  From this, we obtain

$$\begin{aligned}
&(s,t) \in \Phi(R)_{A\to B}\\
\iff &\text{neither } s \text{ nor } t \text{ has a WHNF} \vee\\
&\exists s', t'.\, s \equiv \alpha\, s' \wedge t \equiv \alpha\, t' \wedge (s',t') \in R_{A[\mu X.\,A/X]},
\end{aligned}$$

  just as we did in the case of function types. □

Let us now formulate the main result of this section.[32]

Theorem 5.1.9. *Observational equivalence is the largest observational bisimulation.*

Proof. The proof consists of two steps: We need to show that $\equiv_{\text{obs}}$ is actually an observational bisimulation and that it is the largest one with respect to $\sqsubseteq$. We begin by showing that $\equiv_{\text{obs}}$ is an observational bisimulation, thus we need to show that $\equiv_{\text{obs}} \sqsubseteq \Phi(\equiv_{\text{obs}})$. So we let $t_1 \equiv^A_{\text{obs}} t_2$ for some type A and show that $(t_1, t_2) \in \Phi(\equiv_{\text{obs}})_A$.

- For $A = \mathbf{1}$, the proof is trivial.
- For $A = A_1 + A_2$, we distinguish three cases.
  i) The term t_1 has a WHNF, i.e., there are $i \in \{1,2\}$ and t_1' with $t_1 \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t_1'$. But then there is a t_2' with $t_2 \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t_2'$, for otherwise one of the tests $[\top, \bot]$ or $[\bot, \top]$ would distinguish t_1 and t_2, contradicting $t_1 \equiv_{\text{obs}} t_2$. Thus t_2 has a WHNF, too.

     We show that for all t_1' and t_2' with $t_k \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t_k'$ for all $k \in \{1,2\}$ that $t_1' \equiv_{\text{obs}} t_2'$ must hold. In the case $i = 1$, assume there is a test φ distinguishing t_1' and t_2'. Then $[\varphi, \bot]$ distinguishes t_1 and t_2, contradicting $t_1 \equiv_{\text{obs}} t_2$. Thus we must have $t_1' \equiv_{\text{obs}} t_2'$. The case of $i = 2$ is symmetric. We put $U = \big\{(t_1', t_2') \;\big|\; \forall k \in \{1,2\}.\, t_k \longrightarrow\!\!\!\!\!\rightarrow \kappa_i\, t_k'\big\}$ and by the two arguments above, we have $\iota_i(U) \in F(\equiv_{\text{obs}})_A$ and $\delta(t_k) = \iota_i(\pi_k(X))$ for all $k \in \{1,2\}$, thus $(t_1, t_2) \in \Phi(\equiv_{\text{obs}})_A$.
  ii) Conversely, if t_2 has a WHNF, then $(t_1, t_2) \in \Phi(\equiv_{\text{obs}})_A$ using a symmetric argument.
  iii) If neither t_1 nor t_2 has a WHNF, then $\delta(t_k) = *$ for $k = 1, 2$ and, since $(*, *) \in \overline{F}(\equiv_{\text{obs}})_A$, we have that $(t_1, t_2) \in \Phi(\equiv_{\text{obs}})_A$.

- The case $A = \mu X.\, B$ is proved analogously.
- If $A = B \to C$, we have for each $u \in \mathbf{ON}_B$ and t_k' with $t_k\, u \longrightarrow\!\!\!\!\rightarrow t_k'$ and $k \in \{1,2\}$ that $t_1' \equiv_{\mathrm{obs}} t_2'$. Assume that t_1' and t_2' could be distinguished by a test φ, then the test $[u]\,\varphi$ would distinguish t_1 and t_2, which contradicts $t_1 \equiv_{\mathrm{obs}} t_2$. Thus, $t_1' \equiv_{\mathrm{obs}} t_2'$ for all such t_1' and t_2'. Now let $X(u) = \{(t_1', t_2') \mid \forall k \in \{1,2\}.\, t_k\, u \longrightarrow\!\!\!\!\rightarrow t_k'\}$. By the above discussion, $X \in \overline{F}(\equiv_{\mathrm{obs}})_A$, and by definition, also $\pi_k(X(u)) = \delta(t_k)(u)$ for all $\{1,2\}$ and for all $u \in \mathbf{ON}_B$, hence $(t_1, t_2) \in \Phi(\equiv_{\mathrm{obs}})_A$.
- The cases for products and greatest fixed point types can be proved analogously.

Having proved that $\equiv_{\mathrm{obs}}$ is an observational bisimulation, it remains to prove that $\equiv_{\mathrm{obs}}$ is the largest such bisimulation. So let $R \in \mathrm{Rel}_{\Lambda^=}$ be such that $R \sqsubseteq \Phi(R)$. We show that $R \sqsubseteq \equiv_{\mathrm{obs}}$ by showing that for all $A \in \mathrm{Ty}$, all $(t_1, t_2) \in R_A$ and all $\varphi : A$ that $t_1 \vDash \varphi \Leftrightarrow t_2 \vDash \varphi$ holds. The proof is by induction on φ. The base case for the trivial tests $\top$ and $\bot$ is immediate. We prove the induction step by case distinction in A.

- If $A = A_1 + A_2$, then, since R is a bisimulation, either $\delta(t_1) = \delta(t_2) = *$ or there is an $i \in \{1,2\}$ such that

$$\forall t_1'.\, (t_1 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_1') \implies (\exists t_2'.\, t_2 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_2' \wedge (t_1', t_2') \in R_{A_i}) \tag{5.1}$$
$$\forall t_2'.\, (t_2 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_2') \implies (\exists t_1'.\, t_1 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_1' \wedge (t_1', t_2') \in R_{A_i}) \tag{5.2}$$

  We use this to show that t_1 and t_2 satisfy φ simultaneously. If there is $k \in \{1,2\}$ such that $\delta(t_k) = *$, then t_k does not have a WHNF and both t_1 and t_2 do not satisfy φ. Otherwise, we use that the test φ must be of the form $[\psi_1, \psi_2]$ with $\psi_i :\!\downarrow B_i$. By definition, $t_1 \vDash \varphi \Leftrightarrow t_1' \vDash \psi_i$ for $t_1 \equiv \kappa_i\, t_1'$. The existence of t_1' with $t_1 \equiv \kappa_i\, t_1'$ implies that there is a t_1'' with $t_1 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_1''$. By (5.1), there is a t_2' with $t_2 \longrightarrow\!\!\!\!\rightarrow \kappa_i\, t_2'$ and $(t_1'', t_2') \in R_{A_i}$. Finally, by the induction hypothesis, t_1' and t_2' simultaneously satisfy ψ_i, hence $t_2 \vDash \varphi$. Using (5.2), we prove analogously that $t_2 \vDash \varphi$ implies $t_1 \vDash \varphi$.
- We proceed analogously for least fixed point types.
- If $A = B \to C$, then the assumption $R_A \subseteq \Phi(R)_A$ says that for all $u \in \mathbf{ON}_B$

$$\forall t_1'.\, (t_1\, u \longrightarrow\!\!\!\!\rightarrow t_1') \implies (\exists t_2'.\, t_2\, u \longrightarrow\!\!\!\!\rightarrow t_2' \wedge (t_1', t_2') \in R_C) \tag{5.3}$$
$$\forall t_2'.\, (t_2\, u \longrightarrow\!\!\!\!\rightarrow t_2') \implies (\exists t_1'.\, t_1\, u \longrightarrow\!\!\!\!\rightarrow t_1' \wedge (t_1', t_2') \in R_C) \tag{5.4}$$

  This allows us to show that $t_1 \vDash \varphi \Leftrightarrow t_2 \vDash \varphi$. The test φ must be of the form $[u]\,\psi$ for some $u \in \mathbf{ON}_B$ and $\psi :\!\downarrow C$. By (5.3), there is a $t_2\, u \longrightarrow\!\!\!\!\rightarrow t_2'$ with $(t_1\, u, t_2') \in R_C$, which implies by induction that $t_1\, u$ and t_2' simultaneously satisfy ψ. Moreover, $t_2\, u \longrightarrow\!\!\!\!\rightarrow t_2'$ implies that $t_2\, u$ and t_2' simultaneously satisfy ψ. Hence $t_1\, u \vDash \psi \Rightarrow t_2\, u \vDash \psi$ and thus $t_1 \vDash \varphi \Rightarrow t_2 \vDash \varphi$. Analogously, we prove $t_2 \vDash \varphi \Rightarrow t_1 \vDash \varphi$ by (5.4), thus t_1 and t_2 simultaneously satisfy φ.
- The case for products and greatest fixed points is proved analogously.

Thus $\equiv_{\mathrm{obs}}$ is the largest observational bisimulation, hence it is final for Φ. □

Theorem 5.1.9 gives us a bisimulation proof principle for observational equivalence. That is to say, we can prove that two terms are observationally equivalent by establishing an observational bisimulation relation that contains these two terms. This is an improvement over proofs by induction on tests in two ways: First, bisimulation proofs are usually easier to give. More importantly though, we can enhance the bisimulation proof principle very easily with up-to techniques, see Section 2.5.1.

Up-to techniques allow us to overcome two major problems that the plain bisimulation proof principle from Theorem 5.1.9 has. Note that the transition system on terms has transitions to *all* terms to which reductions are possible. This forces observational bisimulations to be closed under reductions, which makes them unnecessarily complicated. Another problem is that we cannot easily use equivalences we have already proved. Both problems can be solved by closing a bisimulation candidate under observational equivalence.

We make use of Lemma 2.5.11 in the following to establish an up-to technique that solves the above problems of the bisimulation proof principle. For $R \in \mathrm{Rel}_{\Lambda^=}$, we denote by $R^{\equiv_{\mathrm{obs}}} \in \mathrm{Rel}_{\Lambda^=}$ the closure of R under observational equivalence, that is,

$$R^{\equiv_{\mathrm{obs}}} \coloneqq \quad \equiv_{\mathrm{obs}} ; R ; \equiv_{\mathrm{obs}} . \tag{5.5}$$

Since $\overline{F}$ is the canonical lifting of F to relations, we get from [Bon+14] that up-to bisimilarity, that is up-to $\equiv_{\mathrm{obs}}$, is Φ-compatible. Furthermore, we denote by Eq the diagonal relation and by $\sqcup$ the index-wise union. Then we have the following result.

Proposition 5.1.10. *The functor* $C^{\mathrm{obs}} \colon \mathrm{Rel}_{\Lambda^=} \to \mathrm{Rel}_{\Lambda^=}$ *given by*

$$C^{\mathrm{obs}}(R) = (R \sqcup \mathrm{Eq})^{\equiv_{\mathrm{obs}}}$$

is Φ*-compatible.*

Proof. Since $\mathrm{Eq} \sqsubseteq \Phi(\mathrm{Eq})$, we have that the constant functor mappping to Eq is compatible [Bon+14, Prop. 1]. By the same proposition, also $(-)^{\equiv_{\mathrm{obs}}}$ and $\sqcup$ are compatible. Hence the composition C^{obs} is compatible. □

Recall from Definition 2.5.10 that an observational bisimulation (i.e., a Φ-invariant) up to $p \colon \mathrm{Rel} \to \mathrm{Rel}$ is a relation R, such that $R \sqsubseteq \Phi(p(R))$. This allows us to simplify proofs of observational normalisation in the following. Note also that Φ itself is trivially Φ-compatible (Lemma 2.5.11). This might seem like a useless up-to technique. However, it becomes handy if a term only reduces under repeated observations. We will use this, for example, when proving properties of streams. These are usually given by specifying head and tail, which are combined observations: $.\mathrm{hd} = .\mathrm{out}.\mathrm{pr}_1$ and $.\mathrm{tl} = .\mathrm{out}.\mathrm{pr}_2$. Similarly, functions with multiple arguments usually reduce only if all arguments are provided.

Let us put the bisimulation proof principle and up-to techniques to work, starting with a simple example.

Example 5.1.11. In Example 3.2.19 we conjectured that select oddF alt would have the same behaviour as $\underline{1}^{\omega}$. We prove select oddF alt $\equiv_{\mathrm{obs}} \underline{1}^{\omega}$ now by showing that R with

$$\begin{aligned} R_{\mathrm{Nat}^{\omega}} &= \{(\text{select oddF alt}, \underline{1}^{\omega})\} \\ R_A &= \emptyset, \qquad\qquad A \neq \mathrm{Nat}^{\omega} \end{aligned}$$

is an observational bisimulation up to $\Phi \circ C^{\mathrm{obs}}$. Recall from Example 3.2.19 that

$$\begin{array}{l}(\text{select oddF alt}).\mathrm{hd} \equiv \underline{1} \\ (\text{select oddF alt}).\mathrm{tl} \equiv \text{select oddF alt}\end{array} \quad \text{and} \quad \begin{array}{l}(\underline{1}^{\omega}).\mathrm{hd} \equiv \underline{1} \\ (\underline{1}^{\omega}).\mathrm{tl} \equiv \underline{1}^{\omega}\end{array}$$

Since $(\underline{1}, \underline{1}) \in C^{\mathrm{obs}}(R)_{\mathrm{Nat}}$ and $(\text{select oddF alt}, \underline{1}^{\omega}) \in R^{\omega}_{\mathrm{Nat}} \subseteq C^{\mathrm{obs}}(R)_{\mathrm{Nat}^{\omega}}$, we have

$$((\text{select oddF alt}).\mathrm{out}, (\underline{1}^{\omega}).\mathrm{out}) \in \Phi(C^{\mathrm{obs}}(R))_{\mathrm{Nat}\times\mathrm{Nat}^{\omega}}$$

and hence

$$(\text{select oddF alt}, \underline{1}^{\omega}) \in \Phi(\Phi(C^{\mathrm{obs}}(R)))_{\mathrm{Nat}^{\omega}}.$$

So R is indeed an observational bisimulation up to $\Phi \circ C^{\mathrm{obs}}$. ◀

Example 5.1.11 shows that it requires some ingenuity to combine up-to techniques in the correct order for a proof to go through. So it would be useful if there was a way to introduce up-to techniques in a proof whenever necessary without the need to correctly specify the order of use beforehand. Fortunately, this can be achieved by using the *companion* of Φ that we introduced in Definition 2.5.12. For instance, the companion allows us to close a relation backwards under observations, as the following lemma shows.

Lemma 5.1.12. *Let $R \in \mathrm{Rel}_{\Lambda^=}$. The companion γ_Φ of Φ, see Definition 2.5.12, fulfils the following backwards closure conditions.*

- *If for all $u \in \mathbf{ON}_A$ we have $(s\,u, t\,u) \in \gamma_\Phi(R)_B$, then $(s,t) \in \gamma_\Phi(R)_{A\to B}$.*
- *If for all $i \in \{1,2\}$ we have $(\pi_i\,s, \pi_i\,t) \in \gamma_\Phi(R)_{A_i}$, then $(s,t) \in \gamma_\Phi(R)_{A_1\times A_2}$.*
- *If $(s.\mathrm{out}, t.\mathrm{out}) \in \gamma_\Phi(R)_{A[\nu X.\,A/X]}$, then $(s,t) \in \gamma_\Phi(R)_{\nu X.\,A}$.*

Proof. We only prove the case for functions, the other cases are analogous. Recall from Lemma 5.1.8 that $\forall u \in \mathbf{ON}_A.\,(s\,u, t\,u) \in \gamma_\Phi(R)_B$ is equivalent to $(s,t) \in \Phi(\gamma_\Phi(R))_{A\to B}$. Since Φ is compatible, we have $\Phi(\gamma_\Phi(R)) \sqsubseteq \gamma_\Phi(R)$. Thus, $(s,t) \in \gamma_\Phi(R)_{A\to B}$ follows. □

5.1.3. An Extensive Example: Transitivity of the Substream Relation

We will now study an elaborate example that demonstrates the use of the bisimulation proof method, up-to techniques and the companion. In this example, we show that the following substream relation is transitive.

Example 5.1.13. Recall that a stream selector was given as element of the type F, where $F = \nu X.\,\mu X.\,X + Y$. Let $s, t : \mathrm{Nat}^{\omega}$ be stream terms in We say that s is *substream* of t, written $s \leq t$, if there is a selector $x \in \mathbf{ON}_F$ such that $\mathrm{select}\,x\,s \equiv_{\mathrm{obs}} t$. Symbolically, we define

$$s \leq t \coloneqq \exists x \in \mathbf{ON}_F.\, s \equiv_{\mathrm{obs}} \mathrm{select}\,x\,t.$$ ◀

To prove transitivity, we observe that if $r \leq s$ by x_1 and $s \leq t$ by x_2, then

$$t \equiv_{\mathrm{obs}} \mathrm{select}\,x_2\,s \equiv_{\mathrm{obs}} \mathrm{select}\,x_2\,(\mathrm{select}\,x_1\,r). \tag{5.6}$$

We thus expect to be able to relate r and t directly by some form of composition of selectors. Indeed, we can compose selectors in $\lambda\mu\nu{=}$ as follows, where $F_\mu = \mu X.\, X + F$.

$$\begin{aligned}
&\mathrm{comp}\colon F \to F \to F\\
&(\mathrm{comp}\ x\ y).\mathrm{out} = \mathrm{comp}_\mu\ (x.\mathrm{out})\ (y.\mathrm{out})\\
&\mathrm{comp}_\mu\colon F_\mu \to F_\mu \to F_\mu\\
&\mathrm{comp}_\mu\ (\mathrm{pres}\ x)\ (\mathrm{pres}\ y) \quad = \mathrm{pres}\ (\mathrm{comp}\ x\ y)\\
&\mathrm{comp}_\mu\ (\mathrm{pres}\ x)\ (\mathrm{drop}\ v) \quad = \mathrm{drop}\ (\mathrm{comp}_\mu\ (x.\mathrm{out})\ v)\\
&\mathrm{comp}_\mu\ (\mathrm{drop}\ u)\ v \quad = \mathrm{drop}\ (\mathrm{comp}_\mu\ u\ v)
\end{aligned}$$

Note that comp is defined coinductively, whereas comp_μ is defined first by iteration on the first argument, and then in the base case of the first argument by iteration on the second argument. Let us introduce some more visually appealing notation for the composition of selectors:

$$y \bullet x \coloneqq \mathrm{comp}\ x\ y \qquad y \bullet_\mu x \coloneqq \mathrm{comp}_\mu\ x\ y.$$

If we can show that

$$\mathrm{select}\ (y \bullet x) \equiv_{\mathrm{obs}} \mathrm{select}\ y \circ \mathrm{select}\ x, \tag{5.7}$$

then we can continue the chain of equations in (5.6) with

$$t \equiv_{\mathrm{obs}} \mathrm{select}\ x_2\ (\mathrm{select}\ x_1\ r) \equiv_{\mathrm{obs}} \mathrm{select}\ (x_2 \bullet x_1)\ r.$$

Hence, the selector that witnesses $r \le t$ is given by composing the selectors that witness $r \le s$ and $s \le t$, respectively.

The rest of this subsection is devoted to proving (5.7). A first step towards this is to prove an induction principle that allows us to prove observational equivalence of maps out of F_μ. This requires the characterisation of the elements in F_μ, which is the content of the following lemma.

Lemma 5.1.14. *For all $u \in \mathbf{ON}_{F_\mu}$ there are $n \in \mathbb{N}$ and $x \in \mathbf{ON}_F$, such that $u \equiv \mathrm{drop}^n\ (\mathrm{pres}\ x)$.*

Proof. Since $u \in \mathbf{ON}_{F_\mu} \subseteq \mathbf{SN}_{F_\mu}$, by Lemma 3.2.34 there is an $i \in \{1, 2\}$ and an observationally normalising u' with $u \longrightarrow\!\!\!\!\!\rightarrow \alpha\ (\kappa_i\ u')$. If $i = 1$, then $\alpha\ (\kappa_i\ u') = \mathrm{pres}\ u'$, thus $u \equiv \mathrm{drop}^0\ (\mathrm{pres}\ u')$. Hence, the claim holds with $x = u'$. Otherwise, if $i = 2$, then $\alpha\ (\kappa_i\ u') = \mathrm{drop}\ u'$. As u' is a finite term and strongly normalising, we get by induction on the reduction steps towards a normal form of u' that there are $x \in \mathbf{ON}_F$ and $k \in \mathbb{N}$, such that $u' \equiv \mathrm{drop}^k\ (\mathrm{pres}\ x)$. Hence, the claim holds in this case with $n = k + 1$, that is, we have $u \equiv \mathrm{drop}^{k+1}\ (\mathrm{pres}\ x)$. □

Given the characterisation of elements in F_μ, we now construct an operator that closes a relation under the application to arguments in F_μ. The idea is that whenever there are terms s and t related by $R_{F_\mu \to A}$, then we would like to extend R, such that $s\,u$ and $t\,u$ are related for all $u \in F_\mu$. In general, this is not sound as an up-to technique. However, if we assume that $s\,u$ and $t\,u$ are related by R_A in the base case where $u \equiv \mathrm{pres}\ x$ for some $x : F$, and that R is closed under induction steps, then we can safely build a closure of R that contains all pairs $(s\,u, t\,u)$. This closure is given in the following definition.

Definition 5.1.15. We define $H\colon [\mathrm{Rel}_{\Lambda=}, \mathrm{Rel}_{\Lambda=}] \to [\mathrm{Rel}_{\Lambda=}, \mathrm{Rel}_{\Lambda=}]$, where $[\mathrm{Rel}_{\Lambda=}, \mathrm{Rel}_{\Lambda=}]$ is the set of monotone functions on $\mathrm{Rel}_{\Lambda=}$, as follows by using the closure under convertibility $(-)^{\equiv}$.

$$\begin{aligned} H(T)(R)_A = \{(s\,u, t\,u) \mid & (s,t) \in R_{F_\mu \to A} \wedge u \in \mathbf{ON}_{F_\mu} \wedge x \in \mathbf{ON}_F \wedge u \equiv \mathrm{pres}\, x \\ & \wedge (s\,(\mathrm{pres}\, x), t\,(\mathrm{pres}\, x)) \in R_A\} \\ \cup \{(s\,u, t\,u) \mid & (s,t), (s',t') \in R_{F_\mu \to A} \wedge u, u' \in \mathbf{ON}_{F_\mu} \\ & \wedge s\,(\mathrm{drop}\, u') \equiv s'\,u' \wedge t\,(\mathrm{drop}\, u') \equiv t'\,u' \\ & \wedge (s'', t'u') \in T(R)^{\equiv}_A\} \end{aligned}$$

The *F_μ-closure* is the least fixed point $\mu H\colon \mathrm{Rel}_{\Lambda=} \to \mathrm{Rel}_{\Lambda=}$ of H. ◂

We show that the inductive closure μH is sound as an up-to technique, by proving that it is contained in the companion of Φ. Thus, we can use μH in all proofs of bisimilarity up to γ_Φ.

Lemma 5.1.16. *μH is contained in the companion of Φ: $\mu H \sqsubseteq \gamma_\Phi$.*

Proof. To show that $\mu H \sqsubseteq \gamma_\Phi$, we need to prove that $\mu H(R)_A \subseteq \gamma_\Phi(R)_A$ for all $R \in \mathrm{Rel}_{\Lambda=}$ and $A \in \mathrm{Ty}$. Since μH is a fixed point of H, it suffices to prove $H(\mu H)(R)_A \subseteq \gamma_\Phi(R)_A$. So let $(s\,u, t\,u) \in H(\mu H)(R)_A$ for some $(s,t) \in R_{F_\mu \to A}$ by one of the clauses of H. We proceed by induction on n for $u \equiv \mathrm{drop}^n\,(\mathrm{pres}\, x)$, see Lemma 5.1.14.

- If $n = 0$, then the first case of H applies and we have $(s\,(\mathrm{pres}\, x), t\,(\mathrm{pres}\, x)) \in R_A$. Thus also $(s\,(\mathrm{pres}\, x), t\,(\mathrm{pres}\, x)) \in \gamma_\Phi(R)_A$. Since C^{obs} is compatible by Proposition 5.1.10, it is contained in the companion, which means that the companion is $\equiv$-closed. Thus, $(s\,u, t\,u) \in \gamma_\Phi(R)_A$.
- If $n = k+1$, then by the second case of H we get $(s', t') \in R_{F_\mu \to A}$, such that $s\,(\mathrm{drop}\, u') \equiv s'\,u'$, $t\,(\mathrm{drop}\, u') \equiv t'\,u'$ and $(s'\,u', t'\,u') \in \mu H(R)_A$, where $u' = \mathrm{drop}^k\,(\mathrm{pres}\, x)$. By the induction hypothesis for k, we thus obtain $(s'\,u', t'\,u') \in \gamma_\Phi(R)_A$. Appealing again to the $\equiv$-closure of γ_Φ, we thus have $(s\,u, t\,u) \in \gamma_\Phi(R)_A$, as required.

Hence, $\mu H(R)_A = H(\mu H)(R)_A \subseteq \gamma_\Phi(R)_A$ for all R and A, and so $\mu H \sqsubseteq \gamma_\Phi$. □

It is fairly hard to use the definition of μH directly in a bisimilarity proof because it is not clear which conditions R needs to satisfy in order for $\mu H(R)_A$ to contain $(s\,u, t\,u)$ for all $(s,t) \in R_{F_\mu \to A}$ and $u \in \mathbf{ON}_{F_\mu}$. In the lemma below, we establish such conditions on a given relation R and the terms related by $R_{F_\mu \to A}$, which leads to a proof principle that allows us to easily apply μH. These conditions essentially require that R_A relates terms in the base case of F_μ and that R is closed under computations in the induction step.[33]

Lemma 5.1.17. *Let $R \in \mathrm{Rel}_{\Lambda=}$ and $A \in \mathrm{Ty}$. Suppose that for all $(s,t) \in R_{F_\mu \to A}$ and all monotone $T\colon \mathrm{Rel}_{\Lambda=} \to \mathrm{Rel}_{\Lambda=}$ the following conditions hold:*

i) $\forall x \in \mathbf{ON}_F.\, (s\,(\mathrm{pres}\, x), t\,(\mathrm{pres}\, x)) \in R_A$ *and*

ii) $\forall u \in \mathbf{ON}_{F_\mu}.\, \exists (s', t') \in R_{F_\mu \to A}.\, (s'\,u, t'\,u) \in T(R)_A \implies (s\,(\mathrm{drop}\, u), t\,(\mathrm{drop}\, u)) \in T(R)^{\equiv}_A$

Then for all $u \in \mathbf{ON}_{F_\mu}$ and all $(s,t) \in R_{F_\mu \to A}$ we have $(s\,u, t\,u) \in \mu H(R)_A$.

Proof. Let R and A be given as required by the lemma. Suppose now that $u \in \mathbf{ON}_{F_\mu}$. By Lemma 5.1.14, there is an $n \in \mathbb{N}$ and an $x \in \mathbf{ON}_F$, such that $u \equiv \mathrm{drop}^n\,(\mathrm{pres}\,x)$. We proceed by induction on n to prove for all $(s,t) \in R_{F_\mu \to A}$ that $(s\,u, t\,u) \in \mu H(R)_A$.

- If $n = 0$, then $u \equiv \mathrm{pres}\,x$. Condition i) gives us that $(s\,(\mathrm{pres}\,x), t\,(\mathrm{pres}\,x)) \in R_A$. By definition of H (Definition 5.1.15), we thus have $(s\,u, t\,u) \in H(\mu H)(R)_A$, and by μH being a fixed point of H hence $(s\,u, t\,u) \in \mu H(R)_A$.
- If $n = k+1$, then $u \equiv \mathrm{drop}\,u'$ for $u' = \mathrm{drop}^k\,(\mathrm{pres}\,x)$. From condition ii), we get with $T = \mu H$ a pair $(s', t') \in R_{F_\mu \to A}$, such that, $(s'\,u', t'\,u') \in \mu H(R)_A$ would imply $(s\,(\mathrm{drop}\,u'), t\,(\mathrm{drop}\,u')) \in \mu H(R)^{\equiv}_A$. Since by induction we have for any $(s', t') \in R_{F_\mu \to A}$ that $(s'\,u', t'\,u') \in \mu H(R)_A$, we thus obtain $(s\,(\mathrm{drop}\,u'), t\,(\mathrm{drop}\,u')) \in \mu H(R)^{\equiv}_A$. By definition of H, this implies $(s\,u, t\,u) \in H(\mu H)(R)_A$. Hence, again by μH being a fixed point, it follows that $(s\,u, t\,u) \in \mu H(R)_A$.

This induction on n shows for all $(s,t) \in R_{F_\mu \to A}$ that $(s\,u, t\,u) \in \mu H(R)_A$, as required. □

The induction principle that we devised in Lemma 5.1.17 is readily usable, if we want to prove that two functions with one argument in F_μ are observationally equivalent. However, (5.7) requires us to prove an equivalence between functions with *two* arguments in F_μ, see Proposition 5.1.19 below for details. We could apply Lemma 5.1.17 twice in this situation, but note that comp_μ does not do any further pattern matching on the second argument in the last case of its definition. Hence, we only need to distinguish three cases: both arguments are given by the pres-constructor; the first is a pres- and the second drop-constructor; and the first is given by the drop-constructor, while the second argument is arbitrary. The following lemma applies Lemma 5.1.17 twice to obtain a mutual induction principle that captures exactly this situation.[34]

Lemma 5.1.18. *Let $R \in \mathrm{Rel}_{\Lambda=}$ and $A \in \mathrm{Ty}$. Suppose that for all $(s,t) \in R_{F_\mu \to F_\mu \to A}$ and all monotone $T : \mathrm{Rel}_{\Lambda=} \to \mathrm{Rel}_{\Lambda=}$ the following conditions hold:*

i) $\forall x, y \in \mathbf{ON}_F.\ (s\,(\mathrm{pres}\,x)\,(\mathrm{pres}\,y), t\,(\mathrm{pres}\,x)\,(\mathrm{pres}\,y)) \in R_A$,

ii) $\forall v \in \mathbf{ON}_{F_\mu}.\ \exists (s', t') \in R_{F_\mu \to F_\mu \to A}.$
$$(\forall u \in \mathbf{ON}_{F_\mu}.\ (s'\,u\,v, t'\,u\,v) \in T(R)_A) \implies \forall x \in \mathbf{ON}_F.\ (s\,(\mathrm{pres}\,x)\,(\mathrm{drop}\,v), t\,(\mathrm{pres}\,x)\,(\mathrm{drop}\,v)) \in T(R)^{\equiv}_A$$

iii) $\forall u, v \in \mathbf{ON}_{F_\mu}.\ \exists (s', t') \in R_{F_\mu \to F_\mu \to A}.$
$$(s'\,u\,v, t'\,u\,v) \in T(R)_A \implies (s\,(\mathrm{drop}\,u)\,v, t\,(\mathrm{drop}\,u)\,v) \in T(R)^{\equiv}_A$$

Then for all $(s,t) \in R_{F_\mu \to F_\mu \to A}$ and all $u, v \in \mathbf{ON}_{F_\mu}$ we have $(s\,u\,v, t\,u\,v) \in \gamma_\Phi(R)_A$.

Proof. The proof proceeds by applying Lemma 5.1.17 to the second argument to show that we have $(\lambda u.\, s\,u\,v, \lambda u.\, t\,u\,v) \in \mu H(\gamma_\Phi(R))_{F_\mu \to A}$ for all $v \in \mathbf{ON}_{F_\mu}$. Two cases arise from this: First, we need to prove for all $y \in \mathbf{ON}_F$ that $(\lambda u.\, s\,u\,(\mathrm{pres}\,y), \lambda u.\, t\,u\,(\mathrm{pres}\,y)) \in \gamma_\Phi(R)_{F_\mu \to A}$, and, second, that for all operators T and all $v' \in \mathbf{ON}_{F_\mu}$ also $(\lambda u.\, s\,u\,(\mathrm{drop}\,v'), \lambda u.\, t\,u\,(\mathrm{drop}\,v')) \in T(\gamma_\Phi(R))_{F_\mu \to A}$. Both cases are tackled by appealing to Lemma 5.1.12, so that we have to prove for all $u \in \mathbf{ON}_{F_\mu}$ that $(s\,u\,(\mathrm{pres}\,y), t\,u\,(\mathrm{pres}\,y)) \in \gamma_\Phi(R)_{F_\mu \to A}$, and $(s\,u\,(\mathrm{drop}\,v'), t\,u\,(\mathrm{drop}\,v')) \in T(\gamma_\Phi(R))_{F_\mu \to A}$. Finally, both cases are resolved by applying in each of them the inductive closure from Lemma 5.1.17 to u. The first case cases uses thereby the first and third assumption of the lemma, whereas the second case requires the second and third assumption. □

Having all this machinery of up-to techniques set up, we are finally in the position to prove that select is a homomorphism from selector composition to function composition, that is, we can prove $\text{select}\ (y \bullet x) \equiv_{\text{obs}} \text{select}\ y \circ \text{select}\ x$ for all $x, y \in \mathbf{ON}_F$, see Equation (5.7). The idea of the proof is that we first show that select_μ is a homomorphism from $\bullet_\mu$ to function composition, in other words, we have for all $u, v \in \mathbf{ON}_F$ that $\text{select}_\mu\ (v \bullet_\mu u) \equiv_{\text{obs}} \text{select}_\mu\ v \circ \text{select}_\mu\ u$. We prove this equation by proving that the abstractions $\lambda u\ v.\ \text{select}_\mu\ (v \bullet_\mu u)\ s$ and $\lambda u\ v.\ \text{select}_\mu\ v\ (\text{select}_\mu\ u\ s)$ of the involved terms are observationally equivalent. This allows us to apply the induction principle that we derived in Lemma 5.1.18. The identity (5.7) follows then by definition of select in terms of select_μ.

Proposition 5.1.19. *The relation $R \in \text{Rel}_{\Lambda^=}$, given by*

$$\begin{aligned} R_{F_\mu \to F_\mu \to A^\omega} &= \left\{ \left(\lambda u\ v.\ \text{select}_\mu\ (v \bullet_\mu u)\ s, \lambda u\ v.\ \text{select}_\mu\ v\ (\text{select}_\mu\ u\ s)\right) \;\middle|\; s \in \mathbf{ON}_{A^\omega} \right\} \\ R_{A^\omega} &= \{(\text{select}_\mu\ ((\text{pres}\ y) \bullet_\mu (\text{pres}\ x))\ s, \text{select}_\mu\ (\text{pres}\ y)\ (\text{select}_\mu\ (\text{pres}\ x)\ s)) \mid \\ &\qquad s \in \mathbf{ON}_{A^\omega}, x, y \in \mathbf{ON}_F\} \end{aligned}$$

is an observational bisimulation up to γ_Φ. In particular, for all $x, y \in \mathbf{ON}_F$ the equation (5.7)*, that is* $\text{select}\ (y \bullet x) \equiv_{\text{obs}} \text{select}\ y \circ \text{select}\ x$*, holds.*

Proof. We first show that for all $(f, g) \in R_{F_\mu \to F_\mu \to A^\omega}$ and $u, v \in \mathbf{ON}_{F_\mu}$ that

$$(f\ u\ v, g\ u\ v) \in \gamma_\Phi(R)_{A^\omega}. \tag{5.8}$$

So let $(f, g) \in R_{F_\mu \to F_\mu \to A^\omega}$, that is, $f = \lambda u\ v.\ \text{select}_\mu\ (v \bullet_\mu u)\ s$ and $g = \lambda u\ v.\ \text{select}_\mu\ v\ (\text{select}_\mu\ u\ s)$ for $s \in \mathbf{ON}_{A^\omega}$. From Lemma 5.1.18, we obtain $(f\ u\ v, g\ u\ v) \in \gamma_\Phi(R)_{A^\omega}$, by proving the following three conditions.

i) Let $x, y \in \mathbf{ON}_F$, then $(f\ (\text{pres}\ x)\ (\text{pres}\ y), g\ (\text{pres}\ x)\ (\text{pres}\ y)) \in \gamma_\Phi(R)_{A^\omega}$, by definition of R and $\text{id} \sqsubseteq \gamma_\Phi$.

ii) For the second condition, let $v \in \mathbf{ON}_{F_\mu}$. Given $x \in \mathbf{ON}_F$, we first note that we have the following computations:

$$\begin{aligned} f\ (\text{pres}\ x)\ (\text{drop}\ v) &\equiv \text{select}_\mu\ (\text{comp}_\mu\ (\text{pres}\ x)\ (\text{drop}\ v))\ s \\ &\equiv \text{select}_\mu\ (\text{drop}\ (\text{comp}_\mu\ (x.\text{out})\ v))\ s \\ &\equiv \text{select}_\mu\ (\text{comp}_\mu\ (x.\text{out})\ v)\ (s.\text{tl}) \\ &\equiv (\lambda u\ v.\ \text{select}_\mu\ (v \bullet_\mu u)\ (s.\text{tl}))\ (x.\text{out})\ v \end{aligned}$$

and

$$\begin{aligned} g\ (\text{pres}\ x)\ (\text{drop}\ v) &\equiv \text{select}_\mu\ (\text{drop}\ v)\ (\text{select}_\mu\ (\text{pres}\ x)\ s) \\ &\equiv \text{select}_\mu\ v\ ((\text{select}_\mu\ (\text{pres}\ x)\ s).\text{tl}) \\ &\equiv \text{select}_\mu\ v\ (\text{select}\ x\ (s.\text{tl})) \\ &\equiv \text{select}_\mu\ v\ (\text{select}_\mu\ (x.\text{out})\ (s.\text{tl})) \\ &\equiv (\lambda u\ v.\ \text{select}_\mu\ v\ (\text{select}_\mu\ u\ (s.\text{tl})))\ (x.\text{out})\ v. \end{aligned}$$

Thus, we can pick $f' \coloneqq \lambda u\ v.\ \text{select}_\mu\ (v \bullet_\mu u)\ (s.\text{tl})$ and $g' \coloneqq \lambda u\ v.\ \text{select}_\mu\ v\ (\text{select}_\mu\ u\ (s.\text{tl}))$, which are related by $R_{F_\mu \to F_\mu \to A^\omega}$. Assume that $(f'\ u\ v, g'\ u\ v) \in T(R)_{A^\omega}$ for all $u \in \mathbf{ON}_{F_\mu}$. Then by the above computations, $(f\ (\text{pres}\ x)\ (\text{drop}\ v), g\ (\text{pres}\ x)\ (\text{drop}\ v)) \in T(R)^{\equiv}_{A^\omega}$ for all $x \in \mathbf{ON}_F$. Thus, also the second condition of Lemma 5.1.18 is fulfilled.

iii) Let $u, v \in \mathbf{ON}_{F_\mu}$. We note now that

$$\begin{aligned} f\ (\mathrm{drop}\ u)\ v &= (\lambda u\ v.\ \mathrm{select}_\mu\ (v \bullet_\mu u)\ s)\ (\mathrm{drop}\ u)\ v \\ &\equiv \mathrm{select}_\mu\ (v \bullet_\mu (\mathrm{drop}\ u))\ s \\ &= \mathrm{select}_\mu\ (\mathrm{comp}_\mu\ (\mathrm{drop}\ u)\ v)\ s \\ &\equiv \mathrm{select}_\mu\ (\mathrm{drop}\ (\mathrm{comp}_\mu\ u\ v))\ s \\ &\equiv \mathrm{select}_\mu\ (\mathrm{comp}_\mu\ u\ v)\ (s.\mathrm{tl}) \\ &\equiv (\lambda u\ v.\ \mathrm{select}_\mu\ (v \bullet_\mu u)\ (s.\mathrm{tl}))\ u\ v \end{aligned}$$

and

$$\begin{aligned} g\ (\mathrm{drop}\ u)\ v &= (\lambda u\ v.\ \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ u\ s))\ (\mathrm{drop}\ u)\ v \\ &\equiv \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ (\mathrm{drop}\ u)\ s) \\ &\equiv \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ u\ (s.\mathrm{tl})) \\ &\equiv (\lambda u\ v.\ \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ u\ (s.\mathrm{tl})))\ u\ v \end{aligned}$$

Since $s \in \mathbf{ON}$, also $s.\mathrm{tl} \in \mathbf{ON}$. Thus, if we define terms $f' \coloneqq \lambda u\ v.\ \mathrm{select}_\mu\ (v \bullet_\mu u)\ (s.\mathrm{tl})$ and $g' \coloneqq \lambda u\ v.\ \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ u\ (s.\mathrm{tl}))$, then by the above reasoning $(f'\ u\ v, g'\ u\ v) \in T(\gamma_\Phi(R))_{A^\omega}$ implies that $(f\ (\mathrm{drop}\ u)\ v, g\ (\mathrm{drop}\ u)\ v) \in T(\gamma_\Phi(R))^{\equiv}_{A^\omega}$. Hence, also the third condition of Lemma 5.1.18 is fulfilled.

Since all conditions of Lemma 5.1.18 are fulfilled for all $(f, g) \in \gamma_\Phi(R)_{F_\mu \to F_\mu \to A^\omega}$ and $u, v \in \mathbf{ON}_{F_\mu}$, we obtain that $(f\ u\ v, g\ u\ v) \in \gamma_\Phi(R)_{A^\omega}$.

We now prove that $R \sqsubseteq \Phi(\gamma_\Phi(R))$. For the case $(f, g) \in \gamma_\Phi(R)_{F_\mu \to F_\mu \to A^\omega}$, we have shown above that $(f\ u\ v, g\ u\ v) \in \gamma_\Phi(R)_{A^\omega}$ for all $u, v \in \mathbf{ON}_{F_\mu}$. It follows by Lemma 5.1.12 that $(f\ u, g\ u) \in \gamma_\Phi(R)_{F_\mu \to A^\omega}$ for all $u \in \mathbf{ON}_{F_\mu}$, hence $(f, g) \in \Phi(\gamma_\Phi(R))_{F_\mu \to F_\mu \to A^\omega}$.

It remains to prove that $(t_1, t_2) \in R_{A^\omega}$ implies $(t_1, t_2) \in \Phi(\gamma_\Phi(R))_{A^\omega}$. Let $(t_1, t_2) \in R_{A^\omega}$, so that we have for $s \in \mathbf{ON}_{A^\omega}$ and $x, y \in \mathbf{ON}_F$ that $t_1 = \mathrm{select}_\mu\ ((\mathrm{pres}\ y) \bullet_\mu (\mathrm{pres}\ x))\ s$ and $t_2 = \mathrm{select}_\mu\ (\mathrm{pres}\ y)\ (\mathrm{select}_\mu\ (\mathrm{pres}\ x)\ s)$. We now need to show that $(t_1.\mathrm{hd}, t_2.\mathrm{hd}) \in \gamma_\Phi(R)_A$ and $(t_1.\mathrm{tl}, t_2.\mathrm{tl}) \in \gamma_\Phi(R)_{A^\omega}$. Note first that we have the following computations for t_1:

$$\begin{aligned} t_1 &\equiv \mathrm{select}_\mu\ (\mathrm{comp}_\mu\ (\mathrm{pres}\ x)\ (\mathrm{pres}\ y))\ s \\ &\equiv \mathrm{select}_\mu\ (\mathrm{pres}\ (\mathrm{comp}\ x\ y))\ s \\ &\equiv \mathrm{select}_\mu\ (\mathrm{pres}\ (y \bullet x))\ s \end{aligned}$$

Since

$$\begin{aligned} t_1.\mathrm{hd} &\equiv s.\mathrm{hd} \\ &\equiv (\mathrm{select}_\mu\ (\mathrm{pres}\ x)\ s).\mathrm{hd} \\ &\equiv (\mathrm{select}_\mu\ (\mathrm{pres}\ y)\ (\mathrm{select}_\mu\ (\mathrm{pres}\ x)\ s)).\mathrm{hd} \\ &\equiv t_2.\mathrm{hd}, \end{aligned}$$

we obtain $(t_1.\mathrm{hd}, t_2.\mathrm{hd}) \in \gamma_\Phi(R)_A$. Next, we have

$$\begin{aligned} t_1.\mathrm{tl} &\equiv \mathrm{select}\ (y \bullet x)\ (s.\mathrm{tl}) \\ &\equiv \mathrm{select}_\mu\ ((y \bullet x).\mathrm{out})\ (s.\mathrm{tl}) \\ &\equiv \mathrm{select}_\mu\ ((y.\mathrm{out}) \bullet (x.\mathrm{out}))\ (s.\mathrm{tl}) \end{aligned}$$

and

$$\begin{aligned} t_2.\mathrm{tl} &\equiv (\mathrm{select}_\mu\ (\mathrm{pres}\ y)\ (\mathrm{select}_\mu\ (\mathrm{pres}\ x)\ s)).\mathrm{tl} \\ &\equiv \mathrm{select}\ y\ ((\mathrm{select}_\mu\ (\mathrm{pres}\ x)\ s).\mathrm{tl}) \\ &\equiv \mathrm{select}\ y\ (\mathrm{select}\ x\ (s.\mathrm{tl})) \\ &\equiv \mathrm{select}_\mu\ (y.\mathrm{out})\ (\mathrm{select}\ x\ (s.\mathrm{tl})) \\ &\equiv \mathrm{select}_\mu\ (y.\mathrm{out})\ (\mathrm{select}_\mu\ (x.\mathrm{out})\ (s.\mathrm{tl})) \end{aligned}$$

Combining this with (5.8) we thus obtain $(t_1.\mathrm{tl}, t_2.\mathrm{tl}) \in \gamma_\Phi(R)_{A^\omega}$. Applying Lemma 5.1.12 to the result that $(t_1.\mathrm{hd}, t_2.\mathrm{hd}) \in \gamma_\Phi(R)_A$ and $(t_1.\mathrm{tl}, t_2.\mathrm{tl}) \in \gamma_\Phi(R)_{A^\omega}$, we finally obtain that $(t_1, t_2) \in \Phi(\gamma_\Phi(R))_{A^\omega}$. Hence, R is an observational bisimulation up to γ_Φ. So for all $s \in \mathbf{ON}_{A^\omega}$

$$\lambda u\ v.\ \mathrm{select}_\mu\ (v \bullet_\mu u)\ s \equiv_{\mathrm{obs}} \lambda u\ v.\ \mathrm{select}_\mu\ v\ (\mathrm{select}_\mu\ u\ s). \tag{5.9}$$

Finally, we need to show for $x, y \in \mathbf{ON}_F$ that $\mathrm{select}\ (y \bullet x) \equiv_{\mathrm{obs}} \mathrm{select}\ y \circ \mathrm{select}\ x$ holds. Since

$$\mathrm{select}\ (y \bullet x) \equiv \mathrm{select}_\mu\ ((y \bullet x).\mathrm{out}) \equiv \mathrm{select}_\mu\ ((y.\mathrm{out}) \bullet_\mu (x.\mathrm{out}))$$

and

$$\mathrm{select}\ y \circ \mathrm{select}\ x \equiv \mathrm{select}_\mu\ (y.\mathrm{out}) \circ \mathrm{select}_\mu\ (x.\mathrm{out}),$$

we obtain by (5.9) that $\mathrm{select}\,(y \bullet x)\ s \equiv_{\mathrm{obs}} (\mathrm{select}\ y \circ \mathrm{select}\ x)\ s$ for all $s \in \mathbf{ON}_{A^\omega}$. Thus, the equivalence $\mathrm{select}\ (y \bullet x) \equiv_{\mathrm{obs}} (\mathrm{select}\ y \circ \mathrm{select}\ x)$ follows from extensionality (Lemma 4.1.25.(v)). □

5.2. A First-Order Logic for Observational Equivalence

In Section 5.1.2, we have described a proof method for observational equivalence that is based on bisimulation relations. There are several problems with this approach. For one, we have to *explicitly* use several up-to techniques to be able to give reasonable bisimulations. This can be somewhat improved by using the companion, see Definition 2.5.12, but even then we have to establish that each up-to technique, which appears in a proof, is compatible or contained in the companion, see Lemma 5.1.16. Even worse, general bisimulations and the explicit use of up-to techniques does not lend itself very well to automatically verifiable proofs. The other problem with this approach is that we need to come up with a bisimulation of which we know *beforehand* that it proves the desired identity. This is not an easy task, as we have seen in Proposition 5.1.19, but it can be simplified by the use of parametrised coinduction [Hur+13]. Since we aim for automatically verifiable proofs, we now take a more syntactic approach for proving observational equivalence.

More specifically, we will establish in this section a first order logic together with a proof system for observational equivalence, to which we will refer as $\mathbf{FOL}_\blacktriangleright$. This logic has the standard propositional and first order connectives, and a binary relation that represents observational equivalence. The proof system supports both inductive and coinductive reasoning, that is, we can carry out proofs by induction on inductive data types and prove identities on coinductive types. Induction and coinduction are usually introduced by means of induction and coinduction schemes, see for example [Bla+14; Gim95]. Though it is *in principle* possible to use the explicit induction and coinduction schemes to prove most theorems [Wir04], such proofs are notoriously hard to set up

because one needs to guess the induction and coinduction hypotheses beforehand. Thus, we will not rely on the standard induction and coinduction schemes in the proof system.

A more natural approach are *cyclic proofs*, that is, proofs that repeat arguments by referring back to a step made before in the proof. Famous examples of cyclic proofs are the proof of irrationality of the square root of two by an infinite descent argument [Wir04], the proof that the Ackermann function is total, and generally proofs by well-founded induction. It is important to note that cyclic proofs are finite representations of infinite proofs [DHL06; NW96; Stu08], where the latter cannot be automatically verified and are also frequently rejected for philosophical reasons. Cyclic proofs have been extensively studied for purely inductive proofs [Bae09; Bro05; BBC08; BDP11; BG14; BGP12; BS07; BS11; RB17; Sim17; SD03; Wir04], to some extent for coinductive proofs [RL09] and in specific cases for mixed inductive-coinductive proofs [SD03]. Cockett [Coc01] and Santocanale [San02a] have used cyclic proof systems to define maps on inductive and coinductive types, and in the former also to reason about the thus defined maps. We will discuss the relation to all these systems in Section 5.4. Moreover, cyclic proofs have made an implicit appearance through game semantics [NW96] and in parametrised coinduction [Hur+13]. That cyclic proofs are a promising approach is witnessed by the fact that the structural induction principle is subsumed by, sometimes even equivalent to, cyclic proofs [Bro05; Sim17].

A problem of cyclic proofs is that we need to ensure that a proof is actually correct, most importantly, the verification that a cycle leads to a well-defined proof. The various approaches to deal with this problem include trace conditions [Bro05; Sim17], parity conditions [San02a] and size-based termination [AP13; Bar+04; LJB01; SD03; Xi01]. Since these conditions are either global, that is they are conditions on the whole proof object, or require non-trivial constraint solving, both soundness and proof checking are difficult to establish for the above mentioned systems. For this reason, we will use another approach in this exposition.

An elegant way to ensure correctness of recursive definitions (or cyclic proofs in the context of logics) was introduced by Nakano [Nak00]. We adapt this approach here by annotating formulas with a so-called *later modality*. This later modality stems originally from the provability logic, or Gödel-Löb logic, which characterises transitive, well-founded Kripke frames [Sol76], and thus allows one to carry out induction without an explicit induction scheme [Bek99]. Later, Nakano's approach was further developed by Appel et al. [App+07], Atkey and McBride [AM13], Bizjak et al. [Biz+16b] and Møgelberg [Møg14], mostly with the intent to replace syntactic guardedness checks (cf. Example 4.1.6) for coinductive definitions by type-based checks of well-definedness. We will use the later modality in a similar way to control the use of an induction or coinduction hypothesis in a proof, which leads to easily verifiable proofs. In particular, we introduce a logic $\mathbf{FOL}_{\blacktriangleright}$ with a Gentzen-style intuitionistic sequent calculus that internalises the rules of provability logic and combines these with rules that enable step-wise reasoning about observational equivalence, as we briefly explain now.

The intuition behind the logic $\mathbf{FOL}_{\blacktriangleright}$ and especially the interplay of the later modality and the equality relation is the following. Given two terms s and t in $\lambda\mu\nu{=}$, there is a formula $s \sim t$ in $\mathbf{FOL}_{\blacktriangleright}$, which should be read as s and t are observationally equivalent. To be more precise, it means that s and t are related by every step in the ω^{op}-chain that can be used to construct the relation $\equiv_{\mathrm{obs}}$, cf. Theorem 5.1.9 and Section 2.5. Using this interpretation of the equality relation, we can understand the role of the later modality in $\mathbf{FOL}_{\blacktriangleright}$. Given a formula $s \sim t$, we can obtain another formula $\blacktriangleright(s \sim t)$, which should be read as "s and t are later observationally equivalent", that is, a

proof of $\blacktriangleright(s \sim t)$ is a promise that we can prove $s \sim t$ later. We will make this intuition a bit more precise now.

Recall from Definition 2.5.5 that the ω^{op}-chain to approximate the greatest fixed point of a monotone map $F\colon \mathrm{Rel}_X \to \mathrm{Rel}_X$ is given by the chain of inclusions

$$\mathbf{1} \quad \supseteq \quad F(\mathbf{1}) \quad \supseteq \quad F^2(\mathbf{1}) \quad \supseteq \quad \cdots,$$

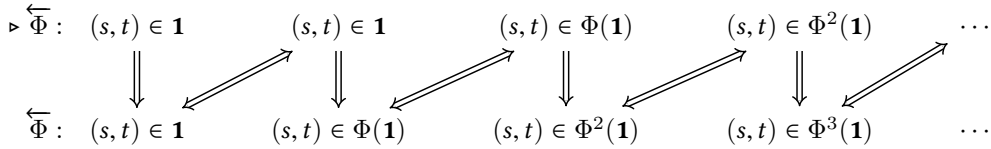
where $\mathbf{1}$ is the full relation on X. We will denote this chain by $\overleftarrow{F}$ with $\overleftarrow{F}(n) = F^n(\mathbf{1})$. The semantics of the relation $\sim$ is given by instantiating F in this construction with the operator Φ, which we used to characterise observational equivalence as largest bisimulation in Section 5.1.2. A formula $s \sim t$ is then interpreted at $n \in \mathbb{N}$ by $(s,t) \in \overleftarrow{\Phi}(n)$. To make sense of a formula $\blacktriangleright(s \sim t)$, we define the chain $\triangleright\, \overleftarrow{\Phi}$ by $(\triangleright\, \overleftarrow{\Phi})(0) = \mathbf{1}$ and $(\triangleright\, \overleftarrow{\Phi})(n+1) = \overleftarrow{\Phi}(n)$, which corresponds to the following picture.

$$\triangleright\, \overleftarrow{\Phi}\colon \qquad \mathbf{1} \quad \supseteq \quad \mathbf{1} \quad \supseteq \quad \Phi(\mathbf{1}) \quad \supseteq \quad \Phi^2(\mathbf{1}) \quad \supseteq \quad \cdots$$

The reason why we introduce the later formula is that the implication $\blacktriangleright(s \sim t) \to s \sim t$ will be interpreted at $n \in \mathbb{N}$ as

$$(s,t) \in (\triangleright\, \overleftarrow{\Phi})(n) \implies (s,t) \in \overleftarrow{\Phi}(n). \tag{5.10}$$

We can depict this implication by the following diagram.



Starting with $(s,t) \in \mathbf{1}$, which is always true, we can move in a zig-zag from left to right through the diagram, thereby proving that $(s,t) \in \overleftarrow{\Phi}(n)$ for all $n \in \mathbb{N}$, cf. Lemma 2.3.5. Thus, we can infer from the implication (5.10) that s and t are observationally equivalent. This proof principle is called *Löb induction* and is captured by the following rule, which is part of the proof system for $\mathbf{FOL}_{\blacktriangleright}$ in a more general form, see Definition 5.2.4.

$$\frac{\blacktriangleright(s \sim t) \vdash s \sim t}{\vdash s \sim t}$$

Besides this rule, there are also rules that allow us to state that if we later have a proof for an equivalence between the observations on terms of a fixed point type, then we get a proof now of the equivalence between the original terms. For instance, we can derive the following rule for streams in $\mathbf{FOL}_{\blacktriangleright}$, see Example 5.2.12.

$$\frac{\vdash \mathrm{hd}\, s \sim_A \mathrm{hd}\, t \qquad \vdash \blacktriangleright(\mathrm{tl}\, s \sim_{A^\omega} \mathrm{tl}\, t)}{\vdash s \sim_{A^\omega} t}$$

Note the similarity of this rule with the bisimulation proof principle for streams, cf. Example 5.1.11. The power of the proof system comes, as one might imagine, from the combination of these two rules. The interested reader might want to compare the bisimulation-based proof in Example 5.1.11 with the syntactic proof in Example 5.2.12 below. Further details on the interpretation of the logic are given in Section 5.2.2.

5.2.1. The Logic FOL$_\blacktriangleright$

In this section, we introduce the formulas of the logic **FOL$_\blacktriangleright$** and its proof system, we prove some basic properties of the logic, and we demonstrate the use of **FOL$_\blacktriangleright$** in some example applications. All the examples in this section have been checked automatically in the prototype implementation of a proof checker for **FOL$_\blacktriangleright$**.

For the remainder of this section, we restrict attention to terms of $\lambda\mu\nu{=}$. Recall that we wrote $\Gamma \vdash t : A$, whenever t is a term of type A in context Γ. For clarity, we will henceforth add a supscript Ty to the judgement and write instead

$$\Gamma \vdash_{\mathrm{Ty}} t : A.$$

Definition 5.2.1. The *formulas* of **FOL$_\blacktriangleright$** are given by

$$\varphi, \psi ::= \bot \mid t \doteq_A s \mid t \sim_A s \mid \blacktriangleright \varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \varphi \to \psi \mid \forall x : A.\, \varphi \mid \exists x : A.\, \varphi,$$

where A ranges over all types in Ty. To disambiguate formulas, we will use parentheses and adapt a few conventions concerning the binding of connectives: The later modality binds stronger than all other logical connectives; the implication is right associative; binding of quantifiers extends from the dot towards the end of the formula; and implication binds stronger than conjunction, which again binds stronger than disjunction. We say that φ is a *well-formed formula* in context Γ, if $\Gamma \Vdash \varphi$ can be derived inductively from the following rules.

$$\frac{}{\Gamma \Vdash \bot} \qquad \frac{\Gamma \vdash_{\mathrm{Ty}} s, t : A}{\Gamma \Vdash s \doteq_A t} \qquad \frac{\Gamma \vdash_{\mathrm{Ty}} s, t : A}{\Gamma \Vdash s \sim_A t} \qquad \frac{\Gamma \Vdash \varphi}{\Gamma \Vdash \blacktriangleright \varphi}$$

$$\frac{\Gamma \Vdash \varphi \qquad \Gamma \Vdash \psi \qquad \square \in \{\wedge, \vee, \to\}}{\Gamma \Vdash \varphi \square \psi} \qquad \frac{\Gamma, x : A \Vdash \varphi}{\Gamma \Vdash \forall x : A.\, \varphi} \qquad \frac{\Gamma, x : A \Vdash \varphi}{\Gamma \Vdash \exists x : A.\, \varphi}$$

If the type A is clear from the context, we usually drop the subscript in $\doteq$- and $\sim$-formulas, and write $s \sim t$ instead of $s \sim_A t$. A finite sequence of formulas Δ is said to be *well-formed* in context Γ, if $\Gamma \Vdash \varphi$ for all φ in Δ. We denote this by $\Gamma \Vdash \Delta$ as well. ◂

The reader will have noticed that there are two binary relations on terms among the formulas of **FOL$_\blacktriangleright$**. Besides the relation $\sim$, which reflects observational equivalence in the logic, there is a relation $\doteq$. This relation will be interpreted as the conversion relation $\equiv$. A formula of the form $s \doteq t$ is then usually paraphrased as s and t are *definitionally equal* [Mar75a]. For the most part, definitional equality can be ignored, we only need it to be able to talk about weak head normal forms inside the logic.

When giving the axioms and later the semantics of **FOL$_\blacktriangleright$** we need to be able to substitute terms for variables in formulas. The following definition introduces substitutions and accompanying notations for the formulas in **FOL$_\blacktriangleright$**.

Definition 5.2.2. The term substitution of $\lambda\mu\nu{=}$ extends to formulas in the expected way by defining $(s \sim t)[u/x] = s[u/x] \sim t[u/x]$, $(\varphi \wedge \psi)[u/x] = \varphi[u/x] \wedge \psi[u/x]$ etc. We denote for a formula φ with $\Gamma \Vdash \varphi$ and a substitution $\sigma \in \mathrm{Subst}(\Lambda^{=}; \Gamma)$ by $\varphi[\sigma]$ the result of substituting for all variables in φ the corresponding term in σ. Given a type A and a term $t \in \Lambda^{=}(A)$, we write $\sigma[x \mapsto t]$ for the *updated substitution* in $\mathrm{Subst}(\Lambda^{=}; \Gamma, x : A)$ that is the same as σ but additionally substitutes t for x. Finally, if the variable x is understood from the context, then we write $\varphi[t]$ for $\varphi[t/x]$. ◂

The following result allows us to switch the order of substitutions, similar to the substitution Lemma 4.2.3 for terms.

Lemma 5.2.3. *If* $\Gamma, x : A, y : B \Vdash \varphi$, $\Gamma \vdash_{\mathrm{Ty}} s : A$ *and* $\Gamma, x : A \vdash_{\mathrm{Ty}} t : B$, *then we have that the equation* $\varphi[t/y][s/x] = \varphi[s/x][t[s/x]/y]$ *holds.*

Proof. This follows by induction on φ from the substitution lemma for $\lambda\mu\nu{=}$, see Lemma 4.2.3. □

We now define the syntactic proof system for $\mathbf{FOL}_{\blacktriangleright}$.

Definition 5.2.4. Let φ be a formula and Δ a sequence of formulas. We say φ is *provable in context* Γ *under the assumptions* Δ, if $\Gamma \mid \Delta \vdash \varphi$ holds. The *provability relation* $\vdash$ is thereby given inductively by the rules in the Figures 5.2, 5.3, 5.4, 5.5 and 5.6. ◂

$$\frac{\Gamma \Vdash \Delta, \varphi}{\Gamma \mid \Delta, \varphi \vdash \varphi}\ \textbf{(Proj)} \qquad \frac{\Gamma \mid \Delta \vdash \varphi \qquad \Gamma \Vdash \psi}{\Gamma \mid \Delta, \psi \vdash \varphi}\ \textbf{(Weak)} \qquad \frac{\Gamma \Vdash \varphi \qquad \Gamma \mid \Delta \vdash \bot}{\Gamma \mid \Delta \vdash \varphi}\ (\bot\textbf{-E})$$

$$\frac{\Gamma \mid \Delta \vdash \varphi \qquad \Gamma \mid \Delta \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \wedge \psi}\ (\wedge\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi_1 \wedge \varphi_2 \qquad i \in \{1,2\}}{\Gamma \mid \Delta \vdash \varphi_i}\ (\wedge_i\textbf{-E})$$

$$\frac{\Gamma \mid \Delta \vdash \varphi_i \qquad \Gamma \Vdash \varphi_j \qquad j \neq i}{\Gamma \mid \Delta \vdash \varphi_1 \vee \varphi_2}\ (\vee_i\textbf{-I}) \qquad \frac{\Gamma \mid \Delta, \varphi_1 \vdash \psi \qquad \Gamma \mid \Delta, \varphi_2 \vdash \psi}{\Gamma \mid \Delta, \varphi_1 \vee \varphi_2 \vdash \psi}\ (\vee\textbf{-E})$$

$$\frac{\Gamma \mid \Delta, \varphi \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \to \psi}\ (\to\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi \to \psi \qquad \Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \psi}\ (\to\textbf{-E})$$

$$\frac{\Gamma, x : A \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \forall x : A.\, \varphi}\ (\forall\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \forall x : A.\, \varphi \qquad t \in \mathbf{ON}_A^\Gamma}{\Gamma \mid \Delta \vdash \varphi[t/x]}\ (\forall\textbf{-E})$$

$$\frac{t \in \mathbf{ON}_A^\Gamma \qquad \Gamma \mid \Delta \vdash \varphi[t/x]}{\Gamma \mid \Delta \vdash \exists x : A.\, \varphi}\ (\exists\textbf{-I}) \qquad \frac{\Gamma \Vdash \psi \qquad \Gamma, x : A \mid \Delta, \varphi \vdash \psi}{\Gamma \mid \Delta, \exists x : A.\, \varphi \vdash \psi}\ (\exists\textbf{-E})$$

Figure 5.2.: Intuitionistic Rules for Standard Connectives

$$\frac{\Gamma \vdash_{\mathrm{Ty}} t, s : A \qquad t \equiv s}{\Gamma \mid \emptyset \vdash t \doteq_A s}\ \textbf{(Def)} \qquad \frac{\Gamma \mid \Delta[s/x] \vdash \varphi[s/x] \qquad \Gamma \mid \Delta \vdash s \doteq_A t}{\Gamma \mid \Delta[t/x] \vdash \varphi[t/x]}\ (\doteq\textbf{-Repl})$$

Figure 5.3.: Rules for Definitional Equality

The rules are grouped as follows. In Figure 5.2, we find the standard proof rules for an intuitionistic first-order logic. The rules for definitional equality are given in Figure 5.3. They are the standard rules for equality, see for example [Jac99, Chap. 3]. The rules in Figure 5.4 describe how equivalences of programs can be proven. Those rules that can be applied in both directions are thereby denoted by a double line. None of these rules should come at a surprise, except for **(ν-Ext)** and **(μ-Cong)**. In these rules, the later modality makes its appearance to control the use of hypotheses that can be introduced through the Löb (or fixed point) rule **(Löb)** in Figure 5.5. The rule **(ν-Ext)** should be read as "to prove an equivalence between two programs on a ν-type, it suffices to give a proof later

$$\frac{\Gamma \vdash_{\mathrm{Ty}} t, s : A \qquad t \equiv s}{\Gamma \mid \emptyset \vdash t \sim_A s}\ \textbf{(Refl)} \qquad \frac{\Gamma \mid \Delta \vdash s \sim_A t}{\Gamma \mid \Delta \vdash t \sim_A s}\ \textbf{(Sym)}$$

$$\frac{\Gamma \mid \Delta \vdash s \sim_A t \qquad \Gamma \mid \Delta \vdash t \sim_A r}{\Gamma \mid \Delta \vdash s \sim_A r}\ \textbf{(Trans)}$$

$$\frac{\Gamma, x : A \mid \Delta \vdash f\ x \sim_B g\ x}{\Gamma \mid \Delta \vdash f \sim_{A \to B} g}\ (\to\textbf{-Ext})$$

$$\frac{\Gamma \mid \Delta \vdash s.\mathrm{pr}_1 \sim_A t.\mathrm{pr}_1 \qquad \Gamma \mid \Delta \vdash s.\mathrm{pr}_2 \sim_B t.\mathrm{pr}_2}{\Gamma \mid \Delta \vdash s \sim_{A \times B} t}\ (\times\textbf{-Ext})$$

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright(s.\mathrm{out} \sim_{A[\nu X.\, A/X]} t.\mathrm{out})}{\Gamma \mid \Delta \vdash t \sim_{\nu X.\, A} s}\ (\nu\textbf{-Ext})$$

$$\frac{\Gamma \mid \Delta \vdash s \sim_{A_i} t}{\Gamma \mid \Delta \vdash \kappa_i\ s \sim_{A_1 + A_2} \kappa_i\ t}\ (+\textbf{-Cong}) \qquad \frac{\Gamma \mid \Delta \vdash \blacktriangleright(s \sim_{A[\mu X.\, A/X]} t)}{\Gamma \mid \Delta \vdash \alpha\ s \sim_{\mu X.\, A} \alpha\ t}\ (\mu\textbf{-Cong})$$

Figure 5.4.: Rules for Equality

$$\frac{\Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi}\ \textbf{(Nec)} \qquad \frac{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi \to \psi)}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi \to \blacktriangleright \psi}\ \textbf{(K)} \qquad \frac{\Gamma \mid \Delta, \blacktriangleright \varphi \vdash \varphi}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Löb)}$$

Figure 5.5.: Rules for the Later Modality

for their unfoldings", see the examples below. Next, the rules in Figure 5.5 are the standard rules of provability logic[35] [Sol76] that also turn $\blacktriangleright$ into an applicative functor with a fixed point operator, see e.g. [AM13]. Finally, the rules in Figure 5.6 allow us to refine elements of inductive types.

We note that if a formula is provable, then the formula and assumptions are well-typed.

Lemma 5.2.5. *If we have for a formula φ and a sequence Δ of formulas that $\Gamma \mid \Delta \vdash \varphi$ holds, then $\Gamma \Vdash \Delta$ and $\Gamma \Vdash \varphi$.*

Proof. This is proved by a straightforward induction on the proof tree for $\Gamma \mid \Delta \vdash \varphi$. □

One might expect there to be a truth constant in the logic, just like the primitive falsity proposition $\bot$.[36] It turns out that it is not necessary to explicitly add such a constant, as we can easily represent it in $\mathbf{FOL}_{\blacktriangleright}$ as follows.

Definition 5.2.6. We define the *truth formula in* $\mathbf{FOL}_{\blacktriangleright}$ by $\top \coloneqq \underline{0} \sim \underline{0}$. ◂

The formula $\top$ has is the expected property that it is provable without further assumptions.

Lemma 5.2.7. *We have $\Gamma \mid \Delta \vdash \top$ in $\mathbf{FOL}_{\blacktriangleright}$ for any Γ and Δ.*

Proof. $\Gamma \mid \Delta \vdash \top$ follows from **(Refl)** and weakening. □

Before we give some examples, let us derive some general rules that will prove useful. First, just as convertibility is included as relation in observational equivalence, we have that definitional equality implies observational equivalence in $\mathbf{FOL}_{\blacktriangleright}$.

$$\frac{\Gamma \mid \Delta[\langle\rangle/x] \vdash \varphi[\langle\rangle/x]}{\Gamma, x : \mathbf{1} \mid \Delta \vdash \varphi}\ \textbf{(1-Ref)}$$

$$\frac{\Gamma, y : A \mid \Delta[\kappa_1\, y/x] \vdash \varphi[\kappa_1\, y/x] \qquad \Gamma, y : B \mid \Delta[\kappa_2\, y/x] \vdash \varphi[\kappa_2\, y/x]}{\Gamma, x : A + B \mid \Delta \vdash \varphi}\ \textbf{(+-Ref)}$$

$$\frac{\Gamma, y : A[\mu X.\, A/X] \mid \Delta[\alpha\, y/x] \vdash \varphi[\alpha\, y/x]}{\Gamma, x : \mu X.\, A \mid \Delta \vdash \varphi}\ \textbf{(μ-Ref)}$$

Figure 5.6.: Refinement Rules

Lemma 5.2.8. *The following rule can be derived in* $\mathbf{FOL}_{\blacktriangleright}$ *for all terms* s *and* t *with* $\Gamma \vdash_{\mathrm{Ty}} s, t : A$.

$$\frac{\Gamma \mid \Delta \vdash s \doteq_A t}{\Gamma \mid \Delta \vdash s \sim_A t}$$

Proof. We put $\varphi \coloneqq s \sim x$, so that $\Gamma, x : A \Vdash \varphi$ and $\Gamma \mid \emptyset \vdash \varphi[s/x]$ by **(Refl)**. Note that $\Delta[t/x] = \Delta$, as x is not in Δ. Through repeatedly applying **(Weak)**, we obtain $\Gamma \mid \Delta \vdash \varphi[s/x]$. Finally, we use **($\doteq$-Repl)** and $\Gamma \mid \Delta \vdash s \doteq_A t$ to deduce $\Gamma \mid \Delta \vdash \varphi[t/x]$. Since $\varphi[t/x] = s \sim t$, we get $\Gamma \mid \Delta \vdash s \sim t$. □

Second, we can obtain a few structural rules related to terms and term variables.[37]

Lemma 5.2.9. *In* $\mathbf{FOL}_{\blacktriangleright}$ *weakening for term variables is derivable for any type* $A \in \mathrm{Ty}$*:*

$$\frac{\Gamma \mid \Delta \vdash \varphi}{\Gamma, x : A \mid \Delta \vdash \varphi}$$

Moreover, the following substitution rule is derivable.

$$\frac{\Gamma, x : A \mid \Delta \vdash \varphi \qquad \Gamma \vdash_{\mathrm{Ty}} u : A}{\Gamma \mid \Delta[u/x] \vdash \varphi[u/x]}\ \textbf{(Subst)}$$

We also have the following exchange rule:

$$\frac{\Gamma \mid \Delta_1, \varphi_1, \varphi_2, \Delta_2 \vdash \psi}{\Gamma \mid \Delta_1, \varphi_2, \varphi_1, \Delta_2 \vdash \psi}\ \textbf{(Exch)}$$

Finally, we can derive the following conversion rule, which is a weak form of replacement.

$$\frac{\Gamma \mid \Delta[s/x] \vdash \varphi[s/x] \qquad s \equiv t}{\Gamma \mid \Delta[t/x] \vdash \varphi[t/x]}\ \textbf{(Conv)}$$

Proof. All the rules weakening, substitution, exchanges and conversion, are derived by induction on the proof trees. Most steps in these induction proofs are direct, so we only go through the critical cases here. For weakening, **(Refl)** is the crucial case, in which we use that the underlying term language has weakening as follows. If $\Gamma \mid \emptyset \vdash t \sim_B s$ is given by **(Refl)** from $\Gamma \vdash_{\mathrm{Ty}} t, s : B$ and $t \equiv s$, then also $\Gamma, x : A \vdash_{\mathrm{Ty}} t, s : B$ and $\Gamma, x : A \mid \emptyset \vdash t \sim_B s$ by **(Refl)**. For **(Subst)**, the **(Refl)** rule is again the crucial case, where we use that $s \equiv t$ implies $s[u/x] \equiv t[u/x]$. For proving **(Exch)**, on the other

hand, the important cases are **(Proj)**, **(→-I)** and **(Löb)**. Each of them is dealt with by using **(Weak)** in a standard manner. Finally, for the conversion rule **(Proj)**, **(Def)** and **(Refl)** are the critical cases. In the **(Proj)**-case, we use that the reduction relation preserves types (Theorem 3.2.22), whereas both the **(Def)**- and **(Refl)**-case are given by transitivity of conversion. This concludes the proof of all four derived rules. □

In many proofs, we will need that the later modality interacts well with other results that can be proven in $\mathbf{FOL}_{\blacktriangleright}$. For instance, if we derived an entailment $\varphi \vdash \psi$, then this entailment should also hold later, that is, $\blacktriangleright\varphi \vdash \blacktriangleright\psi$ should be derivable as well. This is formulated in the following lemma in a more general way, together with some useful consequences like the distribution of the later modality over the universal quantifier.

Lemma 5.2.10. *The following rule, which makes ▶ an applicative functor in the sense of [AM13], is derivable in* $\mathbf{FOL}_{\blacktriangleright}$.

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi \to \psi) \qquad \Gamma \mid \Delta \vdash \blacktriangleright\varphi}{\Gamma \mid \Delta \vdash \blacktriangleright\psi}\ \textbf{(Appl)}$$

More generally, the later modality can be distributed over rules: Let $\Gamma, \Delta, \varphi_1, \dots, \varphi_n$ *and* ψ *be given, such that for every* Δ' *the following is derivable*

$$\frac{\Gamma \mid \Delta, \Delta' \vdash \varphi_1 \qquad \cdots \qquad \Gamma \mid \Delta, \Delta' \vdash \varphi_n}{\Gamma \mid \Delta, \Delta' \vdash \psi} \tag{5.11}$$

in $\mathbf{FOL}_{\blacktriangleright}$, *then also the following is derivable*

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright\varphi_1 \qquad \cdots \qquad \Gamma \mid \Delta \vdash \blacktriangleright\varphi_n}{\Gamma \mid \Delta \vdash \blacktriangleright\psi} \tag{5.12}$$

In particular, the later modality distributes over $\forall$*-quantification and* **(×-Ext)**:

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright(\forall x : A.\, \varphi)}{\Gamma \mid \Delta \vdash \forall x : A.\, \blacktriangleright\varphi} \tag{5.13}$$

$$\frac{\Gamma \mid \Delta \vdash \blacktriangleright(\pi_1\ s \sim_A \pi_1\ t) \qquad \Gamma \mid \Delta \vdash \blacktriangleright(\pi_2\ s \sim_B \pi_2\ t)}{\Gamma \mid \Delta \vdash \blacktriangleright(s \sim_{A\times B} t)} \tag{5.14}$$

Proof. We first derive (5.12) from an arbitrary proof **(R)** of (5.11). Let $\Gamma, \Delta, \varphi_1, \dots, \varphi_n$ and ψ be given as in the assumption. We put $\Delta' := \varphi_1, \dots, \varphi_n$, and derive from (5.11) and repeated applications of **(→-I)**, **(K)** and **(→-E)** the following proof.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \mid \Delta, \Delta' \vdash \varphi_1}\textbf{(Proj)} \quad \cdots \quad \cfrac{}{\Gamma \mid \Delta, \Delta' \vdash \varphi_n}\textbf{(Proj)}}{\Gamma \mid \Delta, \Delta' \vdash \psi}\textbf{(R)}}{\Gamma \mid \Delta \vdash \varphi_1 \to \cdots \to \varphi_n \to \psi}\textbf{(→-I)}}{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi_1 \to \cdots \to \varphi_n \to \psi)}\textbf{(Nec)}}{\Gamma \mid \Delta \vdash \blacktriangleright\varphi_1 \to \cdots \to \blacktriangleright\varphi_n \to \blacktriangleright\psi}\textbf{(K)} \qquad \Gamma \mid \Delta \vdash \blacktriangleright\varphi_1 \quad \cdots \quad \Gamma \mid \Delta \vdash \blacktriangleright\varphi_n}{\Gamma \mid \Delta \vdash \blacktriangleright\psi}\textbf{(→-E)}$$

Both **(Appl)** and (5.14) are then given by instantiating (5.12) with **(→-E)** and **(×-Ext)**, respectively. Finally, (5.13) is given by

$$\dfrac{\dfrac{\Gamma, x : A \mid \Delta \vdash \blacktriangleright(\forall x : A.\, \varphi)}{\Gamma, x : A \mid \Delta \vdash \blacktriangleright \varphi}\,(*)}{\Gamma \mid \Delta \vdash \forall x : A.\, \blacktriangleright \varphi}\,(\forall\textbf{-I})$$

where **(∗)** is obtained from instantiating (5.12) with **(∀-E)**. □

We now demonstrate the proof system of $\mathbf{FOL}_{\blacktriangleright}$ on some examples. To make proofs more readable and to clarify the use of the later modality in combination with the fixed point rule, we will use two types of arrows to connect parts of a proof. Solid arrows connect two proof trees that could be just one tree, but have been split because of size constraints. On the other hand, dashed arrows point from where a hypothesis is used to the proof step where this hypothesis is introduced through the **(Löb)**-rule. This latter graphical notation makes it to easier grasp where hypotheses are used and introduced.

Example 5.2.11. In this first example, we will prove that the addition of natural numbers is commutative. Recall that we have defined in Example 3.1.7 the addition in $\lambda\mu\nu$ by iteration on the first argument. Analogously, we can define the addition in $\lambda\mu\nu=$ as follows.

$$\begin{aligned} \_ + \_ &: \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \\ 0 + m &= m \\ (\mathrm{suc}\, n) + m &= \mathrm{suc}\, (n + m) \end{aligned}$$

The goal is now to prove the formula $\forall n, m.\, n + m \sim m + n$ in $\mathbf{FOL}_{\blacktriangleright}$.

Towards this goal, we begin by proving four intermediate results. Recall that Nat was defined as $\mu X.\, \mathbf{1} + X$, and that 0 and the successor were defined in terms of the constructors for the least fixed point and the sum. Thus, our first result is a combination of the congruence rules for sums and least fixed points in the case of the natural numbers:

$$\left.\begin{array}{c} \dfrac{\dfrac{\Gamma \mid \Delta \vdash \blacktriangleright(n \sim_{\mathrm{Nat}} m)}{\Gamma \mid \Delta \vdash \blacktriangleright(\kappa_2\, n \sim_{\mathbf{1}+\mathrm{Nat}} \kappa_2\, m)}\,(+\textbf{-Cong})}{\Gamma \mid \Delta \vdash \mathrm{suc}\, n \sim_{\mathrm{Nat}} \mathrm{suc}\, m}\,(\mu\textbf{-Cong}) \end{array}\right\}(\mathrm{suc}\textbf{-Cong})$$

Similarly, the second result is a combination of refinement steps for the natural numbers. Note that $\varphi[\mathrm{suc}\, n/n] = \varphi[\alpha\, y/n][\kappa_2\, n/y]$ for some fresh variable y, and similarly for $\varphi[0/n]$ and Δ. Thus, we can apply **(+-Ref)** and **(μ-Ref)** as follows.

$$\left.\begin{array}{c} \dfrac{\dfrac{\Gamma \mid \Delta[0/n] \vdash \varphi[0/n] \qquad \Gamma, n : \mathrm{Nat} \mid \Delta[\mathrm{suc}\, n/n] \vdash \varphi[\mathrm{suc}\, n/n]}{\Gamma, y : \mathbf{1} + \mathrm{Nat} \mid \Delta[\alpha\, y/n] \vdash \varphi[\alpha\, y/n]}\,(+\textbf{-Ref})}{\Gamma, n : \mathrm{Nat} \mid \Delta \vdash \varphi}\,(\mu\textbf{-Ref}) \end{array}\right\}(\mathrm{Nat}\textbf{-Ref})$$

Third, we note that, by definition, the successor distributes over $+$ to the left summand:

$$\vdash \forall n, m.\, (\mathrm{suc}\, n) + m \sim \mathrm{suc}\, (n + m) \tag{5.15}$$

by the following proof in $\mathbf{FOL}_{\blacktriangleright}$.

$$\dfrac{\dfrac{(\mathrm{suc}\, n) + m \equiv \mathrm{suc}\,(n + m)}{n, m : \mathrm{Nat} \mid \emptyset \vdash (\mathrm{suc}\, n) + m \sim \mathrm{suc}\,(n + m)}\,\textbf{(Refl)}}{\vdash \forall n, m.\, (\mathrm{suc}\, n) + m \sim \mathrm{suc}\,(n + m)}\,\textbf{(}\forall\textbf{-I)}$$

Lastly, the analogous property for the right summand, that is

$$\vdash \forall n, m.\, n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m), \tag{5.16}$$

also holds as follows. First, we note that the definition of $+$ gives us the following computations.

$$0 + \mathrm{suc}\, m \equiv \mathrm{suc}\, m \equiv \mathrm{suc}\,(0 + m) \tag{5.17}$$
$$(\mathrm{suc}\, n) + (\mathrm{suc}\, m) \equiv \mathrm{suc}(n + (\mathrm{suc}\, m)) \tag{5.18}$$
$$\mathrm{suc}\,((\mathrm{suc}\, n) + m) \equiv \mathrm{suc}\,(\mathrm{suc}\,(n + m)) \tag{5.19}$$

The proof of the successor distribution to the right is our first proof that uses the **(Löb)**-rule. As discussed, we indicate the recursion step by a dashed arrow in the proof. We put $\Gamma \coloneqq n, m : \mathrm{Nat}$ and $\Delta \coloneqq \blacktriangleright(\forall n, m.\, n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m))$, and prove (5.16) by induction on n as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \blacktriangleright(n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m))}\,\textbf{(Proj)}, (\forall\textbf{-E})}{\Gamma \mid \Delta \vdash \mathrm{suc}\,(n + (\mathrm{suc}\, m)) \sim \mathrm{suc}\,(\mathrm{suc}\,(n + m))}\,(\mathrm{suc}\textbf{-Cong})}{\Gamma \mid \Delta \vdash (\mathrm{suc}\, n) + (\mathrm{suc}\, m) \sim \mathrm{suc}\,((\mathrm{suc}\, n) + m)}\,(5.18), (5.19)\,\textbf{(Conv)}}{\Gamma \mid \Delta \vdash n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m)}\,(5.17)\,(\mathrm{Nat}\textbf{-Ref})}{n : \mathrm{Nat} \mid \Delta \vdash \forall m.\, n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m)}\,(\forall\textbf{-I})}{\Delta \vdash \forall n\, m.\, n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m)}\,(\forall\textbf{-I})}{\vdash \forall n\, m.\, n + (\mathrm{suc}\, m) \sim \mathrm{suc}\,(n + m)}\,\textbf{(Löb)}$$

Finally, we can put the above results together to prove commutativity of addition. In the proof below we use the assumptions Δ and the context Γ, given by $\Delta \coloneqq \blacktriangleright(\forall n, m.\, n + m \sim m + n)$ and $\Gamma \coloneqq n, m : \mathrm{Nat}$, respectively. The proof of commutativity is given as follows.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash 0 + 0 \sim 0 + 0} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \blacktriangleright(0 + m \sim m + 0)}\,\textbf{(Proj)}, (5.13), (\forall\textbf{-E})}{\Gamma \mid \Delta \vdash \mathrm{suc}\,(0 + m) \sim \mathrm{suc}\,(m + 0)}\,(\mathrm{suc}\textbf{-Cong})}{\Gamma \mid \Delta \vdash 0 + (\mathrm{suc}\, m) \sim (\mathrm{suc}\, m) + 0}\,(5.16), \textbf{(Trans)}, (5.15)}{\Gamma \mid \Delta \vdash 0 + m \sim m + 0}\,(\mathrm{Nat}\textbf{-Ref})}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \blacktriangleright(n + m \sim m + n)}\,\textbf{(Proj)}, (5.13), (\forall\textbf{-E})}{\Gamma \mid \Delta \vdash \mathrm{suc}\,(n + m) \sim \mathrm{suc}\,(m + n)}\,(\mathrm{suc}\textbf{-Cong})}{\Gamma \mid \Delta \vdash (\mathrm{suc}\, n) + m \sim m + (\mathrm{suc}\, n)}\,(5.15), \textbf{(Trans)}, (5.16)}{\Gamma \mid \Delta \vdash n + m \sim m + n}\,(\mathrm{Nat}\textbf{-Ref})}{\vdash \forall n, m.\, n + m \sim m + n}\,\textbf{(Löb)} + (\forall\textbf{-I})$$

Note that we used two recursion steps in this proof, as the proof proceeds by induction both on n and on m. Traditionally, this proof would be broken up into two separate induction proofs to fit the usual induction scheme. However, in the present setup we can just follow the natural steps that arise in the proof. ◀

Let us now use $\mathbf{FOL}_{\blacktriangleright}$ to prove some equivalences on coinductive types. In the first example, we look again at the stream equivalence that we proved in Example 5.1.11. This allows us to demonstrate how proofs by coinduction arise from combining the extensionality rules for products and greatest fixed points, and the **(Löb)**-rule.

Example 5.2.12. Recall from Example 3.2.11 that we have defined a map that selects the odd positions of a stream, and that we have proved in Example 5.1.11 that the application of this map to the alternating bit stream results in a constant stream. In Example 5.1.11, we used the bisimulation proof method to show this equivalence. We will now prove it again but this time around in $\mathbf{FOL}_{\blacktriangleright}$. That is, we show that the formula $\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega}$ is derivable in $\mathbf{FOL}_{\blacktriangleright}$.

Intuitively, to show that two streams are equivalent, we need to show that their heads match and their tails continue to be equivalent. This extensionality principle for streams can be derived in $\mathbf{FOL}_{\blacktriangleright}$, similarly to the refinement rule for natural numbers:

$$\left.\begin{array}{l} \textbf{(Nec)}\ \dfrac{\Gamma \mid \Delta \vdash s.\mathrm{hd} \sim_A t.\mathrm{hd}}{\dfrac{\Gamma \mid \Delta \vdash \blacktriangleright(s.\mathrm{hd} \sim_A t.\mathrm{hd}) \qquad \Gamma \mid \Delta \vdash \blacktriangleright(s.\mathrm{tl} \sim_{A^\omega} t.\mathrm{tl})}{\Gamma \mid \Delta \vdash \blacktriangleright(s.\mathrm{out} \sim t.\mathrm{out})}\,(5.14)}{\Gamma \mid \Delta \vdash s \sim_{A^\omega} t}\,(\nu\textbf{-Ext}) \end{array}\right\} (A^\omega\textbf{-Ext})$$

Combined with the fixed point rule **(Löb)**, this extensionality principle allows us to prove stream equivalences, as we will demonstrate now.

We want to show that $\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega}$ is derivable. This is accomplished by the following proof, in which we use Δ given by $\Delta \coloneqq \blacktriangleright(\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega})$, and the following two computations.

$$(\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt}).\mathrm{tl} \equiv \mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \tag{5.20}$$

$$\underline{1}^{\omega}.\mathrm{tl} \equiv \underline{1}^{\omega} \tag{5.21}$$

$$\dfrac{\dfrac{\Delta \vdash (\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt}).\mathrm{hd} \equiv \underline{1}^{\omega}.\mathrm{hd} \qquad \dfrac{\dfrac{}{\Delta \vdash \blacktriangleright(\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega})}\,\textbf{(Proj)} \quad (5.20),\,(5.21)}{\Delta \vdash \blacktriangleright((\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt}).\mathrm{tl} \sim \underline{1}^{\omega}.\mathrm{tl})}\,\textbf{(Conv)}}{\Delta \vdash (\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt}).\mathrm{hd} \sim_A (\underline{1}^{\omega}).\mathrm{hd}}\,(\mathrm{Nat}^\omega\textbf{-Ext})}{\dfrac{\Delta \vdash \mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega}}{\vdash \mathrm{select}\,\mathrm{oddF}\,\mathrm{alt} \sim \underline{1}^{\omega}}\,\textbf{(Löb)}}$$

This proof should be compared to Example 5.1.11, where we proved that the relation R, which was given by $R_{\mathrm{Nat}^\omega} = \{(\mathrm{select}\,\mathrm{oddF}\,\mathrm{alt}, \underline{1}^{\omega})\}$, is an observational bisimulation up-to. However, there we had to use a clever combination of up-to techniques, whereas in the present proof the steps that we have to take arise naturally in the construction of the proof. It should be noted though that up-to techniques still play an important role in the semantics of $\mathbf{FOL}_{\blacktriangleright}$, see Section 5.2.2. ◀

In the next example we prove that addition of streams is commutative. This demonstrates coinduction for streams on a quantified formula.

Example 5.2.13. Recall that streams of natural numbers are given by $\mathrm{Nat}^\omega = \nu X.\, \mathrm{Nat} \times X$, see Example 3.1.2, and from Example 3.1.8 that stream addition is defined from addition on Nat by

$$\begin{aligned} \_ \oplus \_ &: \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \to \mathrm{Nat}^\omega \\ (s \oplus t).\mathrm{hd} &= s.\mathrm{hd} + t.\mathrm{hd} \\ (s \oplus t).\mathrm{tl} &= s.\mathrm{tl} \oplus t.\mathrm{tl} \end{aligned}$$

To show that $\oplus$ is commutative, we first observe that by commutativity of addition on natural numbers (Example 5.2.11) we have for all $s, t : \mathrm{Nat}^\omega$

$$(s \oplus t).\mathrm{hd} \equiv s.\mathrm{hd} + t.\mathrm{hd} \sim t.\mathrm{hd} + s.\mathrm{hd} \equiv (t \oplus s).\mathrm{hd}, \tag{5.22}$$

and from the definition of stream addition we obtain

$$(s \oplus t).\mathrm{tl} \equiv s.\mathrm{tl} \oplus t.\mathrm{tl} \qquad \text{and} \qquad t.\mathrm{tl} \oplus s.\mathrm{tl} \equiv (t \oplus s).\mathrm{tl} \tag{5.23}$$

Let $\Delta \coloneqq \blacktriangleright(\forall s, t : \mathrm{Nat}^\omega.\, s \oplus t \sim t \oplus s)$ and $\Gamma \coloneqq s, t : \mathrm{Nat}^\omega$, then the following is a proof in $\mathbf{FOL}_\blacktriangleright$.

$$\dfrac{\dfrac{(5.22)}{\Gamma \mid \Delta \vdash (s \oplus t).\mathrm{hd} \sim (t \oplus s).\mathrm{hd}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta \vdash \forall s, t : \mathrm{Nat}^\omega.\, \blacktriangleright(s \oplus t \sim t \oplus s)}\,\textbf{(Proj)}, (5.13)}{\Gamma \mid \Delta \vdash \blacktriangleright(s.\mathrm{tl} \oplus t.\mathrm{tl} \sim t.\mathrm{tl} \oplus s.\mathrm{tl})}\,\textbf{(}\forall\textbf{-E)}}{\Gamma \mid \Delta \vdash \blacktriangleright((s \oplus t).\mathrm{tl} \sim (t \oplus s).\mathrm{tl})}\,\textbf{(Conv)}, (5.23)}{\Gamma \mid \Delta \vdash s \oplus t \sim t \oplus s}\,(\mathrm{Nat}^\omega\textbf{-Ext})$$
$$\dfrac{\Gamma \mid \Delta \vdash s \oplus t \sim t \oplus s}{\Delta \vdash \forall s, t : \mathrm{Nat}^\omega.\, s \oplus t \sim t \oplus s}\,\textbf{(}\forall\textbf{-I)}$$
$$\dfrac{\Delta \vdash \forall s, t : \mathrm{Nat}^\omega.\, s \oplus t \sim t \oplus s}{\vdash \forall s, t : \mathrm{Nat}^\omega.\, s \oplus t \sim t \oplus s}\,\textbf{(Löb)}$$

This shows that $\oplus$ is commutative. ◂

In the final example of this section we derive the $\mathbf{FOL}_\blacktriangleright$-version of the induction principle for natural numbers.

Example 5.2.14. Recall from Definition 5.2.2 that for formula φ with $\Gamma, n : \mathrm{Nat} \Vdash \varphi$, we denote for $t : \mathrm{Nat}$ the formula $\varphi[t/n]$ by $\varphi[t]$. Usually, the induction principle for natural numbers reads like

$$\frac{\Gamma \vdash \varphi[0] \qquad \Gamma, m : \mathrm{Nat} \mid \varphi[m] \vdash \varphi[\mathrm{suc}\, m]}{\Gamma, n : \mathrm{Nat} \vdash \varphi}$$

In the context of the logic $\mathbf{FOL}_\blacktriangleright$, we cannot derive the rule in this form but we need to introduce a later modality on the induction hypothesis:

$$\frac{\Gamma \mid \Delta \vdash \varphi[0] \qquad \Gamma \mid \Delta \vdash \forall n : \mathrm{Nat}.\, \blacktriangleright \varphi[n] \to \varphi[\mathrm{suc}\, n]}{\Gamma \mid \Delta \vdash \forall n : \mathrm{Nat}.\, \varphi}\,\textbf{(Nat-Ind)}$$

This rule can be derived by using the fixed point rule and the refinement rules as in the following

proof, in which we let $\Delta' \coloneqq \Delta, \blacktriangleright \forall n.\, \varphi$, and use that $\Delta'[0/n] = \Delta'$ and $\Delta'[\mathrm{suc}\, n/n] = \Delta'$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta' \vdash \forall n : \mathrm{Nat}.\ \blacktriangleright \varphi[n] \to \varphi[\mathrm{suc}\, n]}\,(\forall\text{-}\mathbf{E}) \quad \dfrac{\dfrac{}{\Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \blacktriangleright \forall n.\, \varphi}\,\mathbf{(Proj)}}{\Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \forall n.\ \blacktriangleright \varphi}\,(5.13)}{\Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \blacktriangleright \varphi[m] \to \varphi[\mathrm{suc}\, m]}\,(\forall\text{-}\mathbf{E}) \quad \Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \blacktriangleright \varphi[m]}{\Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \varphi[\mathrm{suc}\, m]}\,(\to\text{-}\mathbf{E})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \Delta' \vdash \varphi[0]} \quad \Gamma, m : \mathrm{Nat} \mid \Delta' \vdash \varphi[\mathrm{suc}\, m]}{\Gamma, n : \mathrm{Nat} \mid \Delta' \vdash \varphi}\,(\mathrm{Nat}\text{-}\mathbf{Ref})}{\Gamma \mid \Delta, \blacktriangleright \forall n.\, \varphi \vdash \forall n.\, \varphi}\,(\forall\text{-}\mathbf{I})}{\Gamma \mid \Delta \vdash \forall n.\, \varphi}\,(\mathbf{Löb})$$

Towards the end of this section, we will see that such an induction principle holds in $\mathbf{FOL}_{\blacktriangleright}$ for a more general class of types. ◀

5.2.2. A Model, Soundness and Incompleteness

Having an intuitive understanding of the logic $\mathbf{FOL}_{\blacktriangleright}$ is all nice and well, but to make the logic actually useful, we need to show that the derivable formulas are also sensible in the interpretation we intend to give them. For instance, we should prove that if $s \sim t$ is derivable in $\mathbf{FOL}_{\blacktriangleright}$, then $s \equiv_{\mathrm{obs}} t$. To this end, we construct a model, in which we can interpret the formulas and for which the axioms are sound. This model follows largely the idea described in the introduction of this section, namely we will interpret formulas relative to a natural number that denotes a stage of the approximation of observational equivalence. As such, we obtain a Kripke-style semantics, in which the worlds of a frame are given by natural numbers.

There are, however, two things that we have to tweak on the naive semantics that we gave in the introduction of this section. First of all, it should be noted that the type-specific rules for the equality in Figure 5.4 on page 130 only use the later modality at fixed point types. As we will see in Lemma 5.2.28, the later modality is used to control the application of the operator Φ that defines observational equivalence. Thus, a naive interpretation of $\sim$ as approximation of $\equiv_{\mathrm{obs}}$ would require us to use the later modality in all rules of Figure 5.4. Since this is not desirable, we first give an alternative way to prove and construct observational equivalence, which allows us to leave the later modality out at non-fixed point types. Second, we need to interpret the first-order connectives of $\mathbf{FOL}_{\blacktriangleright}$. For the sake of an accessible exposition, we will interpret them as Boolean connectives. As it turns out, the easiest way to then give an interpretation to formulas is by assigning to each formula φ a decreasing chain $[\![\varphi]\!] \in [\omega^{\mathrm{op}}, \mathbb{B}]$ over the booleans $\mathbb{B}$, since this presheaf category supports all the Boolean connectives that we require, see Lemma 2.3.3.

The idea to interpret a formula as a decreasing chain in $[\omega^{\mathrm{op}}, \mathbb{B}]$ follows the ideas of Birkedal et al. [Bir+11] and Appel et al. [App+07], who give an interpretation of a type theory with a later modality in terms of chains in $[\omega^{\mathrm{op}}, \mathbf{Set}]$. In contrast, our interpretation is based on relation- and Boolean-valued presheafs in $[\omega^{\mathrm{op}}, \mathrm{Rel}_{\Lambda=}]$ and $[\omega^{\mathrm{op}}, \mathbb{B}]$, respectively. This makes for a more accessible interpretation than casting the model in $[\omega^{\mathrm{op}}, \mathbf{Set}]$, where we would have to work with specific sets that represent truth-values, cf. the proof of Lemma 2.3.3.

Notation 5.2.15. Since we focus on terms in the copattern calculus $\lambda\mu\nu{=}$ in this chapter, we abuse notation and write Rel instead of $\mathrm{Rel}_{\Lambda^=}$ for the complete lattice of (type-indexed) binary relations over terms in $\lambda\mu\nu{=}$.

In what follows, we need to characterise the components of closed types.

Lemma 5.2.16. *Let A be a closed type. Then either there are closed types B and C, such that $A = B \times C$, $A = B + C$ or $A = B \to C$, or there is $X \Vdash B : \mathbf{Ty}$ and $\rho \in \{\mu, \nu\}$ with $A = \rho X.\, B$. In the first case, we refer to B and C as the* components *of A.*

Proof. This follows directly by induction on the derivation of $\Vdash A : \mathbf{Ty}$. □

As we mentioned already above, we need a different way of proving observational equivalence that allows us to leave out intermediate steps in a bisimulation. For example, if we want to prove that R is an observational bisimulation at the type of streams over A, then the results in Section 5.1.2 require us to show for all $(s,t) \in R_{A^\omega}$ that $(s.\mathrm{out}, t.\mathrm{out}) \in C^{\mathrm{obs}}(R)_{A\times A^\omega}$. Typically, we are only interested in the head and tail of streams though. Thus, it would be more natural to leave out the intermediate step and only require that $(s.\mathrm{hd}, t.\mathrm{hd}) \in C^{\mathrm{obs}}(R)_A$ and $(s.\mathrm{tl}, t.\mathrm{tl}) \in C^{\mathrm{obs}}(R)_{A^\omega}$. We can achieve this simplification by using the following notion of compressed bisimulation. Note that the use of compressed bisimulations is very similar to the use of Φ as an up-to technique.

Definition 5.2.17. Let $\Phi_c : \mathrm{Rel} \to \mathrm{Rel}$ be the operator defined by

$$\Phi_c(R)_A = \begin{cases} \Phi(R)_A, & A = \rho X.\, B, \rho \in \{\mu,\nu\} \\ \Phi(\Phi_c(R))_A, & \text{otherwise.} \end{cases}$$

A *compressed observational bisimulation*, Φ_c-bisimulation for short, is a relation $R \in \mathrm{Rel}$, such that $R \sqsubseteq \Phi_c(R)$. ◂

Note that Φ_c in Definition 5.2.17 is well-defined because in the second clause A is a product, a sum or a function space, hence Φ only refers to the components of these types, see Definition 5.1.6 and Lemma 5.1.8. For instance, if $A = B \to C$, then we have that

$$\begin{aligned}\Phi_c(R)_{B\to C} = \Phi(\Phi_c(R))_{B\to C} &= \{(t_1,t_2) \mid (\delta(t_1),\delta(t_2)) \in \overline{F}(\Phi_c(R))_{B\to C}\} \\ &= \{(t_1,t_2) \mid (\delta(t_1),\delta(t_2)) \in \langle F(\pi_1), F(\pi_2)\rangle(F(\Phi_c(R)))_{B\to C}\}\end{aligned}$$

and since $F(\Phi_c(R))_{B\to C} = \mathcal{P}(\Phi_c(R)_B)^{\mathbf{ON}_A}$, the definition of $\Phi_c(R)_{B\to C}$ only depends on $\Phi_c(R)_B$. By Lemma 5.2.16 we eventually arrive at the first case.

As one might expect, a compressed observational bisimulation can be completed to an actual observational bisimulation. For instance, in the stream example we can add all pairs $(s.\mathrm{out}, t.\mathrm{out})$, for which $(s,t) \in R$, $(s.\mathrm{hd}, t.\mathrm{hd}) \in C^{\mathrm{obs}}(R)_A$ and $(s.\mathrm{tl}, t.\mathrm{tl}) \in C^{\mathrm{obs}}(R)_{A^\omega}$. Instead of constructing such a bisimulation, we can show directly that a Φ_c-bisimulation is a Φ-bisimulation up to γ_Φ, where γ_Φ is the companion of Φ from Definition 2.5.12. This approach significantly simplifies the proof because the construction of a Φ-bisimulation from a Φ_c-bisimulation is quite involved.

Lemma 5.2.18. *If R is a Φ_c-bisimulation, then R is a Φ-bisimulation up to γ_Φ. Thus, if two terms s and t are related by R, then $s \equiv_{\mathrm{obs}} t$.*

Proof. Let R be a Φ_c-bisimulation. We prove $R_A \subseteq \Phi(\gamma_\Phi(R))_A$ by induction on A, see Lemma 5.2.16. If A is a fixed point type, that is, if $A = \rho X.\, B$, then

$$R_A \subseteq \Phi_c(R)_A = \Phi(R)_A \subseteq \Phi(\gamma_\Phi(R))_A,$$

by the assumption that R is a Φ_c-bisimulation, the definition of Φ_c and $\mathrm{id} \sqsubseteq \gamma_\Phi$ (Lemma 2.5.13). Otherwise, if A is not a fixed point type, then

$$R_A \subseteq \Phi_c(R)_A = \Phi(\Phi_c(R))_A \subseteq \Phi(\Phi(\gamma_\Phi(R)))_A \subseteq \Phi(\gamma_\Phi(R))_A,$$

by definition of Φ_c, the induction hypothesis that $R_B \subseteq \Phi(\gamma_\Phi(R))_B$ for components B of A, and $\Phi \sqsubseteq \gamma_\Phi$ (Lemma 2.5.13). Thus, $R \sqsubseteq \Phi(\gamma_\Phi(R))$ as required. □

We are now in the position to construct a model for the logic $\mathbf{FOL}_\blacktriangleright$. Since formulas of the form $s \sim t$ will be interpreted in the approximations of Φ_c-bisimilarity, the first step is to define these approximations.

Definition 5.2.19. The chain of approximations of compressed observational bisimilarity is the sequence $\overleftarrow{\Phi_c} \in [\omega^{\mathrm{op}}, \mathrm{Rel}]$ defined inductively by

$$\overleftarrow{\Phi_c}(0) \coloneqq \mathbf{I} \coloneqq \Lambda^= \times \Lambda^=$$
$$\overleftarrow{\Phi_c}(n+1) \coloneqq \Phi_c\big(\overleftarrow{\Phi_c}(n)\big).$$

Note that this is indeed an ω^{op}-chain because $\overleftarrow{\Phi_c}(\mathbf{I})(n) \supseteq \overleftarrow{\Phi_c}(\mathbf{I})(n+1)$ for all $n \in \mathbb{N}$ by $\mathbf{I} \supseteq \Phi(\mathbf{I})$ and monotonicity of Φ_c. ◂

The crucial result is now that we can prove observational equivalence of two terms by proving that they are related by every approximation of Φ_c-bisimilarity.

Lemma 5.2.20. *Let A be a type in* Ty *and s, t terms in $\Lambda^=(A)$. If $(s,t) \in \overleftarrow{\Phi_c}(n)_A$ for all $n \in \mathbb{N}$, then $s \equiv_{\mathrm{obs}} t$.*

Proof. One first proves that Φ_c is ω^{op}-continuous, that is, for any family $\{R_i\}_{i\in\mathbb{N}}$ of $\equiv$-closed relations $R_i \in \mathrm{Rel}$ with $R_{i+1} \sqsubseteq R_i$, we have $\Phi_c(\bigsqcap_{i\in\mathbb{N}} R_i) = \bigsqcap_{i\in\mathbb{N}} \Phi_c(R_i)$. This follows easily from the characterisation of Φ in Lemma 5.1.8, where care has to be taken only in the case of inductive types. There, the limit preservation crucially requires confluence of the reduction relation in $\lambda\mu\nu{=}$. Since Φ_c is ω^{op}-continuous, the chain $\overleftarrow{\Phi_c}$ stabilises, see Definition 2.5.5. Thus, by Lemma 2.5.6, we get that the largest fixed point $\nu\Phi_c \in \mathrm{Rel}$ of Φ_c is given as $\nu\Phi_c = \bigsqcap_{n\in\mathbb{N}} \overleftarrow{\Phi_c}(n)$, see also [CC79; San09]. Thus, s and t are related by $\nu\Phi_c$ if and only if they are related by $\overleftarrow{\Phi}(n)$ for every n. Since $\nu\Phi_c$ is a Φ_c-bisimulation, we obtain $s \equiv_{\mathrm{obs}} t$ from Lemma 5.2.18. □

Recall from Lemma 2.3.3 that $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$ is the two-valued Boolean algebra and that $[\omega^{\mathrm{op}}, \mathbb{B}]$ is Cartesian closed, complete and cocomplete (Proposition 2.3.2). For brevity, we denote this category by

$$\mathbf{T} \coloneqq [\omega^{\mathrm{op}}, \mathbb{B}]\,. \tag{5.24}$$

Given a set I, we denote the I-indexed product and coproduct by $\bigwedge_I \colon \mathbf{T}^I \to \mathbf{T}$ and $\bigvee_I \colon \mathbf{T}^I \to \mathbf{T}$, respectively. If I is a two element set, then we denote the resulting binary product and coproduct by $\wedge$ and $\vee$. Completeness and cocompleteness also give rise to initial and terminal objects that are given by

$$\forall n \in \mathbb{N}. \quad \mathbf{0}(n) = \mathrm{ff} \quad \text{and} \quad \mathbf{1}(n) = \mathrm{tt}.$$

Finally, the exponential in $\mathbf{T}$ is denoted by $\Rightarrow \colon \mathbf{T}^{\mathrm{op}} \times \mathbf{T} \to \mathbf{T}$, see the discussion after Lemma 2.3.3.

To make the semantics of the logic more readable, we interpret the membership for pairs of terms in the chain $\overleftarrow{\Phi_c}$ as a chain in $\mathbf{T}$ as follows.

Definition 5.2.21. For a type $A \in \mathrm{Ty}$ and terms $s, t \in \Lambda^{=}(A)$, we define a chain $s \approx_c t \in \mathbf{T}$:

$$\forall n \in \mathbb{N}. (s \approx_c t)(n) \coloneqq (s, t) \in \overleftarrow{\Phi_c}(n)_A. \tag{5.25}$$

That $s \approx_c t$ is a chain in $\mathbf{T}$ follows directly from the fact that $\overleftarrow{\Phi_c} \in [\omega^{\mathrm{op}}, \mathrm{Rel}]$. ◀

The model of $\mathbf{FOL}_{\blacktriangleright}$ is now given by the following Kripke-style semantics. Here, $\triangleright \colon \mathbf{T} \to \mathbf{T}$ is the functor with $(\triangleright \zeta)(0) = \mathrm{tt}$ and $(\triangleright \zeta)(k+1) = \zeta(k)$ that we introduced in Lemma 2.3.4. Moreover, we denote by $\lambda\!\!\lambda$ the introduction of a set-theoretic function.

Definition 5.2.22. Let φ be a formula of $\mathbf{FOL}_{\blacktriangleright}$ with $\Gamma \Vdash \varphi$, see Definition 5.2.1. We define for $\sigma \in \mathrm{Subst}(\mathbf{ON}; \Gamma)$ a chain $[\![\varphi]\!](\sigma) \in \mathbf{T}$ by induction on φ.

$$\begin{aligned}
[\![\bot]\!](\sigma) &\coloneqq \mathbf{0} \\
[\![s \doteq t]\!](\sigma) &\coloneqq \lambda\!\!\lambda n.\, s[\sigma] \equiv t[\sigma] \\
[\![s \sim t]\!](\sigma) &\coloneqq s[\sigma] \approx_c t[\sigma] \\
[\![\blacktriangleright \varphi]\!](\sigma) &\coloneqq \triangleright [\![\varphi]\!](\sigma) \\
[\![\varphi \wedge \psi]\!](\sigma) &\coloneqq [\![\varphi]\!](\sigma) \wedge [\![\psi]\!](\sigma) \\
[\![\varphi \vee \psi]\!](\sigma) &\coloneqq [\![\varphi]\!](\sigma) \vee [\![\psi]\!](\sigma) \\
[\![\varphi \to \psi]\!](\sigma) &\coloneqq [\![\varphi]\!](\sigma) \Rightarrow [\![\psi]\!](\sigma) \\
[\![\forall x : A.\, \varphi]\!](\sigma) &\coloneqq \textstyle\bigwedge_{t \in \mathbf{ON}_A} [\![\varphi]\!](\sigma[x \mapsto t]) \\
[\![\exists x : A.\, \varphi]\!](\sigma) &\coloneqq \textstyle\bigvee_{t \in \mathbf{ON}_A} [\![\varphi]\!](\sigma[x \mapsto t])
\end{aligned}$$

Suppose Δ is a sequence of formulas with $\Gamma \Vdash \Delta$. Given $n \in \mathbb{N}$, a formula φ and $\sigma \in \mathrm{Subst}(\mathbf{ON}; \Gamma)$, we write

$$\sigma; n \vDash \varphi \quad \coloneqq \quad [\![\varphi]\!](\sigma)(n),$$

and say that φ is *satisfied at stage n and σ*. Next, given a sequence Δ of formulas we say that φ is *satisfied under the assumption in Δ at σ*, if whenever all the formulas in Δ are satisfied, then φ is satisfied:

$$\Delta; \sigma; n \vDash \varphi \quad \coloneqq \quad \text{if } (\forall \psi \in \Delta.\, \sigma; n \vDash \psi), \text{ then } \sigma; n \vDash \varphi.$$

Finally, we say that φ is *satisfied under the assumptions in Δ*, if $\Delta; \sigma; n \vDash \varphi$ holds for all σ:

$$\Gamma \mid \Delta; n \vDash \varphi \quad \coloneqq \quad \forall \sigma \in \mathrm{Subst}(\mathbf{ON}; \Gamma).\, \Delta; \sigma; n \vDash \varphi \qquad ◀$$

We note for $\zeta \in \mathbf{T}$ that $\forall n \in \mathbb{N}.\, \zeta(n)$ holds if and only if there exists a morphism $\mathbf{1} \to \zeta$ in $\mathbf{T}$. Thus, we frequently use the equivalence

$$\forall n \in \mathbb{N}.\, n \vDash \varphi \text{ holds iff the hom-set } \mathbf{T}(\mathbf{1}, [\![\varphi]\!]) \text{ is inhabited.}$$

The following proposition provides a more general formulation of this fact.

Proposition 5.2.23. *Given a sequence Δ of formulas, a formula φ and a substitution σ, we have that*

$$\text{there is a morphism } \left(\textstyle\bigwedge_{\psi \in \Delta} [\![\psi]\!](\sigma)\right) \to [\![\varphi]\!](\sigma) \text{ in } \mathbf{T} \quad \text{iff} \quad \forall n \in \mathbb{N}.\, \Delta \mid ; \sigma; n \vDash \varphi. \qquad \square$$

We now show in a series of lemmas that the axioms of $\mathbf{FOL}_{\blacktriangleright}$ are sound for the interpretation in Definition 5.2.22. The final result we aim for is the following theorem.

Theorem 5.2.24. *The axioms of $\mathbf{FOL}_{\blacktriangleright}$ are sound for the interpretation in Definition 5.2.22, in the sense that for all formulas φ,*

$$\text{if } \Gamma \mid \Delta \vdash \varphi, \text{ then } \forall n \in \mathbb{N}.\ \Gamma \mid \Delta; n \vDash \varphi.$$

In particular, if $\vdash s \sim t$, then $s \equiv_{\mathrm{obs}} t$.

The first step towards proving Theorem 5.2.24 is to show the validity of the rules that make $\sim$ an equivalence relation. We prove this by appealing to the following result that allows us to infer properties of the approximations $\overleftarrow{\Phi}$ from compatible up-to techniques, for which we recall from Definition 2.5.10 that $\Phi_c^{\times m} \colon L^m \to L^m$ is given by point-wise application of Φ_c.

Lemma 5.2.25. *Let $T \colon \mathrm{Rel}^m \to \mathrm{Rel}$ be Φ_c-compatible, that is $T \circ \Phi_c^{\times m} \sqsubseteq \Phi_c \circ T$. Then we have $T(\overleftarrow{\Phi_c}(n)^m) \sqsubseteq \overleftarrow{\Phi_c}(n)$ for all $n \in \mathbb{N}$.*

Proof. We proceed by induction on n. If $n = 0$, then $\overleftarrow{\Phi_c}(0) = \mathbf{I}$ and the claim holds trivially. Otherwise, assume that $T(\overleftarrow{\Phi_c}(n)^m) \sqsubseteq \overleftarrow{\Phi_c}(n)$ holds for $n \in \mathbb{N}$, and we prove $T(\overleftarrow{\Phi_c}(n+1)^m) \sqsubseteq \overleftarrow{\Phi_c}(n+1)$. Indeed, we have

$$\begin{aligned}
T(\overleftarrow{\Phi_c}(n+1)^m) &= T(\Phi_c(\overleftarrow{\Phi_c}(n))^m) \\
&\sqsubseteq \Phi_c(T(\overleftarrow{\Phi_c}(n)^m)) && \text{by compatibility} \\
&\sqsubseteq \Phi_c(\overleftarrow{\Phi_c}(n)) && \text{by IH and monotonicity of } \Phi_c \\
&= \overleftarrow{\Phi_c}(n+1)
\end{aligned}$$

Thus, by induction, $T(\overleftarrow{\Phi_c}(n)^m) \sqsubseteq \overleftarrow{\Phi_c}(n)$ holds for all $n \in \mathbb{N}$. $\square$

Since we have already investigated some up-to techniques for Φ, we wish to reuse these techniques for compressed bisimulations. The following result allows us to transfer compatibility of up-to techniques for Φ to Φ_c, if the up-to technique in question is given on each type separately.

Lemma 5.2.26. *Let $T \colon \mathrm{Rel}^m \to \mathrm{Rel}$ be Φ-compatible and assume that T factors through a family of maps $\{S_A \colon \mathrm{Rel}^m_{\Lambda^=(A)} \to \mathrm{Rel}_{\Lambda^=(A)}\}_{A \in \mathrm{Ty}}$, in the sense that for any m-tuple of relations $\vec{R}$, we have $T(\vec{R})_A = S_A(\overrightarrow{R_A})$. Under these conditions, T is also Φ_c-compatible.*

Proof. We show that for any m-tuple $\vec{R}$ of relations and type A that $T\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)_A \subseteq \Phi_c\big(T\big(\vec{R}\big)\big)_A$ by induction on A. If $A = \rho X.\, B$, then

$$\begin{aligned}
T\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)_A &= S_A\big(\Phi_c^{\times m}\big(\vec{R}\big)_A\big) && \text{by factorisation of } T\\
&= S_A\Phi^{\times m}\big(\vec{R}\big)_A && \text{by definition of } \Phi_c\\
&= T\big(\Phi^{\times m}\big(\vec{R}\big)\big)_A && \text{by factorisation of } T\\
&\subseteq \Phi\big(T\big(\vec{R}\big)\big)_A && \text{by compatibility}\\
&= \Phi_c\big(T\big(\vec{R}\big)\big)_A && \text{by definition of } \Phi_c.
\end{aligned}$$

Otherwise, if A is not a fixed point type, then we have

$$\begin{aligned}
T\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)_A &= S_A\big(\Phi_c^{\times m}\big(\vec{R}\big)_A\big) = S_A\big(\Phi^{\times m}\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)_A\big)\\
&= T\big(\Phi^{\times m}\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)\big)_A && \text{by factorisation } T\\
&\subseteq \Phi\big(T\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)\big)_A && \text{by compatibility}\\
&\subseteq \Phi\big(\Phi_c\big(T\big(\vec{R}\big)\big)\big)_A && \text{by induction Hyp.}\\
&= \Phi_c\big(T\big(\vec{R}\big)\big)_A && \text{by definition of } \Phi_c.
\end{aligned}$$

Thus $T\big(\Phi_c^{\times m}\big(\vec{R}\big)\big)_A \subseteq \Phi_c\big(T\big(\vec{R}\big)\big)_A$ holds for all $\vec{R}$ and A, hence T is Φ_c-compatible. □

We now use Lemma 5.2.25 to obtain soundness of the rules **(Refl)**, **(Sym)** and **(Trans)**.

Lemma 5.2.27. *The chain $\approx_c$ is an equivalence relation and is closed under conversion at each stage. More precisely, for all $A \in \mathrm{Ty}$ and $r, s, t, u \in \Lambda^{=}(A)$, there are the following morphisms in* $\mathbf{T}$.

1. $\mathbf{1} \to t \approx_c t$ *(Reflexivity);*
2. $s \approx_c t \to t \approx_c s$ *(Symmetry);*
3. $r \approx_c s \wedge s \approx_c t \to r \approx_c t$ *(Transitivity); and*
4. $s \approx_c t \to r \approx_c u$, *if* $r \equiv s$ *and* $t \equiv u$ *(Conversion closure).*

Proof. We prove 1–4 by formulating these properties so that we can apply Lemma 5.2.25. This is achieved by using the maps $\mathrm{Refl}\colon \mathbf{1} \to \mathrm{Rel}$, $\mathrm{Sym}\colon \mathrm{Rel} \to \mathrm{Rel}$, $\_;\_\colon \mathrm{Rel}\times\mathrm{Rel} \to \mathrm{Rel}$ and $C^{\equiv}\colon \mathrm{Rel} \to \mathrm{Rel}$, where Refl maps constantly to the diagonal relation Eq, Sym maps a relation to its inverse, $\_\,;\,\_$ is the composition of relations, and $C^{\equiv}$ closes a relation under conversion. The statement of 1–4 is then equivalent to saying for all $n \in \mathbb{N}$ that $\mathrm{Refl}\big(\overleftarrow{\Phi_c}(n)\big) \sqsubseteq \overleftarrow{\Phi_c}(n)$, $\mathrm{Sym}\big(\overleftarrow{\Phi_c}(n)\big) \sqsubseteq \overleftarrow{\Phi_c}(n)$, $\overleftarrow{\Phi_c}(n)\,;\,\overleftarrow{\Phi_c}(n) \sqsubseteq \overleftarrow{\Phi_c}(n)$ and $C^{\equiv}\big(\overleftarrow{\Phi_c}(n)\big) \sqsubseteq \overleftarrow{\Phi_c}(n)$. It remains to show that the maps Refl, Sym, $\_\,;\,\_$ and $C^{\equiv}$ are Φ_c-compatible, since we then can appeal to Lemma 5.2.25 to obtain the above inclusions. Towards this end, one first shows that these maps are Φ-compatible. Compatibility of reflexivity, symmetry and composition has been proved in [Bon+14, Sec 4.2], which uses that Φ is the canonical relation lifting of F, see Definition 5.1.6 and that F preserves weak pullbacks. Weak pullback preservation follows directly from the definition of F in terms of products, coproducts, and the power set functor, see [GS00]. For compatibility of the $\equiv$-closure we use that for any relation

$C^{\equiv}(R) \sqsubseteq C^{\equiv_{\text{obs}}}(R)$ and that $C^{\equiv_{\text{obs}}}$ is compatible, see Proposition 5.1.10. Finally, Φ_c-compatibility of the maps follows from Lemma 5.2.26, as the maps Refl, Sym, _ ; _ and $C^{\equiv}$ are defined on each type separately. □

The next step is to establish the soundness of the extensionality and congruence rules for $\sim$. Since the rules **(ν-Ext)** and **(μ-Cong)** involve the later modality, we first show how this modality can be interpreted over $[\omega^{\text{op}}, \text{Rel}]$ and how it can be used to "guard", cf. [BM13], the unfolding of $\overleftarrow{\Phi_c}$.

Lemma 5.2.28. *Let* $\overline{\Phi_c} \colon [\omega^{\text{op}}, \text{Rel}] \to [\omega^{\text{op}}, \text{Rel}]$ *be given by* $\overline{\Phi_c}(S)(n) \coloneqq \Phi_c(S(n))$. *We have for the functor* $\triangleright \colon [\omega^{\text{op}}, \text{Rel}] \to [\omega^{\text{op}}, \text{Rel}]$ *defined in Lemma 2.3.4 that*

$$\overleftarrow{\Phi_c} = \triangleright(\overline{\Phi_c}(\overleftarrow{\Phi_c})). \tag{5.26}$$

Proof. To show that $\overleftarrow{\Phi_c} = \triangleright(\overline{\Phi_c}(\overleftarrow{\Phi_c}))$ for the $\overleftarrow{\Phi_c}$ from Definition 5.2.19, let $n \in \mathbb{N}$. Then by Lemma 2.3.4

$$\begin{aligned}
\triangleright\Big(\overline{\Phi_c}(\overleftarrow{\Phi_c})\Big)(n) &= \begin{cases} \mathbf{I}, & n = 0 \\ \overline{\Phi_c}(\overleftarrow{\Phi_c})(k), & n = k+1 \end{cases} \\
&= \begin{cases} \mathbf{I}, & n = 0 \\ \Phi_c(\overleftarrow{\Phi_c}(k)), & n = k+1 \end{cases} = \begin{cases} \mathbf{I}, & n = 0 \\ \overleftarrow{\Phi_c}(k+1), & n = k+1 \end{cases} = \overleftarrow{\Phi_c}(n).
\end{aligned}$$

Thus, $\triangleright(\overline{\Phi_c}(\overleftarrow{\Phi_c}))(n) = \overleftarrow{\Phi_c}(n)$ for all $n \in \mathbb{N}$ and so $\overleftarrow{\Phi_c} = \triangleright(\overline{\Phi_c}(\overleftarrow{\Phi_c}))$. □

Lemma 5.2.29. *The rules* **(→-Ext)**, **(×-Ext)**, **(ν-Ext)**, **(+-Cong)** *and* **(μ-Cong)** *are sound for the interpretation in Definition 5.2.22. More precisely, the following are isomorphisms in* **T**.

$$\begin{aligned}
(s \approx_c t) &\cong \bigwedge_{u \in \mathbf{ON}_A} s\,u \approx_c t\,u && \forall s, t : A \to B \\
(s \approx_c t) &\cong (\pi_1\, s \approx_c \pi_1\, t) \wedge (\pi_2\, s \approx_c \pi_2\, t) && \forall s, t : A \times B \\
(s \approx_c t) &\cong \triangleright(t.\text{out} \approx_c s.\text{out}) && \forall s, t : \nu X.\, A \\
(\kappa_i\, s \approx_c \kappa_i\, t) &\cong (t \approx_c s) && \forall i \in \{1, 2\} \text{ and } s, t : A_i \\
(\alpha\, s \approx_c \alpha\, t) &\cong \triangleright(t \approx_c s) && \forall s, t : A[\mu X.\, A/X]
\end{aligned}$$

Proof. We show that the premises of the rules imply the corresponding conclusion in each case.

- For function types $A \to B$, we first note that for $s, t \in \Lambda^{=}(A \to B)$ the existence of an isomorphism $(s \approx_c t) \cong \bigwedge_{u \in \mathbf{ON}_A} s\,u \approx_c t\,u$ simply means that for all $n \in \mathbb{N}$, $(s,t) \in \overleftarrow{\Phi_c}(n)_{A \to B} \iff \forall u \in \mathbf{ON}_A.\, (s\,u, t\,u) \in \overleftarrow{\Phi_c}(n)_B$. For 0 this is trivial since $\overleftarrow{\Phi_c}(0) = \mathbf{I}$. On the other hand, for $n \in \mathbb{N}$ we have

  $$\overleftarrow{\Phi_c}(n+1)_{A \to B} = \Phi_c(\overleftarrow{\Phi_c}(n))_{A \to B} = \Phi(\Phi_c(\overleftarrow{\Phi_c}(n)))_{A \to B} = \Phi(\overleftarrow{\Phi_c}(n+1))_{A \to B}.$$

  By Lemma 5.1.8 $(s,t) \in \Phi(\overleftarrow{\Phi_c}(n+1))_{A \to B}$ is equivalent to $\forall u \in \mathbf{ON}_A.\, (s\,u, t\,u) \in \overleftarrow{\Phi_c}(n+1)$ because $\overleftarrow{\Phi_c}$ is $\equiv$-closed, see Lemma 5.2.27. Thus the desired isomorphism exists.

- The proof of soundness for **(×-Ext)** and **(+-Cong)** is analogous to that for **(→-Ext)**.
- To prove soundness of **(ν-Ext)** and **(μ-Cong)**, we appeal to the identity $\overleftarrow{\Phi_c} = \triangleright\Big(\overline{\Phi_c}\big(\overleftarrow{\Phi_c}\big)\Big)$ from Lemma 5.2.28 as follows. Suppose we are given $s, t \in \Lambda^{=}(\nu X.\, A)$ and $n \in \mathbb{N}$. Then we have

$$\begin{aligned}
(s,t) \in \triangleright\Big(\overline{\Phi_c}\big(\overleftarrow{\Phi_c}\big)\Big)(n)_{\nu X.\, A} &\iff \begin{cases} (s,t) \in \mathbf{I}_{\nu X.\, A}, & n = 0 \\ (s,t) \in \Phi\big(\overleftarrow{\Phi_c}(k)\big)_{\nu X.\, A}, & n = k+1 \end{cases} \\
&\iff \begin{cases} (s.\mathrm{out}, t.\mathrm{out}) \in \mathbf{I}_{A[\nu X.\, A/X]}, & n = 0 \\ (s.\mathrm{out}, t.\mathrm{out}) \in \overleftarrow{\Phi_c}(k)_{A[\nu X.\, A/X]}, & n = k+1 \end{cases} \qquad (*) \\
&\iff (s.\mathrm{out}, t.\mathrm{out}) \in (\triangleright\, \overleftarrow{\Phi_c})(n)_{A[\nu X.\, A/X]},
\end{aligned}$$

  where we can apply Lemma 5.1.8 in the second case of $(*)$, since $\overleftarrow{\Phi_c}$ is closed under convertibility. Combining this relation with (5.26) from Lemma 5.2.28, we obtain

$$\begin{aligned}
(s \approx_c)(n) &\iff (s,t) \in \overleftarrow{\Phi_c}(n) \\
&\iff (s,t) \in \Big(\triangleright\, \overline{\Phi_c}\big(\overleftarrow{\Phi_c}\big)\Big)(n) && \text{by (5.26)} \\
&\iff (s.\mathrm{out}, t.\mathrm{out}) \in (\triangleright\, \overleftarrow{\Phi_c})(n) && \text{see above} \\
&\iff \triangleright(s.\mathrm{out} \approx_c t.\mathrm{out})(n)
\end{aligned}$$

  which shows that **(ν-Ext)** is sound.
- Similarly, one shows for $s, t \in \Lambda^{=}(A[\mu X.\, A/X])$ and $n \in \mathbb{N}$ that

$$(\alpha\, s, \alpha\, t) \in \Big(\triangleright\, \overline{\Phi_c}\big(\overleftarrow{\Phi_c}\big)\Big)(n) \iff (s,t) \in (\triangleright\, \overleftarrow{\Phi_c})(n),$$

  from which we obtain soundness of **(μ-Cong)** by the same reasoning as for **(ν-Ext)**. □

Now that we have proven that the axioms for $\sim$ are correct, we are left with the task to show soundness of the axioms for the later modality (Figure 5.5 on page 130) and of the refinement rules (Figure 5.6, page 131). The rules **(K)** and **(Nec)** for the later modality are instances of the functoriality of $\triangleright$ and the next operator in Lemma 2.3.4 for the category **T**. The soundness of the fixed point rule **(Löb)** has been proven in Lemma 2.3.5.

To prove the refinement rules correct, we first show that we can replace convertible terms in formulas without affecting their validity. Conveniently, this also proves soundness of the replacement rule for definitional equality.

Lemma 5.2.30. *Let φ be a formula with $\Gamma \Vdash \varphi$ and $\sigma, \tau \in \mathrm{Subst}(\mathbf{ON}; \Gamma)$. If $\sigma \equiv \tau$, that is $\sigma(x) \equiv \tau(x)$ for all $x \in \mathrm{dom}(\Gamma)$, then $[\![\varphi]\!](\sigma) \cong [\![\varphi]\!](\tau)$.*

Proof. We proceed by induction on the formula φ. By applying Lemma 5.2.27 to $s[\sigma] \equiv s[\tau]$ and $t[\sigma] \equiv t[\tau]$, we have for all terms s and t that $[\![s \sim t]\!](\sigma) \cong [\![s \sim t]\!](\tau)$, which proves the claim in the base case $\varphi = s \sim t$. All the other cases follow immediately by induction from functoriality of the operations that we used in the definition of $[\![-]\!]$ in Definition 5.2.22. □

Lemma 5.2.30 allows us now to replace terms of inductive type by their weak head normal forms, which allows us to obtain soundness for the refinement rules.

Lemma 5.2.31. *The rules* **(1-Ref)**, **(+-Ref)** *and* **(μ-Ref)** *are sound for the interpretation in Definition 5.2.22, that is, for all $n \in \mathbb{N}$ we have*

i) if $\Gamma \mid \Delta[\langle\rangle/x]; n \vDash \varphi[\langle\rangle/x]$, *then* $\Gamma, x : \mathbf{1} \mid \Delta; n \vDash \varphi$;

ii) if $\Gamma, y_i : A_i \mid \Delta[\kappa_i\, y_i/x]; n \vDash \varphi[\kappa_i\, y_i/x]$ *for all* $i \in \{1,2\}$, *then* $\Gamma, x : A_1 + A_2 \mid \Delta; n \vDash \varphi$;

iii) if $\Gamma, y : A[\mu X.\, A/X] \mid \Delta[\alpha\, y/x]; n \vDash \varphi[\alpha\, y/x]$, *then* $\Gamma, x : \mu X.\, A \mid \Delta; n \vDash \varphi$.

Proof. Suppose that φ is a formula with $\Gamma, x : A_1 + A_2 \Vdash \varphi$, and let us assume for all $n \in \mathbb{N}$ that $\Gamma, y_i : A_i \mid \Delta[\kappa_i\, y_i/x]; n \vDash \varphi[\kappa_i\, y_i/x]$ holds for all $i \in \{1,2\}$. We need to show that $\Gamma, x : A_1 + A_2 \mid \Delta; n \vDash \varphi$ holds for all $n \in \mathbb{N}$. Thus, by Proposition 5.2.23, we need to show that if for all $i \in \{1,2\}$ and all $\tau \in \mathrm{Subst}(\mathbf{ON}; \Gamma, x : A_i)$, there are morphisms $\bigwedge_{\psi\in\Delta} [\![\psi[\kappa_i\, y_i/x]]\!](\tau) \to [\![\varphi[\kappa_i\, y_i/x]]\!](\tau)$, then for all $\sigma \in \mathrm{Subst}(\mathbf{ON}; \Gamma, x : A_1 + A_2)$, there is $\bigwedge_{\psi\in\Delta} [\![\psi]\!](\sigma) \to [\![\varphi]\!](\sigma)$.

So let $\sigma \in \mathrm{Subst}(\mathbf{ON}; \Gamma, x : A_1 + A_2)$. Since $\sigma(x) \in \mathbf{ON}_{A_1+A_2}$, there is an $i \in \{1,2\}$ and a t with $\sigma(x) \equiv \kappa_i\, t$. We then define $\tau := \sigma[x \mapsto \kappa_i\, t]$ and $\gamma := \sigma[y_i \mapsto t]$, so that $\Delta[\kappa_i\, y_i/x][\gamma] = \Delta[\tau]$ and $\varphi[\kappa_i\, y_i/x][\gamma] = \varphi[\tau]$. From the assumptions and Lemma 5.2.30 we then obtain

$$\bigwedge_{\psi\in\Delta} [\![\psi]\!](\sigma) \cong \bigwedge_{\psi\in\Delta} [\![\psi]\!](\tau) \cong \bigwedge_{\psi\in\Delta} [\![\psi[\kappa_i\, y_i/x]]\!](\gamma) \to [\![\varphi[\kappa_i\, y_i/x]]\!](\gamma) \cong [\![\varphi]\!](\tau) \cong [\![\varphi]\!](\sigma),$$

as required. The proofs for **(1-Ref)** and **(μ-Ref)** are completely analogous. □

We can now put everything together to show soundness of $\mathbf{FOL}_{\blacktriangleright}$'s proof system.

Proof of Theorem 5.2.24. To prove that $\Gamma \mid \Delta \vdash \varphi$ implies $\Gamma \mid \Delta; n \vDash \varphi$ for all $n \in \mathbb{N}$, one proceeds by induction on the proof tree that witnesses $\Gamma \mid \Delta \vdash \varphi$. The cases for the standard rules of intuitionistic logic in Figure 5.2 are thereby given by the properties of the Boolean algebra $\mathbb{B}$, see Proposition 2.3.2 and Lemma 2.3.3. It only should be noted that these rules are sound because of the way we handle variables in $\mathbf{FOL}_{\blacktriangleright}$ and because there is no strengthening rule, see the discussions [LS88, p. 131] and [Jac99, p. 231]. Soundness of the rules for definitional equality (Figure 5.3) are given by Lemma 5.2.30 and by convertibility being an equivalence relation. Next, soundness for the $\sim$-rules in Figure 5.4 is given by Lemma 5.2.27 and Lemma 5.2.29. The rules for the later modality in Figure 5.5 are valid by Lemma 2.3.4 and Lemma 2.3.5. Finally, the rules in Figure 5.6 are proved correct in Lemma 5.2.31. The special case for $\varphi = s \sim t$ gives us that $\vdash s \sim t$ implies $(s,t) \in \overleftarrow{\Phi_c}(n)$ for all $n \in \mathbb{N}$. By Lemma 5.2.20 we then obtain $s \equiv_{\mathrm{obs}} t$, as required. □

Now that we have established a model for the logic $\mathbf{FOL}_{\blacktriangleright}$,[38] one might ask whether we can prove any observational equivalence. In other words, we may ask whether $\mathbf{FOL}_{\blacktriangleright}$ is complete for observational equivalence. We answer this question negatively in the following example.

Example 5.2.32 (Incompleteness)**.** We define for each type $A \in \mathrm{Ty}$ a term $\delta\colon \mathrm{Nat} \to A^\omega \to A^\omega$ that iteratively takes the tail of an input stream:

$$\begin{aligned} \delta\, 0 \qquad & s = s \\ \delta\, (\mathrm{suc}\, n)\, s &= \delta\, n\, (s.\mathrm{tl}) \end{aligned}$$

Note that the first argument of inductive type (Nat) introduces an observation on a coinductive type in the second case of δ's definition. This leads to the a problem if we want to prove properties of δ in $\mathbf{FOL}_{\blacktriangleright}$. For example, we clearly have $(\delta\ n\ s).\mathrm{tl} \equiv_{\mathrm{obs}} \delta\ n\ (s.\mathrm{tl})$ for all $n \in \mathbf{ON}_{\mathrm{Nat}}$ and $s \in \mathbf{ON}_{A^\omega}$, because $n \equiv \underline{m}$ for some $m \in \mathbb{N}$ and thus

$$(\delta\ n\ s).\mathrm{tl} \equiv (\delta\ \underline{m}\ s).\mathrm{tl} \equiv (s.\mathrm{tl}^m).\mathrm{tl} \equiv_{\mathrm{obs}} (s.\mathrm{tl}).\mathrm{tl}^m \equiv \delta\ n\ (s.\mathrm{tl}).$$

If we try to prove this in $\mathbf{FOL}_{\blacktriangleright}$ we encounter the following problem though. Let φ be the formula $\forall n.\, \forall s.\, (\delta\ n\ s).\mathrm{tl} \sim \delta\ n\ (s.\mathrm{tl})$, which we aim to prove. We can start a proof for φ:

$$\dfrac{\dfrac{\dfrac{}{(\delta\ 0\ s).\mathrm{tl} \equiv s.\mathrm{tl} \equiv \delta\ 0\ (s.\mathrm{tl})}}{\blacktriangleright\varphi \vdash (\delta\ 0\ s).\mathrm{tl} \sim \delta\ 0\ (s.\mathrm{tl})} \qquad \dfrac{\dfrac{}{n : \mathrm{Nat}, s : A^\omega \mid \blacktriangleright\varphi \vdash (\delta\ n\ (s.\mathrm{tl})).\mathrm{tl} \sim \delta\ n\ (s.\mathrm{tl}.\mathrm{tl})}\ ??}{n : \mathrm{Nat}, s : A^\omega \mid \blacktriangleright\varphi \vdash (\delta\ (\mathrm{suc}\ n)\ s).\mathrm{tl} \sim \delta\ (\mathrm{suc}\ n)\ (s.\mathrm{tl})}}{\dfrac{n : \mathrm{Nat}, s : A^\omega \mid \blacktriangleright\varphi \vdash (\delta\ n\ s).\mathrm{tl} \sim \delta\ n\ (s.\mathrm{tl})}{\dfrac{\blacktriangleright\varphi \vdash \varphi}{\vdash \varphi}}}$$

The problem arises if we try to fill the question marks. By instantiating the quantifiers with n and s.tl, we can obtain by (5.13) and **(∀-E)** from $\blacktriangleright\varphi$ only

$$\blacktriangleright\big((\delta\ n\ (s.\mathrm{tl})).\mathrm{tl} \sim \delta\ n\ (s.\mathrm{tl}.\mathrm{tl})\big),$$

from which we have to remove the later modality to fill the question marks. But since this is not sound, φ cannot be proven in this way. Indeed, the problem is that we have to prove φ by using induction on n, but the resulting later modality does not interact well with the tail observation on the coinductive type of streams. In other words, we are not able to carry out proofs by induction on elements coinductive types. ◀

Let us finish this section with remarking on our use of the terms "induction" and "coinduction" in reference to proofs in $\mathbf{FOL}_{\blacktriangleright}$. One can show that the usual induction and coinduction schemes are derivable in $\mathbf{FOL}_{\blacktriangleright}$, only with the difference that the later modality appears in the induction and coinduction hypotheses. More precisely, one first defines for a type[39] A with $X \Vdash A : \mathbf{Ty}$ a predicate lifting $\overline{A}$ and a relation lifting $\widetilde{A}$ that are subject to the following two rules. In these rules, $A[-]$ is the action of the type A as in Definition 3.3.1.

$$\frac{\Gamma, x : B \Vdash \varphi}{\Gamma, x : A[B] \Vdash \overline{A}(\varphi)} \qquad \frac{\Gamma, x : B \times C \Vdash \varphi}{\Gamma, x : A[B] \times A[C] \Vdash \widetilde{A}(\varphi)}$$

The definition of these liftings follows thereby the definition of liftings by Hermida and Jacobs [HJ97] for polynomial functor.

Given a formula φ on the least fixed point type $\mu X.\, A$, that is, with $\Gamma, x : \mu X.\, A \Vdash \varphi$, one can show that the following induction principle for φ is derivable in $\mathbf{FOL}_{\blacktriangleright}$.

$$\frac{\Gamma \mid \Delta \vdash \forall y : A[\mu X.\, A].\, \overline{A}(\blacktriangleright\varphi)[y] \to \varphi[\alpha\ y]}{\Gamma \mid \Delta \vdash \forall (x : \mu X.\, A).\, \varphi}$$

To establish a coinduction principle, let $B \in \mathrm{Ty}$ and $C \in \mathrm{Ty}$, and let $c \colon B \to A[B]$ and $d \colon C \to A[C]$ be coalgebras. Moreover, assume that we are given a relation between B and C in form of a formula

φ with $\Gamma, x : B \times C \Vdash \varphi$. One can then show that the following coinduction principle holds in $\mathbf{FOL}_{\blacktriangleright}$, where R_ν is the coiterator from Definition 3.3.1.

$$\frac{\Gamma, x : B, y : C \mid \Delta \vdash \varphi[\langle x, y\rangle] \to \blacktriangleright \widetilde{A}(\varphi)[\langle c\ x, d\ y\rangle]}{\Gamma, x : B, y : C \mid \Delta \vdash \varphi[\langle x, y\rangle] \to R_\nu\ c\ x \sim R_\nu\ d\ y}$$

We will not go into the details of how to prove that the induction and coinduction principles are derivable in $\mathbf{FOL}_{\blacktriangleright}$, since they are of limited use and merely show that the proof system subsumes the standard principles.

5.3. (Un)Decidability of Observational Equivalence

In this section, we present two results concerning decidability of observational equivalence. The first just makes precise the intuitive assertion that, in general, observational equivalence is undecidable. The second result, however, establishes a fragment of $\lambda\mu\nu{=}$ on which observational equivalence actually becomes decidable. This fragment is admittedly rather small, but still an interesting start. Since this fragment contains non-terminating terms, the algorithm that checks observational equivalence has to check at the same time for observational normalisation. We thereby show that also observational normalisation is decidable in this fragment of $\lambda\mu\nu{=}$.

5.3.1. Observational Equivalence is Undecidable

Proposition 5.3.1. *Observational inequivalence is semi-decidable on* **ON***-terms.*

Proof. Let t_1, t_2 be two terms in $\mathbf{ON}_A$. To decide whether $t_1 \not\equiv_{\text{obs}} t_2$ we can enumerate all tests on A and check for each of them whether t_1 and t_2 do not satisfy it simultaneously. This gives a procedure that terminates, if $t_1 \not\equiv_{\text{obs}} t_2$. We now show that it is impossible to give a general algorithm that checks observational inequivalence via an encoding of Post's correspondence problem (PCP). This shows that observational equivalence is undecidable.

Let A be a finite alphabet with at least two letters, and $w = (w_1, \ldots, w_N)$ and $v = (v_1, \ldots, v_N)$ sequences of words over A. A solution to the PCP for $\langle w, v\rangle$ is a finite sequence $(i_k)_{1 \le k \le K}$ such that for all $k \in \{1, \ldots, K\}$, $1 \le i_k \le N$, and

$$w_{i_1} \cdots w_{i_K} = v_{i_1} \cdots v_{i_K}. \tag{5.27}$$

It is known to be undecidable whether a solution exists for any given $\langle w, v\rangle$. The idea of the encoding of the PCP, which we are about to give, is to define a decidable predicate on finite sequences that contains all lists for which (5.27) holds. This is carried out in the following way.

We begin by defining the relevant data structures as types, and basic functions on them. Let A have n letters, so that we can encode A as the sum $A = \underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{n}$. Words over A are given by lists $A^* = \mu X.\, \mathbf{1} + A \times X$, thus a word w can be written as a sequence of constructors, and concatenation of lists $\cdot$ can be defined inductively in the usual way. We can also define predicates $eq_A : A \times A \to \text{Bool}$ and $eq_L : A^* \times A^* \to \text{Bool}$ that are computable on observationally normalising arguments, such that

$$\begin{aligned} eq_A\ x \equiv \top \quad &\text{iff} \quad x.\text{pr}_1 \equiv x.\text{pr}_2 \quad \text{and} \\ eq_L\ y \equiv \top \quad &\text{iff} \quad y.\text{pr}_1 \equiv y.\text{pr}_2. \end{aligned}$$

Finally, we encode the set of numbers $\{0, \dots, N-1\}$ with the type $\widehat{N} = \underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{N}$ and finite, non-empty sequences of them by $\widehat{N}^+ = \mu X.\, \widehat{N} + \widehat{N} \times X$.

To reduce an instance $\langle w, v \rangle$ of PCP to observational equivalence, we define a map h which given a sequence $u = (i_k)_{1 \le k \le K}$ computes the pair $(w_{i_1} \cdots w_{i_K}, v_{i_1} \cdots v_{i_K})$, a predicate P which tests whether a sequence is a solution, and the empty predicate $P_\bot$ by

$$
\begin{aligned}
&h : \widehat{N}^+ \to A^* \times A^* \\
&h\, i = \langle w_i, v_i \rangle \\
&h\, (i : u) = \left\langle w_i \cdot ((h\, u).\mathrm{pr}_1), v_i \cdot ((h\, u).\mathrm{pr}_2) \right\rangle
\end{aligned}
$$

$$
\begin{aligned}
&P : \widehat{N}^+ \to \mathrm{Bool} \\
&P = eq_L \circ h
\end{aligned}
\qquad\qquad
\begin{aligned}
&P_\bot : \widehat{N}^+ \to \mathrm{Bool} \\
&P_\bot\, u = \bot
\end{aligned}
$$

Hence, for $u : \widehat{N}^+$, $P\, u \equiv \top$ if and only if u solves the PCP for $\langle w, v \rangle$, and P is clearly computable for **ON** terms. Moreover, since observational inequivalence is witnessed by tests, $P \not\equiv_{\mathrm{obs}} P_\bot$ means that there is a $\varphi \in \mathrm{Tests}_{\widehat{N}^+ \to \mathrm{Bool}}$ such that $P \vDash \varphi$ and $P_\bot \nvDash \varphi$. We may assume that such a φ is of the form $\varphi = [u]\,[\top, \bot]$ for some $u : \widehat{N}^+$ so that $P\, u \equiv \top$, and u is a solution to the PCP. Thus, if we can find φ, we can solve the PCP. In other words, $P \not\equiv_{\mathrm{obs}} P_\bot$ if and only if the PCP for $\langle w, v \rangle$ has a solution.

In summary, if for all terms t_1, t_2 it is decidable whether there is a test that distinguishes t_1 and t_2, then the PCP is decidable, hence observational equivalence cannot be decidable. □

5.3.2. Decidability on a Language Fragment

Even though observational equivalence is undecidable on the full language, there is a fragment of the language on which we can decide it. Analysing the encoding of Post's correspondence problem, we find that the encoding crucially requires functions. Indeed, once we forbid terms of function type, observational equivalence becomes decidable.

More precisely, we fix a declaration block Σ, and consider terms that are well-typed within Σ in the following restricted syntax:

$$t ::= f \mid \kappa_i\, t \mid \alpha\, t \mid t.\mathrm{pr}_1 \mid t.\mathrm{pr}_2 \mid t.\mathrm{out}$$

$$D ::= \left\{ \cdot.\mathrm{pr}_1 \mapsto t_1 \;;\; \cdot.\mathrm{pr}_2 \mapsto t_2 \right\} \mid \{ \cdot.\mathrm{out} \mapsto t \}.$$

That is, in this restricted calculus we cannot form terms of function type, copatterns have only one layer, and λ-abstraction is excluded. It should be noted that one-layer copatterns and excluding λ-abstraction do not pose any limitations, as we can first unroll nested copatterns into nested λ-abstractions and then introduce for each λ-abstraction a new symbol into the signature. These transformations preserve observational equivalence.

In this restricted calculus, Algorithm 1 decides whether two terms are observationally equivalent, returning a witnessing test if they are not or a bisimulation up-to convertibility if they are. The

Algorithm 1: Decide whether two terms are observationally equivalent

```
CheckBisim(t1, t2 ∈ ON_A, R ∈ RelΛ^=) : Tests + RelΛ^=
    Invariant: If t1 R_A t2, then R is a bisimulation up-to convertibility.
    if t1 R_A t2 then return R
    Add (t1, t2) to R
    Bring t1 and t2 into WHNF
    case t_i = f_i with (f_i : A = D_i) ∈ Σ do
        case D1 = {·.pr1 ↦ r1 ; ·.pr2 ↦ r2 ↦ ·.pr2 ↦ r2} and
             D2 = {·.pr1 ↦ s1 ; ·.pr2 ↦ s2 ↦ ·.pr2 ↦ s2} do
            R' ← UpdateTest(λφ. [π1] φ, CheckBisim(r1, s1, R))
            UpdateTest(λφ. [π2] φ, CheckBisim(r2, s2, R'))
        case D1 = {·.out ↦ r} and D2 = {·.out ↦ s} do
            UpdateTest(λφ. [ξ] φ, CheckBisim(r, s, R))
    case t1 = κ_i t1' and t2 = κ_j t2' do
        if i ≠ j then return [⊤, ⊥]
        UpdateTest(CoprodTest(i), CheckBisim(t1', t2', R))
    case t1 = α t1' and t2 = α t2' do
        UpdateTest(λφ. α^-1 φ, CheckBisim(t1', t2', R))
end
UpdateTest(f : Tests → Tests, U ∈ Tests + RelΛ^=) : Tests + RelΛ^=
    case U is a test φ do return f(φ)
    case U is a relation R do return R
end
CoprodTest(i, φ)
    if i = 1 then return [φ, ⊥] else return [⊥, φ]
end
```

notation we use in the algorithm is similar to that of monadic Haskell-code, where we treat $\mathrm{Tests} + (-)$ as a monad and we use the left-arrow notation.[1]

Informally, the algorithm works as follows. It compares recursively the given terms according to what it requires to fulfil the same tests. If that fails, it builds up a test witnessing this, while returning from the recursion. Otherwise, it puts the given pair of terms in the bisimulation candidate R and tries to close R recursively. Once it arrives at a pair that has already been compared, it returns the constructed relation, which is closed at that point.

Termination of the algorithm is ensured by the fact that a term t in the fixed declaration block Σ essentially generates a finite subsystem of the term transition system, as we explain now. Modulo reduction to WHNF, the only way of creating a term of inductive type is by a finite sequence of constructors, hence we can remove only finitely many such. For coinductive types, on the other hand, a WHNF must be a symbol in Σ, hence we must eventually reach a pair of symbols that are already in the relation, as there are only $|\Sigma|^2$ such pairs. Therefore, the algorithm terminates.

We will now make these arguments precise.

[1] Implementation: `https://github.com/hbasold/ObservationalEquiv/blob/master/DecideEquiv`

Theorem 5.3.2. *Let $t_1, t_2 \in \mathbf{ON}_A$ be in the restricted language. Then the following holds.*

(i) If $\texttt{CheckBisim}(t_1, t_2, \emptyset)$ *returns a test φ, then $(t_1 \vDash \varphi) \neq (t_2 \vDash \varphi)$.*

(ii) If $\texttt{CheckBisim}(t_1, t_2, \emptyset)$ *returns a relation R, then R is a bisimulation up-to convertibility and we have $(t_1, t_2) \in R_A$.*

(iii) $\texttt{CheckBisim}(t_1, t_2, \emptyset)$ *terminates.*

Proof. (i) This is very easy to see, as we only stop with a test if the constructors for elements of a sum type do not match and then trace back the observations we made to get to the sum constructors.

(ii) We prove the invariant given at the beginning of `CheckBisim`: If t_1 and t_2 are related by R, then R is already a bisimulation up-to convertibility. Since $\emptyset$ fulfils this and we return R without further changes, the statement of the theorem follows.

So assume that t_1 and t_2 are not yet related by R, in which case the pair is added and we continue on the WHNF of these terms. In all cases, we recurse on elements of $\delta(t_1)$ and $\delta(t_2)$, where δ is the transition system that we defined in Section 5.1.1. This means that, in the recursion step, if these elements are already in R, we indeed have found a bisimulation. For example, if $A = B_1 \times B_2$ and $t_i = f_i$ with $(f_i : A = D_i) \in \Sigma$, then $\delta(t_i)(j) = \{t' : B_j \mid t' \equiv \pi_j\, t\}$ for $j = 1, 2$ and, in particular, $r_j \in \delta(f_1)(j)$ and $s_j \in \delta(f_2)(j)$. Since, as a result of $\texttt{CheckBisim}(r_1, s_1, R)$, R' is a bisimulation up-to convertibility and contains (r_1, s_1), the result of $\texttt{CheckBisim}(r_2, s_2, R')$ contains (r_1, s_1), (r_2, s_2) and is a bisimulation up-to convertibility as well. Therefore, the invariant is preserved.

(iii) We use the following two termination measures: n, the maximum of the sizes of the terms t_1 and t_2, and $m = |\Sigma|^2 - \#\text{pairs of symbols in } R$. On the recursive calls of `CheckBisim` for inductive types, n strictly decreases and for coinductive types, m strictly decreases (though n might increase in this case). Thus m becomes eventually 0, meaning that all symbols of Σ have been related with each other. From here on, n must decrease until it becomes 1, at which point t_1 and t_2 must be symbols from Σ and are thus related. Hence, `CheckBisim` stops and returns R. □

After having proved that we can decide observational equivalence on observationally normalising terms, one might ask whether observational normalisation is a decidable property. The answer to this question is indeed yes and we describe the idea for a decision procedure in the following.

We have seen that **ON** is the largest predicate that is contained in **SN** and is closed under δ-steps. This can be leveraged, just as we did for observational equivalence, by constructing recursively a predicate that contains strongly normalising terms, giving, again as before, a terminating procedure, if we can decide strong normalisation. In the restricted calculus, we only need to decide whether there is a WHNF though, since

1. there is always a unique reduction sequence and
2. we check strong normalisation by continuing to check recursively for observational normalisation under constructors.

So we are left with the task to decide whether a term has a WHNF. This can be done by trying to reduce the term to a WHNF and storing every term in the reduction sequence in a predicate that witness whether a term has *no* WNHF. If we reach a term a second time in the reduction sequence, we know that there can be no WHNF. For the same reasons as before, this eventually terminates due to the fact that terms can make only finitely many transitions.

We illustrate this with an example of a term that has no WHNF.

Example 5.3.3. Let Σ be the following declaration block.

$$\begin{aligned} &\mathrm{grow} : \mathrm{Nat} \times \mathrm{Nat}^{\omega} \\ &\mathrm{grow.pr}_1 = 0 \\ &\mathrm{grow.pr}_2 = \mathrm{grow.pr}_2.\mathrm{out.pr}_2 \end{aligned}$$

The term $\mathrm{grow.pr}_2$ leads to the following reduction sequence

$$\mathrm{grow.pr}_2 \longrightarrow \mathrm{grow.pr}_2.\mathrm{out.pr}_2 \longrightarrow \mathrm{grow.pr}_2.\mathrm{out.pr}_2.\mathrm{out.pr}_2 \longrightarrow \cdots$$

which is obviously diverging. We can show this with the following predicate that the decision procedure constructs.

$$\begin{aligned} &P_{\mathrm{Nat}^{\omega}} = \{\mathrm{grow.pr}_2, \mathrm{grow.pr}_2.\mathrm{out.pr}_2\} \\ &P_{\mathrm{Nat}\times\mathrm{Nat}^{\omega}} = \{\mathrm{grow.pr}_2.\mathrm{out}\} \\ &P_C = \emptyset, \text{ all other types } C \end{aligned}$$

That P indeed proves that $\mathrm{grow.pr}_2$ has no WHNF is seen as follows. In order to reduce $\mathrm{grow.pr}_2$, we need to reduce $\mathrm{grow.pr}_2.\mathrm{out.pr}_2$, thus we need to make a reduction step on $\mathrm{grow.pr}_2.\mathrm{out}$, which in turn needs a reduction of $\mathrm{grow.pr}_2$. Since all of these terms are in P, we have found a loop in the reduction sequence, hence $\mathrm{grow.pr}_2$ has no WHNF. □

This procedure to decide the (non-)existence of a weak head normal form can also be found in the above mentioned implementation.

5.4. Discussion

The purpose of the present chapter was to find better ways of proving observational equivalence for $\lambda\mu\nu{=}$-programs than mere induction on tests. This led us to study three very different approaches: a coinduction proof principle, a syntactic proof system and automatic proofs for a fragment of $\lambda\mu\nu{=}$. We obtained the first method, the coinduction principle, by showing that there is a labelled transition system on terms, for which the largest bisimulation relation is given by observational equivalence. A major advantage of the coinduction principle over the test induction is that the former can be drastically improved by using so-called up-to techniques. In Section 5.1.3, we demonstrated on an example how the coinduction principle and up-to techniques can be used to prove complex properties.

The example in Section 5.1.3 showed also that, even if observational equivalence can be characterised coinductively, equivalences between functions with inductive domain require an induction

principle for inductive types. In that example, we implemented such an induction principle in form of an up-to technique, but this came at the cost of largely obscuring the proof. We found that the reason for this obscurity is that the program equivalence in Section 5.1.3 would naturally be shown by a mutual coinduction and two induction proofs. This mutual proof is, however, stratified in the approach that we took by implementing induction as an up-to technique for the coinduction principle. Such a stratification leads then to complicated proof goals, which take away from the actual result we set out to prove. Thus, the next step is to find a proof method that supports induction and coinduction equally well.

In Section 5.2, we introduced the logic $\mathbf{FOL}_{\blacktriangleright}$ for observational equivalence that has two important features: it treats induction and coinduction on a par, and it allows the discovery of proof goals in the proof construction. The need for the former was discussed above. Discovering proof goals while constructing a proof, on the other hand, lifts another burden of the usual coinduction principle, namely that one has to guess a bisimulation relation beforehand. This can be a tricky task, as we have seen in the extensive example. The need for guessing a bisimulation can be overcome by using so-called parameterised coinduction, but we refrain here from introducing another complication. Instead, we endowed the logic $\mathbf{FOL}_{\blacktriangleright}$ with a proof system, which allows us to refer back to previous steps in a proof and thereby removing the need for guessing proof goals up-front, similarly to what cyclic proof systems achieve.

An interesting aspect of the logic $\mathbf{FOL}_{\blacktriangleright}$, or actually its soundness proof, is that up-to techniques are still used "under the hood", in the sense that we use up-to techniques in order to show that the axioms of the logic are sound for observational equivalence, see Lemma 5.2.27. The difference between the axioms in the logic and the up-to techniques is that in the former case we do not need to explicitly assemble the up-to techniques that are used in a proof, see also the discussion after Example 5.1.11. In Section 5.1.3 we avoided the explicit assembly of compatible up-to techniques by using the companion instead. Appealing to the companion in a proof is of course fine as far as correctness is concerned, but one loses any sense of what the actual content of that proof is. More specifically, the companion is defined *impredicatively*, that is, the companion itself is already included in the join that defines the companion, see Definition 2.5.12. This fact is what makes it impossible to extract from proofs that involve the companion a bisimulation. Thus, such proof "feel magical" and lose any constructive content. The proof trees of $\mathbf{FOL}_{\blacktriangleright}$, on the other hand, implicitly record all uses of up-to techniques, since each proof step is annotated by the corresponding proof rule, which in turn might refer to an up-to technique through the soundness proof. Thus, we can recover from a proof tree all up-to techniques that appear in a proof, without having to explicitly assemble an up-to technique for that proof.

In Section 5.3, we concerned ourselves with the question whether observational equivalence can be automatically proven. We showed there that observational equivalence is indeed decidable on a fragment of the calculus $\lambda\mu\nu{=}$, by exhibiting an algorithm that outputs either a bisimulation or a counterexample. Moreover, we also proved that observational equivalence is in general undecidable, which shows the limit of the automatic approach to proving observational equivalence.

Related Work

Program Properties as Coinductive Predicates In Chapter 4, we discussed some work that relates to our notion of observational equivalence. One particular piece of work that we discussed there is also relevant here: Abramsky [Abr90] introduces, what he calls, *applicative bisimilarity*

to reason about the equivalence of λ-terms. What is interesting about his work, is that Abramsky defines applicative bisimilarity using the ω^{op}-chain construction, which appears here in Lemma 5.2.20, and then shows that applicative bisimilarity coincides with a notion of contextual equivalence. This approach is very similar to the result of Section 5.1.2, namely that observational equivalence is the largest observational bisimulation. However, it is clear that we are dealing with a different notion of (program) context, since Abramsky compares programs in any context, whereas we restrict to those contexts that arise from tests. In particular, we only allow observationally normalising terms as function arguments, whereas in the contextual equivalence in [Abr90] any term is allowed as function argument.

There is another crucial difference with Abramsky's work: In order to prove two λ-terms M, N to be applicative bisimilar, one tries to reduce both terms to a λ-abstraction, say $\lambda x.\, M'$ and $\lambda x.\, N'$, and then shows that for any term P that $M'[P/x]$ and $N'[P/x]$ are applicative bisimilar. If we adopt this definition to our setting, then this leads to another notion of equivalence because we then also take weak head normalisation for terms of function type (or more generally, of coinductive type) into account. That is to say, to show that terms s, t of type $A \to B$ are applicative bisimilar, we first try to reduce s to either $\lambda\{\cdot\, x \mapsto s'\}$ or to f with $(f : A \to B = \{\cdot\, x \mapsto s'\}) \in \Sigma$, and similarly for t and some t'. Having found these WHNFs, we require then that for all $u \in \mathbf{ON}_A$ that $s'[u/x]$ and $t'[u/x]$ are again applicative bisimilar.[40] It is clear that this leads to a different notion of equivalence because first of all we take normalisation of function to WHNFs into account. But more importantly, this way of observing the behaviour of terms is not part of the language $\lambda\mu\nu{=}$, thus it might be possible to show that up-to-context is a sound up-to technique for this notion of applicative bisimilarity, cf. Note 33.

Some evidence that supports this conjecture is provided by the work of Sangiorgi et al. [SKS11], where applicative bisimilarity is replaced by so-called *environmental bisimilarity*. An environmental bisimulation is thereby a relation between terms relative to a set of possible arguments for the related terms (the environment). The bisimulation condition is very similar to that for applicative bisimilarity, namely that one first tries to find a WHNF and then substitute arguments, only that the arguments in this case come from the corresponding environment. This definition allows Sangiorgi et al. to show that environmental bisimilarity is a congruence and that up-to-context is a sound up-to technique.

Another notion of bisimilarity between terms has been studied by Sumii and Pierce [SP07]. The intention there was to find an alternative to logical relations [Pit00] for languages with recursive types. Also here, the same differences arise from comparing terms on the basis of weak head normal forms, as we discussed above for applicative and environmental bisimilarity.

This also gives us the opportunity to discuss logical relations as another way of reasoning about program equivalence or, more specifically, contextual equivalence. These were introduced by Pitts [Pit00] because properties of contextual equivalence are hard or even impossible to prove by induction on contexts. For the polymorphic lambda calculus $\lambda 2$ (or System F) [Gir72; Rey74], logical relations have turned out to be a valuable proof tool. However, logical relations cannot be used that easily for systems with recursive types because logical relations are usually defined by induction on types, which is impossible for recursive types. There has been some work to overcome this problem, see [BH99; CH07] and the discussion in [SP07], but it seems that bisimulations are a better choice to reason in the presence of recursive types. Moreover, the definition of what a bisimulation on terms is can be varied, depending on which kind of observations on programs one is interested,

cf. [San11].

Apart from studies specifically about program equivalences, the work on bisimilarity, and coinductive predicates in general, is relevant here. Especially useful were to us coalgebras and general notions of bisimilarity [Has+13; HJ97; Rut00; Sta11] and the investigations on up-to techniques, both lattice theoretically [Pou07; Pou16; PS11] and category theoretically [Bon+14; PR17; Rot+17]. The usefulness of the coalgebraic approach to defining and proving program properties stems from its wide applicability, which arises because coalgebras and the accompanying notions are defined independently of any programming language or semantics. However, we have also seen that there are currently limits to the applicability of existing tools, see in particular Note 33 and Note 32.

A First-Order Logic for Observational Equivalence Let us now discuss the work that is related to our logic $\mathbf{FOL}_{\blacktriangleright}$ for observational equivalence. First of all, since the logic contains intuitionistic first order logic and has a Gentzen-style sequent calculus as proof system, the corresponding pieces of work like [Gen35] and [TvD88, Chap. 10] apply. Similarly relevant is also the literature about the provability logic of Löb as origin of the later modality as it was used by Nakano [Nak00] and later by Appel et al. [App+07] for (semantical) type construction and type checking. In particular, the correspondence between the Löb logic and transitive, well-founded Kripke-frames [SV82; Smo85; Sol76] is of interest here. This correspondence tells us that the Löb logic characterises (well-founded) induction [Bek99] and even fixed points [Smo85, Thm. 3.15] in a parameter-free way.

But let us be a bit more specific in the comparison. Upon discussing coinductive proof systems that are not based on explicit bisimulations, the most prominent example that comes to mind is the system Circ by Roşu and Lucanu [RL09]. When it comes to universally quantified equivalences, Circ is likely to be as expressive as $\mathbf{FOL}_{\blacktriangleright}$. The more complicated first-order formulas that are available in $\mathbf{FOL}_{\blacktriangleright}$ are not expressible in Circ. Also, induction was not available in the original system but has been added later to the implementation [Gor+11]. Interestingly, this induction principle also arises in Circ through the same so-called freezing mechanism as coinduction does. This suggests that freezing is strongly related to the way we use the later modality in $\mathbf{FOL}_{\blacktriangleright}$. In particular, it seems that the rule `[Derive]` in [RL09] is a combination of our **(Löb)**-rule with **(ν-Ext)**. However, there is a major difference between the two approaches: the soundness proof for Circ is monolithic in the sense that one can only show that a completed proof is correct. In contrast, in $\mathbf{FOL}_{\blacktriangleright}$ the correctness of a proof is ensured locally for each proof rule, thereby allowing for a modular soundness proof, see Section 5.2.2, and easier proof checking, see Definition 5.2.4.

More generally, the proof system of $\mathbf{FOL}_{\blacktriangleright}$ shares with cyclic proof systems that it has no explicit induction or coinduction scheme, see also in the introduction, therefore proof discovery becomes much more natural. On the other hand, $\mathbf{FOL}_{\blacktriangleright}$ has the advantage over cyclic proof systems that there is no global correctness condition on proofs, rather each derivation step ensures the correctness of a proof locally. This drastically simplifies the soundness proof and proof checking for $\mathbf{FOL}_{\blacktriangleright}$, as we have discussed above in the comparison to Circ.

Another approach to proving properties about programs and to logic in general is (dependent) type theory. This view on logic is going to be the subject of the next two chapters, thus we postpone a further discussion of this approach for now. Let us just mention Agda [Agd15] as implementation of an advanced type theory, in which transitivity of the substream relation has a short and crisp proof, in contrast to our development in Section 5.1.3. There are several reasons for that. First of all, we had to develop an induction principle from scratch, which is already part of Agda's type

theory. Second, Agda carries out all the computations automatically that we had to derive by hand in the proof of Proposition 5.1.19. Finally, the stratification of induction and coinduction makes the bisimulation-based proof more involved than necessary. Despite the deficits of the approach in Section 5.1.3, this should be seen of a study of the principles behind mixed inductive-coinductive reasoning, which should enable us to develop useful and well-understood proof techniques.

On the semantical side, Dreyer et al. [DAB11] have used approximations of logical relations that are indexed by their corresponding approximation depth. This solves the problem, which we mentioned above, that logical relations cannot be directly used to prove contextual equivalence for programs in languages with recursive types. Dreyer et al. then go ahead and devise a syntax for a logic that encompasses first-order logic over term and type variables, second order logic for (logical) relations, and fixed points of relation transformers. They ensure thereby that the fixed points of relation transformers are well-defined by requiring that the fixed point variable only occurs under a later modality. This guarantees that the relation transformer is *contractive*, which in turn allows the approximation of its fixed point through an ω^{op}-chain construction. The outcome of the efforts in [DAB11] is then a collection of inference rules that hold on the semantics of these formulas. These rules are close to that of $\mathbf{FOL}_{\blacktriangleright}$, only that they are tailored towards capturing contextual equivalence, cf. the discussion above.

In later work, Birkedal et al. [Bir+11] developed the approach of step-indexing further to give an axiomatic description of domain theoretic constructions by using the later modality. Based on this, Birkedal and Møgelberg [BM13] and Møgelberg [Møg14] developed programming languages, in which well-definedness of recursive definitions is ensured through type-annotations. In [Biz+16b], this language was then extended to a dependent type theory. Since a dependent type theory can serve as a vehicle for intuitionistic logic, one might expect that our logic $\mathbf{FOL}_{\blacktriangleright}$ could be construed in the theory of Bizjak et al. [Biz+16b]. However, there is a crucial point that prevents this. Take for example the type of streams over A and observational equivalence on that type. Recall that we have defined streams to be given by the type $\nu X.\, A \times X$, thus the obligations for proving $s \sim_{A^\omega} t$ were that $s.\mathrm{hd} \sim_A t.\mathrm{hd}$ and $\blacktriangleright(s.\mathrm{tl} \sim_{A^\omega} t.\mathrm{tl})$, see Example 5.2.12. In contrast in the type theory proposed ibid., streams over A are given by the type Str_A with $\mathrm{Str}_A = \mathrm{fix}\, X.\, A \times \blacktriangleright X$, where fix is a general fixed point operator and $\blacktriangleright$ is the type theoretical equivalent of the later modality that we used here. For an extension of the type theory [Bir+16], it was then shown that the identity type on streams (cf. Chapter 7), which is denoted by $\mathrm{Id}_{\mathrm{Str}_A}(s,t)$ for s and t of type Str_A, is isomorphic to $\mathrm{Id}_A(\mathrm{hd}\, s, \mathrm{hd}\, t) \times \mathrm{Id}_{\blacktriangleright \mathrm{Str}_A}(\mathrm{tl}\, s, \mathrm{tl}\, t)$. This resembles our Lemma 5.2.29 with the slight, but very crucial, difference that here the later modality is pushed into the underlying data type, whereas the later modality stays in our setting on the level of propositions about that data type. Thus, these extensions of dependent type theory by Birkedal et al. cannot be applied directly here.

Since we just came across the result in Lemma 5.2.29, we should briefly mention that similar correspondences, for example, between the equality on a product type and the conjunction of equality on the product components appear also in other places. Undoubtedly, this correspondence between equality proofs can be found in many places, but let us just mention two instances here. An example is of course the obligatory reference to homotopy type theory with the univalence axiom [Uni13]. But already in observational type theory [AMS07], this correspondence is taken to be the definition of equality on product types.

Contributions

It should be clear by now that there is plenty of work closely related to both the bisimulation proof method and $\mathbf{FOL}_{\blacktriangleright}$. So a clarification of the contributions made in this chapter is in order.

Since observational equivalence for the inductive-coinductive calculus $\lambda\mu\nu{=}$ has not been studied before, also the characterisation in terms of bisimulations is new. The real novelty is though the use of an up-to technique that enables the use of induction in a bisimulation proof. To my knowledge, such an up-to technique has not been studied before. Another contribution is the more organised approach that was taken here to characterise observational equivalence by choosing a transition system that encodes the relevant observations, from which observational equivalence is obtained as the canonical notion of bisimulation. This facilitates the reuse of existing work on, for example, up-to techniques.

The logic $\mathbf{FOL}_{\blacktriangleright}$ is, as far as I know, the first logic for mixed inductive-coinductive programs that supports proof discovery, without requiring the user to find an induction or coinduction hypothesis upfront. Moreover, due to the local proof correctness condition, the proof rules and the soundness proof are fairly easy to understand. This is in contrast to cyclic proof systems, which have global correctness criteria.

Finally, the approach to automatically prove program equivalences by constructing a bisimulation up-to is a novel instance of a similar technique that has been studied for finite automata and finite transition systems [Bon+13; BP13; CH89]. In the setting of inductive-coinductive programs, these techniques were not used before to devise algorithms for equivalence checking.

Future Work

There are plenty of open questions concerning the material of this chapter. I will list a few that strike me as important, but surely the attentive reader will find more.

First of all, there is some general improvement that can be made on the different notions of bisimulation for (higher-order) languages by Sumii and Pierce [SP07] and Sangiorgi et al. [SKS11]. Recall that we have worked with bisimulations indexed by types in Section 5.1.2, which allowed us to define observational bisimulations very smoothly by appealing to standard coalgebraic machinery. This definition also enabled us to reuse existing work on up-to techniques to improve the bisimulation proof method. Up-to techniques are also used in [SKS11], but the notion of compatibility has not been used there yet, and all the soundness proofs for these up-to techniques are quite ad-hoc. Concerning [SP07], the situation is even worse, since a bisimulation there is not just a relation between terms but between terms, type contexts and assignments of types to type variables. This leads to the problem that bisimulations in [SP07] are not even closed under unions, which prevents the construction of a largest bisimulation. In both cases, one can work instead with binary relations between terms that are indexed by environments. Such environments are assignments of terms to variables in the case of [SKS11] and assignments of types to type variables for [SP07]. By using indexed binary relations, it becomes then possible to use standard coalgebraic methods like compatible up-to techniques for these notions of bisimulation, just like we did in Section 5.1.

An obvious deficit of $\mathbf{FOL}_{\blacktriangleright}$, which we discussed in Example 5.2.32, is its incompleteness in mixed inductive-coinductive cases. The problem here is that the later modality allows us to derive an induction principle for inductive types only because observational equivalence on such a type can equivalently be characterised inductively. This claim is supported by the exhaustive set of the tests

on natural numbers that we gave in Example 4.1.20. So to prove the property in Example 5.2.32, we either need to add an explicit induction principle to the logic or we need to devise another way of using the later modality to obtain a true induction principle. Adding an explicit induction principle is possible and should be sound, since we can obtain an up-to technique that models induction for (separable) types, just as we did in Section 5.1.3 for the substream example. However, adding an induction principle is also unsatisfactory because this breaks proof discovery, a proof method that we embraced when constructing $\mathbf{FOL}_{\blacktriangleright}$. So it would be better, if we could obtain instead the missing induction principle by revisiting the way we use the later modality. A possibility might be to have several later modalities, as proposed by Atkey and McBride [AM13], so as to control the use of the **(Löb)**-rule in different parts of a proof.

Overcoming the lack of a proper induction principle leads also to the question of whether the techniques we developed to prove soundness apply to more general predicates and relations. It should certainly be possible to construe logics for general coinductive predicates like we did here for observational equivalence. If the sets over which such predicates are constructed have some inductive flavour, then this will be reflected in the logic, as we have seen for observational equivalence on inductive types. What about inductive or even mixed inductive-coinductive predicates though? The problem here is of course that these are not constructed as ω^{op}-chains but as ω-chains, thus as increasing rather than decreasing chains. However, both forms of chains are given by induction on natural numbers. Therefore, the fact that the provability logic characterises well-founded induction, see the discussion above, should enable us to construct a logic, possibly with several later modalities, that also accounts for inductively defined predicates.

Another direction for extending $\mathbf{FOL}_{\blacktriangleright}$ is to allow more general (coinductive) predicates that cannot be constructed as ω^{op}-chain, but which require transfinite induction instead, cf. the proof of Lemma 5.2.20. Also here, the remark that provability logic characterises well-founded frames applies. Thus, it should not even be necessary to change the logic but just the interpretation, in the sense that a model has general ordinals as index rather than just natural numbers.

Finally, it would be interesting to see how far the language fragment of $\lambda\mu\nu{=}$ of Section 5.3.2 can be extended so that observational equivalence is still decidable on that fragment. A possible source of inspiration can be here the work on higher order model checking by Kobayashi and Igarashi [KI13], Ong [Ong15], and others.

Notes

[30] One can see the reduction steps, which can occur in between observations in the above transition system, as *internal computations*. By this we mean computation steps that are not labelled and therefore not visible from outside the system. Thus, the above definition of $\longrightarrow\!\!\triangleright\!\!\triangleright$ from $\longrightarrow\!\!\!\!\!\rightarrow$ and $\longrightarrow\!\!\triangleright$ is the so-called *saturation* of a transition system that has two types of transitions: the labelled transitions given by $\longrightarrow\!\!\triangleright$, and unlabelled transitions given by reduction steps through $\longrightarrow$. Under this reinterpretation of the transition system given by $\longrightarrow\!\!\triangleright\!\!\triangleright$, the notion of bisimilarity, which we will study in Section 5.1.2, is actually weak bisimilarity in this alternative transition system, cf. [San11, Sec. 4.2].

[31] One may object to this way of casting the transition system $\longrightarrow\!\!\triangleright\!\!\triangleright$ into the coalgebra δ for two reasons. First, it seems to obscure that fact that δ comes from a labelled transition system. However,

just having one set of labels is problematic, because the labels that the transition system may take dependent on the type of the terms from which a transition originates. Second, the $(-) + \mathbf{1}$ part of the functor F might be seen as superfluous, as in the case that a term has no WHNF one may also choose the empty set for the outgoing transitions. The problem with this approach is that in the case of sum types we cannot choose whether to use the empty set in $\mathcal{P}(\Lambda^{=}(A_1))$ or $\mathcal{P}(\Lambda^{=}(A_2))$.

[32] We already mentioned that Theorem 5.1.9 is often referred to as expressiveness of the testing logic for bisimilarity. Klin [Kli07] proves a general result that allows one to derive expressiveness of a logic for bisimilarity from some conditions on the transition system and logic at hand. More specifically, one has to establish that the logic and the transition system interact through a so-called distributive law. There is a way, albeit rather complex, to describe the testing logic in the appropriate form and give the necessary distributive law. However, this distributive law violates the injectivity condition that is required in [Kli07].

Another approach to obtain adequacy results is the bialgebra framework of Turi and Plotkin [TP97]. Unfortunately, also their results do not apply here, see Note 33 for a discussion.

[33] The reader might be wondering why we need a complicated up-to technique that essentially acts like a contextual closure, namely by extending a relation R for each pair $(s,t) \in R_{F_\mu \to A}$ and each $u \in \mathbf{ON}_{F_\mu}$ by $(s\,u, t\,u)$. We remarked earlier that without further conditions on R, such a closure is not sound for observational bisimulation proofs. This can be seen as follows.

Suppose there is a general, Φ-compatible contextual closure $C^{\mathrm{ctx}} \colon \mathrm{Rel}_{\Lambda^{=}} \to \mathrm{Rel}_{\Lambda^{=}}$. Now for any two types $A, B \in \mathrm{Ty}$ and terms $s, t : A \to B$, we can form the relation $R_{A\to B} = \{(s,t)\}$. The contextual closure gives us for all $u \in \mathbf{ON}_A$ that the pair $(s\,u, t\,u)$ is related by $C^{\mathrm{ctx}}(R)_B$. We then also have that $(s\,u, t\,u) \in C^{\mathrm{obs}}(C^{\mathrm{ctx}}(R))_B$. Since C^{obs} is $\equiv$-closed, we obtain from Lemma 5.1.8 that $(s,t) \in \Phi(C^{\mathrm{obs}}(C^{\mathrm{ctx}}(R)))_B$. By Φ-compatibility of C^{ctx}, we find that R is an observational bisimulation up to $C^{\mathrm{obs}} \circ C^{\mathrm{ctx}}$, hence $s \equiv_{\mathrm{obs}} t$. Because s and t are arbitrary terms and since by Proposition 4.1.28 there are terms that are not observationally equivalent, it cannot be the case that C^{ctx} is a compatible up-to technique.

It now follows that there can be no bialgebraic description, in the style of Turi and Plotkin [TP97], of the transition system in Section 5.1.1: Suppose we had such a description. It would then follow that there is a compatible contextual closure, see [Bon+14, Cor. 3]. Therefore, such a description cannot exist by the above argument. Analysing the above argument further, we see that the deeper reason is that the term language of $\lambda\mu\nu{=}$ contains the observations of the transition system as operations. It is known [Kli11] that specifications that allow observations on the underlying transition system need to be given through so-called coGSOS specifications (the dual of Generalised Operational Semantics). As the language $\lambda\mu\nu{=}$ also allows complex terms as output, we need to combine coGSOS with GSOS specifications, see [Kli11]. Thus, to express the transition system in Section 5.1.1 as bialgebra, we would need a format for operational semantics that accommodates both GSOS and coGSOS. However, this combination fails in general, as shown by Klin [Kli11] and Klin and Nachyla [KN14]. Also the tyft/tyxt formats [GV92; Sta08] for operational semantics do not help in this case, as reduction steps in the transition system involve complex terms as source. So to summarise, it is not clear whether there is a way to express the transition system from Section 5.1.1 by using bialgebras, in such a way that some results, like congruence, can be derived from more general results, cf. Note 32.

[34] It might help the reader to compare Lemma 5.1.18 to the following mutual induction principle for

predicates on natural numbers. Suppose that $\varphi(n, m)$ is a first order formula over pairs of natural numbers. If the following three conditions, corresponding to those in Lemma 5.1.18, are fulfilled, then $\varphi(n, m)$ holds for all natural numbers $n, m \in \mathbb{N}$:

i) $\varphi(0, 0)$ holds;

ii) If $m \in \mathbb{N}$ and $\forall n.\, \varphi(n, m)$ holds, then $\varphi(0, m + 1)$ holds; and

iii) If $n, m \in \mathbb{N}$ and $\varphi(n, m)$ holds, then $\varphi(n + 1, m)$ holds.

One then shows that $\varphi(n, m)$ holds for all $m, n \in \mathbb{N}$ through proving by induction on m that the formula $\psi(m)$ with $\psi(m) \coloneqq \forall n.\, \varphi(n, m)$ holds. This corresponds to the outer induction that we employed in the proof of Lemma 5.1.18. Both, $\psi(0)$ and $\psi(m+1)$ are then again proved by induction on n, where one uses the corresponding conditions on φ.

[35] A reader who is familiar with modal logic, and with provability logic in particular, will have noticed that both the necessitation rule **(Nec)** and the implication introduction rule **(→-I)** are part of $\mathbf{FOL}_{\blacktriangleright}$. Hence, we can derive $\vdash \varphi \to \blacktriangleright \varphi$ for any formula φ. In the usual provability logic [Sol76], this implication would not be valid, only the sequent $\varphi \vdash \blacktriangleright \varphi$ is valid there. The reason is that the sequent is external in the provability interpretation, that is, we can prove that if φ is a theorem, then there is a proof of φ in the logic under consideration. In contrast, the implication means that *inside* the considered system we can show that if φ holds, then φ is provable. Clearly, this does not hold in a consistent logic, see [Smo85, Rem. 14, p. 66] for a discussion. The reason why we can allow the implication here is because we choose a specific model, see Section 5.2.2, which is different from the provability interpretation and which fulfils this implication.

[36] **Negation and Inequality** One might be tempted to introduce falsity, similarly to how we introduced $\top$ in Lemma 5.2.7, by defining $\bot' \coloneqq \underline{0} \sim \underline{1}$. However, we cannot derive the elimination scheme $\bot' \to \varphi$ for an arbitrary formula φ. This can be seen as follows. Let us use the usual definitions of negation $\neg\varphi \coloneqq \varphi \to \bot$ and inequality $s \nsim t \coloneqq \neg(s \sim t)$. Then, in fact, one of the axioms of Peano arithmetic is $\underline{0} \nsim \underline{1}$, which reads as $\bot' \to \bot$ for the above definition of $\bot'$. This axiom, in turn, is independent of the other axioms in Peano arithmetic. Since the present system does not connect $\bot$ and $\sim$ other than through **(⊥-E)**, it is likely that $\bot' \to \bot$ cannot be derived, just as it cannot be derived in Peano arithmetic.

We might expect at least though that the above definition of $\bot'$ allows us to derive for $s : A$ and $t : B$ the implication $\kappa_1\, s \sim_{A+B} \kappa_2\, t \to \bot'$. This, however, is again not possible because s and t are fixed, thus we cannot infer from $\kappa_1\, s \sim \kappa_2\, t$ any information about $\underline{0} \sim \underline{1}$. As the same problem is shared by $\bot$ in $\mathbf{FOL}_{\blacktriangleright}$, we could remedy this problem by adding a new rule that generalises the axiom $\underline{0} \nsim \underline{1}$ of Peano arithmetic:

$$\frac{\Gamma \vdash_{\mathrm{Ty}} s : A \qquad \Gamma \vdash_{\mathrm{Ty}} t : B}{\Gamma \mid \Delta \vdash \kappa_1\, s \nsim \kappa_2\, t}$$

This rule allows us to directly derive both $\bot' \to \bot$ and $\kappa_1\, s \sim \kappa_2\, t \to \bot'$.

The addition of this rule would be fine in a classical logic, since there we would be able to derive $\neg(s \nsim t) \to s \sim t$ by appealing to the law of excluded middle. If, however, we want to have a robust set of axiom and thus stick to the intuitionistic axioms we used to so far, then we need to have

a different way of dealing with inequality. The common way to do so is to introduce a so-called *apartness relation* into the logic, see [TvD88]. In the setting of **FOL$_\blacktriangleright$**, we would introduce for any two terms s, t a formula $s \# t$, which should be read as "s and t are apart". As an intuition, one can think of $s \# t$ as stating that there is a test that distinguishes s and t. The axiom that makes the apartness relation work, is then $\neg(s \# t) \leftrightarrow s \sim t$. We will comment on the possible semantics of the apartness relation and the validity of the axiom in Note 38.

[37] It is important to note that both the rules **(≐-Repl)** in Figure 5.3 and **(Conv)** in Lemma 5.2.9 only work for definitionally equal and convertible terms, respectively, and *not* for observationally equivalent terms. The reason is that, if we were be able to derive the more general replacement rule

$$\frac{\Gamma \mid \Delta[s/x] \vdash \varphi[s/x] \qquad \Gamma \mid \Delta \vdash s \sim t}{\Gamma \mid \Delta[t/x] \vdash \varphi[t/x]}$$

then the proof system would be inconsistent. This can be seen as follows.

First, we can derive from the hypothetical replacement rule that $\sim$ is a congruence on function terms, that is, for all terms $u, v : A$ and $f : A \to B$, $u \sim_A v$ implies $f\, u \sim_B f\, v$. Suppose now that we are given $s, t : A^\omega$. By using $f = \lambda x.(x.\mathrm{tl})$ we can, in combination with **(K)**, derive in **FOL$_\blacktriangleright$** that $\blacktriangleright(s \sim t)$ implies $\blacktriangleright(s.\mathrm{tl} \sim t.\mathrm{tl})$. We can combine this with (5.14) and **(ν-Ext)**, see also Example 5.2.12 below, to prove that $\blacktriangleright(s \sim t)$ implies $s \sim t$. So, finally, we can apply **(Löb)** to obtain $s \sim t$. Since s and t are arbitrary streams, we obtain from the (hypothetical) replacement rule that the proof system would become inconsistent relative to the desirable property that $\underline{0} \nsim \underline{1}$. This property holds in the model that we give in Section 5.2.2, thus a general replacement rule would be unsound for this model.

[38] Recall from Note 36 that we discussed a possible replacement for inequalities $s \nsim t$, namely an apartness relation $s \# t$. This relation ought to fulfil the axiom $\neg(s \# t) \leftrightarrow s \sim t$. We also hinted in Note 36 at a possible interpretation: s and t are considered to be apart if there is a test that distinguishes them. Given the development in this section, we can define

$$n \Vdash s \#_A t \coloneqq (s, t) \notin \overleftarrow{\Phi_c}(n).$$

One then has to prove for $n \in \mathbb{N}$ that $(s, t) \in \overleftarrow{\Phi_c}(n)$ is the same as saying that s and t cannot be distinguished by any test that makes at most n observations on fixed point types, cf. Theorem 5.1.9, Lemma 5.2.18 and Definition 5.2.19. For example, the tests $\varphi_k :\downarrow \mathrm{Nat}$, which we devised in Example 4.1.20, makes at most k observations on fixed point types. Let us write $\ell(\varphi) \leq n$, if φ makes at most n observations on fixed point types. This characterisation of $(s, t) \in \overleftarrow{\Phi_c}(m)$ allows us to equivalently give semantics to $s \# t$ by

$$n \Vdash s \#_A t = \exists \varphi \in \mathrm{Tests}_A.\, \ell(\varphi) \leq n \wedge \neg(s \Vdash \varphi \iff t \Vdash \varphi).$$

Since $s \vDash \varphi$ is decidable for any term and test, we can now derive

$$\begin{aligned}
n \vDash \neg(s \#_A t) &= \neg(\exists \varphi \in \mathrm{Tests}_A.\, \ell(\varphi) \leq n \wedge \neg(s \vDash \varphi \iff t \vDash \varphi)) && (*)\\
&= \forall \varphi \in \mathrm{Tests}_A.\, \neg(\ell(\varphi) \leq n) \wedge (\neg(\neg(s \vDash \varphi \iff t \vDash \varphi))) \\
&= \forall \varphi \in \mathrm{Tests}_A.\, \ell(\varphi) > n \wedge (s \vDash \varphi \iff t \vDash \varphi) && (**)\\
&= \forall m > n.\, (s,t) \in \overleftarrow{\Phi_c}(m) && (*)\\
&= (s,t) \in \overleftarrow{\Phi_c}(n) && (***)\\
&= n \vDash s \sim t,
\end{aligned}$$

where the identities $(*)$ hold by the above discussion, $(**)$ holds because $s \vDash \varphi \iff t \vDash \varphi$ is decidable, and $(***)$ is obtained from the fact that $\overleftarrow{\Phi_c}$ is an ω^{op}-chain (i.e., $n < m$ implies $\overleftarrow{\Phi_c}(m) \sqsubseteq \overleftarrow{\Phi_c}(n)$). This proves that the desired axiom $\neg(s \# t) \leftrightarrow s \sim t$ indeed holds.

[39] The class of type for which the induction and coinduction principles can be derived has actually to be restricted to non-mutual fixed point types. The reason for this is that **FOL$_\blacktriangleright$** does not have fixed point formulas, which would be necessary to formulate the induction and coinduction hypotheses for mutual types.

[40] In general, we have to deal with more complicated copatterns, but we shall ignore this technicality for the present discussion.

Categorical Logic Based on Inductive-Coinductive Types

The greatest art in theoretical and practical life consists in changing the problem into a postulate; that way one succeeds.

— J. W. von Goethe in a letter to C. F. Zelter.[41]

In the previous chapters we have studied mixed inductive-coinductive programs and some reasoning principles for those programs. A prototypical example is the stream filtering program from Example 3.2.11. What is more, we have also used this program in Example 5.1.13 to define the substream relation, which is a mixed inductive-coinductive relation, as we have have seen in Section 5.1.3. Thus, we need a language for expressing more general inductive-coinductive predicates and relations, and reason with them.

One could extend the logic $\mathbf{FOL}_{\blacktriangleright}$ with fixed point operators that would allow us to express inductive-coinductive predicates, cf. Note 39. This is a route that has been taken by Aho and Ullman [AU79] and Gurevich and Shelah [GS86] with the goal that inductive predicates, like the transitive closure of a relation, can be expressed in an extended first-order logic without having to invoke full second-order logic. The problem with this approach is that we would have to repeat the type structure of the languages $\lambda\mu\nu$ and $\lambda\mu\nu=$ in the formulas and the proofs. That is to say, we will have on the one hand a calculus with fixed point types and the corresponding term formation rules, and on the other hand a logic with fixed point formulas, which come again with a set of proof rules. Since both, term formation and proof rules, would look essentially the same, we would repeat all the work that we did for the calculus $\lambda\mu\nu$ again for the logic over that calculus. Thus, it would save us a lot of work, if we can unify programming and reasoning into one system, as we then only need to give one set of rules that subsume term formation and proof rules. This is where dependent type theory enters the picture.

In a dependent type theory, one allows terms to occur inside types. Thus, it would be more precise to say that a dependent type theory is a theory of *types that can depend on terms*. But what does this actually mean and how can such a theory be useful? We will answer these questions in the introductory Section 6.1, and thereby lay the foundation for the remainder of this thesis. Before that, let us see how the actual content of this chapter is structured though.

Structure of the Chapter

As we mentioned above, the aim of this and the following chapter is to develop a theory of dependent inductive-coinductive types. It turns out that inductive and coinductive types are enough to account for propositional and first-order connectives, hence the title "Categorical Logic Based on Inductive-Coinductive Types". In this chapter, we first develop dependent inductive-coinductive types in the context of category theory. This allows us to hide many difficulties arising from a syntactic approach, which we still have to face in Chapter 7, such as the need for variable renaming or reduction relations.

The basic idea to describe dependent inductive-coinductive types is to extend the approach of Hagino [Hag87] from simple to dependent types. Hagino gave in his seminal paper a calculus of, what he called, categorical data types. These types correspond to initial and final dialgebras for functors that determine the type of the constructors and destructors, respectively, of data types. More specifically, an inductive type with n constructors is category theoretically given in [Hag87] as an initial (F, Δ)-dialgebra, where $F, \Delta \colon \mathbf{C} \to \prod_{i=1}^{n} \mathbf{C}$ are functors, of which Δ is the diagonal and F determines the domain of each constructor. This setup allows Hagino to represent coproduct and product as inductive and coinductive types, respectively. For example, he uses that the coproduct is left-adjoint to the diagonal functor, so that for the choice $F \colon \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ with $F(X) = (A, B)$, we have that $A + B \in \mathbf{C}$ is an initial (F, Δ)-dialgebra. Since dependent types can be seen as objects in fibres of a fibration, we just replace the category $\mathbf{C}$ with the fibres of a given fibration $P \colon \mathbf{E} \to \mathbf{B}$. For example, inductive types that are indexed by $I \in \mathbf{B}$ arise then as initial (F, G_u)-dialgebras for functors $F, G_u \colon \mathbf{P}_I \to \mathbf{P}_{J_1} \times \cdots \times \mathbf{P}_{J_n}$, where G_u generalises the diagonal functor to reindexing as follows. If $u = (u_1, \ldots, u_n)$ are morphisms $u \colon J_k \to I$ in $\mathbf{B}$, then G_u is given by reindexing along all these morphisms: $G_u = \langle u_1^*, \ldots, u_n^* \rangle$. How this allows us to describe dependent types and thereby generalises Hagino's idea is the subject of the first Section 6.2. The outcome of this endeavour is the notion of *dependent recursive type closed categories (μPCC)*, which allow the interpretation of (strictly positive) dependent inductive-coinductive types.

Throughout the remainder of the chapter we study μPCCs further. In Section 6.3, we show how to construct types in locally Cartesian closed categories from only initial algebras of polynomial functors. We obtain thereby a class of models for dependent recursive type closed categories. This result extends the work of Abbott [Abb03], Abbott et al. [AAG05] and Gambino and Kock [GK13] to the fairly general dependent types that are the subject of the present chapter.

With a dependent type theory comes naturally a first-order logic, as we will see in Section 6.1. Thus, we go ahead in Section 6.4 and analyse this logic further. We do so by constructing a fibration of predicates over types of a μPCC. In this fibration, we show that the validity of the induction principle for inductive types is equivalent to having *strong coproducts*. This result is not surprising, given the development in [FGJ11; HJ97], but it puts the strong elimination for sum types in a broader perspective, as its validity has been discussed in the past.[42] Similarly, we show that coinductive types always come with a bisimulation proof principle in μPCCs, which is again not very surprising as these types come about as final dialgebras. These two results give us the basic principles to reason about inductive-coinductive types in the logic that arises from data type closed categories.

To interpret sum and product types over a fibration, one has to require the so-called Beck-Chevalley condition to hold, see Definition 2.4.4. This condition allows us to move reindexing functors over products and coproducts, which ensures that, respectively, all products and coproducts are essentially the same throughout the fibres. As such, this condition corresponds to the expected definition of substitution on sum types: $(\Sigma x : A[y].\, B[x, y])[t/y] = \Sigma x : A[t].\, B[x, t]$, where x and y are distinct variables. We generalise in Section 6.5 the Beck-Chevalley condition to the inductive-coinductive types that we introduce in Section 6.2. Moreover, we show that our Beck-Chevalley condition is equivalent to the usual one for products and coproducts, and that binary products, binary coproducts and final objects are fibred if we construct these as data types that satisfy the Beck-Chevalley condition.

Original Publication The sections 6.2 and 6.3 are based on [Bas15a], but have been expanded in explanation and detail. Another difference is that the terminology has been changed slightly to something more accurate: Data type complete categories are now called dependent recursive type complete categories, μP-complete categories for short, and data type closed categories are accordingly now called dependent recursive type closed categories (μPCC instead of DTCC). Both sections 6.4 and 6.5 stem from unpublished notes.

6.1. Hitchhiker's Guide to Dependent Type Theory

I will try to give here a short overview over what dependent type theory is, the various perspectives on dependent types in programming, logic and category theory, and the origins of dependent types. If the reader is familiar with all of this, then she might wish to skip ahead to Section 6.2.

Dependent Types in Programming

Our first stop is the use of dependent types in programming. As an example, suppose we want to write a program involving lists. Upon implementing the function head$\colon A^* \to A$, which shall return the first element of the input list, we find that this function is undefined on the empty list. We can solve this by aborting the program, if "head" gets applied to the empty list. But this leaves us with potential run-time errors, if we are not careful in using head. Another example that we might encounter is the function zipWith$\colon (A \to B \to C) \to A^* \to B^* \to C^*$, which is intended to combine two lists entry-wise with a given function. The problem here is that if the input lists do not have the same length, then the output can only be as long as the shorter of the two input list. Such behaviour can easily lead to bugs in programs, since in this form zipWith silently discards parts of the input. A possible way to prevent both faulty and unexpected behaviour is the use of dependent types, as we will explain in what follows.

Before getting into dependent types, let us briefly digress to discuss another way to deal with the above problems. In functional programming, one often alters the return type of head and zipWith, as to allow the function caller to handle possible errors. For example, one can give head the type $A^* \to A + \mathbf{1}$ and return $\kappa_2\, \langle\rangle$ in case the input list is empty. For zipWith we face a choice however: Do we use $C^* + \mathbf{1}$ as return type to allow signalling an error, or do we use $C^* \times (\mathbf{1} + A^* + B^*)$ to give the user the opportunity to handle the remainder of the longer list, if the inputs do not have the same length? Since possible errors have to be handled explicitly, altering the return type clearly leads to more complex code, which one might want to avoid. This is what we can achieve by using dependent types.

So how can dependent types be helpful to prevent software bugs? A common slogan in typed programming is that programs that type check "can never go wrong". The actual content of this statement depends, however, on the expressiveness of the type system at hand. For instance, the type system of the programming language C can be utilised to make some guarantees, but of course a lot of things still go wrong because they are not captured by the type system: wrong indexing into an array, memory leaks, etc. In a dependent type theory, terms may appear in types, thus we are able to use arbitrary run-time information during type checking. This is what makes dependent types such a powerful tool to specify aspects of run-time behaviour through types and capture errors during type checking.

Let us employ dependent types to capture the possible errors that might occur from the use of the list functions, which we described above. Suppose that for $n : \mathrm{Nat}$ there is a type "$\mathrm{List}_A\ n$" of lists that have exactly length n, also known as vectors of length n. Then the situation regarding the use of head and zipWith can be improved by implementing for all values $n : \mathrm{Nat}$, functions head_n and $\mathrm{zipWith}_n$ of type $\mathrm{List}_A\ (n+1) \to A$ and $(A \to B \to C) \to \mathrm{List}_A\ n \to \mathrm{List}_B\ n \to \mathrm{List}_C\ n$, respectively. These types say that head_n can only be applied to non-empty lists, and zipWith expects lists of the same length as input and also returns a list of that length. This reading allows us to implement head_n so that it is well-defined for every input, and $\mathrm{zipWith}_n$ so that it does not discard any input.

Of course, we do not want to implement for each $n : \mathrm{Nat}$ the functions head and zipWith separately, but rather have in each case one function *for all* $n : \mathrm{Nat}$. This can be achieved by using dependent products as follows. Consider for a set I and a family X of sets $\{X_i\}_{i \in I}$ the usual set-theoretic product of X given by

$$\prod_{i \in I} X_i = \left\{ f : I \to \bigcup_{i \in I} X_i \;\middle|\; \forall i \in I.\, f(i) \in X_i \right\}.$$

To obtain the dependent product of a type, we mimic this definition by a type as follows. Suppose that A is a type, and that the variable x occurs freely in the type $B[x]$ and ranges over A. We denote this situation for the moment by $x : A \vdash B[x] : \mathbf{Type}$. The intuition for the dependent product, written as $\Pi x : A.\, B[x]$, is thus that it consists of functions f, such that for all $t : A$ we have $f\ t : B[t]$, where $B[t]$ is the type $B[x]$ in which t is substituted for x, cf. Definition 5.2.2. This dependent product type allows us to assign general types to head and zipWith:

$$\begin{aligned} &\mathrm{head} : \Pi n : \mathrm{Nat}.\, \mathrm{List}_A\ (n+1) \to A \\ &\mathrm{zipWith} : (A \to B \to C) \to (\Pi n : \mathrm{Nat}.\, \mathrm{List}_A\ n \to \mathrm{List}_B\ n \to \mathrm{List}_C\ n). \end{aligned}$$

For example, we can apply head to $2 : \mathrm{Nat}$ to obtain the function $\mathrm{head}\ 2 : \mathrm{List}_A\ 3 \to A$, which we called head_2 above.[43]

It is interesting to note that if $B[x]$ does not use the variable x, then this type family is constant. Hence for $f : \Pi x : A.\, B[x]$, we have that $f\ t : B$ for all $t : A$, which is to say that f is a function of type $A \to B$. More generally, we can define for types A and B, where x does not occur in B, the function type $A \to B$ to be given by $\Pi x : A.\, B$.[44]

Continuing our programming journey, we may find that we have to maintain invariants in the program and ensure through appropriate interfaces that these invariants are never violated. The following example from [Jac99] illustrates how dependent types allow us to utilise the type checker to enforce such invariants. Suppose our program may involve date calculations, so we want to represent dates as a type. The typical approach would be to use triples of type $\mathrm{Nat} \times \mathrm{Nat} \times \mathrm{Nat}$, in which the components are year, month and day. The problem with this representation is that there are invalid combinations, since a year has only twelve months and also the day of a month is limited. Even worse, the number of days depend on the month and the year. A way to enforce the invariant that all date triples are valid can be achieved, for example, with interfaces in object oriented languages or with polymorphism in functional languages [Pie02]. In both cases, we as programmers must still be disciplined enough to ensure that the invariant is never violated inside the implementation of the interface for the date type. Therefore, it would be better if the programming language provided a mechanism to enforce the correctness invariant on the type itself. What we actually would like to have in the above example is that for $n : \mathrm{Nat}$ there is a type $\mathrm{Pos}\ n$ of positive numbers bounded by

n. Then the correct type for months would be Pos 12. However, what do we do with the day? The answer to this lies in using types that mimic coproducts of set families. Recall that the coproduct of an I-indexed family X is given by

$$\coprod_{i \in I} X_i = \{\langle i, x\rangle \mid i \in I, x \in X_i\}.$$

The types that correspond to the coproduct in dependent type theory are the dependent sum types. These are denoted for a type $B[x]$ with $x : A \vdash B[x] : \mathbf{Type}$ by $\Sigma x : A.\, B[x]$. Elements of a sum type are dependent pairs, that is, given $s : A$ and $t : B[s]$, there is an element $\langle s, t\rangle$ of type $\Sigma x : A.\, B[x]$. In the case of dates, the type we are after is given by

$$\Sigma y : \mathrm{Nat}.\, \Sigma m : \mathrm{Pos}\ 12.\, \mathrm{Pos}\ (\mathrm{days}\ y\ m),$$

where "days y m" computes the numbers of days in the month m of the year y. As required, any element of this type is now a valid date, and the promise that computations on dates only return valid dates can be validated through *type checking*.

There are many more examples that underpin the usefulness of dependent types to ensure more complex correctness properties of programs. As such, dependent types appear in an increasing number of programming languages: Extensions of Haskell allow the simulation of some aspects of dependent types; Agda [Agd15] was designed in the first place as a total programming language; and Idris [Idr17] combines, just like Epigram [AMM05], partial programs with dependent types. Lastly, dependent types also allow the verification of correctness properties of hardware specifications [BMH07; HDL90; PS15]. So we may say that dependent types have proven to be a useful programming tool.

However, as we all know, there is no such thing as a free lunch. In the case of dependent type theories, the express power comes at the cost that type checking becomes more difficult because we may have to evaluate some computations in the process. For instance, we claimed above that head 2 is of type $\mathrm{List}_A\ 3 \to A$, whereas, strictly speaking it is of type $\mathrm{List}_A\ (2 + 1) \to A$. Thus, we first need to compute $2 + 1$ to be able to apply head 2 to a list of type $\mathrm{List}_A\ 3$. We will see in Chapter 7 how this works in a concrete type theory. Other problem that come with moving to dependent type theories is that deciding whether a type is inhabited is in general undecidable. This is not so surprising though, as dependent type theories correspond to first-order logic, as we will see.

We this, it is now time to move on to the second use of dependent types, and their actual origin as a correspondence with first-order logic.

Dependent Types in Intuitionistic Logic

Logical reasoning and the axiomatic method had been known already by the ancient Greek, but it was only by the turn of the 20th century that logic became an independent branch of mathematics. This was caused by the need for a rigorous foundation of mathematics to avoid the contradictions in arguments and paradoxes that were discovered around that time. The development of mathematical logic led to several logical systems but also to the philosophical question of what we accept as reasoning principles. Most famously, we distinguish today between classical and constructive logic, where the latter requires proofs to have a certain constructive content. For example, constructivists therefore reject the principle of excluded middle and the axiom of choice. Requiring proofs to be given by explicit constructions was an idea that already lurked around in algebra and analysis by

the end of the 19th century[45], but it was only in the work of Brouwer and his successors that this idea was put into shape [Tro11].

These days, we associate with constructive logic the so-called *Brouwer-Heyting-Kolmogorov interpretation* (*BHK-interpretation*), which encapsulates the idea that a proof is given by an explicit construction. For instance, a proof of the formula $\forall x : A.\, \exists y : B.\, \varphi(x, y)$ is a procedure that for any a in A constructs an element b in B, such that $\varphi(a, b)$ holds, cf. [TvD88, Sec. 1.3]. With the advent of the study of recursion and computable arithmetic, this interpretation of proofs was pushed even further to require that proofs are machine computable constructions. One of the most famous instances of the use of recursion is the encoding used by Gödel in his incompleteness proof, but also realisability of proofs and Markov's principle arose from there, see [TvD88, Chap. 4]. This idea, the computability of proofs, is today the main driving force behind the type theoretic approach to the BHK-interpretation of intuitionistic logic.

Types made their first appearance in Russel and Whitehead's *Principia Mathematica*, but were not studied there in their own right. It was only Church [Chu40] who used types to represent propositions so that proofs of these propositions were given by λ-terms [Chu32]. Coincidentally, it turned out that these very same λ-terms could be used to describe *all* computable functions [Chu36], albeit being untyped. The simple type theory of Church made a severe restriction to the domain of functions that could be described by typeable λ-terms. This restriction ruled out, for example, the option of encoding all of higher-order arithmetic of System T, thus also the encoding of Gödel's Dialecta interpretation [Göd58; BDS13, Sec. 5.3]. Girard [Gir71; Gir72] overcame this problem by introducing System F,[46] which extends Church's simple types with impredicative polymorphism. Besides capturing a larger class of total higher-order functions, System F includes universal second-order quantification. However, it is lacking first-order quantification, which is what dependent types can achieve.

Dependent types were first conceived by de Bruijn for his Automath project, see [dBru66] for an informal overview and [dBru68] for a fully formal description. The goal of the Automath project was the formalisation of mathematics in a human-readable language, assisted by a computer that checks the correctness of proofs. Later, other logicians picked up on the idea of dependent types to account for first-order logic. The dependent type theory that is most popular today is Martin-Löf's type theory (MLTT), which he first developed in 1972 but never published, and then revised in [Mar75b] to overcome an inconsistency [Gir72]. It is the work of Martin-Löf that forms the basis of many modern proof assistants. For example, Coq [Coq12] is based on the Calculus of (Inductive) Constructions [CH88; Pau15], and extends MLTT with, among other things, a special type of propositions. Agda, on the other hand, is directly based on MLTT [Nor07]. There have been plenty of other type systems that feature dependent types like Pure Type Systems [GN91; KLN04], Nuprl [Con97], and the Extended Calculus of Constructions [Luo89], which is the type theory of the proof assistant LEGO. A more detailed discussion can be found in the introduction of [NGdV94] or in [NG14]. Due to its tight connection to logic and category theory, we base our understanding of dependent types here on the work of Martin-Löf though.

After this history lesson about constructivism and dependent types, we shall now take our second stop to see how dependent types are viewed from the perspective of logic. Recall that we devised in Chapter 5 the logic $\mathbf{FOL}_{\blacktriangleright}$ that extends standard first-order intuitionistic logic. This logic came with a proof system, in which we represented all proofs as finite trees. These trees are thereby built up using the proof rules of the logic. The idea of Church [Chu32] was to represent such proofs as

λ-terms. Let us illustrate this idea on the two quantifiers.

Among the formulas of $\mathbf{FOL}_\blacktriangleright$ there were universally quantified formulas. To remind ourselves of the rules for those quantified formulas, let us reintroduce some notation. Recall that the logic $\mathbf{FOL}_\blacktriangleright$ came with a judgement $\Gamma \mid \Delta \vdash \varphi$ that holds if the formula φ uses only variables from the context Γ and is derivable in the proof system of $\mathbf{FOL}_\blacktriangleright$ from the assumptions in Δ. We write $\Gamma \vdash_{\mathcal{L}} t : A$, if t is a term from some underlying language $\mathcal{L}$, like $\lambda\mu\nu$, $\lambda\mu\nu=$ or $\mathbf{ON}^{\Gamma}_{A}$, of type A in the object context Γ. Moreover, we write $\Gamma \Vdash \varphi$ if φ is a formula that is type correct for the variables in context Γ, and we denote the substitution of t for x in a formula φ by $\varphi[t]$, if x is uniquely determined. The universal quantifier was then determined by the following two rules.

$$\frac{\Gamma, x : A \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \forall x : A.\, \varphi}\ (\forall\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \forall x : A.\, \varphi \qquad \Gamma \vdash_{\mathcal{L}} t : A}{\Gamma \mid \Delta \vdash \varphi[t]}\ (\forall\textbf{-E})$$

Instead of coding proofs as trees, we can associate to each proof rule a term constructor, thereby turning the above *proof construction rules* into *typing rules*:

$$\frac{\Gamma, x : A \mid \Delta \vdash p : \varphi}{\Gamma \mid \Delta \vdash (\lambda x.\, p) : (\forall x : A.\, \varphi)} \qquad \frac{\Gamma \mid \Delta \vdash p : (\forall x : A.\, \varphi) \qquad \Gamma \vdash_{\mathcal{L}} t : A}{\Gamma \mid \Delta \vdash p\ t : \varphi[t]}$$

These rules should be read as follows. For the first, if p is a proof of φ for an arbitrary $x : A$, then the λ-abstraction $\lambda x.\, p$ proves $\forall x : A.\, \varphi$. The second rule, implements precisely the idea of the BHK-interpretation that a proof p for $\forall x : A.\, \varphi$ is a procedure that produces for each t a proof for $\varphi[t]$. Similarly, we can turn the rules for the existential quantifier

$$\frac{\Gamma \vdash_{\mathcal{L}} t : A \qquad \Gamma \mid \Delta \vdash \varphi[t]}{\Gamma \mid \Delta \vdash \exists x : A.\, \varphi}\ (\exists\textbf{-I}) \qquad \frac{\Gamma \Vdash \psi \qquad \Gamma, x : A \mid \Delta, \varphi \vdash \psi}{\Gamma \mid \Delta, \exists x : A.\, \varphi \vdash \psi}\ (\exists\textbf{-E})$$

into typing rules. How can this be achieved? Well, in the introduction rule we need to provide a witness and a proof that this witness satisfies the corresponding proposition. Thus, a proof of an existential quantifier is given by the pair of this witness and the proof:

$$\frac{\Gamma \vdash_{\mathcal{L}} t : A \qquad \Gamma \mid \Delta \vdash p : \varphi[t]}{\Gamma \mid \Delta \vdash \langle t, p \rangle : (\exists x : A.\, \varphi)}$$

The second rule is a bit more delicate, since we now assign names to proofs, which would lead us to introduce a fresh variable in the conclusion of the elimination rule. A better way is to add the proof of the existentially quantified proposition as an extra assumption. Keeping this in mind, we can also turn **(∃-E)** into a type checking rule for a proof term.

$$\frac{\Gamma \Vdash \psi \qquad \Gamma, x : A \mid \Delta, \alpha : \varphi \vdash p : \psi \qquad \Gamma \mid \Delta \vdash q : (\exists x : A.\, \varphi)}{\Gamma \mid \Delta \vdash \textbf{unpack}\ q\ \textbf{as}\ \langle x, \alpha \rangle\ \textbf{in}\ p : \psi}$$

The proof term "**unpack** q **as** $\langle x, \alpha \rangle$ **in** p" allows us to inspect the existential proof q and use the fact that there is a witness $x : A$ and a proof $\alpha : \varphi$. However, we do not get further information about them, which is why both the witness and the proof are variables in p.

Let us put the proof terms for universal and existential quantifier to use and derive a small tautology of intuitionistic logic:

$$\exists x : A.\, \forall y : B.\, \varphi[x, y] \vdash \forall y : B.\, \exists x : A.\, \varphi[x, y]. \tag{6.1}$$

This requires us to find a proof term p that uses the assumption on the left, which can be accessed through a proof variable α, with

$$\alpha : (\exists x : A.\, \forall y : B.\, \varphi[x, y]) \vdash p : (\forall y : B.\, \exists x : A.\, \varphi[x, y]).$$

Using λ-abstraction, we can define $p \coloneqq \lambda y.\, q$, if we have a proof q with

$$y : B \mid \alpha : (\exists x : A.\, \forall y : B.\, \varphi[x, y]) \vdash q : (\exists x : A.\, \varphi[x, y]).$$

Since we need an element of type A to prove q, we have to extract this element from α. This can be done by using the elimination for the existential quantifier, hence we put $q \coloneqq \mathbf{unpack}\ \alpha\ \mathbf{as}\ \langle x, \beta\rangle\ \mathbf{in}\ r$ for some proof r with

$$y : B, x : A \mid \beta : (\forall y : B.\, \varphi[x, y]) \vdash r : (\exists x : A.\, \varphi[x, y]).$$

The proof r is now a proof of an existential quantifier, thus we use the dependent pairing and the proof β to obtain $r \coloneqq \langle x, \beta\, y\rangle$. Putting all these steps into one proof, we thus have

$$\alpha : (\exists x : A.\, \forall y : B.\, \varphi[x, y]) \vdash \lambda y.\, \mathbf{unpack}\ \alpha\ \mathbf{as}\ \langle x, \beta\rangle\ \mathbf{in}\ \langle x, \beta\, y\rangle : (\forall y : B.\, \exists x : A.\, \varphi[x, y]).$$

From the perspective of the BHK-interpretation, this proof is also an intuitive computational process through which the global choice of x for all y in α is turned into a local choice for all y.[47]

It would be natural to turn the mere derivability in (6.1) into the following implication.

$$\vdash (\exists x : A.\, \forall y : B.\, \varphi[x, y]) \to \forall y : B.\, \exists x : A.\, \varphi[x, y] \tag{6.2}$$

To do so, we need proof terms for the implication, which are basically the same as for the function type or the universal quantifier, only that this time the "domain" of the implication ranges over proofs. Thus, we can form $\lambda\alpha.\, p : \varphi \to \psi$, where p is a proof for ψ that may use the assumption φ through the proof variable α. The implication in (6.2) is then proved by $\lambda\alpha.\, \lambda y.\, \mathbf{unpack}\ \alpha\ \mathbf{as}\ \langle x, \beta\rangle\ \mathbf{in}\ \langle x, \beta\, y\rangle$.

At this point, we have introduced three different types of application and λ-abstraction, all of them only differentiated by their types, as we can see from the following table.

Role	Application	Abstraction
Function Type	$t\, s : B$ for $t : A \to B$ and $s : A$	$\lambda x.\, t : A \to B$
Implication	$p\, q : \psi$ for $p : \varphi \to \psi$ and $q : \varphi$	$\lambda\alpha.\, p : \varphi \to \psi$
Universal Quantification	$p\, t : \varphi[t]$ for $p : (\forall x : A.\, \varphi)$ and $t : A$	$\lambda x.\, p : \forall x : A.\, \varphi$

That there are so many different flavours of abstraction and application is owed to the fact that Church's proof system in [Chu32] has a strict separation between objects and proofs, just like our logic $\mathbf{FOL}_{\blacktriangleright}$. However, if we push further the idea of expressing proofs as elements of types that represent propositions, then we arrive at the *propositions-as-types* interpretation.[48] The idea of this interpretation is that we can associate to each proposition of an intuitionistic logic (propositional, first-order, etc.) an appropriate type and to each proof a term. In the present context, we associate to first-order formulas types of a dependent type theory. More specifically, universal quantifiers correspond to the dependent product types (Π-types) and existential quantifiers to dependent sums (Σ-types) that we already used above. This allows us to subsume all the different rules for applications and abstractions by two rules, as we shall explain now.

Recall that we introduced earlier the dependent product type $\Pi x : A.\, B[x]$ to mimic the set-theoretic product of set-families and that both come with a notion of function application. Now suppose that for a family $\{X_i\}_{i\in I}$ we have made a choice of elements $u_i \in X_i$ for each $i \in I$. Then we can construct a function $f \in \prod_{i\in X} X_i$ with $f(i) \in X_i$ by putting $f(i) = u_i$. This function formation, together with function application, defines the family product. Therefore, we characterise dependent product types by the following two formation rules.

$$\frac{\Gamma, x : A \vdash t : B[x]}{\Gamma \vdash (\lambda x.\, t) : (\Pi x : A.\, B[x])} \qquad \frac{\Gamma \vdash t : (\Pi x : A.\, B[x]) \qquad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B[s]}$$

Let us compare these two rules to those for the universal quantifier above. Two things have happened here. First, the context of assumptions Δ has vanished from the picture. This is because we interpret now propositions as types, thus the assumptions that used to be in Δ appear now in the context Γ. But this also means that the types that appear in Γ may use variables that appear before in Γ. For instance, a possible context would be $(n : \mathrm{Nat}, v : \mathrm{List}_A\, n)$. Second, did objects in the universal quantifier instantiation range before over terms of an external language $\mathcal{L}$, then these objects stem now from the type theory itself. This is seen in the application rule for the dependent product type. Thus, the dependent product type streamlines universal quantification and functions into one type. Similarly, the dependent sum type is given by the following two rules.

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B[s]}{\Gamma \vdash \langle s, t\rangle : (\Sigma x : A.\, B[x])} \qquad \frac{\Gamma, x : A, y : B[x] \vdash t : C \qquad \Gamma \vdash s : (\Sigma x : A.\, B[x])}{\Gamma \vdash \textbf{unpack}\ s\ \textbf{as}\ \langle x, y\rangle\ \textbf{in}\ t : C}$$

These term formation rules unify products and sums with the first-order quantifiers.

Let us exemplify this unification by showing the correspondence between dependent products and sums, and universal and existential quantifier. Suppose that we inductively translate formulas into types and that we have already translated a formula φ into a type $\underline{\varphi}$. We can extend the translation to the quantified versions of this formula by defining

$$\underline{\forall x : A.\, \varphi} = \Pi x : A.\, \underline{\varphi} \qquad \text{and} \qquad \underline{\exists x : A.\, \varphi} = \Sigma x : A.\, \underline{\varphi}.$$

The logical implication is just given by the function type, which in turn is defined in terms of the dependent product, as we have seen earlier. Coming back to our example from equation (6.1) above, we want to show that there is a term t of type

$$(\Sigma x : A.\, \Pi y : B.\, C[x, y]) \to \Pi y : B.\, \Sigma x : A.\, C[x, y].$$

The term that achieves this is literally the one we have used before, only to be interpreted now in the type theory: $\lambda\alpha.\, \lambda y.\, \textbf{unpack}\ \alpha\ \textbf{as}\ \langle x, \beta\rangle\ \textbf{in}\ \langle x, \beta\ y\rangle$. Thus, after the translation of formulas to types, we can just keep the same proof terms as before.

The reader might have noticed that in our table one possibility of abstraction is missing:

Role	Application	Abstraction
Data Extraction	$t\ p : A$ for $t : \varphi \to A$ and $p : \varphi$	$\lambda\alpha.\, t : \varphi \to A$

That is, the extraction of data, which is contained in a proof, back into the object language. Such an extraction is usually not part of a logic,[49] but it can be derived if we approach logic through

dependent types. A typical example for the use of this extraction mechanism is the recovery of the witness of an existential proof:

$$(\lambda y.\, \mathbf{unpack}\ y\ \mathbf{as}\ \langle x, \alpha\rangle\ \mathbf{in}\ x) : (\Sigma x : A.\, B[x]) \to A$$

This extraction of witnesses from sum types will be discussed again in Section 6.4 and Chapter 7.

We round off the type theoretic view on logic by a more concrete example. Suppose we want to use our highly sophisticated type theory to reason about something mundane as holidays. Suppose we have encoded our favourite holidays as a type isHoliday$[y, m, d]$, with

$$y : \mathrm{Nat}, m : \mathrm{Pos}\ 12, d : \mathrm{Pos}\ (\mathrm{days}\ y\ m) \vdash \mathrm{isHoliday}[y, m, d] : \mathbf{Type},$$

where "days" calculates again the days in the given month of a year. The question of how to actually define isHoliday is left for later. From the logic point of view, isHoliday is a predicate (or relation) on years, months and days that should hold if the given date is a holiday.[50] The type theoretic analogue is the existence of a term (a proof) of type isHoliday$[y, m, d]$, provided the given date is a holiday. We then say that the type isHoliday$[y, m, d]$ is *inhabited*. Analogous to the set-theoretic notion of comprehension, which allows us to form a set that contains exactly those elements for which a predicate holds, we can form a type Holiday that is inhabited only by dates that are provably holidays. This type is given by

$$\mathrm{Holiday} \coloneqq \Sigma y : \mathrm{Nat}, m : \mathrm{Pos}\ 12, d : \mathrm{Pos}\ (\mathrm{days}\ y\ m).\, \mathrm{isHoliday}[y, m, d],$$

where we abbreviate the nested sum types by just one Σ. As a type, Holiday resembles the set $\{(y, m, d) \mid \mathrm{isHoliday}[y, m, d]\}$, with the difference that each date in the type Holiday comes with a proof that it actually is a holiday.

Dependent Types in Categorical Logic

It is now time to get to our final destination: the category theoretical perspective on dependent type theory. This completes our trip through the three-fold relation between computations, logic and category theory. The reason for discussing the category theoretical perspective is that it provides us with an elegant framework for the model theory of types, and it will allow us to discuss concepts of inductive-coinductive types from an abstract perspective. But more is true: Categories and type theories complement each other. The former can serve also as deductive systems, while allowing us to sweep many technicalities of syntactic type theories, like variable binding, under the rug. However, it is much easier to give and check proofs in syntactic theories, because category theory makes every application of structural rules like weakening explicit. That category theory is so explicit leads to an enormous amount of bookkeeping, which is necessary to construct even simple terms in the language of category theory. This is contrary to syntactic theories, where we leave weakening steps implicit and are thereby able to manage even complex proofs. We will not delve deeply in how the relation between category and type theory developed, but rather refer to the wonderful introduction by Lambek and Scott [LS88].

It is a well-known fact that the simply typed λ-calculus with finite product types, finite sum types and function types corresponds to Cartesian closed categories with finite coproducts, if we impose enough equations on the terms of the calculus, cf. Section 4.2. In fact, the simply typed λ-calculus

gives rise to an initial object in the category of Cartesian closed categories, as shown in [LS88, Chap. I.11]. So what is then the category theoretical counterpart to dependent type theory?

When we introduced the dependent product and sum types, we got our intuition for their defining properties from set-families. In the preliminaries (Section 2.4) it is shown how set-families can be organised into a fibration by letting $\mathrm{Fam}(\mathbf{Set})$ be the category of families and $P : \mathrm{Fam}(\mathbf{Set}) \to \mathbf{Set}$ be the functor that maps a family $\{X_i\}_{i\in I}$ to the index set I. This fibration is the prototypical example for a model of dependent type theory. But what are the integral features that a fibration must have to interpret such a theory? The answer to this question is best found by organising a dependent type theory itself into a fibration.

So we wish construct a fibration $T\colon \mathbf{T} \to \mathbf{C}$ from a given dependent type theory, the *classifying fibration* of that theory. The base category $\mathbf{C}$ of that fibration has contexts as objects and substitutions as morphisms. By a substitution $\sigma\colon \Gamma_1 \to \Gamma_2$ we mean here an n-tuple $(t_1, \ldots, t_n)$ of terms, where $\Gamma_2 = x_1 : A_1, \ldots, x_n : A_n$ and $\Gamma_1 \vdash t_i : A_i[t_1/x_1, \ldots, t_{i-1}/x_{i-1}]$. Such substitutions can be composed and there is an identity substitution, both of which are given by extending the corresponding notions from classifying categories in Section 4.2.1. The total category $\mathbf{T}$ of the fibration consists of types and terms in contexts. More precisely, the objects of $\mathbf{T}$ are judgements $\Gamma \vdash A : \mathbf{Type}$ and the morphisms $\Gamma_1 \vdash A : \mathbf{Type} \to \Gamma_2 \vdash B : \mathbf{Type}$ are pairs (σ, t), where σ is a substitution $\Gamma_1 \to \Gamma_2$ and t is a term with $\Gamma_1, x : A \vdash t : B[\sigma]$. The fibration is then given by the obvious projection functor $T\colon \mathbf{T} \to \mathbf{C}$ with $P(\Gamma \vdash A : \mathbf{Type}) = \Gamma$. Note that a fibre $\mathbf{T}_\Gamma$ of T over a context Γ consists of types in that context and the morphisms are pairs (id_Γ, t), where id_Γ is the identity substitution and $\Gamma, x : A \vdash t : B$. Using this fact, we can easily describe the Cartesian lifting of a substitution. Since the fibration P will be cloven (and even split), it suffices to define for a substitution $\sigma\colon \Gamma_1 \to \Gamma_2$ the reindexing functor $\sigma^*\colon \mathbf{T}_{\Gamma_2} \to \mathbf{T}_{\Gamma_1}$. As one might expect, this functor is given by applying the substitution: $\sigma^*(\Gamma_2 \vdash A : \mathbf{Type}) = \Gamma_2 \vdash A[\sigma] : \mathbf{Type}$ and $\sigma^*(\mathrm{id}_{\Gamma_2}, t) = (\mathrm{id}_{\Gamma_1}, t[\sigma])$. One can easily check that T is a fibration with these definitions, provided that the dependent type theory satisfies some conditions like a substitution lemma, cf. Section 4.2.1.

Let us now connect dependent types to set-families. The idea is essentially that a dependent type can be seen as a family of types that is indexed by other types. More precisely, a context $\Gamma \in \mathbf{C}$ corresponds to an index set $I \in \mathbf{Set}$ and a type A in that context to a set family X indexed by I. The exact interpretation of contexts as index sets and types as set families depends of course on the concrete theory at hand. But as an illustration, let A be a type in an empty context and B be a type with $x : A \vdash B : \mathbf{Type}$. Then the context $x : A$ is an element of $\mathbf{C}$ and the type B is in $\mathbf{T}_{x:A}$. Now suppose that we (inductively) interpreted the context $x : A$ as a set I and the type B as a family $X \in \mathbf{Set}^I$, where $\mathbf{Set}^I$ is the fibre of the set-family fibration over I. In the type theory we can then form the sum type over B with $\emptyset \vdash \Sigma x : A.\, B : \mathbf{Type}$, which is thus an element of $\mathbf{T}_\emptyset$. On the side of set families, we can form the singleton family $\{\coprod_{i\in I} X_i\}_{\star\in\mathbf{1}}$ that consists just of the coproduct of all sets in the family X and is an element of $\mathbf{Set}^{\mathbf{1}}$. This example illustrates how contexts correspond to index sets, in particular the empty context to the singleton set, and how type families give rise to set families.

As a further example, we return now to the types that we cooked up to model calendar days. To this end, we need the family $\{\mathrm{Pos}_n\}_{n\in\mathbb{N}}$ with $\mathrm{Pos}_n = \{k \in \mathbb{N} \mid 1 \le k \le n\}$, which is an object in $\mathbf{Set}^{\mathbb{N}}$. Earlier, we formed the type of correct dates as a sum type by using a type of bounded positive numbers. We can take now Pos as the corresponding family. So the question is: How we can properly represent sum types as operations on (fibres of) the set-family fibration? Recall that

the date type consists of two sums, where the inner one was taken over a type that corresponds to the family

$$D = \{\mathrm{Pos}_{\mathrm{days}\ y\ m}\}_{(y,m)\in\mathbb{N}\times\mathrm{Pos}_{12}}.$$

We can now form the coproduct over the index for the month to obtain the first sum:

$$\mathrm{DM} = \left\{\coprod_{m\in\mathrm{Pos}_{12}} D_{(y,m)}\right\}_{y\in\mathbb{N}}.$$

Finally, the date type corresponds to the following family, where we write $\{X\}_{\mathbf{1}}$ instead of $\{X\}_{\star\in\mathbf{1}}$.

$$\left\{\coprod_{y\in\mathbb{N}} \mathrm{DM}_y\right\}_{\mathbf{1}}$$

Since we are using the terminology "coproduct" here, we need to explain what the appropriate universal property is. This will allow us to relate the rules of the type theory to those of the set-fibration.

In essence, the universal property of the coproducts above will correspond to the term formation rules of sum types, as we demonstrate now. So let A and $B[x]$ be a types, where $x : A$ occurs freely in $B[x]$. We can see $B[x]$ as a family of types indexed by A, and therefore consider A and $B[x]$ analogous to an index set I and a set-family X in $\mathbf{Set}^I$, respectively. Recall now that sum types came with a pairing of type $x : A, y : B[x] \vdash \langle x, y\rangle : \Sigma x : A.\, B[x]$. Note that we can use weakening to view the type $\Sigma x : A.\, B[x]$ as a family with $x : A \vdash \Sigma x : A.\, B[x] : \mathbf{Type}$. By analogy, the singleton family $\{\coprod_{i\in I} X_i\}_{\mathbf{1}}$ gives rise to the I-indexed family $\{\coprod_{i\in I} X_i\}_{j\in I}$, in which the index j is not used. Formally, this weakening is achieved by considering the unique map $!\colon I \to \mathbf{1}$ from I to the singleton set. This map gives rise to the reindexing functor, see Section 2.4,

$$!^*\colon \mathbf{Set}^{\mathbf{1}} \to \mathbf{Set}^I$$

by $!^*(Y) = \{Y_\star\}_{j\in I}$, where $\star$ denotes the only element in $\mathbf{1}$. With this notation for weakening at hand, we can now see the pairing for the coproduct as a map

$$\eta_X\colon X \to !^*\Big(\coprod_{i\in I} X_i\Big)$$

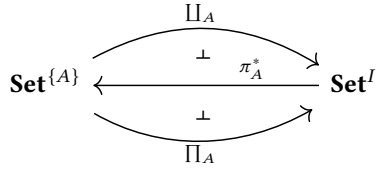
in $\mathbf{Set}^I$ by putting $\eta_X = \{\eta_{X,i}\}_{i\in I}$ with $\eta_{X,i}(x) = (i, x)$. A reader familiar with adjoint functors recognises probably already the notation for the unit of an adjunction lurking in here, so let us make the guess that the coproduct is left adjoint to the weakening functor $!^*$. Indeed, let us write $\coprod_I X$ for the coproduct $\coprod_{i\in I} X_i$. We then have the following adjoint correspondence between maps on indexed sets.

$$\frac{f\colon \quad \coprod_I X \longrightarrow Y \qquad \mathbf{Set}^{\mathbf{1}}}{g\colon \qquad X \longrightarrow\ !^* Y \quad \mathbf{Set}^I}$$

From top to bottom, we have that $f = \{f_\star\colon \coprod_I X \to Y\}$ gives the family $g = \{g_i\colon X_i \to Y_\star\}_{i\in I}$ by putting $g_i(x) = f(i, x)$. Going upwards, we define f from g by putting $f_\star(i, x) = g_i(x)$. This latter definition corresponds to what we achieved with unpacking for sum types, in the sense that $f_\star(s)$ corresponds to **unpack** s **as** $\langle i, x\rangle$ **in** $g_i(x)$. The universal property of the coproduct of set-families is

now given by the adjunction $\coprod_I \dashv !^*$, which, as we have seen, captures precisely the term formation rules for sum types.[51]

So far, we have explained the sum type of date types with only one free variable. But what about set-families like D that have multiple free variables? To tackle this case, we first need to discuss how free variables are introduced in the first place. Type theoretically, one usually has a context formation rule that, given a valid context Γ and a type A in that context, allows us to expand the context Γ to a new context $\Gamma, x : A$. Since a context Γ corresponds to an index set I and a type A in this context to a family X indexed by I, we thus need an operation that turns a set-family into a set. In other words, we want to have a functor $\{-\}\colon \mathrm{Fam}(\mathbf{Set}) \to \mathbf{Set}$, which is usually called *comprehension*. Recall that we saw X as an object (I, X) in the category $\mathrm{Fam}(\mathbf{Set})$ of set-families. Then we can define the comprehension functor by $\{(I, X)\} = \coprod_{i\in I} X_i$. Note that if X is a constant family, that is, if there is a set A with $X_i = A$ for all $i \in I$, then $\{(I, X)\} \cong I \times A$. We can now define a map $\pi_X\colon \{X\} \to I$ by putting $\pi_X(i, x) = i$. Reindexing along this map corresponds to weakening of types. For example, if I is a set, then we can form the singleton family $\{I\}_{\star\in\mathbf{1}}$, from which we obtain $\{\{I\}_{\mathbf{1}}\} \cong \mathbf{1} \times I \cong I$. The projection $\pi_{\{I\}_{\mathbf{1}}}\colon \{\{I\}_{\mathbf{1}}\} \to \mathbf{1}$ is then nothing but the unique map $!\colon I \to \mathbf{1}$ that we used earlier to express the universal property of the coproduct $\coprod_I X$. Indeed, given a family $Y \in \mathbf{Set}^{\mathbf{1}}$, we have that $\pi^*(Y) = \{Y_\star\}_{j\in I}$, which is the set-theoretic analogue to weakening of a type X without dependencies, in that we introduce a fresh variable j of type I into the (empty) context of Y. Therefore, we also have that the coproduct functor $\coprod_I$ is left-adjoint to π^*. More generally, coproduct and products for set-families are captured by the following adjoint situation, in which $A \in \mathbf{Set}^I$ and $\pi_A\colon \{A\} \to I$.



Let us now see by the holiday example how the comprehension functor allows the introduction of fresh variables into context. Recall that we considered the family D, where $D_{(y,m)}$ contains exactly those positive numbers that correspond to the days in the month m of the year y. Comprehension now allows us to view the predicate isHoliday as a set-family in $\mathbf{Set}^{\{D\}}$: Define

$$\mathrm{isHoliday}_{((y,m),d)} \coloneqq \{\text{set of proofs that d.m.y is a holiday we acknowledge}\}$$

for $(y, m) \in \mathbb{N} \times \mathrm{Pos}_{12}$ and $d \in D_{(y,m)}$, or, if we are only interested in provability and not the proofs themselves, we can use the non-constructive definition

$$\mathrm{isHoliday}_{((y,m),d)} \coloneqq \begin{cases} \mathbf{1}, & \text{d.m.y is a holiday we acknowledge} \\ \emptyset, & \text{otherwise} \end{cases}.$$

To construct the set-family that corresponds to the type of holidays we need to go through a small complication. This complication should be embraced though, as it shows how category theory makes the use of weakening explicit. We noted earlier that the comprehension of a constant family is isomorphic to a product, thus for the map $!_{\mathbb{N}}\colon \mathbb{N} \to \mathbf{1}$ we obtain $\{!_{\mathbb{N}}^*(\mathrm{Pos}_{12})\} \cong \mathbb{N} \times \mathrm{Pos}_{12}$. In other

Dependent Type Theory	Fibrations
Context Γ	Object I in base category $\mathbf{B}$
Type in context Γ	Object X in fibre $\mathbf{P}_I$ of fibration $P\colon \mathbf{E} \to \mathbf{B}$ over I
Context expansion	Comprehension $\{-\}\colon \mathbf{E} \to \mathbf{B}$
Weakening	Reindexing along projection $\pi_X\colon \{X\} \to PX$
Dependent Product/function space Π	Product with $\pi_X^* \dashv \prod_X$
Dependent sum Σ	Coproduct with $\coprod_X \dashv \pi_X^*$
Equality type (see Example 6.2.12)	Equality $\mathrm{Eq}\colon \mathbf{P}_I \to \mathbf{P}_{I\times I}$, left adjoint to contraction δ^* with $\delta\colon I \to I \times I$

Table 6.1.: Comparison of Notions from Dependent Type Theory and Fibrations

words, the index of D is equivalently given by $\{!^*_{\mathbb{N}}(\mathrm{Pos}_{12})\}$. This allows us to use the description of coproducts as left-adjoint functors to form the (singleton) family Holiday by the following repeated coproduct.

$$\mathrm{Holiday} \coloneqq \coprod_{\mathbb{N}} \coprod_{!^*_{\mathbb{N}}(\mathrm{Pos}_{12})} \coprod_{D} \mathrm{isHoliday}$$

In this definition, the coproducts are, from right to left, left-adjoint to the reindexing functors coming from $\pi_D\colon \{D\} \to \{!^*_{\mathbb{N}}(\mathrm{Pos}_{12})\}$, $\pi_{!^*_{\mathbb{N}}(\mathrm{Pos}_{12})}\colon \{!^*_{\mathbb{N}}(\mathrm{Pos}_{12})\} \to \mathbb{N}$, and $\pi_{\mathbb{N}}\colon \mathbb{N} \to \mathbf{1}$, respectively. This formation of the family Holiday illustrates the bookkeeping that is introduced when encoding type families that use weakening.

Of course, one has to appropriately axiomatise the notion of comprehension, but the basic idea is that it is a functor from the total into the base category of a fibration that comes with projection morphisms like the ones we used above. Further technical details are deferred to Section 6.2.4. Let us just mention that there are other notions than fibrations with comprehension that can be utilised to model dependent type theory: display map categories [Tay99], categories with attributes [Car78; and Dyb95; Mog89; Pit01], categories with families [Dyb95], and contextual categories [Car86; KL16; Str91]. These notions do not differ very much from each other though, see [Jac93] and [nLa17]. Finally, to keep track of the correspondence between type theoretic and category theoretical concepts, the reader might find Table 6.1 useful.

This ends our tour through the three different worlds of dependent type theory. Other views, further explanation and more references concerning dependent type theory may be found in, among other places, [GTL89; Hof97; Jac99; Mar75b; NGdV94; NG14; and Pie04].

6.2. Categorical Dependent Recursive Types

In this section, we give a category theoretical description of dependent inductive-coinductive types. The word "type" is somewhat loosely used here in the sense that we do not mean by "type" a syntactic label, rather we mean an entity (here: an object in a category) that determines the values (morphisms from a final object, see [LS88]) for variables of that type and the behaviour of maps (morphisms) on it, cf. [PSW76]. This allows us to construct a type theory that corresponds to these entities. For example, for objects A and B, the maps into $A \times B$ are completely determined by maps on A and B through the projections and pairing together with the universal mapping property. This gives rise to product types with the appropriate term constructors, cf. Section 3.1. That being said,

we will identify in Definition 6.2.9 an inductively generated class of objects and functors that model so-called *strictly positive types*. In other words, we effectively give a syntactic description of objects that we study here as dependent inductive-coinductive types.

To motivate the formal definition of dependent types, we will first study in Section 6.2.1 the prototypical example of length-indexed lists, which we have already seen in the introductory guide. From this example, we extract in two steps the relevant ingredients of dependent inductive-coinductive types. First, we show in Section 6.2.2 that such types are given as initial (respectively final) dialgebras, in which the functor that determines the codomain (resp. the domain) of these dialgebras is of very special shape. This is where we generalise the categorical data types of Hagino [Hag87] to dependent types. The second step is to identify domain functors for inductive types and codomain functors for coinductive types. This leads us to the definition of *recursive dependent type complete categories (μP-complete categories)* in Section 6.2.3, which allow the interpretation of strictly positive, dependent, inductive-coinductive types.

One may wonder why we restrict ourselves to strictly positive types. The problem is not so much the description of more general types. In fact, conceiving initial or final dialgebras that fulfil the restrictions in Section 6.2.2 as types is perfectly fine. However, we can of course not guarantee the existence of inductive (resp. coinductive) types for arbitrary domain (resp. codomain) functors. So we restrict to a reasonable closure condition that will allow us to construct the arising types from simpler structures in Section 6.3.

We will finish this section by fusing μP-completeness with comprehension categories into *recursive dependent type closed categories (μPCC)*, in which we can construct strictly positive types that may depend on previously constructed types. For example, we are able to construct the types that we met in the introductory guide: the type of natural numbers Nat, the type $\mathrm{List}_A\, n$ of lists of length n for n : Nat, and the type $\mathrm{Pos}\, n$ of positive numbers bounded by n. Other examples that can be constructed in μPCCs include bisimilarity and the substream relation for streams. Thus, μPCCs provide a framework for general dependent inductive-coinductive types.

6.2.1. Introductory Example

Before diving into the formal definition of dependent recursive types, we start with an example, which allows us to first gain some intuition and from which we will extract the essence of dependent recursive types. The type we examine are lists indexed by their length, frequently also called *vectors*.

Example 6.2.1. For a given set A, we denote by A^n the n-fold product of A. *Length-indexed lists*[52] or *vectors* over A are given by the set-family $\mathrm{List}\, A = \{A^n\}_{n\in\mathbb{N}}$, which is an object in the category $\mathbf{Set}^{\mathbb{N}}$ of families indexed by natural numbers (Section 2.4).

Vectors come with two constructors: $\mathrm{nil}_\star \colon \mathbf{1} \to A^0$ for the empty vector, and $\mathrm{cons}_n \colon A \times A^n \to A^{n+1}$ for prefixing of vectors with elements of A. We note that $\mathrm{nil} \colon \{\mathbf{1}\}_{\mathbf{1}} \to \{A^0\}_{\mathbf{1}}$ is a morphism in the category $\mathbf{Set}^{\mathbf{1}}$ of families indexed by the one-element set $\mathbf{1}$, whereas $\mathrm{cons} = \{\mathrm{cons}_n\}$ is a morphism $\{A \times A^n\}_{n\in\mathbb{N}} \to \{A^{n+1}\}_{n\in\mathbb{N}}$ in $\mathbf{Set}^{\mathbb{N}}$. Let $F, G \colon \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$ be functors into the product of the categories $\mathbf{Set}^{\mathbf{1}}$ and $\mathbf{Set}^{\mathbb{N}}$ given by

$$F(X) = (\{\mathbf{1}\}_{\mathbf{1}}, \{A \times X_n\}_{n\in\mathbb{N}}) \qquad G(X) = (\{X_0\}_{\mathbf{1}}, \{X_{n+1}\}_{n\in\mathbb{N}}).$$

Using these functors, we see that $(\mathrm{nil}, \mathrm{cons}) \colon F(\mathrm{List}\, A) \to G(\mathrm{List}\, A)$ is an (F, G)-dialgebra. In fact, $(\mathrm{nil}, \mathrm{cons})$ is the *initial* (F, G)-dialgebra, see Section 2.5. ◂

At this stage, the reader might be wondering why we are using initial dialgebras rather than initial algebras to describe the inductive data type in Example 6.2.1. There are two reasons for this. Firstly, as we will see in Definition 6.2.9, using dialgebras allows us to give a straightforward definition of (strictly positive) dependent types. Such a definition is also possible for algebras, cf. Theorem 6.3.1, but much more complicated to carry out. The second reason for using dialgebras is that these allow us the representation of dependent sum and dependent product types as initial and, respectively, final dialgebras. This view will lead to a minimal type theory in Chapter 7.

In Example 6.2.1, we have presented vectors as an initial dialgebra for certain functors. As it turns out, the functor that describes the codomain of the constructors has a special form.

Example 6.2.2. First, we note that the fibre of Fam(**Set**) above I is isomorphic to $\mathbf{Set}^I$. Let then $z\colon \mathbf{1} \to \mathbb{N}$ and $s\colon \mathbb{N} \to \mathbb{N}$ be defined by $z(*) = 0$ and $s(n) = n + 1$, giving us reindexing functors $z^*\colon \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}}$ and $s^*\colon \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$. By their definition, $z^*(X) = \{X_0\}_{\mathbf{1}}$ and $s^*(X) = \{X_{n+1}\}_{n\in\mathbb{N}}$, hence the functor G, we used to describe vectors as dialgebra, is $G = \langle z^*, s^*\rangle$. In Sec. 6.2.2, we address the structure of F.

6.2.2. Signatures and Recursive Types

The idea that underlies the categorical data types of Hagino [Hag87] is that the outcome of a constructor for a type must be of that type itself. More concretely, a (non-dependent) inductive type T with n constructors is given by a declaration like that on the left in Figure 6.1. Hagino then

data T **where**	**data** $\Gamma \vdash$ T' **where**
$c_1 : F_1(\mathrm{T}) \to \mathrm{T}$	$\Gamma_1 \vdash c_1' : F_1'(\mathrm{T'}) \to \mathrm{T'}[\sigma_1]$
$\vdots$	$\vdots$
$c_n : F_n(\mathrm{T}) \to \mathrm{T}$	$\Gamma_1 \vdash c_n' : F_n'(\mathrm{T'}) \to \mathrm{T'}[\sigma_n]$

Figure 6.1.: Declarations of Non-Dependent and Dependent Inductive Types

notices that T corresponds to an initial (F, G)-dialgebra in a category $\mathbf{C}$ with $F, G\colon \mathbf{C} \to \prod_{k=1}^{n} \mathbf{C}$, $F = \langle F_1, \ldots, F_n\rangle$ and $G = \langle \mathrm{Id}, \ldots, \mathrm{Id}\rangle$. The dialgebra structure on T is then given by the tuple $(c_1, \ldots, c_n)$, which is a morphism in the category $\prod_{k=1}^{n} \mathbf{C}$. We now extend this idea to dependent types. Suppose that T' is a dependent type as in Figure 6.1 with variables in context Γ. Each constructor c_k' of T' may live in a different local context Γ_k, thus we have to apply a substitution σ_k to the type T' to bring it into the local context, see the codomain of the constructors c_k' in Figure 6.1. We encountered this already in the example in Section 6.2.1, as the type of vectors over A corresponds to the following type declaration.

$$\begin{aligned} &\textbf{data } n : \mathrm{Nat} \vdash \mathrm{List}\, A\, n \textbf{ where} \\ &\qquad \vdash \mathrm{nil} : \qquad \mathbf{1} \to \mathrm{List}\, A\, 0 \\ &k : \mathrm{Nat} \vdash \mathrm{cons} : \mathrm{List}\, A\, k \to \mathrm{List}\, A\, (k+1) \end{aligned}$$

Assume that we are given a fibration $P\colon \mathbf{E} \to \mathbf{B}$ and that there is an index $I \in \mathbf{B}$ that corresponds to the context Γ. To view T' as an object in the fibre $\mathbf{P}_I$, we assume that each Γ_k corresponds to an object $J_k \in \mathbf{B}$ as well. Given this setup, we note that c_k' must be a morphism in $\mathbf{P}_{J_k}$, hence $F_k'(T')$

must also be an object in the same fibre. This leads us for each $1 \leq k \leq n$ to the assumption that F'_k is a functor $\mathbf{P}_I \to \mathbf{P}_{J_k}$. Similarly, each σ_k is a substitution that assigns to each variable in the context Γ a term that lives in context Γ_k. Therefore, we also assume that there is a morphism $u_k \colon J_k \to I$ in the base $\mathbf{B}$, which in turn gives rise to the reindexing functors $u_k^* \colon \mathbf{P}_I \to \mathbf{P}_{J_k}$. These assumptions allow us to view each constructor c'_k as a morphism $F'_k(T') \to u_k^*(T')$ in $\mathbf{P}_{J_k}$ or, equivalently, as morphism $(c'_1, \ldots, c'_n)$ in $\prod_{k=1}^n \mathbf{P}_{J_k}$. Putting this all together, we will conceive $(c'_1, \ldots, c'_n)$ as an initial dialgebra on T' for the functors F' and G_u with $F' = \langle F'_1, \ldots, F'_n \rangle$ and $G_u = \langle u_1^*, \ldots, u_n^* \rangle$. This development is summed in the following definition of a signature and the definition of a recursive type below in Definition 6.2.7.

Definition 6.2.3. Let $P \colon \mathbf{E} \to \mathbf{B}$ be a fibration. A *(dependent) recursive-type signature* with parameters in a category $\mathbf{C}$, is a pair (F, u) consisting of

- a functor $F \colon \mathbf{C} \times \mathbf{P}_I \to \mathbf{D}$, where $\mathbf{D} = \prod_{k=1}^n \mathbf{P}_{J_k}$ is a product of categories for some $n \in \mathbb{N}$, objects $J_k \in \mathbf{B}$ with $k = 1, \ldots, n$ and $I \in \mathbf{B}$, and
- a family u of n morphisms in $\mathbf{B}$ with $u_k \colon J_k \to I$ for $k = 1, \ldots, n$.

A family u as above induces a functor $G_u \colon \mathbf{P}_I \to \mathbf{D}$ given by $G_u \coloneqq \langle u_1^*, \ldots, u_n^* \rangle$. ◂

Signatures form the basis of the types we will introduce in Definition 6.2.7. Before we get to them, let us give the official signature for the type of vectors.

Example 6.2.4. In Example 6.2.2, we have seen that the codomain of the constructors for vectors are given by a functor G with $G = \langle z^*, s^* \rangle$. According to Definition 6.2.3, we have $G = G_u$ with $u = (z, s)$, thus the signature of vectors is (F, u), with the functor $F \colon \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$, as given in Example 6.2.1.

Note that the parameter category $\mathbf{C}$ is trivial ($\mathbf{C} = \mathbf{1}$) in this case. To make the set A, over which we considered vectors, a parameter we can use instead the functor $F' \colon \mathbf{Set} \times \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$ given by $F'(Y, X) = (\{\mathbf{1}\}_{\mathbf{1}}, \{Y \times X_n\}_{n \in \mathbb{N}})$. Note that we then have $F'(A, -) = F$ for any set A. This gives us a signature (F', u) with parameters in $\mathbf{Set}$. ◂

We showed in Example 6.2.1 that vectors are an initial (F, G_u)-dialgebra. This is an instance of what we will call an inductive type for a signature. In Example 6.2.4, we added the set over which we consider vectors as a parameter to the signature. So to define types for arbitrary signatures (with parameters), we need to be able to deal with parameterised dialgebras, which are dialgebras that take further parameters. More technically, these are dialgebras in functor categories for a lifting of the functors F and G_u that we will define now. This is analogous to, for example, [Kim10].

Definition 6.2.5. Suppose that we are given a signature (F, u), where $F \colon \mathbf{C} \times \mathbf{P}_I \to \mathbf{D}$. For each $V \in \mathbf{C}$ we define a functor $F(V, -) \colon \mathbf{P}_I \to \mathbf{D}$ by putting $F(V, -)(X) \coloneqq F(V, X)$. Let us, moreover, assume that for all V an initial $(F(V, -), G_u)$-dialgebra $\alpha_V \colon F(V, \Phi_V) \to G_u(\Phi_V)$ and final $(G_u, F(V, -))$-dialgebra $\xi_V \colon G_u(\Omega_V) \to F(V, \Omega_V)$ exists, where $G_u = \langle u_1^*, \ldots, u_n^* \rangle$.

Given these assumptions, we define functors on functor categories with fixed domain $\mathbf{C}$:

$$\widehat{F}, \widehat{G_u} : [\mathbf{C}, \mathbf{P}_I] \to [\mathbf{C}, \mathbf{D}] \qquad \widehat{F}(H) \coloneqq F \circ \langle \mathrm{Id}_{\mathbf{C}}, H \rangle \qquad \widehat{G_u}(H) \coloneqq G_u \circ H. \tag{6.3}$$

We use the dialgebras Φ_V and Φ_V to build functors $\mu(\widehat{F},\widehat{G_u})\colon \mathbf{C}\to\mathbf{P}_I$ and $\nu(\widehat{G_u},\widehat{F})\colon \mathbf{C}\to\mathbf{P}_I$ by

$$\begin{aligned}
\mu(\widehat{F},\widehat{G_u})(V) &\coloneqq \Phi_V & \mu(\widehat{F},\widehat{G_u})(f\colon V\to W) &\coloneqq (\alpha_W\circ F(f,\mathrm{id}_{\Phi_W}))^{-}\\
\nu(\widehat{G_u},\widehat{F})(V) &\coloneqq \Omega_V & \nu(\widehat{G_u},\widehat{F})(f\colon V\to W) &\coloneqq (F(f,\mathrm{id}_{\Omega_V})\circ\xi_V)^{\sim},
\end{aligned}$$

where the bar and tilde superscripts denote the inductive and coinductive extensions, that is, the unique homomorphism given by initiality and finality, respectively. ◂

To understand the definition of the functors $\mu(\widehat{F},\widehat{G_u})$ and $\nu(\widehat{G_u},\widehat{F})$, it helps to visualise their definition as homomorphism diagrams:

$$\begin{array}{ccc}
F(V,\Theta_V) & \xrightarrow{F\left(V,\mu\left(\widehat{F},\widehat{G_u}\right)\right)(f)} & F(V,\Theta_W)\\
\Big\downarrow{\scriptstyle \alpha_V} & & \Big\downarrow{\scriptstyle F\left(f,\mathrm{id}_{\Phi_W}\right)}\\
 & & F(W,\Theta_W)\\
 & & \Big\downarrow{\scriptstyle \alpha_W}\\
G_u(\Theta_V) & \xrightarrow{\mu\left(\widehat{F},\widehat{G_u}\right)(f)} & G_u(\Theta_W)
\end{array}
\qquad\qquad
\begin{array}{ccc}
G_u(\Omega_V) & \xrightarrow{\nu\left(\widehat{G_u},\widehat{F}\right)(f)} & G_u(\Omega_W)\\
\Big\downarrow{\scriptstyle \xi_V} & & \\
F(V,\Omega_V) & & \Big\downarrow{\scriptstyle \xi_W}\\
\Big\downarrow{\scriptstyle F\left(f,\mathrm{id}_{\Omega_V}\right)} & & \\
F(W,\Omega_V) & \xrightarrow{F\left(W,\nu\left(\widehat{G_u},\widehat{F}\right)\right)(f)} & F(W,\Omega_W)
\end{array}$$

Now note that $\widehat{F}\big(\mu(\widehat{F},\widehat{G_u})\big)(V)=F(V,\Phi_V)$ and $\widehat{G_u}\big(\mu(\widehat{F},\widehat{G_u})\big)(V)=G_u(\Phi_V)$, so that the α given in in Definition 6.2.5 is a family of morphisms, which is moreover natural by uniqueness of inductive extensions. Since the same also holds for ξ, we obtain the following lemma.

Lemma 6.2.6. *The morphisms α_V and ξ_V in Definition 6.2.5 form natural transformations*

$$\alpha\colon \widehat{F}\big(\mu(\widehat{F},\widehat{G_u})\big)\Rightarrow\widehat{G_u}\big(\mu(\widehat{F},\widehat{G_u})\big) \qquad \textit{and} \qquad \xi\colon \widehat{G_u}\big(\nu(\widehat{G_u},\widehat{F})\big)\Rightarrow\widehat{F}\big(\nu(\widehat{G_u},\widehat{F})\big). \tag{6.4}$$

Moreover, α and ξ are, respectively, initial and final dialgebras in the functor category $[\mathbf{C},\mathbf{D}]$.

Having set up the notations for parameterised initial and final dialgebras, we can finally define what (dependent) recursive types are.

Definition 6.2.7. Let (F,u) be a recursive-type signature with parameters in $\mathbf{C}$ and $F\colon \mathbf{C}\times\mathbf{P}_I\to\mathbf{D}$. An *inductive type* for (F,u) is an initial $(\widehat{F},\widehat{G_u})$-dialgebra with carrier $\mu(\widehat{F},\widehat{G_u})$. Dually, a *coinductive type* for (F,u) is a final $(\widehat{G_u},\widehat{F})$-dialgebra, note the order, with the carrier being denoted by $\nu(\widehat{G_u},\widehat{F})$ Both carriers are functors with

$$\mu(\widehat{F},\widehat{G_u})\colon \mathbf{C}\to\mathbf{P}_I \quad \text{ and } \quad \nu(\widehat{G_u},\widehat{F})\colon \mathbf{C}\to\mathbf{P}_I.$$

If $\mathbf{C}=\mathbf{1}$, then we view these carriers as objects in $\mathbf{P}_I$ and drop the hats from the notation. ◂

We have seen in the introductory guide that dependent coproducts and products play a central role in dependent type theory. Moreover, we have claimed that these would arise as inductive and coinductive types, respectively. The following example makes this statement precise.

Example 6.2.8. Recall from Section 2.4 that a fibration $P\colon \mathbf{E} \to \mathbf{B}$ is said to have dependent coproducts and products along a morphism $f\colon I \to J$ in $\mathbf{B}$, if there are functors $\coprod_f$ and $\prod_f$ from $\mathbf{P}_I$ to $\mathbf{P}_J$ that are, respectively, left and right adjoint to f^*. For each $X \in \mathbf{P}_I$, we can define a signature, such that $\coprod_f(X)$ and $\prod_f(X)$ arise as recursive types for these signatures, as follows. Define the constant functor by

$$K_X\colon \mathbf{P}_J \to \mathbf{P}_I \qquad K_X(Y) = X \qquad K_X(g) = \mathrm{id}_X.$$

With this definition, (K_X, f) is the signature for coproducts and products. For example, the unit η of the adjunction $\coprod_f \dashv f^*$ is the initial (K_X, f^*)-dialgebra $\eta_X\colon K_X(\coprod_f(X)) \to f^*(\coprod_f(X))$, using that $K_X(\coprod_f(X)) = X$.

To actually get a functor $\coprod_f\colon \mathbf{P}_I \to \mathbf{P}_J$ we need to move to a parameterised signature. The observation we use is that the projection functor $\pi_1\colon \mathbf{P}_I \times \mathbf{P}_J \to \mathbf{P}_I$ is a "parameterised" constant functor, since we have $K^J_A = \pi_1(A, -)$. If we are given $f\colon I \to J$ in $\mathbf{B}$, then we use the signature (π_1, f), and define $\coprod_f \coloneqq \mu(\widehat{\pi_1}, \widehat{f^*})$ and $\prod_f \coloneqq \nu(\widehat{f^*}, \widehat{\pi_1})$. We claim now that we indeed get adjunctions $\coprod_f \dashv f^* \dashv \prod_f$. This will be proven in Thm. 6.2.11. For now, let us just mention that, for example, the unit $\eta\colon \mathrm{Id} \Rightarrow f^* \coprod_f$ is given by the initial dialgebra on $\mu(\widehat{\pi_1}, \widehat{f^*})$. ◀

6.2.3. Recursive-Type Complete Categories

We now define a class of signatures and functors that should be seen as category theoretical language for, what is usually called, strictly positive types [AM09], positive generalised abstract data types [HF11] or descriptions [Cha+10; DM13]. Note, however, that none of these treat coinductive types. A *non-dependent* version of strictly positive types that include coinductive types are given in [AAG05].

Let us first introduce some notation. Given categories $\mathbf{C}_1$ and $\mathbf{C}_2$ and an object $A \in \mathbf{C}_1$, we denote by $K^{\mathbf{C}_1}_A\colon \mathbf{C}_1 \to \mathbf{C}_2$ the functor mapping constantly to A. The projections on product categories are denoted, as usual, by $\pi_k\colon \mathbf{C}_1 \times \mathbf{C}_2 \to \mathbf{C}_k$. Using these notations, we can define what we understand to be a recursive type.

Definition 6.2.9. Let $P\colon \mathbf{E} \to \mathbf{B}$ be a cloven fibration, $\mathcal{S}$ a class of signatures in P and $\mathcal{D}$ a class of functors. We denote by $\mathcal{S}_{\mathbf{C}\to\mathbf{D}} \subseteq \mathcal{S}$ the class of all signatures $(F, u) \in \mathcal{S}$ such that $F\colon \mathbf{C} \to \mathbf{D}$. Similarly, $\mathcal{D}_{\mathbf{C}\to\mathbf{D}} \subseteq \mathcal{D}$ is the class of all functors $F \in \mathcal{D}$ with $F\colon \mathbf{C} \to \mathbf{D}$.

We say that P is a *recursive-type complete category* (*μP-complete category*) if there is a class $\mathcal{S}$ of signatures in P and a class $\mathcal{D}$ of functors subject to the following closure rules. The class $\mathcal{S}$ must be closed under formation of signatures over $\mathcal{D}$ with arbitrary reindexing morphisms in $\mathbf{B}$:

$$\frac{\mathbf{D} = \prod_{i=1}^n \mathbf{P}_{J_i} \qquad F \in \mathcal{D}_{\mathbf{C}\times\mathbf{P}_I\to\mathbf{D}} \qquad u = (u_1\colon J_1 \to I, \ldots, u_n\colon J_n \to I) \text{ morphisms in } \mathbf{B}}{(F, u) \in \mathcal{S}_{\mathbf{C}\times\mathbf{P}_I\to\mathbf{D}}}$$

The class $\mathcal{D}$ must contain base types and must be closed under pairing, composition and the formation

of recursive types from signatures in $\mathcal{S}$:

$$\frac{A \in \mathbf{P}_J}{K_A^{\mathbf{P}_I} \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_J}} \qquad \frac{\mathbf{C} = \prod_{i=1}^n \mathbf{P}_{I_i}}{\pi_k \in \mathcal{D}_{\mathbf{C} \to \mathbf{P}_{I_k}}} \qquad \frac{f \colon J \to I \text{ in } \mathbf{B}}{f^* \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_J}}$$

$$\frac{F_1 \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_K} \qquad F_2 \in \mathcal{D}_{\mathbf{P}_K \to \mathbf{P}_J}}{F_2 \circ F_1 \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_J}} \qquad \frac{F_i \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_{J_i}} \qquad i = 1, 2}{\langle F_1, F_2 \rangle \in \mathcal{D}_{\mathbf{P}_I \to \mathbf{P}_{J_1} \times \mathbf{P}_{J_2}}}$$

$$\frac{(F, u) \in \mathcal{S}_{\mathbf{C} \times \mathbf{P}_I \to \mathbf{D}}}{\mu(\widehat{F}, \widehat{G_u}) \in \mathcal{D}_{\mathbf{C} \to \mathbf{P}_I}} \qquad \frac{(F, u) \in \mathcal{S}_{\mathbf{C} \times \mathbf{P}_I \to \mathbf{D}}}{\nu(\widehat{G_u}, \widehat{F}) \in \mathcal{D}_{\mathbf{C} \to \mathbf{P}_I}}$$

If $\mathcal{S}$ and $\mathcal{D}$ are the minimal classes that are closed under these rules, then we call $\mathcal{S}$ the class of *strictly positive signatures* and $\mathcal{D}$ the class of *strictly positive types.* ◀

A brief note about terminology: The name μP-complete is an adaption of the λ-cube terminology, where the calculus with just dependent types is called λP, see [Bar91; NG14]. The letter "P" stands in this context for "predicate" and signifies the involvement of dependent types. With the Greek letter μ, we intend to refer to recursive types and not just inductive types. The reader should think here of general μ-recursion as a mnemonic, similar to the naming of μ-bicomplete categories [San02b].

The following assumption reflects that $\mathcal{D}$ contains exactly the strictly positive types we will consider in Chapter 7.

Assumption 6.2.10. We assume, without loss of generality, that for a μP-complete category, the classes $\mathcal{S}$ and $\mathcal{D}$ are exactly its strictly positive signatures and types.

One feature of recursive-type complete categories is that basic types, like fibrewise and dependent products and coproducts, arise naturally as inductive and coinductive types. This is an instance of the following, more general, result.

Theorem 6.2.11. *Suppose $P \colon \mathbf{E} \to \mathbf{B}$ is a μP-complete category. Let $\mathbf{C} = \prod_{i=1}^n \mathbf{P}_{J_i}$ and $\pi_1 \colon \mathbf{C} \times \mathbf{P}_I \to \mathbf{C}$ be the first projection. If (π_1, u) is a signature such $u_k \colon J_i \to I$, then we have for $G_u = \langle u_1^*, \ldots, u_n^* \rangle$ the following adjoint situation:*

$$\mu(\widehat{\pi_1}, \widehat{G_u}) \dashv G_u \dashv \nu(\widehat{G_u}, \widehat{\pi_1}).$$

Proof. We only show how the adjoint transposes are obtained in the case of inductive types. Concretely, for a tuple $V \in \mathbf{C}$ and an object $A \in \mathbf{P}_I$, we need to prove the correspondence

$$\frac{f \colon \quad \mu(\widehat{\pi_1}, \widehat{G_u})(V) \longrightarrow A \qquad \text{in } \mathbf{P}_I}{g \colon \qquad\qquad V \longrightarrow G_u A \qquad \text{in } \mathbf{C}}$$

Let us use the notation $H = \mu(\widehat{\pi_1}, \widehat{G_u})$, then the choice of π_1 implies that the initial $(\widehat{\pi_1}, \widehat{G_u})$-dialgebra is of type $\alpha \colon \mathrm{Id}_{\mathbf{C}} \Rightarrow G_u \circ H$, since $\widehat{\pi_1}(H) = \pi_1 \circ \langle \mathrm{Id}_{\mathbf{C}}, H \rangle = \mathrm{Id}_{\mathbf{C}}$ and $\widehat{G_u}(H) = G_u \circ H$. This allows us to use as transpose of f the morphism $V \xrightarrow{\alpha_V} G_u(H(V)) \xrightarrow{G_u f} G_u A$. As transpose of g, we use the inductive extension of $\widehat{\pi_1}(K_A^{\mathbf{C}})(V) = V \xrightarrow{g} G_u A = \widehat{G_u}(K_A^{\mathbf{C}})(V)$. The proof that this correspondence is natural and bijective follows straightforwardly from initiality. For coinductive types, the result is given by duality. □

From this result we can recover the usual binary, fibrewise products and coproducts, and the dependent products and coproducts as shown in the following table.

Construction	Parameter $\mathbf{C}$	Reindexing u	Definition
Binary Coproduct $+_I \colon \mathbf{P}_I \times \mathbf{P}_I \to \mathbf{P}_I$	$\mathbf{P}_I$	$(\mathrm{id}_I, \mathrm{id}_I)$	$+_I = \mu(\widehat{\pi_1}, \widehat{G_u})$
Binary Product $\times_I \colon \mathbf{P}_I \times \mathbf{P}_I \to \mathbf{P}_I$	$\mathbf{P}_I$	$(\mathrm{id}_I, \mathrm{id}_I)$	$\times_I = \nu(\widehat{G_u}, \widehat{\pi_1})$
Dependent Coproduct $\coprod_f \colon \mathbf{P}_I \to \mathbf{P}_J$	$\mathbf{1}$	$(f \colon I \to J)$	$\coprod_f = \mu(\widehat{\pi_1}, \widehat{G_u})$
Dependent Product $\coprod_f \colon \mathbf{P}_I \to \mathbf{P}_J$	$\mathbf{1}$	$(f \colon I \to J)$	$\prod_f = \nu(\widehat{G_u}, \widehat{\pi_1})$

Many more examples, like initial and final objects etc., can be obtained in a μP-complete category, but we defer their construction to the examples in Chapter 7 to reduce repetition. Let us give two examples that become relevant in Section 6.4 though.

Example 6.2.12 (Propositional Equality)**.** There are several definable notions of equality, provided that $\mathbf{B}$ has binary products. A generic one is *propositional equality* $\mathrm{Eq}_I \colon \mathbf{P}_I \to \mathbf{P}_{I\times I}$, the left adjoint to the so-called *contraction* functor $\delta^* \colon \mathbf{P}_{I\times I} \to \mathbf{P}_I$, where δ is the diagonal $\delta \colon I \to I \times I$. Thus, propositional equality can be obtained from the table above as the dependent coproduct along δ: $\mathrm{Eq}_I \coloneqq \coprod_\delta$. Since Eq_I is a left adjoint, it comes for each $X \in \mathbf{P}_{I\times I}$ with a constructor $\mathrm{refl}_X \colon X \to \delta^*(\mathrm{Eq}_I(X))$.

Let us clarify this definition by instantiating it over set families. For $Y \in \mathbf{Set}^{I\times I}$ and $i \in I$, we have $\delta^*(Y)_i = Y_{\delta(i)} = Y_{(i,i)}$. The equality type on $X \in \mathbf{Set}^I$ and $(i,j) \in I \times I$ is given on the other hand by

$$\mathrm{Eq}_I(X)_{(i,j)} = \coprod_{\substack{k\in I \\ \delta(k)=(i,j)}} X_k \cong \begin{cases} X_i, & i = j \\ \emptyset, & i \neq j \end{cases}.$$

The constructor refl_X is given for each $i \in I$ as a map $X_i \to \mathrm{Eq}_I(X)_{(i,i)} \cong X_i$, and is essentially the identity map. It should be noted that one is usually more interested in the instance $\mathrm{Eq}_I(\mathbf{1}_I)$, where $\mathbf{1}_I$ in the final object in $\mathbf{P}_I$. For this reason, we define

$$\mathrm{Eq}(I) \coloneqq \mathrm{Eq}_I(\mathbf{1}_I). \tag{6.5}$$

In Fam($\mathbf{Set}$), one obtains thus

$$\mathrm{Eq}(I) = \mathrm{Eq}_I(\mathbf{1}_I) \cong \begin{cases} \mathbf{1}, & i = j \\ \emptyset, & i \neq j \end{cases},$$

which is the definition of equality on I that might be more familiar. ◂

Example 6.2.13 (Bisimilarity for Streams)**.** Assume that there is an object A^ω in $\mathbf{B}$ of streams over A, together with projections $\mathrm{hd} \colon A^\omega \to A$ and $\mathrm{tl} \colon A^\omega \to A^\omega$, giving us the head and tail of a stream. We can define bisimilarity between streams as coinductive type for the signature

$$F, G_u \colon \mathbf{P}_{A^\omega \times A^\omega} \to \mathbf{P}_{A^\omega \times A^\omega} \times \mathbf{P}_{A^\omega \times A^\omega}$$
$$F = \left\langle (\mathrm{hd} \times \mathrm{hd})^* \circ K_{\mathrm{Eq}(A)}, (\mathrm{tl} \times \mathrm{tl})^* \right\rangle \qquad \text{and} \qquad u = (\mathrm{id}_{A^\omega \times A^\omega}, \mathrm{id}_{A^\omega \times A^\omega}).$$

Let us put this definition of bisimilarity into a broader perspective. There is a category $\mathrm{Rel}(\mathbf{E})$ of binary relations in $\mathbf{E}$ by forming the pullback of P along $\Delta\colon \mathbf{B} \to \mathbf{B}$ with $\Delta(I) = I \times I$, see [HJ97]:

$$\begin{array}{ccc} \mathrm{Rel}(\mathbf{E}) & \longrightarrow & \mathbf{E} \\ {\scriptstyle \mathrm{Rel}(P)}\downarrow & \lrcorner & \downarrow{\scriptstyle P} \\ \mathbf{B} & \xrightarrow{\Delta} & \mathbf{B} \end{array}$$

Since P is cloven, also the resulting fibration $\mathrm{Rel}(P)$ is cloven ([Jac99, Lem. 1.5.1]). Thus, we obtain reindexing functors between the fibres of $\mathrm{Rel}(\mathbf{E})$, which we will denote by $(-)^{\#}$. Note that the fibres of $\mathrm{Rel}(\mathbf{E})$ can be presented by the isomorphism $\mathrm{Rel}(\mathbf{E})_X \cong \mathbf{P}_{X\times X}$, so that reindexing in $\mathrm{Rel}(P)$ is effectively given by $f^{\#} = (f \times f)^*$. We can now reinterpret F and G_u in $\mathrm{Rel}(\mathbf{E})$ by

$$\begin{aligned} &F, G_u\colon \mathrm{Rel}(\mathbf{E})_{A^\omega} \to \mathrm{Rel}(\mathbf{E})_{A^\omega} \times \mathrm{Rel}(\mathbf{E})_{A^\omega} \\ &F = \langle \mathrm{hd}^{\#} \circ K_{\mathrm{Eq}(A)}, \mathrm{tl}^{\#} \rangle \qquad \text{and} \qquad G_u = \langle \mathrm{id}^{\#}_{A^\omega}, \mathrm{id}^{\#}_{A^\omega} \rangle. \end{aligned}$$

The final (G_u, F)-dialgebra on $\mathrm{Bisim}_A \in \mathrm{Rel}(\mathbf{E})_{A^\omega}$ is a pair of morphisms

$$\big(\mathrm{hd}^{\sim}_A\colon \mathrm{Bisim}_A \to \mathrm{hd}^{\#}(\mathrm{Eq}(A)), \mathrm{tl}^{\sim}_A\colon \mathrm{Bisim}_A \to \mathrm{tl}^{\#}(\mathrm{Bisim}_A)\big).$$

Bisim_A should be thought of to consist of all bisimilarity proofs. Coinductive extensions yield the usual coinduction proof principle, allowing us to prove bisimilarity by establishing a bisimulation relation $R \in \mathrm{Rel}(\mathbf{E})_{A^\omega}$ together with $h\colon R \to \mathrm{hd}^{\#}(\mathrm{Eq}(A))$ and $t\colon R \to \mathrm{tl}^{\#}(R)$. This says, as usual, that the heads of streams related by R are equal and that the tails of related streams are again related. Later, in Example 6.4.13, we will see how bisimilarity arises canonically for coinductive types. ◀

Let us now come to an example of a truly dependent, inductive-coinductive type. This type will represent the substream relation that we encountered before.

Example 6.2.14. Recall that we have defined in Example 5.1.13 the substream relation that relates two streams s and t, if all the elements in s occur in order in t. We denoted this relation symbolically by $s \leq t$. In Section 5.1.3, we found out that this relation could be expressed by selecting elements from the stream t through an inductive-coinductive type. From this, we inferred that $\leq$ is an inductive-coinductive relation. The purpose of this example is to make this somewhat more precise by defining $\leq$ directly as a recursive type. In Section 7.5.3 we will relate both definitions.

To give an intuitive description of the direct definition of the substream relation, let us use for a moment the language of the copattern calculus $\lambda\mu\nu{=}$. Let s and t thus be of type A^ω. We consider s to be substream of t, written $s \leq t$, if we can obtain a stream t' by *finitely* often applying .tl to t, such that

1. $s.\mathrm{hd} = t'.\mathrm{hd}$, and
2. $s.\mathrm{tl} \leq t'.\mathrm{tl}$.

Figure 6.2 should give an idea of how this definition works for streams of natural numbers.

Let us now derive from this description a definition of the substream relation in a μP-complete category. As a first towards this goal consider the following mutually recursive declaration of the dependent types $s \leq t$ and $s \leq_\mu t$.

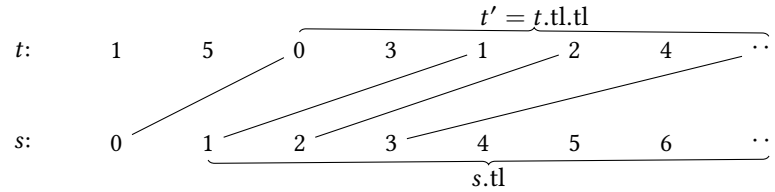


Figure 6.2.: Operational View on Substream Relation

codata $\mathrm{s},\mathrm{t} : A^\omega \vdash \mathrm{s} \leq \mathrm{t}$ **where**
  out : $\mathrm{s} \leq \mathrm{t} \to \mathrm{s} \leq_\mu \mathrm{t}$

data $\mathrm{s},\mathrm{t} : A^\omega \vdash \mathrm{s} \leq_\mu \mathrm{t}$ **where**
  match : $(\mathrm{s.hd} = \mathrm{t.hd}) \times (\mathrm{s.tl} \leq \mathrm{t.tl}) \to \mathrm{s} \leq_\mu \mathrm{t}$
  skip : $\mathrm{s} \leq_\mu \mathrm{t.tl} \to \mathrm{s} \leq_\mu \mathrm{t}$

The first type is coinductive, which is signified by the keyword **codata**, whereas the second is an inductive type. Of course, the first type should represent the substream relation. But what is the role of the type $s \leq_\mu t$? Note that in the description of the substream relation above we said that for $s \leq t$ to hold there must be a stream t' that is obtained by finitely often applying the tail destructor to t. These *skipping* steps can be exactly achieved by using the "skip" constructor of $s \leq_\mu t$ and, since this is an inductive type, we eventually need to use the "match" constructor. This constructor, in turn, requires us to show that the heads of s and t match and that the tail of s is a substream of the tail of t'. Hence, "match" formalises exactly the two conditions we posed above on t'.

In Figure 6.3, we give a representation of a proof tree for the fact that the constant stream 1^ω is a substream of the alternating bit stream alt. Since the tail of alt starts with a 1, we use for clarity the notation $1 : \mathrm{alt}$ instead of alt.tl in the tree. Moreover, the dashed line shows how the proof tree is build through the coinductive type $1^\omega \leq \mathrm{alt}$. We will give a formal definition of this proof by using finality of the substream relation.

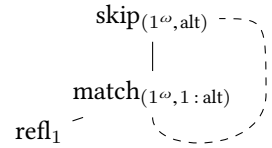


Figure 6.3.: Conceptual Representation of a Proof of $1^\omega \leq \mathrm{alt}$

After developing an intuition for how the substream relation arises as an inductive-coinductive type, it is now our job to construct this relation by using the machinery of μP-complete categories. This is done in three steps: First, we define a functor $S'_\mu : \mathrm{Rel}(\mathbf{E})_{A^\omega} \to \mathrm{Rel}(\mathbf{E})_{A^\omega}$, such that $S'_\mu(R)$ corresponds to $\leq_\mu$ but with all occurrences of $\leq$ in its declaration replaced by R. Second, we define $S \in \mathrm{Rel}(\mathbf{E})_{A^\omega}$, which corresponds to $\leq$. Finally, the type $\leq_\mu$ itself is interpreted as $S'_\mu(S)$. Since $S'_\mu : \mathrm{Rel}(\mathbf{E})_{A^\omega} \to \mathrm{Rel}(\mathbf{E})_{A^\omega}$ should be an inductive type, we define $S'_\mu = \mu\big(\widehat{F}, \widehat{G_u}\big)$ with

$$F : \mathrm{Rel}(\mathbf{E})_{A^\omega} \times \mathrm{Rel}(\mathbf{E})_{A^\omega} \to \mathrm{Rel}(\mathbf{E})_{A^\omega} \times \mathrm{Rel}(\mathbf{E})_{A^\omega}$$
$$F(S,R) = \big(\mathrm{hd}^\#(\mathrm{Eq}(A)) \times \mathrm{tl}^\#(S), (\mathrm{id} \times \mathrm{tl})^*(R)\big)$$

and $u = (\mathrm{id}_{A^\omega}, \mathrm{id}_{A^\omega})$. On top of this type, we can now define the substream relation $S \in \mathrm{Rel}(\mathbf{E})_{A^\omega}$ by $S = \nu\big(G_{\mathrm{id}_{A^\omega}}, S'_\mu\big)$. Notice how these definitions reflect exactly the type declarations above because with $S_\mu = S'_\mu(S)$ we obtain the following operations, since S and S_μ are final and initial dialgebras.

$$\begin{aligned} \mathrm{out}&\colon S \to S_\mu \\ \mathrm{match}&\colon \mathrm{hd}^\#(\mathrm{Eq}(A)) \times \mathrm{tl}^\#(S) \to S_\mu \\ \mathrm{skip}&\colon (\mathrm{id} \times \mathrm{tl})^*(S_\mu) \to S_\mu \end{aligned}$$

Coming back to the example that 1^ω is a substream of alt, we can give in $\mathrm{Rel}(\mathrm{Fam}(\mathbf{Set}))_{A^\omega}$ a formal description of the proof tree in Figure 6.3. To do so, recall that $\mathrm{Rel}(\mathrm{Fam}(\mathbf{Set}))_{A^\omega} \cong \mathbf{Set}^{A^\omega \times A^\omega}$, which allows us to define relations easily as set families. Thus, let us define $R \in \mathbf{Set}^{A^\omega \times A^\omega}$ to be

$$R_{(\sigma,\tau)} = \begin{cases} \mathbf{1}, & \sigma = 1^\omega \text{ and } \tau = \mathrm{alt} \\ \emptyset, & \text{otherwise} \end{cases}.$$

We can then define morphism $c\colon R \to S'_\mu(R)$ in $\mathbf{Set}^{A^\omega \times A^\omega}$ by

$$c_{(1^\omega,\mathrm{alt})}(\star) = \mathrm{skip}_{(1^\omega,\mathrm{alt})}\big(\mathrm{match}_{(1^\omega,1:\mathrm{alt})}(\mathrm{refl}_1(\star), \star)\big).$$

Since $\mathrm{id}^\#(R) \cong \mathrm{Id}(R) = R$, we have that c is a $\big(G_{\mathrm{id}}, S'_\mu\big)$-dialgebra and can thus be extended to a homomorphism $\widetilde{c}\colon R \to S$. By applying this map, we obtain the tree in Figure 6.3 as the element $\widetilde{c}_{(1^\omega,\mathrm{alt})}(\star)$ in S.

Summing up, we have shown how to construct the substream relation in any μP-complete category. We show that this direct definition is equivalent to the one given by stream filtering in Section 5.1.3. ◀

Up to this point, all coinductive types that we encountered only used the identity substitution to determine the domain of their destructors. In the last example of this section we study a dependent coinductive type that uses a non-trivial substitution instead. The point of this example is to illustrate how non-trivial substitutions allow us to block the application of destructors.[53] [54]

Example 6.2.15. A partial stream is a stream together with a, possibly infinite, depth up to which it is defined. Assume that there is an object $\mathbb{N}^\infty$ of natural numbers extended with infinity and a successor map $s_\infty\colon \mathbb{N}^\infty \to \mathbb{N}^\infty$ in $\mathbf{B}$, we will see how these can be defined in Example 6.2.19. The partial streams correspond to the following type declaration, the idea of which is that the head and tail of partial streams are defined only on those partial streams that are defined in, at least, the first position.

codata n : $\mathbb{N}^\infty$ ⊢ PStr A n **where**
  k : $\mathbb{N}^\infty$ ⊢ hd k : PStr A (s_∞ k) → A
  k : $\mathbb{N}^\infty$ ⊢ tl k : PStr A (s_∞ k) → PStr A k

As set of partial functions, we can define partial streams as a family indexed by $n \in \mathbb{N}^\infty$:

$$\mathrm{PStr}(A)_n = \{s\colon \mathbb{N} \rightharpoonup A \mid \forall k < n.\, k \in \mathrm{dom}\, s \wedge \forall k \geq n.\, k \notin \mathrm{dom}\, s\},$$

where the order on $\mathbb{N}^\infty$ is given by extending that of the natural numbers with ∞ as strict top element, that is, such that $k < \infty$ for all $k \in \mathbb{N}$.

The interpretation of $\mathrm{PStr}(A)$ for $A \in \mathbf{P_1}$ in a μP-complete category is given as the carrier of the final (G_u, F)-dialgebra with

$$G_u, F \colon \mathbf{P}_{\mathbb{N}^\infty} \to \mathbf{P}_{\mathbb{N}^\infty} \times \mathbf{P}_{\mathbb{N}^\infty} \qquad G_u = \left\langle s_\infty^*, s_\infty^* \right\rangle \qquad F = \left\langle K_{A'}^{\mathbb{N}^\infty}, \mathrm{Id} \right\rangle,$$

where $A' = !_{\mathbb{N}^\infty}^*(A) \in \mathbf{P}_{\mathbb{N}^\infty}$ is the weakening of A with $!_{\mathbb{N}^\infty} \colon \mathbb{N}^\infty \to \mathbf{1}$. On set families, partial streams are given by the dialgebra $\xi = (\mathrm{hd}, \mathrm{tl})$ with $\mathrm{hd}_k \colon \mathrm{PStr}(A)_{(s_\infty\, k)} \to A$ and $\mathrm{tl}_k \colon \mathrm{PStr}(A)_{(s_\infty\, k)} \to \mathrm{PStr}(A)_k$ for every $k \in \mathbb{N}^\infty$.

We can make this construction functorial in A, using the same "trick" as for sums and products. To this end, we define the functor $H \colon \mathbf{P_1} \times \mathbf{P}_{\mathbb{N}^\infty} \to \mathbf{P}_{\mathbb{N}^\infty} \times \mathbf{P}_{\mathbb{N}^\infty}$ with $H = \left\langle !_{\mathbb{N}^\infty}^* \circ \pi_1, \pi_2 \right\rangle$, where π_1 and π_2 are corresponding projection functors, so that $H(A, X) = F(X)$. This gives, by recursive-type completeness, rise to a functor $\nu(\widehat{G_u}, \widehat{F}) \colon \mathbf{P}_{\mathbb{N}^\infty} \to \mathbf{P}_{\mathbb{N}^\infty}$, which we denote by PStr, together with a pair $(\mathrm{hd}, \mathrm{tl})$ of natural transformations. □

6.2.4. Recursive-Type Closed Categories

We have seen in the introduction that one of central notions in a (dependent) type theory is context extension. It turned out that a possible category theoretical counterpart was to have a comprehension functor. The following definition gives a precise description of the necessary ingredients that a fibration with comprehension needs to have, so that dependently typed contexts can be interpreted over that fibration. This definition can be found, for example, in [Jac99, Lem. 1.8.8, Def. 10.4.7] and [FGJ11].

Definition 6.2.16. Let $P \colon \mathbf{E} \to \mathbf{B}$ be a fibration. If each fibre $\mathbf{P}_I$ has a final object $\mathbf{1}_I$ and these are preserved by reindexing, then there is a fibred *final object functor* $\mathbf{1}_{(-)} \colon \mathbf{B} \to \mathbf{E}$. (Note that then $P(\mathbf{1}_I) = I$.) P is a *comprehension category with unit* (*CCU*), if $\mathbf{1}_{(-)}$ has a right adjoint $\{-\} \colon \mathbf{E} \to \mathbf{B}$, called *comprehension*. This gives rise to a functor $\mathcal{P} \colon \mathbf{E} \to \mathbf{B}^\to$ into the arrow category over $\mathbf{B}$, by mapping $\mathcal{P}(A) = P(\varepsilon_A) \colon \{A\} \to P(A)$ and $\mathcal{P}(f) = (\{f\}, Pf)$, where $\varepsilon \colon \mathbf{1}_{\{-\}} \Rightarrow \mathrm{Id}$ is the counit of $\mathbf{1}_{(-)} \dashv \{-\}$. We often denote $\mathcal{P}(A)$ by π_A and call it the *projection* of A. Finally, P is said to be a *full CCU*, if $\mathcal{P}$ is a fully faithful functor.

Note that, in a μP-complete category, we can define final objects in each fibre, though the preservation of them by reindexing needs to be required separately, see Section 6.5.

Let us record for later use what the fullness conditions for a CCU means explicitly. Recall that a morphism between projections π_A and π_B in the arrow category $\mathbf{B}^\to$ is given by a commuting diagram of the following form.

$$\begin{array}{ccc} \{A\} & \xrightarrow{f} & \{B\} \\ {\scriptstyle \pi_A}\downarrow & & \downarrow{\scriptstyle \pi_B} \\ PA & \xrightarrow{g} & PB \end{array}$$

That $\mathcal{P}$ is a fully faithful functor means now that there is a unique $h \colon A \to B$ in $\mathbf{E}$ with $\mathcal{P}(h) = (f, g)$. By definition of $\mathcal{P}$, we thus have $f = \{h\}$ and $g = Ph$.

The following example describes comprehension for the set-family fibration, which he have encountered already in the introduction.

Example 6.2.17. In Fam(**Set**), the final object functor is given by $\mathbf{1}_I = (I, \{\mathbf{1}\}_{i\in I})$, where $\mathbf{1}$ is the singleton set. Comprehension is defined to be $\{(I,X)\} = \coprod_{i\in I} X_i$ and the projections π_I map then an element of $\coprod_{i\in I} X_i$ to its component i in I.

Using comprehension, we can give a general account to dependent recursive types.

Definition 6.2.18. We say that a fibration $P\colon \mathbf{E} \to \mathbf{B}$ is a *recursive-type closed category* (μPCC), if it is a CCU, has a terminal object in $\mathbf{B}$ and is μP-complete.

As already mentioned, the purpose of introducing comprehension is that it allows us to use recursive types defined in $\mathbf{E}$ again as index. The terminal object in $\mathbf{B}$ is used to introduce recursive types without dependencies, like natural numbers. Let us reiterate on Ex. 6.2.15 to illustrate this.

Example 6.2.19. Recall that we assumed the existence of extended naturals $\mathbb{N}^\infty$ and the successor map s_∞ on them to define partial streams. We are now in the position to define in a recursive-type closed category everything from scratch as follows.

Having defined $+ : \mathbf{P_1} \times \mathbf{P_1} \to \mathbf{P_1}$, see Thm. 6.2.11, we put $\mathbb{N}^\infty \coloneqq \nu(\mathrm{Id}, \mathbf{1} + \mathrm{Id})$ with its final dialgebra structure being the predecessor pred. The successor map $s_\infty\colon \mathbb{N}^\infty \to \mathbb{N}^\infty$ can then be defined as the unique homomorphism in the following diagram.

$$\begin{array}{ccc} \mathbb{N}^\infty & \xrightarrow{s_\infty} & \mathbb{N}^\infty \\ \downarrow{\scriptstyle \kappa_2} & & \downarrow{\scriptstyle \mathrm{pred}} \\ \mathbf{1} + \mathbb{N}^\infty & \xrightarrow{\mathrm{id}+s_\infty} & \mathbb{N}^\infty \end{array}$$

Partial streams $\mathrm{PStr}\colon \mathbf{P}_{\{\mathbb{N}^\infty\}} \to \mathbf{P}_{\{\mathbb{N}^\infty\}}$ are then given as in Ex. 6.2.15 by the final $(\widehat{G_u}, \widehat{F})$-dialgebra, only this time we need to apply the comprehension functor to s_∞ to obtain a map in the base category $\mathbf{B}$. Thus, we put $u = (\{s_\infty\}, \{s_\infty\})$ and $F = \left\langle !^*_{\mathbb{N}^\infty} \circ \pi_1, \pi_2 \right\rangle$. ◂

6.3. Constructing Recursive Types as Polynomials

In Section 6.2, we have given conditions on fibrations that allow us to use them as dependent type theory based only on inductive and coinductive types. The important notion there was that of μP-completeness (Definition 6.2.9). This raises of course the question whether there are any examples of μP-complete categories. We will answer this question affirmatively by showing that there is a class of categories, called Martin-Löf categories, that admit an interpretation of all necessary recursive types. This works by reducing all recursive types to polynomial functors [GK13] (also called container [AAG05]). An example of this class is then the codomain fibration over the category of sets.

The construction proceeds in two steps. First, we reduce in Theorem 6.3.1 the dialgebras arising from signatures to algebras and coalgebras, respectively. Second, we construct the necessary initial algebras and final coalgebras as fixed points of polynomial functors, analogous to the construction of strictly positive types in [AAG05]. This second step is again an assembly of results. We prove in Theorem 6.3.10 that fixed points of dependent polynomial functors with parameters are again polynomial functors. This allows us to construct nested recursive types, and is the significant result of this section. Next, we show that dependent recursive types can be constructed from non-dependent

recursive types. For coinductive types the corresponding result is given in Theorem 6.3.6, whereas the analogous result for inductive types is already in [AAG05] and [GK13]. Finally, the construction of non-dependent, coinductive types can be reduced to that of inductive types, which is again proved by Abbott et al. [AAG05].

We begin by showing that inductive and coinductive types correspond to certain initial algebras and final coalgebras, respectively.

Theorem 6.3.1. *Let $P\colon \mathbf{E} \to \mathbf{B}$ be a fibration with fibrewise coproducts and dependent sums. If (F, u) with $F\colon \mathbf{P}_I \to \mathbf{P}_{J_1} \times \cdots \times \mathbf{P}_{J_n}$ is a signature, then there is an isomorphism*

$$\mathrm{DiAlg}(F, G_u) \cong \mathrm{Alg}\left(\coprod_{u_1} \circ F_1 +_I \cdots +_I \coprod_{u_n} \circ F_n\right)$$

where $F_k = \pi_k \circ F$ is the kth component of F. In particular, existence of inductive types and initial algebras coincide. Dually, if P has fibrewise and dependent products, then

$$\mathrm{DiAlg}(G_u, F) \cong \mathrm{CoAlg}\left(\prod_{u_1} \circ F_1 \times_I \cdots \times_I \prod_{u_n} \circ F_n\right).$$

In particular, existence of coinductive types and final coalgebras coincide.

Proof. The first result is given by a simple application of the adjunctions $\coprod_{k=1}^n \dashv \Delta_n$ between the (fibrewise) coproduct and the diagonal, and $\coprod_{u_k} \dashv u_k^*$:

$$\begin{array}{rl} FX \longrightarrow G_u X & (\text{in } \mathbf{P}_{J_1} \times \cdots \times \mathbf{P}_{J_n}) \\ \hline\hline (\coprod_{u_1}(F_1 X), \ldots, \coprod_{u_n}(F_n X)) \longrightarrow \Delta_n X & (\text{in } \mathbf{P}_I^n) \\ \hline\hline \coprod_{k=1}^n \coprod_{u_k}(F_k X) \longrightarrow X & (\text{in } \mathbf{P}_I) \end{array}$$

That homomorphisms are preserved follows at once from naturality of the used Hom-set isomorphisms. The correspondence for coinductive types follows by duality. □

To be able to reuse existing work, we work in the following with the codomain fibration $\mathrm{cod}\colon \mathbf{B}^{\to} \to \mathbf{B}$ for a category $\mathbf{B}$ with pullbacks. Moreover, we assume that $\mathbf{B}$ is finitely complete and locally Cartesian closed. The following definition allows to connect this to our setup of fibrations.

Definition 6.3.2 ([Jac99, Def. 10.5.3])**.** Let $P\colon \mathbf{E} \to \mathbf{B}$ be a comprehension category with coproducts, that is, for every $A \in \mathbf{E}$ with projection $\pi_A\colon \{A\} \to PA$ there is an adjunction $\coprod_A \dashv \pi_A^*$ that satisfies the Beck-Chevalley condition, see Definition 2.4.4. We denote the unit of these coproducts by $\eta\colon \mathrm{Id} \Rightarrow \pi_A^* \coprod_A$, and we let $\overline{\pi_A}\,(\coprod_A X)$ be the Cartesian lifting of π_A as in the following diagram.

$$\begin{array}{ccc} \pi_A^* \coprod_A X & \xrightarrow{\overline{\pi_A}\,(\coprod_A X)} & \coprod_A X \\ & \top & \\ & \Big\downarrow P & \\ \{A\} & \xrightarrow{\pi_A} & PA = P(\coprod_A X) \end{array}$$

The fibration P has *strong coproducts*, if the dependent pairing $\kappa\colon \{X\} \to \{\coprod_I X\}$, given by

$$\kappa = \left\{\overline{\pi_A}\left(\coprod_A X\right) \circ \eta_X\right\},$$

is an isomorphism. We call P a *closed comprehension category* (*CCompC*), if it is a full CCU with products, strong coproducts and has a final object in the base category $\mathbf{B}$. ◀

By [Jac99, Thm 10.5.5], a finitely complete category $\mathbf{B}$ is locally Cartesian closed if and only if the codomain fibration is $\mathrm{cod}\colon \mathbf{B}^{\to} \to \mathbf{B}$ is a closed comprehension category. Another assumption that we need is that the binary coproducts in $\mathbf{B}$ are disjoint, which means that the injections are monomorphisms and form the following pullback square.

$$\begin{array}{ccc} \mathbf{0} & \longrightarrow & J \\ \downarrow & \lrcorner & \downarrow{\scriptstyle \kappa_2} \\ I & \xrightarrow{\kappa_1} & I + J \end{array}$$

Under the assumption that coproducts are disjoint, we obtain an equivalence $\mathbf{B}/_{I+J} \simeq \mathbf{B}/_I \times \mathbf{B}/_J$, see [Jac99, Prop. 1.5.4].

Assumption 6.3.3. In this section we assume that $\mathbf{B}$ is a finitely complete, locally Cartesian closed category with disjoint coproducts.

Definition 6.3.4. A *dependent polynomial* Γ indexed by I on variables indexed by J is given by a triple of morphisms

$$J \xleftarrow{s} B \xrightarrow{f} A \xrightarrow{t} I$$

If $J = I = \mathbf{1}$, f is said to be a *(non-dependent) polynomial*. The *extension of a polynomial* Γ is given by the composite

$$[\![\Gamma]\!] = \mathbf{B}/_J \xrightarrow{s^*} \mathbf{B}/_B \xrightarrow{\Pi_f} \mathbf{B}/_A \xrightarrow{\amalg_t} \mathbf{B}/_I,$$

which we denote by $[\![f]\!]$ if Γ is non-dependent. A functor $F\colon \mathbf{B}/_J \to \mathbf{B}/_I$ is a *dependent polynomial functor*, if there is a dependent polynomial Γ such that $F \cong [\![\Gamma]\!]$.

Note that polynomials are called *containers* by Abbott et al. [Abb03; AAG05], and a polynomial $\Gamma = 1 \xleftarrow{!} B \xrightarrow{f} A \xrightarrow{!} 1$ would be written as $A \triangleright f$ there. Similarly, dependent polynomials are a slight generalisation of *indexed containers* [AM09]. Container morphisms, however, are different from those of dependent polynomials, as the latter correspond to strong natural transformations [GK13, Prop. 2.9], whereas the former are in exact correspondence with all natural transformations between extensions [AAG05, Thm. 3.4]. Because of this relation, we will apply results for containers, which do not involve morphisms, to polynomials. Another difference in terminology between the theory of polynomial functors and containers is that initial algebras and final coalgebras for functors associated to containers are called and *W-types* and *M-types*, respectively. We will adopt this terminology in the following and denote for $f\colon A \to B$ by W_f the carrier of the initial algebra and by M_f the carrier of the final coalgebra for $[\![f]\!]$. In particular, we can use this to translate [AAG05, Prop. 4.1], giving us a construction of final coalgebras for polynomial functors from initial algebras for polynomial functors.

Assumption 6.3.5. We assume additionally that **B** is closed under the formation of W-types, thus is a *Martin-Löf category* in the terminology of Abbott et al. [AAG05]. By the above remark, **B** then also has all M-types.

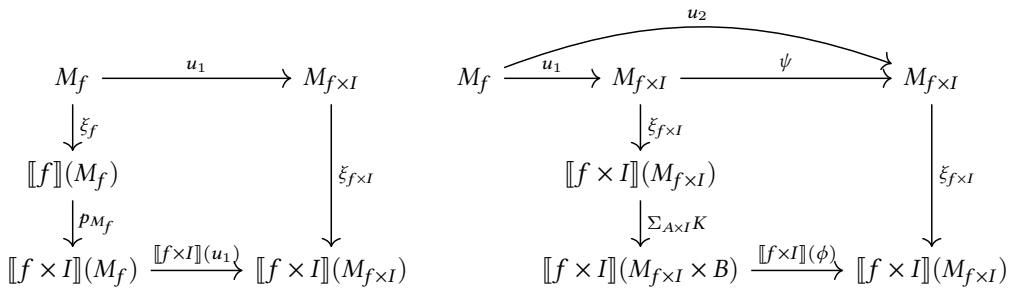
Analogously to how [GH04, Thm. 12] extends [MP00, Prop. 3.8], we extend here [vdBdM04, Thm 3.3]. That is to say, we show that final coalgebras for dependent polynomial functors can be constructed from final coalgebras of non-dependent polynomial functors. One reviewer of [Bas15a] had pointed out that this result can be derived from [vdBdM07, Thm 3.1], the published version of [vdBdM04]. However, we include a proof here, since it gives some insight into the handling of dependencies.

Theorem 6.3.6. *Dependent polynomial functors* $\mathbf{B}/I \to \mathbf{B}/I$ *have final coalgebras.*

Proof. Let $\Gamma = I \xleftarrow{s} B \xrightarrow{f} A \xrightarrow{t} I$ be a dependent polynomial. We construct, analogously to [GH04], the carrier of a final coalgebra V of $[\![\Gamma]\!]$ as an equaliser as in the following diagram, in which $f \times I$ is a shorthand for $B \times I \xrightarrow{f \times \mathrm{id}_I} A \times I$ and $M_{f\times I}$ is the carrier of the final $[\![f \times I]\!]$-coalgebra.

$$V \xrightarrow{g} M_f \underset{u_2}{\overset{u_1}{\rightrightarrows}} M_{f\times I} \tag{6.6}$$

First, we give u_1 and u_2, whose definitions are summarised in the following two diagrams.



These diagrams shall indicate that u_1 is given as coinductive extension and ψ as one-step definition, using that $M_{f\times I}$ is a final coalgebra, see Appendix B. The maps involved in the diagram are given as follows, which we sometimes spell out in the internal language of the codomain fibration, see for example [Abb03], as this is sometimes more readable.[55]

- The natural transformation $p\colon \Sigma_A \Pi_f \Rightarrow \Sigma_{A\times I}\Pi_{f\times I}$ sends (a, v) to $(a, t(a), v)$. It is given by the extension $[\![\alpha, \beta]\!]\colon [\![f]\!] \Rightarrow [\![f \times I]\!]$ of the morphism of polynomials [AAG05]

$$\begin{array}{ccc} B & \xrightarrow{f} & A \\ {\scriptstyle\beta}\downarrow & \lrcorner & \downarrow{\scriptstyle\alpha} \\ B\times I & \xrightarrow{f\times I} & A\times I \end{array}$$

  where $\alpha = \langle \mathrm{id}, t\rangle$ and $\beta = \langle \mathrm{id}, t \circ f\rangle$.

- The map $K\colon \Pi_{f\times I}(M_{f\times I}) \to \Pi_{f\times I}(M_{f\times I}\times B)$ is given as transpose of

$$\langle \varepsilon_{M_{f\times I}}, \pi_1 \circ \pi\rangle\colon\ (f\times I)^*(\Pi_{f\times I}(M_{f\times I})) \to M_{f\times I}\times B,$$

  where ε is the counit of the product (evaluation) and π is the context projection. In the internal language K is given by $K\ v = \lambda(b,i).(v\ (b,i),b)$.

- $\phi\colon M_{f\times I}\times B \to M_{f\times I}$ is constructed as coinductive extension as in the following diagram

$$\begin{array}{ccc} M_{f\times I}\times B & \xrightarrow{\quad\phi\quad} & M_{f\times I} \\ \downarrow{\scriptstyle \xi_{f\times I}\times \mathrm{id}} & & \\ [\![f\times I]\!](M_{f\times I})\times B & & \downarrow{\scriptstyle \xi_{f\times I}} \\ \downarrow{\scriptstyle e} & & \\ [\![f\times I]\!](M_{f\times I}\times B) & \xrightarrow{[\![f\times I]\!](\phi)} & [\![f\times I]\!](M_{f\times I}) \end{array}$$

  Here e is given by $e((a,i,v),b) = (a, s\ b, \lambda(b',s\ b).(v\ (b',i),b'))$, arising from the so-called Frobenius property, see [Jac99, Lem. 1.9.12].

We note that the elements of the subobject V of M_f, as given in (6.6), can be described by

$$\begin{aligned} &x : V_i \iff \\ &\qquad \xi_f\ x = (a : A, v : \Pi_f M_f),\ t\ a = i \text{ and } (\forall b : B.f\ b = a \Rightarrow v\ b : V_{s\ b}). \end{aligned}$$

The direction from left to right is given by simple a calculation, whereas the other direction can be proved by establishing a bisimulation between $u_1\ x$ and $u_2\ x$. We defer these details to Appendix B.

This characterisation of V allows us to prove that $\xi_f\colon M_f \to [\![f]\!](M_f)$ restricts to $\xi'\colon V \to [\![\Gamma]\!](V)$ and that ξ' is a final coalgebra. Hence, V is indeed the carrier of a final $[\![\Gamma]\!]$-coalgebra in $\mathbf{B}/I$. □

Before we continue, let us briefly illustrate the construction of final coalgebras for dependent polynomials from non-dependent polynomials in $\mathbf{Set}^{\to}$.

Example 6.3.7. Recall from Ex. 6.2.15 that partial streams are given by

codata n : $\mathbb{N}^\infty$ ⊢ PStr A n **where**
  k : $\mathbb{N}^\infty$ ⊢ hd k : PStr A (s_∞ k) → A
  k : $\mathbb{N}^\infty$ ⊢ tl k : PStr A (s_∞ k) → PStr A k

and that PStr is the final $(\widehat{G_u}, \widehat{H})$-dialgebra, where we have that $H\colon \mathbf{Set}^{\mathbf{1}}\times\mathbf{Set}^{\mathbb{N}^\infty} \to \mathbf{Set}^{\mathbb{N}^\infty}\times\mathbf{Set}^{\mathbb{N}^\infty}$ with $H = \left\langle !^*_{\mathbb{N}^\infty}\circ\pi_1, \pi_2\right\rangle$ and $u = (s_\infty, s_\infty)$. By Thm. 6.3.1, we can construct PStr as the final coalgebra of $F\colon \mathbf{Set}/\mathbf{1}\times\mathbf{Set}/\mathbb{N}^\infty \to \mathbf{Set}/\mathbb{N}^\infty$ with $F(A,X) = \prod_{s_\infty} !^*_{\mathbb{N}^\infty} A \times_{\mathbb{N}^\infty} \prod_{s_\infty} X$.

Let us now fix an object $A \in \mathbf{Set}/\mathbf{1}$ and put $A' := !^*_{\mathbb{N}^\infty}(A)$. To represent $F(A,-)$ as a polynomial functor, we will need the following pullback

$$\begin{array}{ccc} Q & \xrightarrow{\ f\ } & \coprod_{\mathbb{N}^\infty}\prod_{s_\infty} A' \\ {\scriptstyle p}\downarrow & \lrcorner & \downarrow{\scriptstyle \pi} \\ \mathbb{N}^\infty & \xrightarrow{\ s_\infty\ } & \mathbb{N}^\infty \end{array} \tag{6.7}$$

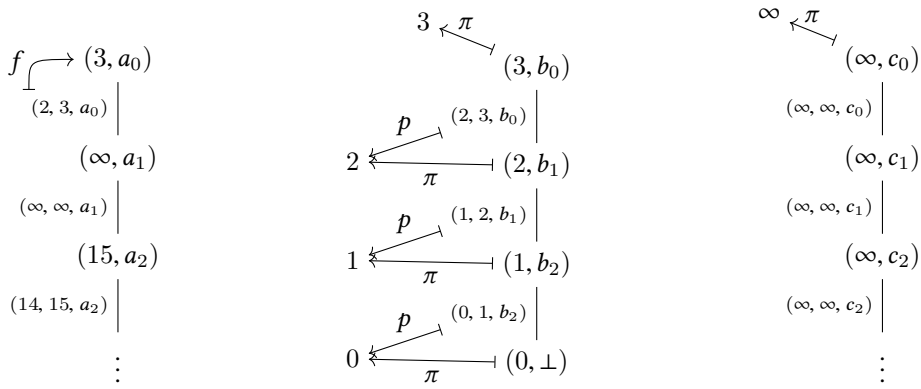


Figure 6.4.: Intermediate trees constructed in Thm. 6.3.6; only the last two are valid

where π is the evident coproduct projection. Now we have

$$\begin{aligned} F(A,X) &= \textstyle\prod_{s_\infty} A' \times_{\mathbb{N}^\infty} \prod_{s_\infty} X \\ &\cong \textstyle\coprod_{\pi} \pi^*\Big(\prod_{s_\infty} X\Big) && \times \text{ represented by } \coprod \\ &\cong \textstyle\coprod_{\pi} \prod_{f} p^* X && \text{Beck-Chevalley with (6.7)} \\ &= [\![\Gamma]\!](X), \end{aligned}$$

where Γ is the polynomial given by

$$\Gamma = \mathbb{N}^\infty \xleftarrow{p} Q \xrightarrow{f} \textstyle\coprod_{\mathbb{N}^\infty} \prod_{s_\infty} A' \xrightarrow{\pi} \mathbb{N}^\infty.$$

To be able to picture trees in M_f and in the final coalgebra of $[\![\Gamma]\!]$, we note that

$$Q \cong \Big\{(k,(n,v)) \in \mathbb{N}^\infty \times \textstyle\coprod_{\mathbb{N}^\infty} \prod_{s_\infty} A' \;\Big|\; s_\infty k = n\Big\}.$$

Moreover, we denote a pair $(n,v) \in \coprod_{\mathbb{N}^\infty} \prod_{s_\infty} A'$ by (n,a) if there in a $k \in \mathbb{N}^\infty$ with $n = s_\infty\, k$ and $v\, k = a$, or if $n = 0$ by $(0,\bot)$.

Recall that we construct in Thm. 6.3.6 the final coalgebra of $[\![\Gamma]\!]$ as a subobject of M_f. In Figure 6.4, we present three trees that are elements of M_f, where only the second and third are actually selected by the equaliser taken in Thm. 6.3.6. The matching of indices, which is used to form the equaliser, is indicated in the second tree. This tree is then an element of $\mathrm{PStr}(A)_3$, whereas the third is in $\mathrm{PStr}(A)_\infty$. ◀

Both in [AAG05, Prop. 4.1] and [vdBdM07, Cor. 4.3] it is shown that a final coalgebra for a polynomial functor can be constructed in a Martin-Löf category from an initial algebra and an equaliser. This is closely related the result of Barr [Bar93], which states that a final coalgebra of an (accessible, ω-continuous) functor F with $F\emptyset \neq \emptyset$ is the Cauchy completion of its initial algebra. We can combine this with Theorem 6.3.6 to obtain final coalgebras for *dependent* polynomial functors from initial algebras of (non-dependent) polynomial functors. This gives us already the possibility

of interpreting non-nested fixed points in any Martin-Löf category. So it remains to show that we can interpret nested fixed points.

We proceed as follows. Suppose that we are given a parameterised signature (F, u) with $F\colon \mathbf{C} \times \mathbf{B}/_I \to \mathbf{D}$, $\mathbf{C} = \prod_{i=1}^m \mathbf{B}/_{K_i}$, $\mathbf{D} = \prod_{i=1}^n \mathbf{B}/_{J_i}$, and $u_i\colon J_i \to I$. Then we need to show that $\mu(\widehat{F}, \widehat{G_u})$ and $\nu(\widehat{F}, \widehat{G_u})$ are both polynomial functors of type $\mathbf{C} \to \mathbf{B}/_I$. To do so, we first use Thm. 6.3.1, thus we have to show that for each $A \in \mathbf{C}$ the intial algebra respectively final coalgebra of the functors

$$H_1(A,-) = \coprod_{u_1} \circ F_1(A,-) +_I \cdots +_I \coprod_{u_n} \circ F_n(A,-)$$
$$H_2(A,-) = \prod_{u_1} \circ F_1(A,-) \times_I \cdots \times_I \prod_{u_n} \circ F_n(A,-)$$

give rise to polynomial functors, assuming that $F_1, \ldots, F_n$ are polynomial functors. Let K be given by $K = K_1 + \cdots + K_n$, then one can use the equivalence $\mathbf{C} \times \mathbf{B}/_I \simeq \mathbf{B}/_{K+I}$ to show that there are polynomial functors $H'_k\colon \mathbf{B}/_{K+I} \to \mathbf{B}/_I$ that are naturally isomorphic to $\mathbf{C} \times \mathbf{B}/_I \simeq \mathbf{B}/_{K+I} \xrightarrow{H_k} \mathbf{B}/_I$, see [Abb03, Prop. 4.4.2]. Such functors H'_k are captured by the following definition.

Definition 6.3.8. We call a dependent polynomial *parametric*, if it is of the form

$$K + I \xleftarrow{\ s\ } B \xrightarrow{\ f\ } A \xrightarrow{\ t\ } I.$$

As we have seen, such polynomials represent polynomial functors $\mathbf{B}/_K \times \mathbf{B}/_I \to \mathbf{B}/_I$ and allow us speak about nested fixed points. What thus remains is that fixed points of parametric dependent polynomial functors are again dependent polynomial functors. Towards this, we first bring parametric polynomials into a convenient form.

Lemma 6.3.9. *Let*

$$\Gamma: \qquad K + I \xleftarrow{\ s\ } B \xrightarrow{\ f\ } A \xrightarrow{\ t\ } I,$$

be a parametric polynomial. Then there are two polynomials

$$\begin{array}{ccccc} K & \xleftarrow{s_1} & B_1 & \xrightarrow{f_1} & \\ & & & & A \xrightarrow{\ t\ } I \\ I & \xleftarrow[s_2]{} & B_2 & \xrightarrow[f_2]{} & \end{array}$$

such that for all $Z \in \mathbf{B}/_{K+I}$

$$\coprod_t \Big(\prod_{f_1} s_1^* \kappa_1^* Z \times_A \prod_{f_2} s_2^* \kappa_2^* Z \Big) \cong [\![\Gamma]\!](Z),$$

where $\kappa_1\colon K \to K + I$ and $\kappa_2\colon I \to K + I$.

Proof. We form the pullbacks

$$\begin{array}{ccc} B_1 & \xrightarrow{s_1} & K \\ {\scriptstyle p_1}\downarrow & \lrcorner & \downarrow{\scriptstyle \kappa_1} \\ B & \xrightarrow{s} & K + I \end{array} \qquad \begin{array}{ccc} B_2 & \xrightarrow{s_2} & I \\ {\scriptstyle p_2}\downarrow & \lrcorner & \downarrow{\scriptstyle \kappa_2} \\ B & \xrightarrow{s} & K + I \end{array}$$

and put $f_i = f \circ p_i$. Then, using the abbreviations $X = \kappa_1^* Z$ and $Y = \kappa_2^* Z$, we have

$$\begin{aligned}
&\coprod_t \Big(\prod_{f_1} s_1^* X \times_A \prod_{f_2} s_2^* Y \Big) \\
&\cong \coprod_t \Big(\prod_f \prod_{p_1} s_1^* X \times_A \prod_f \prod_{p_2} s_2^* Y \Big) && \prod_{f\circ p_i} \cong \prod_f \prod_{p_i} \\
&\cong \coprod_t \Big(\prod_f s^* \prod_{\kappa_1} X \times_A \prod_f s^* \prod_{\kappa_2} Y \Big) && \text{Beck-Chevalley} \\
&\cong \coprod_t \prod_f s^* \Big(\prod_{\kappa_1} X \times_{K+I} \prod_{\kappa_2} Y \Big) && \text{fibred product} \\
&\cong \coprod_t \prod_f s^* \Big(\prod_{\kappa_1} \kappa_1^* Z \times_{K+I} \prod_{\kappa_2} \kappa_2^* Z \Big) && \text{Def. } X, Y \\
&\cong \coprod_t \prod_f s^* Z && \text{Disjoint coprod.} \\
&= [\![F]\!](Z) && \qquad \square
\end{aligned}$$

We are now in the position to prove that initial algebras and final coalgebras of parametric polynomial functors are again polynomial functors, the final step towards constructing recursive types. The proof of this fact follows that for container [Abb03, Sec. 5.3-5.5] or for non-dependent polynomials [GH04], except that we need to take care of the indices.

Theorem 6.3.10. *Initial algebras and final coalgebras of parametric (dependent) polynomial functors are again polynomial functors. More formally, let $F\colon \mathbf{B}/_{K+I} \to \mathbf{B}/_I$ be a parametric polynomial functor. Then there are polynomial functors $\mu F, \nu F\colon \mathbf{B}/_K \to \mathbf{B}/_I$ and natural transformations*

$$\alpha\colon \widehat{F}(\mu F) \Rightarrow \mu F \qquad \text{and} \qquad \xi\colon \nu F \Rightarrow \widehat{F}(\nu F),$$

so that α is an initial algebra and ξ a final coalgebra for $\widehat{F}\colon [\mathbf{B}/_K, \mathbf{B}/_I] \to [\mathbf{B}/_K, \mathbf{B}/_I]$.

Proof. Suppose F is given by the parametric polynomial

$$\Gamma: \qquad K + I \xleftarrow{\ s\ } B \xrightarrow{\ f\ } A \xrightarrow{\ t\ } I.$$

Then, by Lemma 6.3.9, we obtain two polynomials

$$\begin{array}{lccccc}
\Gamma_1: & K & \xleftarrow{s_1} & B_1 & \xrightarrow{f_1} & \\
 & & & & & A \xrightarrow{t} I \\
\Gamma_2: & I & \xleftarrow[s_2]{} & B_2 & \xrightarrow[f_2]{} &
\end{array}$$

that represent $[\![\Gamma]\!]$. Let $H\colon \mathbf{B}/_K \times \mathbf{B}/_I \to \mathbf{B}/_I$ be the functor given by

$$H(X,Y) \coloneqq \coprod_t \Big(\prod_{f_1} s_1^* X \times_A \prod_{f_2} s_2^* Y \Big). \tag{6.8}$$

The goal is then to show that there is a polynomial functor $G\colon \mathbf{B}/_K \to \mathbf{B}/_I$ that carries the structure of an initial algebra or final coalgebra, respectively.

As a first step, let us assume that such a G exists and is given by $G \cong [\![\Delta]\!]$ with

$$\Delta : \qquad K \xleftarrow{x} Q \xrightarrow{h} P \xrightarrow{y} I.$$

Then we must have for each $X \in \mathbf{B}/_K$ an isomorphism $H(X, G(X)) \cong G(X)$. We now want to calculate, as in [Abb03, Sec. 5.3-5.4], how Δ needs to be given.

The first step is to define $y\colon P \to I$, an easy calculation shows that there must be an isomorphism

$$\psi\colon [\![\Gamma_2]\!](y) \xrightarrow{\cong} y. \tag{6.9}$$

So, analogously to [Abb03; GH04], we construct y as the carrier of an initial algebra or final coalgebra, respectively, of $[\![\Gamma_2]\!]$, depending on whether G should be the carrier of an initial algebra or final coalgebra.

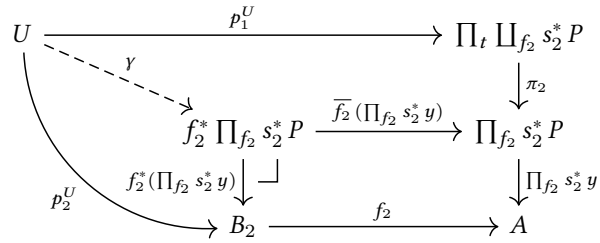
To define h and x, we first introduce a few morphisms obtained from pullbacks. Their purpose will become clear once we give the constraints introduced by the isomorphism $H(X, G(X)) \cong G(X)$. First, we reindex y along s_2 by means of the following pullback.

$$\begin{array}{ccc} s_2^* P & \xrightarrow{\overline{s_2}\, y} & P \\ {\scriptstyle s_2^* y}\downarrow & \lrcorner & \downarrow{\scriptstyle y} \\ B_2 & \xrightarrow{s_2} & I \end{array}$$

Next, we will use the pullback

$$\begin{array}{ccc} U & \xrightarrow{p_2^U} & B_2 \\ {\scriptstyle p_1^U}\downarrow & \lrcorner & \downarrow{\scriptstyle f_2} \\ \prod_t \coprod_{f_2} s_2^* P & \xrightarrow{\pi_1} & A \end{array}$$

from which we define a morphism $\gamma\colon U \to f_2^* \prod_{f_2} s_2^* P$ as in the following diagram, i.e., by using the lifting property of the Cartesian morphism $\overline{f_2}\,(\prod_{f_2} s_2^* y)$.



Finally, we use will use the counit $\varepsilon\colon f_2^* \prod_{f_2} \Rightarrow \mathrm{Id}$ of the adjunction $f_2^* \dashv \prod_{f_2}$. With all this notation set up, we demand there to be an isomorphism

$$\varphi\colon \pi_1^*(f_1) + \coprod_{p_1^U} (\overline{s_2}\, y \circ \varepsilon_{s_2^* y} \circ \gamma)^*(h) \xrightarrow{\cong} \psi^*(h), \tag{6.10}$$

where the binary coproduct is taken in $\mathbf{B}/\prod_t \coprod_{f_2} s_2^* P$. Since

$$\coprod_{\psi}\left(\pi_1^*(f_1) + \coprod_{p_1^U} (\overline{s_2}\, y \circ \varepsilon_{s_2^* y} \circ \gamma)^*\right)$$

is a polynomial functor, we can construct, analogously to [Abb03], φ as initial dialgebra by using Theorem 6.3.1.

The last ingredient to the polynomial Δ, which represents G, is the map $x\colon Q \to K$. It can be defined as the inductive extension of

$$\begin{aligned}&\chi\colon\ \pi_1^*(f_1) + \coprod_{p_1^U} (\overline{s_2}\, y \circ \varepsilon_{s_2^* y} \circ \gamma)^*(K^Q) \xrightarrow{\cong} \psi^*(K^Q)\\ &\chi = [s_1, \pi_2].\end{aligned}$$

We thus have

$$x = [s_1, x \circ \pi_2] \circ \varphi^{-1}. \tag{6.11}$$

By using ψ and φ, we can now define mutually inverse maps

$$\alpha_X : H(X, G(X)) \xleftrightarrow{\cong} G(X) : \beta_X,$$

where $G = [\![\Delta]\!]$. In other words, we show that

$$\coprod_t\Big(\prod_{f_1} s_1^* X \times_A \prod_{f_2} s_2^* \coprod_y \prod_h x^* X\Big) \cong \coprod_y \prod_h x^* X$$

by means of α_X and β_X. To simplify the presentation, we give these maps in the internal language of $\mathbf{B}^{\to}$ and annotate steps with types. Thus, let $i : I$ and define α_X by

$$\alpha_{X,i}\left(a : A, u : \left(\prod_{f_1} s_1^* X\right)_a, v : \left(\prod_{f_2} s_2^* \coprod_y \prod_h x^* X\right)_a \,\middle|\, t\ a = i\right) = (p, w)$$

where

$$\begin{aligned}&(p : P \mid y\ p = i) = \psi(a, \pi_1 \circ v)\\ &w : (\textstyle\prod_h x^* X)_p\\ &w\ (q : h \mid h\ q = p) = [w_1, w_2](\varphi^{-1}\ q)\\ &w_1 : (\pi_1^*(f_1))_{(a,\pi_1\circ v)} \to (x^* X)_{(a,\pi_1\circ v)}\\ &w_1\ (b_1 : B_1 \mid f_1\ b_1 = a) = u\ b_1\\ &w_2 : ((\overline{s_2}\, y \circ \varepsilon_{s_2^* y} \circ \gamma)^*(h))_{(a,\pi_1\circ v)} \to (x^* X)_{(a,\pi_1\circ v)}\\ &w_2\ ((b_2 : B_2, a' : A, z : (\textstyle\prod_{f_2} s_2^* y)_{a'}), q' : Q\\ &\qquad \mid f_2\ b_2 = a' = a \text{ and } z = \pi_1 \circ v \text{ and } h\ q' = z\ b_2 = \pi_1(v\ b_2))\\ &\quad = \pi_2\ (v\ b_2)\ q'.\end{aligned}$$

Note that p is in the correct fibre, since

$$y\ p = y\ (\psi\ (a, u')) = (\textstyle\coprod_t \prod_{f_2} s_2^* y)(a, u') = t\ a = i,$$

and that w_1 and w_2 are well-defined because

$$X\,(u\,b_1) = s_1\,b_1 = [s_1, x \circ \pi_2]\,(\kappa_1\,b_1) = [s_1, x \circ \pi_2]\,(\varphi^{-1}\,q) = x\,q$$

and

$$X\,(\pi_2\,(v\,b_2)\,q') = x\,q' = [s_1, x \circ \pi_2]\,(\kappa_2\,((b_2, a', z), q')) = [s_1, x \circ \pi_2]\,(\varphi^{-1}\,q) = x\,q.$$

Dually, β_X is given for $i : I$ by

$$\beta_{X,i}\left(p : P, w : \left(\prod\nolimits_h x^*\,X\right)_p \;\middle|\; y\,p = i\right) = (a, u, v)$$

where

$$\left(a : A, v' : \left(\prod\nolimits_{f_2} s_2^*\,y\right)_a \;\middle|\; t\,a = i\right) = \psi_i^{-1}\,p$$

$$u : \left(\prod\nolimits_{f_1} s_1^*\,X\right)_a$$

$$u\,(b_1 : B_1 \mid f_1\,b_1 = a) = w\,(\varphi_{(a,v')}\,(\kappa_1\,b_1))$$

$$v : \left(\prod\nolimits_{f_2} s_2^* \coprod\nolimits_y \prod\nolimits_h x^*\,X\right)_a$$

$$v\,(b_2 : B_2 \mid f_2\,b_2 = a) = (v'\,b_2, w')$$

where

$$w'\,(q : Q \mid h\,q = v'\,b_2) = w\,(\varphi_{(a,v')}\,(\kappa_2\,(b_2, a, v', q))).$$

Let us also check that the indices for β_X match up. For well-definedness of u we have

$$h\,(\varphi_{(a,v')}\,(\kappa_1\,b_1)) = \psi_i\,(a, v') = \psi_i\,(\psi_i^{-1}\,p) = p,$$

thus the application $w\,(\varphi_{(a,v')}\,(\kappa_1\,b_1))$ is well-defined, and we have

$$X\,(w\,(\varphi\,(\kappa_1\,b_1))) = x\,(\varphi\,(\kappa_1\,b_1)) = [s_1, \pi_2]\,(\varphi^{-1}\,(\varphi\,(\kappa_1\,b_1))) = [s_1, \pi_2]\,(\kappa_1\,b_1) = s_1\,b_1,$$

hence $u : (\prod_{f_1} s_1^*\,X)_a$. For v we note that

$$y\,(v'\,b_2) = s_2\,b_2,$$

thus $v'\,b_2 : (s_2^*\,y)_{b_2}$. Moreover, we have

$$h\,(\varphi_{(a,v')}\,(\kappa_2\,(b_2, a, v', q))) = \psi_i\,(a, v') = p,$$

so that the application $w\,(\varphi_{(a,v')}\,(\kappa_2\,(b_2, a, v', q)))$ is well-defined. Finally, we find

$$\begin{aligned} X\,(w\,(\varphi\,(\kappa_2\,(b_2, a, v', q)))) &= x\,(\varphi\,(\kappa_2(b, a, v', q))) \\ &= [s_1, x]\,(\varphi^{-1}\,(\varphi\,(\kappa_2(b, a, v', q)))) \\ &= [s_1, x]\,(\kappa_2(b, a, v', q)) \\ &= x\,q. \end{aligned}$$

So $w : (\prod_h x^* X)_{v'\, b_2}$ and, putting this together with $v'\, b_2 : (s_2^* y)_{b_2}$, we have that v has the claimed type. Hence β_X is well-defined.

It is now also straightforward to check that α_X and β_X are mutually inverse natural transformations. Finally, to show that $G(X)$ is the initial algebra (resp. final coalgebra) for $H(X, -)$ follows the same line of argument as [Abb03]. □

Summing up, we obtain the following result.

Corollary 6.3.11. *All recursive types for strictly positive signatures can be constructed in the Martin-Löf category* **B**.

6.4. Internal Reasoning Principles

In the introduction we have seen that first-order intuitionistic logic arises through the Brouwer-Heyting-Kolmogorov interpretation in dependent type theories. The comprehension categories (Definition 6.2.16) allowed us to give a category theoretical account for dependent types, and in Theorem 6.2.11 we saw that that important connectives, like conjunction and disjunction, universal and existential quantification, falsum etc., are definable as recursive types in a μPCC. Thus, we are able to encode first-order formulas in a μPCC. This raises the question, under what conditions we are able to prove formulas in a μPCC by appealing to induction and coinduction principles for truly recursive types like natural numbers or streams.

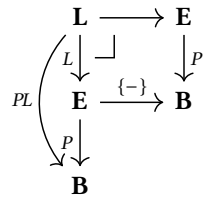
More specifically, given an inductive type, like the natural numbers, we want to prove propositions on that type by induction. On the other hand, for coinductive types, like streams, we would like to prove identities between elements of that type by establishing a bisimulation relation. Both proof methods are well-understood for fibrations $L\colon \mathbf{L} \to \mathbf{D}$, where $\mathbf{L}$ is to be thought of as a logic with free variables ranging over recursive types in $\mathbf{D}$, see the work of Hermida and Jacobs [HJ97], and Fumex et al. [FGJ11]. However, the way to think about dependent types is that these simultaneously define types and a logic for these types, so that we can actually construct $\mathbf{L}$ for a μPCC $P\colon \mathbf{E} \to \mathbf{B}$. It is then this logic, in which we want to derive induction and coinduction principles.

This section is structured as follows. First, we construct said logic over P, and give a weak induction principle within this logic that can be proved without further assumptions. Then we instantiate, under additional assumptions, the framework for "fibred" induction and coinduction, defined in [FGJ11]. It turns out that the crucial assumption for obtaining induction is a strong elimination principle for coproducts. Next, we show that strong elimination is equivalent to dependent recursion, where the latter seems to be a more natural requirement. Finally, we derive a simple coinduction principle for μPCCs, which holds without any further assumptions. This last fact tells us that μPCC model, as expected, extensional rather than intensional type theories.

6.4.1. Internal Logic

We begin by defining a notion of predicates φ over types in a μPCC $P\colon \mathbf{E} \to \mathbf{B}$. For a type A, these are given as objects in $\mathbf{P}_{\{A\}}$, that is, types depending on A. This situation is captured by the pullback

(change of base) below.



We call $L\colon \mathbf{L} \to \mathbf{E}$ the *internal logic* of P. Explicitly, objects in $\mathbf{L}$ are pairs (A, φ) with $\varphi \in \mathbf{P}_{\{A\}}$ and morphisms $(A, \varphi) \to (B, \psi)$ are pairs (f, g) of morphisms in $\mathbf{E}$ with $f\colon A \to B$, $g\colon \varphi \to \psi$ and $\{f\} = Pg$. The internal logic fibration L restricts for each $I \in \mathbf{B}$ to a fibration $L^I\colon \mathbf{PL}_I \to \mathbf{P}_I$, where $\mathbf{PL}_I$ is the fibre of $P \circ L$, the left side in the diagram, above I. The reindexing functor $u^{\#}\colon \mathbf{PL}_J \to \mathbf{PL}_I$ for $u\colon I \to J$ in $\mathbf{B}$ is then given by

$$u^{\#}(A, \varphi) = (u^* A, \{\overline{u}\, A\}^* \varphi) \qquad u^{\#}(\mathrm{id}_A, f) = (\mathrm{id}_{u^* A}, \{\overline{u}\, A\}^* f)$$

where $\overline{u}\, A\colon u^* A \to A$ is the Cartesian lifting of u, see Def. 2.4.1.

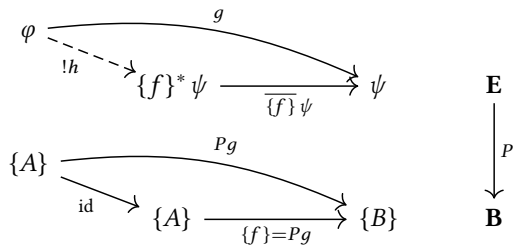
As we have seen in Theorem 6.2.11, every μPCC P has coproducts $\coprod_f\colon \mathbf{P}_I \to \mathbf{P}_J$ along all $f\colon I \to J$ in $\mathbf{B}$ as left adjoint to f^*, in other words, P is a *bifibration*, see [Jac99, Lem. 9.1.2]. We use these coproducts mostly to turn a predicate over A into a type in $\mathbf{P}_I$, where $I = PA$. In this case, we use the notation $\coprod_A \coloneqq \coprod_{\pi_A}$ for the projection $\pi_A\colon \{A\} \to I$ given in Definition 6.2.16. Crucially, these projections come, by their definition, in the form of a natural transformation $\pi\colon \{-\} \Rightarrow P$. This allows us to define a fibrewise comprehension-like functor $\{-\}^I\colon \mathbf{PL}_I \to \mathbf{P}_I$ for L_I.

Lemma 6.4.1. *For each $I \in \mathbf{B}$, we have the following.*

1. *$L^I\colon \mathbf{PL}_I \to \mathbf{P}_I$ is a bifibration, given for $f\colon A \to B$ in $\mathbf{P}_I$ by $\coprod_f(A, \varphi) = \big(B, \coprod_{\{f\}} \varphi\big)$.*
2. *$L^I\colon \mathbf{PL}_I \to \mathbf{P}_I$ has a right adjoint $\mathbf{1}^I_{(-)}\colon \mathbf{P}_I \to \mathbf{PL}_I$, given by $\mathbf{1}^I_A = \big(A, \mathbf{1}_{\{A\}}\big)$.*
3. *There is a functor $\{-\}^I\colon \mathbf{PL}_I \to \mathbf{P}_I$, given by $\{(A, \varphi)\}^I = \coprod_A \varphi$, with projections $\pi^I\colon \{-\}^I \Rightarrow L^I$.*

Proof.

1. (Bi)fibrations are preserved under change of base, see for example [Jac99, Lem. 1.51 and Lem. 9.1.2].
2. Fibred final objects are preserved under change of base [Jac99, Lem 1.8.4], hence we get a final object functor $\mathbf{1}_{(-)}\colon \mathbf{E} \to \mathbf{L}$. Since this is a fibred functor, we can restrict it to $\mathbf{1}^I_{(-)}\colon \mathbf{P}_I \to \mathbf{PL}_I$.
3. To define the action of $\{-\}^I$ on morphisms $(f, g)\colon (A, \varphi) \to (B, \psi)$, we use that there is a unique mediating morphism $h\colon \varphi \to \{f\}^* \psi$ above the identity with $\overline{\{f\}}\, \psi \circ h = g$ as in the following diagram, see Definition 2.4.1.



Note, moreover, that there is a canonical natural transformation $\beta\colon \coprod_A \{f\}^* \Rightarrow P(f)^* \coprod_B$, as used in the Beck-Chevalley condition, which arises because of the identity $P(f) \circ \pi_A = \pi_B \circ \{f\}$:

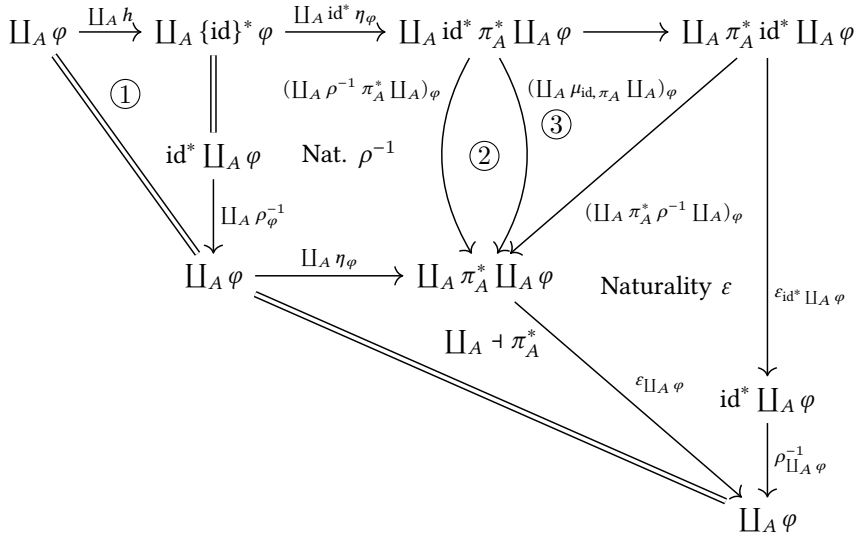
$$\coprod_A \{f\}^* \Rightarrow \coprod_A \{f\}^* \pi_B^* \coprod_B \overset{\cong}{\Rightarrow} \coprod_A \pi_A^* (Pf)^* \coprod_B \Rightarrow (Pf)^* \coprod_B$$

Then we can put

$$\{(f,g)\}^I = \coprod_A \varphi \xrightarrow{\coprod_A h} \coprod_A \{f\}^* \psi \xrightarrow{\beta_\psi} (Pf)^* \left(\coprod_B \psi\right) \xrightarrow{\cong} \coprod_B \psi,$$

using the isomorphism $P(f)^* = \mathrm{id}_I^* \cong \mathrm{Id}$. To check that this indeed gives rise to a functor, one needs to check that $\{\mathrm{id}_{(A,\varphi)}\}^I = \mathrm{id}_{\{(A,\varphi)\}^I}$ and $\{(f_2,g_2)\circ(f_1,g_1)\}^I = \{(f_2,g_2)\}^I \circ \{(f_1,g_1)\}^I$. Both follow, with enough patience, from the coherence conditions of a cloven fibration and of the adjunction $\coprod_A \dashv \pi_A^*$, naturality of the involved morphism, and the uniqueness of h.

We demonstrate this for $\{\mathrm{id}_{(A,\varphi)}\}^I = \{(\mathrm{id}_A, \mathrm{id}_\varphi)\}^I$ and $\mathrm{id}_{\{(A,\varphi)\}^I} = \mathrm{id}_{\coprod_A \varphi}$ in the following diagram, in which $\{(\mathrm{id}_A,\mathrm{id}_\varphi)\}^I$ is the outer, upper triangle.



The triangle ① commutes because ρ is defined as h in this case, thus $\coprod_A \rho_\varphi^{-1} \circ \coprod_A h = \mathrm{id}$ by functoriality of $\coprod_A$. The parts ② and ③ both commute by the coherence conditions of a cloven fibration. For the other parts, the reason for commutativity have been indicated in the diagram.

The projections are obtained as follows. For each A, we can define a morphism $\mathrm{prj}_A\colon \mathbf{1}_{\{A\}} \to \pi_A^* A$ as the unique map mediating the counit $\varepsilon_A\colon \mathbf{1}_{\{A\}} \to A$ of $\mathbf{1} \dashv \{-\}$ and the Cartesian lifting $\overline{\pi_A} A\colon \pi_A^* A \to A$. This gives a map

$$\varphi \xrightarrow{!_\varphi} \mathbf{1}_{\{A\}} \xrightarrow{\mathrm{prj}_A} \pi_A^* A,$$

hence a unique morphism $\pi^I_\varphi \colon \coprod_A \varphi \to A$. Naturality of π^I follows from final objects being fibred, naturality of the involved morphism, the coherence conditions of $\coprod_B \dashv \pi^*_B$ and uniqueness of Cartesian extensions. □

6.4.2. Induction and Dependent Iteration

We are now going to study dependent iteration and induction principles for inductive types in the internal logic we introduced in Section 6.4.1. First, we derive a weak dependent iteration principle that can be defined in any μPCC. Second, we show that we can obtain a proper dependent iteration, or induction, principle if we have strong sums. Since the second projection of strong sums can be defined by using dependent iteration, we arrive at the conclusion that having strong sums is the equivalent to having dependent iteration in a μPCC.

Before we get into the technical details, let us first discuss what dependent iteration is. As a starting point, let us consider the type Nat of natural numbers, which is determined by the constructors $0 : \mathrm{Nat}$ and $\mathrm{suc} : \mathrm{Nat} \to \mathrm{Nat}$. The usual iterations principle would read something like the following rule

$$\frac{\vdash X : \mathbf{Type} \qquad \Gamma \vdash x_0 : X \qquad \Gamma \vdash s : X \to X}{\Gamma \vdash \mathrm{iter}(x_0, s) : \mathrm{Nat} \to X}$$

together with the equations

$$\mathrm{iter}(x_0, s)\ 0 = x_0 \quad \text{and} \quad \mathrm{iter}(x_0, s)\ (\mathrm{suc}\ n) = s\ (\mathrm{iter}(x_0, s)\ n).$$

That is to say, given a type X, a starting value x_0 and a successor map s, we can construct for each natural number n an element in X by iterating s n-times on x_0. Dependent iteration means now that the type X may vary with (that is, depends on) the input value. This is a direct adaptation of the induction principle to the type theoretic setting.[56] Of course, we have to extend the input accordingly, which leads us to the following rule.

$$\frac{n : \mathrm{Nat} \vdash X[n] : \mathbf{Type} \qquad \Gamma \vdash x_0 : X[0] \qquad \Gamma, k : \mathrm{Nat} \vdash s : X[k] \to X[\mathrm{suc}\ k]}{\Gamma, n : \mathrm{Nat} \vdash \mathrm{iter}(x_0, s)\ n : X[n]}$$

The equations that specify the behaviour of dependent iteration have of course be changed accordingly:

$$\mathrm{iter}(x_0, s)\ 0 = x_0 \quad \text{and} \quad \mathrm{iter}(x_0, s)\ (\mathrm{suc}\ n) = s[n]\ (\mathrm{iter}(x_0, s)\ n).$$

Thus far, we have explained dependent iteration only for a special type and also by using type theoretic syntax. So what is the general, category theoretical picture behind dependent iteration? Suppose that (A, α) an initial (F, G_u)-dialgebra in $\mathbf{P}_I$ for a signature (F, u) with $F \colon \mathbf{P}_I \to \mathbf{C}$. First of all, we need a type that depends on the elements of A, that is, we need a predicate over A. In other words, we suppose to be given $(A, \varphi) \in \mathbf{PL}_I$. The usual way to describe dependent iteration or induction, would be to define liftings $\overline{F}$ and $\overline{G_u}$ to functors $\mathbf{PL}_I \to \mathbf{D}$, where $\mathbf{D}$ lifts $\mathbf{C}$ to predicates over the correct indices, just as $\mathbf{P}_I$ is lifted to $\mathbf{PL}_I$. Dependent iteration arises then, if for any dialgebra $\overline{F}(A, \varphi) \to \overline{G_u}(A, \varphi)$ there is a morphism $\mathbf{1}_{\{A\}} \to (A, \varphi)$ in $\mathbf{PL}_I$. Note that $\mathbf{1}_{\{A\}}$ models the full predicate on A and thus expresses that all elements of A are present in φ. This is the result we will have at the end of this section, but it requires strong coproducts to prove. So before that, we will establish a weaker dependent iteration principle that works without strong coproducts. The

idea is that we can track dependencies using sums, that is, given $(A, \varphi) \in \mathbf{PL}_I$, we can pack this predicate to $\{(A,\varphi)\}^I = \coprod_A \varphi \in \mathbf{P}_I$, see Lemma 6.4.1. This trick allows us to avoid having to move to dialgebras on predicates and instead formulate dependent iteration simply on data types, as the following proposition shows.

Proposition 6.4.2. *Let (F,u) be a signature with $F\colon \mathbf{P}_I \to \mathbf{C}$, (A,α) an initial (F,G_u)-dialgebra, and $d\colon F\{(A,\varphi)\}^I \to G_u\{(A,\varphi)\}^I$, such that $\pi^I_\varphi\colon \{(A,\varphi)\}^I \to A$ is a homomorphism. Then there is a unique homomorphism $h\colon (A,\alpha) \to (\{(A,\varphi)\}^I, d)$ with $\pi^I_\varphi \circ h = \mathrm{id}_A$ as in the following diagrams.*

$$\begin{array}{ccccc} F(A) & \xrightarrow{Fh} & F\{(A,\varphi)\}^I & \xrightarrow{F\pi^I_\varphi} & F(A) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle \alpha} \\ G_u(A) & \xrightarrow{G_u h} & G_u\{(A,\varphi)\}^I & \xrightarrow{G_u\pi^I_\varphi} & G_u(A) \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{!h} & \{(A,\varphi)\}^I \\ & \searrow & \downarrow{\scriptstyle \pi^I_\varphi} \\ & & A \end{array}$$

Proof. The homomorphism h is the inductive extension of d and, since $\pi^I_\varphi \circ h$ is an endomorphism on α, it must be the identity by uniqueness of inductive extensions. □

Note that the condition π^I_φ being a homomorphism states that d only changes the element of φ, but preserves the elements of A. The morphism h maps the elements of A to their corresponding elements of φ, and the assertion $\pi^I_\varphi \circ h = \mathrm{id}$ tells us that h keeps the index from A intact. Let us demonstrate the dependent iteration principle by deriving the classical induction principle for vectors.

Example 6.4.3. Recall that we defined vectors as the initial (F,G_u)-dialgebra with $F = \langle \mathbf{1}, K_A \times \mathrm{Id}\rangle$ and $G_u = \langle z^*, s^*\rangle$, giving rise to the constructor $\alpha = (\mathrm{nil}, \mathrm{cons})$. In this case, the condition that $\pi^{\mathbb{N}}_\varphi$ is a homomorphism for a map $d = (d_1, d_2)$ becomes

$$\begin{aligned} &d_1\colon \mathbf{1} \to z^*\left(\textstyle\coprod_{\mathrm{Vec}\,A}(\varphi)\right) && z^*\big(\pi^{\mathbb{N}}_\varphi\big) \circ d_1 = \mathrm{nil} \\ &d_2\colon A \times \textstyle\coprod_{\mathrm{Vec}\,A}(\varphi) \to s^*\left(\textstyle\coprod_{\mathrm{Vec}\,A}(\varphi)\right) && s^*\big(\pi^{\mathbb{N}}_\varphi\big) \circ d_2 = \mathrm{cons} \circ \big(\mathrm{id}_A \times \pi^{\mathbb{N}}_\varphi\big), \end{aligned}$$

which are, modulo the use of coproducts, the usual condition for induction base and step.

Let us derive from Proposition 6.4.2 an induction principle for predicates on vectors that have only trivial proofs for membership. More precisely, let $\Psi \sqsubseteq \mathrm{Vec}\,A$ be a predicate in $\mathbf{Set}^{\mathbb{N}}$, where $\sqsubseteq$ denotes the index-wise set inclusion. We can derive from Ψ a predicate $\varphi \in \mathbf{Set}^{\{\mathrm{Vec}\ A\}}$ in the internal logic of $\mathrm{Fam}(\mathbf{Set})$ by defining

$$\varphi_{(n,v)} = \begin{cases} \mathbf{1}, & v \in \Psi_n \\ \emptyset, & \text{otherwise} \end{cases}.$$

Our goal is now to derive the following induction principle for Ψ by using Proposition 6.4.2 on φ.

$$\frac{\mathrm{nil} * \in \Psi_0 \qquad \forall n \in \mathbb{N}.\, \forall a \in A.\, \forall v \in (\mathrm{Vec}\ A)_n.\, v \in \Psi_n \Longrightarrow \mathrm{cons}\, a\, v \in \Psi_{n+1}}{\mathrm{Vec}\,A \sqsubseteq \Psi} \tag{6.12}$$

The premises of the rule (6.12) allow us to define maps d_1 and d_2 as above by putting

$$d_1(\star) \coloneqq (\mathrm{nil}\,\star, \star) \qquad\qquad d_2(a,(v,\star)) \coloneqq (\mathrm{cons}\, a\, v, \star).$$

That the preconditions of Proposition 6.4.2 hold with respect to these maps is clear, thus we obtain a map $h\colon \mathrm{Vec}\,A \to \coprod_{\mathrm{Vec}\,A}\phi$ with $h(v) = (v, \star)$. Hence for each $n \in \mathbb{N}$ and $v \in (\mathrm{Vec}\,A)_n$, we have that $\star \in \varphi_{(n,v)}$ and so $v \in \Psi_n$. This gives us the conclusion $\mathrm{Vec}\,A \sqsubseteq \Psi$ of the induction rule (6.12). ◀

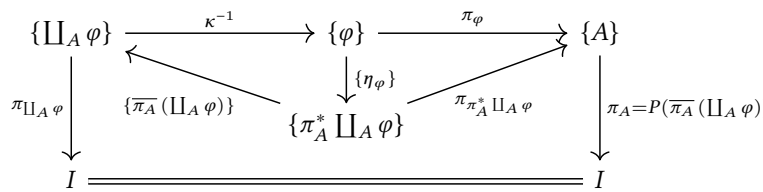
Note that in the last example we have used only a special case of the dependent iteration principle that we derived in Proposition 6.4.2: each element of the family φ was defined to be either the empty set or a singleton. Such definitions are usually called *proof irrelevant* because the truth of φ_v is not witnessed by an informative proof. Thus, we only get to know *that* φ_v holds but not *why*. In a type theoretic setting, we would usually define φ_v to be the set of all proofs that witness that φ holds for v, which will be empty if φ does not hold for v. This explains the reason why we called the content of Proposition 6.4.2 a *weak* dependent iteration principle: It allows us to prove that a predicate φ holds but we cannot extract the proof for this fact, thus it is fairly useless as a general iteration principle.

To see this, suppose for instance that we define an inductive type $\mathrm{Elem} \in \mathbf{P}_{\{\mathrm{Vec}\,A\}}$ that describes positions in vectors. We can use Proposition 6.4.2 only to define maps $h\colon \mathrm{Vec}\,A \to \coprod_{\mathrm{Vec}\,A}\mathrm{Elem}$. However, we cannot extract the actual element from the vector, as we do not have access to Elem itself.

The principle given in Proposition 6.4.2 has another shortcoming: We can use it only on initial dialgebras, since we otherwise cannot construct the homomorphism with the required property. This and the problem described above can be fixed if we assume strong coproducts, see Definition 6.3.2, as we show in the following. In a first step towards this, let us record that if the coproducts in a given μPCC are strong, then its internal logic fibration is a closed comprehension category (CCompC, Definition 6.3.2) as well. This is proved for a general CCompC, which an μPCC with strong coproducts is, in [Jac91, Prop. 4.4.10].

Lemma 6.4.4. *If P admits strong coproducts and full comprehension, then $L\colon \mathbf{L} \to \mathbf{E}$ is a CCompC and in particular we have $\mathbf{1}^I_{(-)} \dashv \{-\}^I$.*

Strong coproducts also allow us do define the index projection $\mathrm{fst}_\varphi\colon \coprod_A \varphi \to A$ as a morphism in $\mathbf{P}_I$ in a simple way. First, note that the following diagram commutes



by definition of κ and the unit η_φ being vertical (i.e., $P(\eta_\varphi) = \mathrm{id}_{\{A\}}$). We thus obtain from fullness of the CCU P a unique map $\mathrm{fst}_\varphi\colon \coprod_A \varphi \to A$ with $\{\mathrm{fst}_\varphi\} = \pi_\varphi \circ \kappa^{-1}$ and $P(\mathrm{fst}_\varphi) = \mathrm{id}_I$. Note that in the case $\varphi = \mathbf{1}_{\{A\}}$ the map $\mathrm{fst}_{\mathbf{1}_{\{A\}}}\colon \coprod_A \mathbf{1}_{\{A\}} \to A$ is an isomorphism by P being a full comprehension category, see [Jac99, Exercise 10.5.4(iii)].

Combining Proposition 6.4.2 with Lem. 6.4.4 we obtain a proper induction principle for initial dialgebras.

Proposition 6.4.5. *Let $P\colon \mathbf{E} \to \mathbf{B}$ be a μPCC with strong coproducts, (F, u) a signature with $F\colon \mathbf{P}_I \to \mathbf{C}$ and (A, α) an initial (F, G_u)-dialgebra. Then we have functors*

$$\ddot{F}, \ddot{G}_u\colon \mathbf{PL}_I \to \mathbf{C} \qquad \ddot{F} \coloneqq F \circ \{-\}^I \qquad \ddot{G}_u \coloneqq G_u \circ \{-\}^I,$$

such that the dialgebra $\ddot{\alpha}\colon \ddot{F}(\mathbf{1}_A^I) \to \ddot{G}_u(\mathbf{1}_A^I)$ given by

$$\ddot{F}(\mathbf{1}_A^I) = F\Big(\coprod\nolimits_A \mathbf{1}_{\{A\}}\Big) \xrightarrow{\cong} FA \xrightarrow{\alpha} G_uA \xrightarrow{\cong} G_u\Big(\coprod\nolimits_A \mathbf{1}_{\{A\}}\Big) = \ddot{G}_u(\mathbf{1}_A^I),$$

is initial among those $(\ddot{F}, \ddot{G}_u)$-dialgebras $((A, \varphi), d)$ for which $\pi^I_\varphi\colon \{(A,\varphi)\}^I \to A$ is a (F, G_u)-dialgebra homomorphism. That is to say, there is a unique $h\colon \mathbf{1}_A^I \to (A, \varphi)$, such that the following diagram commutes.

$$\begin{array}{ccccc}
\ddot{F}(\mathbf{1}_A^I) & \xrightarrow{\ddot{F}h} & \ddot{F}(A,\varphi) = F\{(A,\varphi)\}^I & \xrightarrow{F\pi^I_\varphi} & F(A) \\
\downarrow{\scriptstyle \ddot{\alpha}} & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle \alpha} \\
\ddot{G}_u(\mathbf{1}_A^I) & \xrightarrow{\ddot{G}_u h} & \ddot{G}_u(A,\varphi) = G_u\{(A,\varphi)\}^I & \xrightarrow{G_u\pi^I_\varphi} & G_u(A)
\end{array}$$

Proof. Let $\ddot{\alpha} = G_u(\mathrm{fst}^{-1}) \circ \alpha \circ F(\mathrm{fst})$ be the indicated $(\ddot{F}, \ddot{G}_u)$-dialgebra, where $\mathrm{fst}\colon \coprod_A \mathbf{1}_{\{A\}} \to A$ is the projection that we discussed above. We then have

$$G_u\,(\mathrm{fst}) \circ \ddot{\alpha} = G_u\,(\mathrm{fst}) \circ G_u(\mathrm{fst}^{-1}) \circ \alpha \circ F(\mathrm{fst}) = \alpha \circ F\,(\mathrm{fst})\,,$$

thus fst is a homomorphism as indicated in the left square of (6.13) below. Next, let d be a dialgebra for which π^I_φ is a homomorphism. From Proposition 6.4.2 we obtain a unique $h\colon A \to \{(A,\varphi)\}^I$, making the right square of the following diagram commute.

$$\begin{array}{ccccc}
\ddot{F}(\mathbf{1}_A^I) & \xrightarrow{F(\mathrm{fst})} & F(A) & \xrightarrow{Fh} & \ddot{F}(A,\varphi) \\
\downarrow{\scriptstyle \ddot{\alpha}} & & \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle d} \\
\ddot{G}_u(\mathbf{1}_A^I) & \xrightarrow{G_u(\mathrm{fst})} & G_u(A) & \xrightarrow{G_u h} & \ddot{G}_u(A,\varphi)
\end{array} \tag{6.13}$$

By fullness, see Lemma 6.4.4, we then obtain a unique $\check{h}\colon \mathbf{1}_A^I \to (A,\varphi)$ with $\{\check{h}\}^I = h \circ \pi^I$ and $PL(\check{h}) = \mathrm{id}_A$, since the following diagram commutes by Proposition 6.4.2.

$$\begin{array}{ccccc}
\{\mathbf{1}_A^I\}^I & \xrightarrow{\pi^I} & A & \xrightarrow{h} & \{A,\varphi\}^I \\
\downarrow{\scriptstyle \pi^I} & & & & \downarrow{\scriptstyle \pi^I} \\
A & = & A & = & A
\end{array}$$

Let us now show that $\check{h}$ is a homomorphism $\ddot{\alpha} \to d$, using that $\pi^I_{\mathbf{1}_{\{A\}}} = \mathrm{fst}$, see proof of Lemma 6.4.1.3.

$$\begin{aligned}
d \circ \ddot{F}(\check{h}) &= d \circ F(\{\check{h}\}^I) && \text{Def. of } \ddot{F} \\
&= d \circ F(h \circ \pi^I) && \text{Def. of } \check{h} \\
&= G_u(h \circ \pi^I) \circ \ddot{\alpha} && \text{By (6.13) and using } \pi^I_{\mathbf{1}_{\{A\}}} = \mathrm{fst} \\
&= \ddot{G}_u(\check{h}) \circ \ddot{\alpha} && \text{Def. of } \check{h} \text{ and } \ddot{G}_u
\end{aligned}$$

Uniqueness of $\check{h}$ is immediate by uniqueness of h and by the comprehension being faithful. Hence $\ddot{\alpha}$ is initial indeed. □

Now that we have obtained a dependent iteration (or induction) principle from uniqueness of inductive extensions and strong coproducts, one may ask if we can also go the other way round. That is to say, can we state a dependent iteration principle without appealing to strong coproducts and then derive the existence of an inverse for the coproduct injection from there? We will answer the first question affirmatively here, but leave the second to the syntactic development in Chapter 7. The reason for deferring the second part is that the answer to the first question is unfortunately very technical and should merely be seen as a possibility result, not a practically usable result. One can formulate the result differently as to obtain a better usable result but that requires us to manually define a lifting for the strictly positive type functors in Definition 6.2.9. This is certainly possible but then we would do the same again in the syntactic development, thereby repeating a lot of work. Cutting a long story short, we will now adapt the development of an induction principle by Fumex et al. [FGJ11] to our setting of dependent iteration for inductive types without appealing to strong coproducts.

In [FGJ11], Fumex et al. define a generic lifting for functors to predicates to be able to formulate an induction principle. A remarkable fact of this lifting is that it does not use any further properties of the functor that it lifts, only the fact that comprehension is right adjoint to the final object functor. However, in our case we would thus need for each $I \in \mathbf{B}$ that $\{-\}^I$ is right adjoint to $\mathbf{1}^I_{(-)}$, which in turn needs that the coproducts in the fibration P are strong, see Lemma 6.4.4. Since we want to avoid requiring that, we restate the result in [GJF12, Thm. 4.14] under weaker conditions, namely without using the full adjunction $\mathbf{1}^I_{(-)} \dashv \{-\}^I$.

Lemma 6.4.6. *If $P\colon \mathbf{E} \to \mathbf{B}$ is a μPCC, then there are functors $\mathcal{L}^I\colon \mathbf{PL}_I \to \mathbf{P}_I^{\to}$ and $U^I\colon \mathbf{P}_I^{\to} \to \mathbf{PL}_I$, of which $\mathcal{L}^I$ is given by $\mathcal{L}^I(A,\varphi) = \pi^I_\varphi\colon \coprod_A \varphi \to A$. Moreover, given a functor $F\colon \mathbf{P}_I \to \mathbf{P}_J$, then there is a functor $F^{\to}\colon \mathbf{P}_I^{\to} \to \mathbf{P}_J^{\to}$ defined by $F^{\to}(f\colon A \to B) = Ff\colon FA \to FB$. Out of these, U^I and $F^{\to}$ preserve final objects and, if P has full comprehension, then also $\mathcal{L}^I$ preserves final objects.*

Proof. Functoriality of $\mathcal{L}^I$ follows readily from π^I being a natural transformation. U^I is given, as in [GJF12], by

$$U^I(f\colon A \to B) = \coprod\nolimits_f(\mathbf{1}^I_A) = \coprod\nolimits_f(A, \mathbf{1}_{\{A\}}) = \Big(B, \coprod\nolimits_{\{f\}} \mathbf{1}_{\{A\}}\Big),$$

where the coproduct $\coprod_f$ is taken in L^I (Lemma 6.4.1) The definition of U^I on morphisms is slightly more complicated, see loc. cit., we note however that it only uses properties of cartesian liftings. Finally, $F^{\to}$ is the canonical lifting of a functor to arrow categories.

The proof that U^I preserves final objects is given by Ghani et al. [GJF12, Thm. 4.14], and the preservation of final objects by $F^{\to}$ is a routine proof. That $\mathcal{L}^I$ preserves final objects means that it maps $\mathbf{1}^I_A = \big(A, \mathbf{1}_{\{A\}}\big)$ to an isomorphism. This is indeed the case here, as, by fullness, the map $\pi^I_\varphi\colon \coprod_A \mathbf{1}_{\{A\}} \to A$ is an isomorphism, see [Jac99, Ex. 10.5.4]. The rest of the proof in [GJF12] can be preserved. □

As in [FGJ11], we can combine these functors into a lifting of functors $F\colon \mathbf{P}_I \to \mathbf{P}_J$ to predicates.

Definition 6.4.7. The *canonical predicate lifting of a functor* $F\colon \mathbf{P}_I \to \mathbf{P}_J$ is given by

$$\overline{F} \coloneqq \mathbf{PL}_I \xrightarrow{\mathcal{L}^I} \mathbf{P}_I^{\to} \xrightarrow{F^{\to}} \mathbf{P}_J^{\to} \xrightarrow{U^J} \mathbf{PL}_J.$$

This extends to a *canonical predicate lifting of a signature* (F, u) with $F\colon \mathbf{P}_I \to \prod_{k=1,\dots,n} \mathbf{P}_{J_k}$ to $(\overline{F}, u)$ with $\overline{F}\colon \mathbf{PL}_I \to \prod_{k=1,\dots,n} \mathbf{PL}_{J_k}$ by putting

$$\overline{F}_k \coloneqq \mathbf{PL}_I \xrightarrow{\mathcal{L}^I} \mathbf{P}_I^{\to} \xrightarrow{F_k^{\to}} \mathbf{P}_{J_k}^{\to} \xrightarrow{U^{J_k}} \mathbf{PL}_{J_k}.$$

Note that the functor $\overline{G}_u$ associated to this signature is then given by

$$\overline{G}_u = \langle u_1^{\#}, \dots, u_n^{\#} \rangle\colon \mathbf{P}_I \to \prod\nolimits_{k=1}^{n} \mathbf{PL}_{J_k},$$

where the reindexing is now taken in the internal logic L.

The liftings $\overline{F}$ and $\overline{G}_u$ preserve fibred final objects by Lem. 6.4.6 and by reindexing preserving fibred structure, respectively. This gives us the following important property of the canonical liftings, which enables us to express a dependent iteration principle.

Lemma 6.4.8. *The canonical predicate lifting of a signature* (F, u) *is* truth-preserving*, in the sense that*

$$\overline{F} \circ \mathbf{1}^I_{(-)} \cong \left(\prod_{k=1}^{n} \mathbf{1}^{J_k}_{(-)}\right) \circ F \quad \textit{and} \quad \overline{G_u} \circ \mathbf{1}^I_{(-)} \cong \left(\prod_{k=1}^{n} \mathbf{1}^{J_k}_{(-)}\right) \circ G_u. \qquad \square$$

Let us briefly discuss what shape the dialgebras for these lifted functors take in $\mathrm{Fam}(\mathbf{Set})$.

Example 6.4.9. Recall that an object in $\mathbf{PL}_I$ is a pair (A, φ) with $\varphi \in \mathbf{P}_{\{A\}}$. Let us first spell out how $\overline{F}$ and $\overline{G}_u$ act component-wise on such objects.

$$\begin{aligned}
\overline{F}_k(A,\varphi) &= U^{J_k} F_k^{\to} \mathcal{L}^I(A,\varphi) \\
&= U^{J_k} F_k^{\to} \Big(\mathrm{fst}_\varphi\colon \coprod\nolimits_A \varphi \to A\Big) \\
&= U^{J_k} \Big(F_k(\mathrm{fst}_\varphi)\colon F_k\Big(\coprod\nolimits_A \varphi\Big) \to F_k A\Big) \\
&= \left(F_k A, \coprod\nolimits_{\{F_k(\mathrm{fst}_\varphi)\}} \mathbf{1}_{\{F_k(\coprod_A \varphi)\}}\right)
\end{aligned}$$

and

$$\begin{aligned}
\overline{G}_{u,k}(A,\varphi) &= u_k^{\#}(A,\varphi) \\
&= \left(u_k^* A, \{\overline{u_k}\, A\}^*\, \varphi\right) \\
&= (G_{u,k} A, \{\overline{u_k}\, A\}^*\, \varphi)
\end{aligned}$$

A morphism $r\colon \overline{F}(A,\varphi) \to \overline{G_u}(A,\varphi)$ in $\prod_{k=1}^{n} \mathbf{PL}_{J_k}$ is now an n-tuple of pairs $((d_1, b_1), \dots, (d_n, b_n))$, such that $(d_1, \dots, d_n)\colon FA \to G_u A$ is an (F, G_u)-dialgebra, and each b_k is a morphism

$$b_k\colon \coprod\nolimits_{\{F_k(\mathrm{fst}_\varphi)\}} \mathbf{1}_{\{F_k(\coprod_A \varphi)\}} \to \{\overline{u_k}\, A\}^*\, \varphi$$

with $\{d_k\} = P(b_k)$. Recall that comprehension in Fam(**Set**) was given by taking coproducts. Thus, for $A \in \mathbf{Set}^I$ we have $\{F_k A\} = \coprod_{j \in J_k} (F_k A)_j$ and the elements of $\{F_k A\}$ are of the form (j, x) with $j \in J_k$ and $x \in (F_k A)_j$. Given $\varphi \in \mathbf{Set}^{\{A\}}$, we have for $(j, x) \in \{F_k A\}$

$$\left(\coprod_{\{F_k(\mathrm{fst}_\varphi)\}} \mathbf{1}_{\{F_k(\coprod_A \varphi)\}}\right)_{(j,x)} = \coprod_{\substack{(j',y) \in \{F_k(\coprod_A \varphi)\} \\ j'=j \text{ and } F_k(\mathrm{fst}_\varphi)_j(y) = x}} \mathbf{1} \cong \coprod_{\substack{y \in F_k(\coprod_A \varphi)_j \\ F_k(\mathrm{fst}_\varphi)_j(y) = x}} \mathbf{1},$$

from where we obtain that the components of b_k have the following type.

$$(b_k)_{(j,x)} : \coprod_{\substack{y \in F_k(\coprod_A \varphi)_j \\ F_k(\mathrm{fst}_\varphi)_j(y) = x}} \mathbf{1} \longrightarrow \varphi_{(u_k(j), d_{k,j}(x))}.$$

Most importantly, note the use of d_k in the codomain of b_k, which comes from the fact that $\{d_k\} = P(b_k)$. The intuition is that each b_k proves, from the induction hypothesis that φ holds at the argument of the constructor d_k, that φ holds at $d_{k,j}(x)$.

Let us spell this example further out in the case when A is the family of vectors of a set B. Thus, we assume, see Example 6.2.4, that $F : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbf{1}} \times \mathbf{Set}^{\mathbb{N}}$ with $F_1 = K_{\mathbf{1}}$ and $F_2 = !^*_{\mathbb{N}}(B) \times \mathrm{Id}$, and $G_{(u_1,u_2)} = \langle z^*, s^* \rangle$. Recall that we called the constructors for vectors nil and cons, so that we now put $d_1 = \mathrm{nil}$ and $d_2 = \mathrm{cons}$. Given a predicate $\varphi \in \mathbf{Set}^{\{\mathrm{List}\, B\}}$ that we want to prove by induction, we need to provide maps b_1 and b_2 of the following type. For b_1, note that $\{F_1(\mathrm{List}\, B)\} = \coprod_{x \in \mathbf{1}} \mathbf{1} \cong \mathbf{1}$, so that there only needs to be a single map b_1 of type

$$b_1 : \coprod_{\substack{y \in \mathbf{1} \\ \mathrm{id}(y) = \star}} \mathbf{1} \longrightarrow \varphi_{(0, \mathrm{nil}(\star))}.$$

For b_2, we note that elements of $\{F_2(\mathrm{List}\, B)\}$ are given by pairs $(n, (b, x))$ where $n \in \mathbb{N}$ and $(b, x) \in B \times (\mathrm{List}\, B\, n)$. Thus, we have

$$(b_2)_{(n,(b,x))} : \coprod_{\substack{(b,u) \in F_2(\coprod_{\mathrm{List}B} \varphi)_n \\ \mathrm{fst}_\varphi(u) = x}} \mathbf{1} \longrightarrow \varphi_{(n+1, \mathrm{cons}_n(b,x))}.$$

We can now read the types of b_1 and b_2 as follows. Firstly, the map b_1 picks out a proof that φ holds for the empty vector $\mathrm{nil}(\star)$. Second, for a fixed $n \in \mathbb{N}$, $b \in B$ and $x \in \mathrm{List}\, B\, n$, we can refine the domain of $(b_2)_{(n,(b,x))}$ further. Note that $u \in (\coprod_{\mathrm{List}B} \varphi)_n$ is a pair (x', p), where $x' \in \mathrm{List}\, B\, n$ and $p \in \varphi_{(n,x')}$. The constraint that $\mathrm{fst}_\varphi(u) = x$ ensures then that $x' = x$. In other words, the domain of b_2 consists essentially of all proofs of $\varphi_{(n,x)}$. These observations allow us to rewrite the type of b_2 to the following.

$$(b_2)_{(n,(b,x))} : \varphi_{(n,x)} \to \varphi_{(n+1, \mathrm{cons}_n(b,x))}$$

In this form, we recognise immediately the assumptions of the expected induction scheme for vectors, if we use a proof-irrelevant φ as in Example 6.4.3.

$$\frac{\varphi_{(0,\mathrm{nil}(\star))} \qquad \forall n \in \mathbb{N}.\, \forall b \in B.\, \forall x \in \mathrm{List}\, B\, n.\, \varphi_{(n,x)} \to \varphi_{(n+1, \mathrm{cons}_n(b,x))}}{\forall n \in \mathbb{N}.\, \forall x \in \mathrm{List}\, B\, n.\, \varphi_{(n,x)}}$$

We will discuss the case for a predicate φ that is not defined to be proof-irrelevant in Example 6.4.11. ◀

What remains is to define dependent iteration using the canonical predicate lifting.

Definition 6.4.10. There is a functor

$$\mathrm{DiAlg}(\mathbf{1})\colon \mathrm{DiAlg}(F, G_u) \to \mathrm{DiAlg}(\overline{F}, \overline{G_u})$$

given on objects $d\colon FX \to G_uX$ by

$$\mathrm{DiAlg}(\mathbf{1})(d) = \overline{F}\Big(\mathbf{1}^I_X\Big) \cong \prod_{k=1}^{n} \mathbf{1}^{J_k}_{F(X)} \xrightarrow{\prod_{k=1}^n \mathbf{1}_d^{J_k}} \prod_{k=1}^{n} \mathbf{1}^{J_k}_{G_u(X)} \cong \overline{G_u}\Big(\mathbf{1}^I_X\Big),$$

using that the canonical lifting is truth-preserving, see Lemma 6.4.8. We say that a μPCC $P\colon \mathbf{E} \to \mathbf{B}$ admits *dependent iteration*, if $\mathrm{DiAlg}(\mathbf{1})$ preserves initial dialgebras, cf. [HJ97]. ◀

Let us explicate this form of dependent iteration for vectors, thereby continuing Example 6.4.9.

Example 6.4.11. Recall that for a predicate $\varphi \in \mathbf{Set}^{\{\mathrm{List}\, B\}}$ we were able to reduce the conditions on a dialgebra $r\colon \overline{F}(\mathrm{List}\, B, \varphi) \to \overline{G_u}(\mathrm{List}\, B, \varphi)$ with $r = ((\mathrm{nil}, b_1), (\mathrm{cons}, b_2))$ to

$$b_1\colon \mathbf{1} \to \varphi_{(0,\mathrm{nil}(\star))} \quad \text{and} \quad (b_2)_{(n,(b,x))}\colon \varphi_{(n,x)} \to \varphi_{(n+1,\mathrm{cons}_n(b,x))}.$$

If P happens to admit dependent iteration, then there is a unique homomorphism $h\colon \mathbf{1}^I_{\mathrm{List}\, B} \to (\mathrm{List}\, B, \varphi)$. Since $\mathbf{1}^I_{\mathrm{List}\, B} = (\mathrm{List}\, B, \mathbf{1}_{\{\mathrm{List}\, B\}})$, we have that $h = (f, g)$, where f and g are maps with $f\colon \mathrm{List}\, B \to \mathrm{List}\, B$ and $g\colon \mathbf{1}_{\{\mathrm{List}\, B\}} \to \varphi$. Moreover, for f the following diagram commutes.

$$\begin{array}{ccc} F(\mathrm{List}\, B) & \xrightarrow{Ff} & F(\mathrm{List}\, B) \\ {\scriptstyle (\mathrm{nil},\mathrm{cons})}\downarrow & & \downarrow{\scriptstyle (\mathrm{nil},\mathrm{cons})} \\ G_u(\mathrm{List}\, B) & \xrightarrow{G_uf} & G_u(\mathrm{List}\, B) \end{array}$$

From uniqueness of inductive extensions, we thus immediately conclude that f is the identity map. For the map g, the following diagram commutes for any choice of $n \in \mathbb{N}$, $b \in B$ and $x \in \mathrm{List}\, B\, n$.

$$\begin{array}{ccc} (\mathbf{1}, (\mathbf{1}_{\{\mathrm{List}\, B\}})_{(n,x)}) & \xrightarrow{\left(\mathrm{id}, g_{(n,x)}\right)} & (\mathbf{1}, \varphi_{(n,x)}) \\ \downarrow{\scriptstyle (\mathrm{id}_{\mathbf{1}}, \mathrm{id}_{\mathbf{1}})} & & \downarrow{\scriptstyle \left(b_1, (b_2)_{(n,(b,x))}\right)} \\ ((\mathbf{1}_{\{\mathrm{List}\, B\}})_{(0,\mathrm{nil}(\star))}, (\mathbf{1}_{\{\mathrm{List}\, B\}})_{(n+1,\mathrm{cons}_n(b,x))}) & \xrightarrow{\left(g_{(0,\mathrm{nil}(\star))}, g_{(n+1,\mathrm{cons}_n(b,x))}\right)} & (\varphi_{(0,\mathrm{nil}(\star))}, \varphi_{(n+1,\mathrm{cons}_n(b,x))}) \end{array}$$

This diagram expresses the expected computational behaviour of dependent recursion.

To sum the above discussion up, let us state the following dependent iteration rule for vectors in $\mathrm{Fam}(\mathbf{Set})$, in which we use dependent products to express the type constraints of b_1 and b_2 more conveniently.

$$\frac{b_1\colon \mathbf{1} \to \varphi_{(0,\mathrm{nil}(\star))} \qquad b_2\colon \Pi n \in \mathbb{N}.\, \Pi b \in B.\, \Pi x \in \mathrm{List}\, B\, n.\, \varphi_{(n,x)} \to \varphi_{(n+1,\mathrm{cons}_n(b,x))}}{g\colon \Pi n \in \mathbb{N}.\, \Pi x \in \mathrm{List}\, B\, n.\, \mathbf{1} \to \varphi_{(n,x)}}$$

The second diagram above expresses then indeed the expected computational behaviour:

$$g(0, \mathrm{nil}(\star))(\star) = b_1(\star)$$
$$g(n+1, \mathrm{cons}_n(b,x))(\star) = (b_2)_{(n,(b,x))}(g_{(n,x)}(\star))$$ ◀

With this, we end our exploration of dependent iteration and induction in the context of categorical dependent inductive types. It is time to move on to its dual: coinduction.

6.4.3. Coinduction

In the last section we have discussed dependent iteration principles for inductive types. Under the BHK-interpretation, these dependent iteration principles give us then also induction principles, with which we can prove predicates to hold on all elements of an inductive type. This raises of course the question what the corresponding principle for coinductive types is. Since elements of coinductive types allow the description and comparison of behaviour, Hermida and Jacobs [HJ97] observed that the canonical proof principle associated with coalgebras is concerned with proving equality. More specifically, the principle of *coinduction* grants us the possibility to prove an identity between elements of a coinductive type by establishing a bisimulation relation. The goal of this section is to provide a simple coinduction principle for coinductive types in a recursive-type closed category (μPCC).

To state what a bisimulation relation for a coinductive type is, we first need to define relations internal to a μPCC, where by relation we mean here binary relations on one carrier. Intuitively, such relations are given as predicates on the binary product of that carrier with itself. Since the carrier might be a dependent type, we need to take those dependencies into account. These considerations leads us to define the category of relations in the internal logic of P as the following pullback of L^I along Δ, where Δ is the diagonal given by $\Delta(A) = A \times_I A$ and the binary product is taken in $\mathbf{P}_I$, see Theorem 6.2.11.

$$\begin{array}{ccc} \mathrm{Rel}(\mathbf{P}_I) & \longrightarrow & \mathbf{PL}_I \\ \downarrow & \lrcorner & \downarrow{\scriptstyle L^I} \\ \mathbf{P}_I & \xrightarrow{\Delta} & \mathbf{P}_I \end{array}$$

We will usually leave out the subscript of the binary product, if the index I is understood from the context. Intuitively, $\mathrm{Rel}(\mathbf{P}_I)$ consists of objects (A, R) with $A \in \mathbf{P}_I$ and $R \in \mathbf{P}_{\{A\times A\}}$. The morphisms in $\mathrm{Rel}(\mathbf{P}_I)$ are pairs $(f, g)\colon (A, R) \to (A', R')$ with $f\colon A \to A'$, $g\colon R \to R'$ and $Pg = \{f \times_I f\}$, cf. Section 6.4.1.

To define relation liftings, Fumex et al. [FGJ11] use a so-called quotient functor $Q\colon \mathrm{Rel}(\mathbf{P}_I) \to \mathbf{P}_I$. Such a functor intuitively takes (A, R) to the quotient of A by the equivalence closure of R. We will not need a full quotient here to formulate a coinduction principle, rather it suffices to take instead the collection of pairs in A together with a proof that these are related. The reason for this is that the coinduction principle is only valid on final dialgebras, hence there is no need to take a quotient to compare states of other dialgebras under behavioural equivalence. So let us define a functor $Q\colon \mathrm{Rel}(\mathbf{P}_I) \to \mathbf{P}_I$ by

$$Q(A, R) \coloneqq \{(A \times A, R)\}^I = \coprod\!\!\coprod_{A\times A} R. \tag{6.14}$$

Similar to the weak dependent iteration principle, we will now use Q to establish a coinduction principle by appealing to the fact that coinductive types admit unique coinductive extensions. To

this end, let $\delta\colon \mathrm{Id}_{\mathbf{P}_I} \Rightarrow \Delta$ be the diagonal with $\delta_A = \langle \mathrm{id}, \mathrm{id}\rangle$, and let $\nabla^H\colon H(A\times A) \to H(A)\times_I H(A)$ be the canonical morphism given by $\nabla^H = \langle H(\pi_1), H(\pi_2)\rangle$. Using this notation, we can derive the following coinduction principle.

Proposition 6.4.12. *Let (F,u) be a signature with $F\colon \mathbf{P}_I \to \mathbf{C}$. Suppose that (A,ξ) a final (G_u,F)-dialgebra, $(A,R) \in \mathrm{Rel}(\mathbf{P}_I)$ and $d\colon G_u(Q(A,R)) \to F(Q(A,R))$, such that the following diagram commutes.*

$$\begin{array}{ccc} G_u(Q(A,R)) & \xrightarrow{\nabla^{G_u} \circ G_u(\pi_R^I)} & G_u(A)^2 \\ \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle \xi\times_I \xi} \\ F(Q(A,R)) & \xrightarrow{\nabla^{F} \circ F(\pi_R^I)} & F(A)^2 \end{array}$$

Then there is a unique $h\colon Q(A,R) \to A$ with $\delta_A \circ h = \pi_R^I$.

Proof. By definition, we have for $i = 1,2$ that $\pi_i \circ \nabla^{G_u} = G_u(\pi_i)$, hence we find

$$\pi_i \circ \nabla^{G_u} \circ G_u(\pi_R^I) = G_u(\pi_i \circ \pi_R^I),$$

and analogously for F. Thus $\pi_i \circ \pi_R^I$ is a homomorphism from d to ξ and must therefore be equal to h. Since this holds for $i = 1,2$, we have $\pi_R^I = \delta_A \circ h$. □

What is the intuition behind this coinduction principle? Given that the relation R is a bisimulation relation, which is witnessed by a dialgebra d as in the proposition, then for each related pair there is a an element of A. Moreover, this element shows exactly the behaviour that is modelled commonly by both related elements.[57] To clarify this form of coinduction further, let us instantiate Proposition 6.4.12 to streams.

Example 6.4.13. Let $F = \langle K_A, \mathrm{Id}\rangle$ and $u = (\mathrm{id}_{\mathbf{1}}, \mathrm{id}_{\mathbf{1}})$, so that streams A^ω over A are given by $\nu(G_u,F)$ with $(\mathrm{hd},\mathrm{tl})\colon G_u(A^\omega) \to F(A^\omega)$. The conditions for a dialgebra $G_u(\coprod_{A^\omega\times A^\omega}(R)) \to (A, \coprod_{A^\omega\times A^\omega}(R))$ over a relation R on A^ω then capture the usual bisimulation proof principle: Suppose we have (d_1,d_2) with $d_1\colon \coprod_{A^\omega\times A^\omega}(R) \to A$ and $d_2\colon \coprod_{A^\omega\times A^\omega}(R) \to \coprod_{A^\omega\times A^\omega}(R)$. Then the conditions read as $\delta \circ d_1 = \mathrm{hd} \circ \pi_R$ and $\pi_R \circ d_2 = (\mathrm{tl}\times\mathrm{tl}) \circ \pi_R$. The first condition expresses thereby that the two related streams must have the same head, which is actually computed by d_1. As for the second condition, this says that d_2 maps a proof of two streams being related by R to a proof that their tails are again related by R. The reader will recognise these conditions as the usual ones for bisimulation relations. ◀

Note that this coinduction principle allows us to prove external equality, which is the equality of morphisms in $\mathbf{E}$, between elements of coinductive types. This includes also the (dependent) function space, see Theorem 6.2.11. In particular, we can show in this case that functions are equal by proving that they agree on every argument. Hence, for a theory in which functions are only externally equal if they are convertible, we need to relax the requirement that coinductive types are final dialgebras. This is not so surprising, and we will discuss in Section 6.6 possibilities to overcome this mismatch between category theory and syntactic type theory.

6.5. A Beck-Chevalley Condition for Recursive Types

As already mentioned in the introduction, if we want to recover the expected property that substitutions distribute over sum types, then we need to require that the so-called Beck-Chevalley condition holds for coproducts. More specifically, given a term t and variables x and y, such that x is distinct from all the variables in t and from y, one defines substitution on sum types by

$$(\Sigma x : A[y].\, B[x,y])[t/y] = \Sigma x : A[t].\, B[x,t].$$

Notice that there is something subtle is going on here. First, the sum on the right is a different one because the type of the variable x has changed. Second, we assumed that the variable x is distinct from all other variables around, which allows to avoid having to rename x. More generally, we could introduce a fresh name, rename the bound occurrences of x in B and then define the substitution on sums by

$$(\Sigma x : A[y].\, B[x,y])[t/y] = \Sigma x' : A[t].\, B[x',t]. \tag{6.15}$$

We will now use the classifying fibration for a dependent type theory, which we introduced in Section 6.1, to explain how the Beck-Chevalley condition captures this definition of substitution category theoretically. Recall that we defined the coproduct as a left-adjoint functor to the projections $\pi_A \colon \{A\} \to PA$. In the context of the classifying fibration for a dependently typed calculus, we defined comprehension as context extension and the projections are the weakening substitutions. More precisely, for a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, a type A with $\Gamma \vdash A : \mathbf{Type}$ and a fresh variable x comprehension is given by $\{\Gamma \vdash A : \mathbf{Type}\} = \Gamma, x : A$. The corresponding projection $\pi_A \colon \Gamma, x : A \to \Gamma$ is given by $\pi_A = (x_1, \ldots, x_n)$. This allows us to express the equation (6.15) in category theoretical terms by using the coproduct as follows. Suppose that $y : C \vdash A : \mathbf{Type}$ and $\Gamma \vdash t : C$, then equation (6.15) reads as

$$\Big(\coprod_{\pi_A} B[x,y]\Big)[t/y] = \coprod_{\pi_{A[t]}} B[x',t].$$

Using that reindexing amounts to the application of substitutions, we can further rewrite this to:

$$t^*\left(\coprod_{\pi_{A[y]}} B[x,y]\right) = \coprod_{\pi_{A[t]}} (t,x')^*(B[x,y]).$$

Note that the involved substitutions and projections can be organised in the following pullback diagram.

$$\begin{array}{ccc} \Gamma, x' : A[t] & \xrightarrow{\pi_{A[t]}} & \Gamma \\ {\scriptstyle (t,x')}\downarrow & \lrcorner & \downarrow{\scriptstyle t} \\ y : C, x : A & \xrightarrow{\pi_A} & y : C \end{array}$$

The fact that this is a pullback diagram captures the essence of the variable renaming and thereby also the interaction between the involved coproduct and reindexing functors.

Let us now generalise this interaction between reindexing and coproducts to arbitrary coproducts. The above considerations lead us then to assume that we are given a pullback square such as the

following.

$$\begin{array}{ccc} K & \xrightarrow{g} & J \\ {\scriptstyle v}\downarrow & \lrcorner & \downarrow{\scriptstyle u} \\ L & \xrightarrow{f} & I \end{array}$$

Given such a commuting square, we get an isomorphism $v^* \circ f^* \cong g^* \circ u^*$. Together with the unit $\eta^f : \mathrm{Id} \Rightarrow f^* \coprod_f$ and the counit $\varepsilon^g : \coprod_g g^* \Rightarrow \mathrm{Id}$, we obtain thus the following canonical morphism.

$$\coprod_g v^* \xRightarrow{\coprod_g v^* \eta^f} \coprod_g v^* f^* \coprod_f \xRightarrow{\cong} \coprod_g g^* u^* \coprod_f \xRightarrow{\varepsilon^g u^* \coprod_f} u^* \coprod_f$$

The *Beck-Chevalley condition for coproducts* requires now that this morphism has an inverse. This allows us to generalise the definition of substitution in (6.15) to the following isomorphism.

$$u^* \coprod_f \cong \coprod_g v^*$$

The aim of this section is now to generalise the Beck-Chevalley condition to arbitrary recursive types, thereby enabling the expected notion of substitution on recursive types. After having introduced the generalised Beck-Chevalley condition, we prove that it is equivalent to the usual one for products and coproducts. Moreover, we show that binary products, binary coproducts and final objects are fibred if we construct these as recursive types that satisfy our generalised Beck-Chevalley condition.

We start by developing some notation that we will need to define the Beck-Chevalley condition for recursive types. Let (F, u) be a signature with $F \colon \mathbf{C} \times \mathbf{P}_I \to \mathbf{D}$ and $u_i \colon J_i \to I$ for $\mathbf{D} = \prod_{i=1}^n \mathbf{P}_{J_i}$. Assume that we are given for each $1 \leq i \leq n$ a pullback square as follows.

$$\begin{array}{ccc} L_i & \xrightarrow{w_i} & J_i \\ {\scriptstyle x_i}\downarrow & \lrcorner & \downarrow{\scriptstyle u_i} \\ K & \xrightarrow{v} & I \end{array} \tag{6.16}$$

Note that we are now given a pullback for each constructor/destructor in contrast to the situation for the Beck-Chevalley condition on coproducts that we described above. From the pullbacks in (6.16), we obtain isomorphisms

$$\iota^{u,i} \colon x_i^* \circ v^* \cong (v \circ x_i)^* = (u_i \circ w_i)^* \cong w_i^* \circ u_i^*, \tag{6.17}$$

since P is a cloven fibration. We can merge all morphisms that occur in the pullbacks together into functors $G_x \colon \mathbf{P}_K \to \prod_{i=1}^n \mathbf{P}_{L_i}$ and $W \colon \prod_{i=1}^n \mathbf{P}_{J_i} \to \mathbf{P}_{L_i}$ that we define by $G_x = \langle x_1^*, \ldots, x_n^* \rangle$ and $W = w_1^* \times \cdots \times w_n^*$. If we put $\iota^{u,x} = \iota^{u,1} \times \cdots \times \iota^{u,n}$, which is given for each $A \in \mathbf{P}_I$ by

$$(\iota^{u,1} \times \cdots \times \iota^{u,n})_A = (\iota_A^{u,1}, \ldots, \iota_A^{u,n}) \qquad \text{in } \prod_{i=1}^n \mathbf{P}_{L_i},$$

then we obtain an isomorphism

$$\iota^{u,x} \colon G_x \, v^* \cong W G_u.$$

Let us now assume that we are given a functor $F^{\upsilon}\colon \mathbf{C}\times\mathbf{P}_K \to \prod_{i=1}^{n}\mathbf{P}_{L_i}$ and another isomorphism $\iota^F\colon F^{\upsilon}(\mathrm{Id}\times\upsilon^*) \Rightarrow WF$, as indicated in the following diagram.

$$\begin{array}{ccc} \mathbf{C}\times\mathbf{P}_I & \xrightarrow{F} & \prod_{i=1}^{n}\mathbf{P}_{J_i} = \mathbf{D} \\ {\scriptstyle \mathrm{Id}\times\upsilon^*}\downarrow & \overset{\cong}{\swarrow} & \downarrow{\scriptstyle W} \\ \mathbf{C}\times\mathbf{P}_K & \xrightarrow{F^{\upsilon}} & \prod_{i=1}^{n}\mathbf{P}_{L_i} \end{array}$$

Using this data, we can define a functor $\mathfrak{C}^{\upsilon}\colon \mathrm{DiAlg}\big(\widehat{G_u},\widehat{F}\big) \to \mathrm{DiAlg}\big(\widehat{G_x},\widehat{F^{\upsilon}}\big)$ as follows. Suppose that $\delta\colon \widehat{G_u}(H) \Rightarrow \widehat{F}(H)$ is a dialgebra, then $\mathfrak{C}^{\upsilon}(\delta)$ is the composition

$$\begin{aligned} \widehat{G_x}(\upsilon^* H) &= G_x\,\upsilon^* H \\ &\overset{\iota^u H}{\Longrightarrow} WG_uH = W\widehat{G_u}(H) \\ &\overset{W\delta}{\Longrightarrow} W\widehat{F}(H) = WF\langle \mathrm{Id}_{\mathbf{C}}, H\rangle \\ &\overset{(\iota^F)^{-1}\langle \mathrm{Id}_{\mathbf{C}}, H\rangle}{\Longrightarrow} F^{\upsilon}(\mathrm{Id}_{\mathbf{C}}\times\upsilon^*)\langle \mathrm{Id}_{\mathbf{C}}, H\rangle = F^{\upsilon}\langle \mathrm{Id}_{\mathbf{C}}, \upsilon^* H\rangle = \widehat{F^{\upsilon}}(\upsilon^* H). \end{aligned}$$

For a dialgebra homomorphism $\beta\colon H_1 \Rightarrow H_2$ from δ_1 to δ_2, we put $\mathfrak{C}^{\upsilon}(\beta) = \upsilon^*\beta$, which is a $(\widehat{G_x},\widehat{F^{\upsilon}})$-dialgebra homomorphism since the following diagram commutes.

$$\begin{array}{ccc} \widehat{G_x}(\upsilon^* H_1) & \xRightarrow{\widehat{G_x}(\upsilon^*\beta)} & \widehat{G_x}(\upsilon^* H_2) \\ {\scriptstyle \iota^u H_1}\Downarrow & \iota^u \text{ nat.} & \Downarrow{\scriptstyle \iota^u H_2} \\ W\widehat{G_u}(H_1) & \xRightarrow{W\widehat{G_u}(\beta)} & W\widehat{G_u}(H_2) \\ {\scriptstyle W\delta_1}\Downarrow & \beta \text{ hom.} & \Downarrow{\scriptstyle W\delta_2} \\ W\widehat{F}(H_1) & \xRightarrow{W\widehat{F}(\beta)} & W\widehat{F}(H_2) \\ {\scriptstyle (\iota^F)^{-1}\langle \mathrm{Id}, H_1\rangle}\Downarrow & \iota^F \text{ nat.} & \Downarrow{\scriptstyle (\iota^F)^{-1}\langle \mathrm{Id}, H_2\rangle} \\ \widehat{F^{\upsilon}}(\upsilon^* H_1) & \xRightarrow{\widehat{F^{\upsilon}}(\upsilon^*\beta)} & \widehat{F^{\upsilon}}(\upsilon^* H_2) \end{array}$$

The functor laws for $\mathfrak{C}^{\upsilon}$ follow from functoriality of υ^*. In the dual way, we can also define a functor $\mathfrak{I}^{\upsilon}\colon \mathrm{DiAlg}\big(\widehat{F},\widehat{G_u}\big) \to \mathrm{DiAlg}\big(\widehat{F^{\upsilon}},\widehat{G_x}\big)$ by

$$\begin{aligned} \mathfrak{I}^{\upsilon}(\delta\colon \widehat{F}(H) \Rightarrow \widehat{G_u}(H)) &= (\iota^u)^{-1}H \circ W\delta \circ \iota^F\langle \mathrm{Id}_{\mathbf{C}}, H\rangle \\ \mathfrak{I}^{\upsilon}(\beta) &= \upsilon^*\beta. \end{aligned}$$

These two functors allow us to define what it means that recursive types are preserved by reindexing.

Definition 6.5.1. A coinductive type (Ω,ξ) for (F,G_u) fulfils the *Beck-Chevalley condition*, if for every family of pullbacks $\upsilon\times_I u_i$ as in (6.16), there is a functor F^{υ} and an isomorphism ι^F as described above, such that the coinductive extension $h\colon\ \upsilon^*\Omega \Rightarrow \Omega^{\upsilon}$ of $\mathfrak{C}^{\upsilon}(\xi)$ to the coinductive type $(\Omega^{\upsilon},\xi^{\upsilon})$

for (F^υ, G_x), as in the following diagram, is an isomorphism $\mathfrak{C}^\upsilon(\xi) \cong \xi^\upsilon$ of dialgebras.

$$\begin{array}{ccc} \widehat{G_x}(\upsilon^*\,\Omega) & \xRightarrow{\widehat{G_x}(h)} & \widehat{G_x}(\Omega^\upsilon) \\ {\scriptstyle \mathfrak{C}^\upsilon(\xi)}\Downarrow & & \Downarrow{\scriptstyle \xi^\upsilon} \\ \widehat{F^\upsilon}(\upsilon^*\,\Omega) & \xRightarrow{\widehat{F^\upsilon}(h)} & \widehat{F^\upsilon}(\Omega^\upsilon) \end{array}$$

The Beck-Chevalley condition for inductive types is defined dually using $\mathfrak{I}^\upsilon$.

As the name suggests, this definition is supposed to capture the known Beck-Chevalley conditions. To prove this, we first need a small technical following.

Lemma 6.5.2. *Let $F_1 \colon \mathbf{C}_1 \to \mathbf{C}_2$ be a functor and (F_2, u) be a signature with $F_2 \colon \mathbf{C}_2 \times \mathbf{P}_I \to \mathbf{C}_3$. We define a functor $F \colon \mathbf{C}_1 \times \mathbf{P}_I \to \mathbf{C}_3$ by $F = F_2 \circ (F_1 \times \mathrm{Id})$. With this definition, we have $\mu(\widehat{F}, \widehat{G_u}) = \mu(\widehat{F_2}, \widehat{G_u}) \circ F_1$ and $\alpha^{(\widehat{F}, \widehat{G_u})} = \alpha^{(\widehat{F_2}, \widehat{G_u})} F_1$. Dually, we also have that $\nu(\widehat{G_u}, \widehat{F}) = \nu(\widehat{G_u}, \widehat{F_2}) \circ F_1$ and $\xi^{(\widehat{F}, \widehat{G_u})} = \xi^{(\widehat{F_2}, \widehat{G_u})} F_1$.*

Proof. This follows simply from the definitions. Let $V \in \mathbf{C}_1$, then

$$\begin{aligned} \mu(\widehat{F}, \widehat{G_u})(V) &= \mu(F(V,-), G_u) = \mu((F_2 \circ (F_1 \times \mathrm{Id}))(V,-), G_u) \\ &= \mu(F_2(F_1(V),-), G_u) = \mu(\widehat{F_2}, \widehat{G_u})(F_1(V)) \\ &= (\mu(\widehat{F_2}, \widehat{G_u}) \circ F_1)(V) \end{aligned}$$

and the equality of the natural transformations follows immediately from their definition and the functor equality above. □

We can now show the usual Beck-Chevalley condition and the one given in Definition 6.5.1 are equivalent for the adjunctions we obtained as recursive types in Theorem 6.2.11.

Theorem 6.5.3. *The adjunctions constructed in Theorem 6.2.11 are fibred iff they fulfil the Beck-Chevalley condition for recursive types.*

Proof. We prove this for the constructed right adjoints, the proof for the left adjoints follows by duality. Recall that we defined for $\mathbf{C} = \prod_{i=1}^n \mathbf{P}_{J_i}$, a right adjoint for all $G_u = \langle u_1^*, \ldots, u_n^* \rangle$ by $\nu(\widehat{G_u}, \widehat{\pi_1}) \colon \mathbf{D} \to \mathbf{P}_I$ with $\pi_1 \colon \mathbf{C} \times \mathbf{P}_I \to \mathbf{C}$. Given pullbacks $(x_i, w_i) = \upsilon \times_I u_i$ as in (6.16), we put $\mathbf{D} = \prod_{i=1}^n \mathbf{P}_{L_i}$, $W \colon \mathbf{C} \to \mathbf{D}$ with $W = w_1^* \times \cdots \times w_1^*$ and $G_x = \langle x_1^*, \ldots, x_n^* \rangle$. We find the right adjoint of G_x to be $\nu(\widehat{G_x}, \widehat{\pi_1'})$ with $\pi_1' \colon \mathbf{D} \times \mathbf{P}_K \to \mathbf{D}$. The Beck-Chevalley condition (B-C condition) for adjoint functors requires that the canonical natural transformation $\gamma \colon \upsilon^* \circ \nu(\widehat{G_u}, \widehat{\pi_1}) \Rightarrow \nu(\widehat{G_x}, \widehat{\pi_1'}) \circ W$ is an isomorphism. We show that this coincides with our definition for recursive types.

Let ξ be the final $(\widehat{G_u}, \widehat{\pi_1})$-dialgebra and ξ^υ the final $(\widehat{G_x}, \widehat{F^\upsilon})$-dialgebra. We note that for the projection $\pi_1'' \colon \mathbf{C} \times \mathbf{P}_K \to \mathbf{C}$, we have $\pi_1' \circ (W \times \mathrm{Id}) = W\pi_1''$. Thus, we can choose for the B-C condition $F^\upsilon = W\pi_1''$ and get, by Lemma 6.5.2, $\nu(\widehat{G_x}, \widehat{F^\upsilon}) = \nu(\widehat{G_x}, \widehat{\pi_1'}) \circ W$. By finality, this means that γ is in fact the unique morphism from $\mathfrak{C}^\upsilon(\xi)$ to ξ^υ. The B-C condition for recursive types requires this to be an isomorphism, just as the B-C condition for adjoints does. This means that the constructed adjunctions are fibred iff they satisfy the B-C condition for recursive types with the choice $F^\upsilon = W\pi_1''$. □

In the same way, we can relate fibred final objects and our Beck-Chevalley condition.

Theorem 6.5.4. *Terminal objects* $\mathbf{1}_I$*, constructed as* $v(\mathrm{id}_I^*, \mathrm{Id}_{\mathbf{P}_I})$*, are fibred iff they satisfy the Beck-Chevalley condition for recursive types.*

Proof. We note that the pullback (6.16) degenerates, in this case, to only give a morphism $v\colon K \to I$. Then we pick $\mathrm{Id}^v = v^*$ and the rest follows as in Theorem 6.5.3. □

6.6. Discussion

In the previous Chapter 5 we established several approaches to reasoning about mixed inductive-coinductive programs. Among these approaches was a syntactic logic $\mathbf{FOL}_{\blacktriangleright}$ together with a proof system. We realised at the end of Section 5.2 that we could implement the standard induction and coinduction principles for non-mutual inductive or coinductive types but not for more general types, see Note 39, since the logic $\mathbf{FOL}_{\blacktriangleright}$ is lacking the possibility to form fixed point formulas. However, adding fixed point formulas would force us to also add the corresponding proof rules to the logic, which are a mere adaption of the term formation rules for fixed point types in the calculus $\lambda\mu\nu$. To save ourselves from duplicating rule sets and from repeating proofs of properties for the logic, we decided to merge programming and reasoning into one language.

In the present chapter, we established a dependent type theory in the language of category theory, in which we can write the same programs as in the calculus $\lambda\mu\nu$, as well as reason about these programs. The idea of this category theoretical language was to extend the ideas of Hagino [Hag87] from a simply typed calculus to a dependently typed one. Specifically, this meant that we viewed inductive and coinductive types as initial and final dialgebras in the fibres of a fibration. These fibres replaced thereby the simple categories that Hagino used and allowed us to manipulate the dependencies, as we have seen in the principle example of vectors in Section 6.2.1. Starting from this view on dependent types, we developed notions of signatures and strictly positive types that form a general basis for dependent inductive-coinductive types. All these considerations led us to the definition of recursive type closed categories (μPCC) as a theory of inductive-coinductive dependent types. Besides the truly recursive types like vectors, (partial) streams, bisimilarity, and the substream relation, we were also able to encode the standard connectives of Martin-Löf type theory as (non-recursive) inductive and coinductive types. This will enable us in the next Chapter 7 to construct a syntactic type theory that is purely based on recursive types, while still subsuming standard Martin-Löf type theory.

After we introduced the basic category theoretical setup, we showed that the developed theory is actually sensible by showing that the initial and final dialgebras can be constructed as initial algebras and final coalgebras of polynomial functors. This opens up a wide class of models for the theory. Moreover, we demonstrated that the unique mapping properties of inductive and coinductive types give rise to a simple dependent iteration (or induction) principle and a coinduction principle. Under the further assumption that coproducts are strong, that is, they admit projections for both components, we could also obtain a proper dependent iteration principle. Finally, we established a Beck-Chevalley condition for recursive types to allow the interpretation of a corresponding syntactic theory of inductive-coinductive dependent types in a μPCC.

Related Work

Let us briefly discuss existing work that is related to the content of this chapter. Most of the relevant work has been already discussed in the corresponding places, but a more streamlined overview seems appropriate.

Categorical Dependent Recursive Types The basic setup, which we chose here to construe dependent inductive-coinductive types category theoretically, is essentially a combination of existing ideas. Specifically, we took Hagino's dialgebra approach for simple types [Hag87] and merged it with the fibrational view on dependent types [Jac99]. This fibrational view is, in my opinion, the most modular and transparent presentation of dependent types in terms of categories, since all requirements made on a fibration have a clearly defined purpose. It is this modularity that makes fibrations very suitable for reusing existing notions in the context of inductive-coinductive dependent types. As mentioned at the end of Section 6.1, there is an abundant supply of category theoretical models for dependent types. However, since these are specifically tailored towards dependent types, it is often the case that the straightforward approach of using dialgebras cannot be implemented there. Moreover, many of these models are related or even equivalent to fibrational models [Jac93]. Finally, we should mention that there are other category theoretical approaches to recursive dependent types, but none of them actually treats coinductive dependent types, see the introduction of Section 6.2.3.

Constructing Recursive Types as Polynomials Polynomial functors are the go-to class of functors that is used whenever reasonably general but also well-behaved class of functors are required. One of the main reasons for that is that they preserve certain limits and colimits [GK13], which are important for constructing initial algebras [GH04; GK13; Koc11] and final coalgebras [Jac16; vdBdM07]. Another reason is that it is straightforward to associate a syntactic theory to initial algebras [Fio12; HF11] and to final coalgebras [BRS09; Gol01] of polynomial functors, since these are, respectively, finite and potentially infinite trees. This latter reason is also why polynomial functors [GK13; HF11] or container [Abb03; AAG05] are taken as foundation for data types. Fortunately, this interest led to many existing results that we were able to use in our reduction of theory of dependent recursive types to polynomial functors in Section 6.3. The main sources that we used here were [GK13] and [AAG05].

Internal Reasoning Principles Both induction and coinduction principles have been studied extensively in the context of category theory. The clearest expositions on these principles are [HJ97] and the extension [FGJ11] thereof to indexed sets. In Section 6.4, we instantiated some of the results from these two publications and proved other results under weaker conditions. This is discussed in detail in the corresponding places in Section 6.4.

A Beck-Chevalley Condition for Recursive Types The Beck-Chevalley condition is well- and widely known for products and coproducts, but our analogue for recursive types has not been studied anywhere. That being said, since products and coproducts arise as recursive types, one may ask how the classical Beck-Chevalley condition relates to ours in those cases. Indeed, in Theorem 6.5.3 we showed that the classical condition is equivalent to ours for the presentation of products and coproducts as recursive types.

Contributions

After summarising the content and context of the chapter, let us also clearly mark the contributions made in this chapter. First of all, the adaption of Hagino's categorical data types to dependent types has not been explored before. Moreover, the resulting theory explains the types of Martin-Löf type theory in terms of inductive-coinductive types, which was considered to some extend folklore but has never been formally explored somewhere. The construction of recursive types through polynomials required one new result (Theorem 6.3.10), which is the main contribution of Section 6.3. Next, the internal logic of a dependent type theory has, to the best of my knowledge, not been presented and studied as we did in Section 6.4. In particular, the resulting (weak) induction and coinduction principles were not studied in the absence of strong coproducts and quotients. Since there was no category theoretical study of general dependent inductive-coinductive types, also the Beck-Chevalley condition from Section 6.5 is a novel contribution.

Future Work

Much of the content in this chapter is fairly stable and explored. There are a few directions for future research though. As we will see in Chapter 7, the (unique) mapping principle of the recursive types from Section 6.2 correspond to simple iteration and coiteration rules on the syntactic side. The problem with this is that it usually much easier to write programs by giving a set of equations, similar to what we found in Section 3.2. This raises of course the question whether there is a way to give equational specifications of morphisms on dependent recursive types. A possible approach might be to adapt, at least in the coinductive case, completely iterative algebras [Acz+03] to dependent types. However, the equational specifications there need to be extended with pattern matching for the specification of functions on inductive types, and one needs to be careful with identity types, as pattern matching might lead to stronger than expected theories, cf. [CDP16; GMM06].

Since in Section 6.4 we found the internal reasoning principles were based on the fact that inductive and coinductive types come with unique mapping principles, we concluded that the category theoretical approach we took so far makes too strong requirements on types. The reason is that uniqueness of inductive and coinductive extensions are problematic in syntactic theories because it usually breaks decidability of type checking. A possibility to overcome this problem is to make uniqueness proof-relevant, in the sense that whenever we show the uniqueness of a homomorphism, then there is an explicit proof object in the category that witnesses this fact. This can be realised, for example, by using 2-categories or higher generalisations thereof. But since this takes us to far astray, we leave the problem of matching the category theoretical semantics better intensional type theories for the future.

Notes

[41] Translation taken from "Substance and Function and Einsteins Theory of Relativity" by E. Cassirer, p.371, 1923.

[42] For instance, having both projections for sum types in an impredicative calculus like the Calculus of Constructions, leads to inconsistencies, see [Coq89]. More precisely, the fact that there is a

proposition isomorphic to the coproduct of a family of propositions is independent of having an impredicative universe of propositions [Str89]. Similarly, a logic that combines computational classical logic in form of control operators with strong sum types also becomes inconsistent, as has been shown by Herbelin [Her05].

[43] The alert reader might note that we only obtain $(\mathrm{List}_A\,(n+1))[2] = \mathrm{List}_A\,(2+1)$. To get to $\mathrm{List}_A\,3$ we actually need to use the computation $2+1 \equiv 3$ inside the type. Since this is not really relevant to the present introduction, we sweep this problem under the rug for now and come back to it in the next chapter.

[44] A more modern notation for dependent function types, which emphasises the close relation to the simple function type, is $(x : A) \to B[x]$ for $\Pi x : A.\, B[x]$. However, since the dependent function space is given category theoretically as a product, we stick to the traditional notation here.

[45] A historical overview can be found in [TvD88, Sec. 1.4].

[46] In modern expositions, System F is called $\lambda 2$, see for example [NG14].

[47] The reverse direction of

$$\exists x : A.\, \forall y : B.\, \varphi[x, y] \vdash \forall y : B.\, \exists x : A.\, \varphi[x, y]$$

is the constructive version of the axiom of choice:

$$\alpha : (\forall x : A.\, \exists y : B.\, \varphi[x, y]) \vdash p : (\exists f : A \to B.\, \forall x : A.\, \varphi[x, f\,x]).$$

To give such a proof p, we need to assume though something more about the existential quantifier, namely that there are projections $z : (\exists x : A.\, \psi[x]) \vdash \pi_1\, z : A$ and $z : (\exists x : A.\, \psi[x]) \vdash \pi_2\, z : \psi[\pi_1\, z]$. How these can be obtained is discussed towards the end of Section 6.1 and in the next chapter. If we have these projections, then the proof p is given by

$$p \coloneqq \big\langle \lambda x.\, \pi_1\,(\alpha\, x), \lambda x.\, \pi_2\,(\alpha\, x) \big\rangle.$$

[48] The propositions-as-types interpretation is often also referred to as *Curry-Howard correspondence*. I will, however, avoid this terminology here, since it is neither descriptive nor very accurate. There are many more people that would have to be mentioned in this name, first and foremost, Church is missing from the list.

[49] An exception is Frege's definite description operator ι that allows one to name an object that uniquely satisfies a proposition, see e.g. [TvD88, Par. 2.2.9] and [NG14, Sec. 12.7].

[50] Of course, we can see isHoliday also as a predicate over the date type that we introduced above. However, we use the present definition for illustrative purposes later.

[51] The adjunction $\coprod_I \dashv !^*$ consists in fact also of some equations that need to hold. As it turns out, one of these equations gives the computation rule for sum types: **unpack** $\langle u, v\rangle$ **as** $\langle i, x\rangle$ **in** $t \longrightarrow t[u/i, v/x]$. The other equation ensures that **unpack** y **as** $\langle i, x\rangle$ **in** t is the unique term with that computational property, and as such is usually not part of a syntactic theory. Thus, the category theoretical

approach makes slightly to strong requirements here, see also the discussion in Section 4.2 for the analogous situation in non-dependent calculi. This can be solved in the case of sum types by allowing η-conversions: **unpack** s **as** $\langle i, x\rangle$ **in** $t[\langle i, x\rangle/y] \longrightarrow t[s/y]$. However, η-conversions are difficult to manage and do not generalise well to recursive types.

[52] There are two principal ways to conceive of vectors: Either we group the vectors of the same length into a set, thereby forming a family of sets, or we see vectors as the set of all lists together with the map that assigns to each list its length. In Example 6.2.1, we use the first view, but we will later return to the second when we consider dependent types as objects in slice categories.

[53] For an Agda-versed reader it might be interesting to note that the use of non-trivial substitution in the domain of destructors is currently not available in Agda. It is questionable though whether adding this capability is useful in an intensional theory like Agda. The reason is that if we are given $n : \mathbb{N}^\infty$ and $t : \mathrm{PStr}\, A\, n$, then to apply hd to t we would have to establish that $n = s_\infty\, k$ for some $k : \mathbb{N}^\infty$. However, since $\mathbb{N}^\infty$ is a coinductive type, we will rarely find ourselves in the luxury position that we can prove an equality, rather we may only find that n is bisimilar to $s_\infty\, k$ for some k. But an even better way would be to define PStr differently:

$$
\begin{array}{l}
\textbf{codata}\ \mathrm{n} : \mathbb{N}^\infty \vdash \mathrm{PStr}\ \mathrm{A}\ \mathrm{n}\ \textbf{where} \\
\quad \mathrm{hd}\ :\ \mathrm{PStr}\ \ \mathrm{A}\ \mathrm{n} \longrightarrow (\Pi\, \mathrm{k} : \mathbb{N}^\infty.\, (\mathrm{n}\ .\mathrm{out} = \kappa_2\ \mathrm{k}) \longrightarrow \mathrm{A}) \\
\quad \mathrm{tl}\ \ :\ \mathrm{PStr}\ \ \mathrm{A}\ \mathrm{n} \longrightarrow (\Pi\, \mathrm{k} : \mathbb{N}^\infty.\, (\mathrm{n}\ .\mathrm{out} = \kappa_2\ \mathrm{k}) \longrightarrow \mathrm{PStr}\ \mathrm{A}\ \mathrm{k})
\end{array}
$$

Given an element $s : \mathrm{PStr}\, A\, n$ for some $n : \mathbb{N}^\infty$, we can always apply, say, tl to s. This gives us then a term of type $\Pi\, k.\, (n.\mathrm{out} = \kappa_2\ k) \longrightarrow \mathrm{PStr}\, A\, k$. Thus, if we have a k of type $\mathbb{N}^\infty$ and a proof p for $n.\mathrm{out} = \kappa_2\ k$, then we obtain the tail of s by $\mathrm{tl}\, s\, k\, p : \mathrm{PStr}\, A\, k$. Such a definition works much better in an intensional theory like Agda and, in contrast to a bisimilarity-based definition, still enables the use of automatic reductions.

[54] Even though μP-complete categories offer more flexibility for the domain of the destructors of a coinductive type, see the note 53, the coinductive types in Agda provide some further power. More specifically, the destructors of a coinductive type in Agda can refer to previously declared destructors. This allows the declaration of a strong dependent sum, see [Jac99, Sec. 10.1 and Def. 10.5.2]. We discuss this in Section 6.4, and see also Note 47.

[55] If $X \in \mathbf{B}/I$ and $Y \in \mathbf{B}/J$, then a morphism $f : X \longrightarrow Y$ in $\mathbf{B}$ can be defined in the internal language of the codomain fibration by statements of the form "$f_i\, p = e$", where p is a pattern and e an expression that are given by the following grammar.

$$
\begin{aligned}
e &::= x \mid f \mid e\ e \mid \lambda p.\, e \mid (e, e) \\
p &::= x \mid x : T \mid (p, p) \mid (p \mid c) \\
c &::= e = e \\
T &::= I \mid X_e
\end{aligned}
$$

The intention is that x is a variable, f a morphism in $\mathbf{B}$, $\lambda p.\, e$ an element of a product, and (e, e) an element of a coproduct. A pattern is either a, possibly typed, variable, or it matches dependent pairs of a coproduct, or it introduces some constraints that arise from products or coproducts. Those constraints may just be identities between expressions that can involve only the variables of the

corresponding pattern. Finally, a type is either an object I of $\mathbf{B}$ or it selects a fibre X_e of X by means of an expression. Typing constraints can then be given in the form $e_1 : X_{e_2}$, which expresses that $X\, e_1 = e_2$, using the fact that X is a morphism $U \to I$ for some object U in $\mathbf{B}$. Expressions, patterns and constraints are all subject to the obvious typing constraints. For instance, given a morphism $h\colon I \to J$ in $\mathbf{B}$ and a morphism $f\colon X \to Y$ in $\mathbf{B}/_I$, we can define $\coprod_h f\colon \coprod_h X \to \coprod_h Y$ by

$$\left(\coprod\nolimits_h f\right)_j (i, x : X_i \mid h\, i = j) = (i, f\, x).$$

We now need to show that $(i, f\, x) : (\coprod_h Y)_j$, i.e., $h(Y\, (f\, x)) = j$. Since f is a morphism in $\mathbf{B}/_I$, we have that $Y \circ f = X$. This gives us the required identity by using the constraint $h\, i = j$ in the pattern of $\coprod_h f$: $h\, (Y\, (f\, x)) = h\, (X\, x) = h\, i = j$.

[56] Besides enabling induction, the dependent iteration allows also for interesting computational specifications. For example, primitive recursion can be implemented much more efficiently by using dependent iteration. In particular, the predecessor map $\mathrm{pred}\colon \mathrm{Nat} \to \mathrm{Nat}$ is given by $\mathrm{iter}(0, \lambda n.\, k)$ in the notation of the introduction of Section 6.4.2, where $k : \mathrm{Nat} \vdash \lambda n.\, k : \mathrm{Nat}$. The computational rules give then $\mathrm{pred}\, 0 = 0$ and $\mathrm{pred}\, (s\, n) = (\lambda n.\, k)[n]\, (\mathrm{pred}\, n) = n$, as expected.

[57] This is very similar to the span-based definition of bisimilarity that is studied in the context of the theory of coalgebras, see e.g. [Sta11].

Constructive Logic Based on Inductive-Coinductive Types

> *I would like to argue [...] that without a system of formal constraints there are no creative acts; specifically, in the absence of intrinsic and restrictive properties of mind, there can be only 'shaping of behavior' but no creative acts of self-perfection.*
>
> — Noam Chomsky, "Language and Freedom", 1970.[58]

In Chapter 6, we have established a category theoretical approach to dependent type theory that was solely based on inductive-coinductive types. The key idea was to model inductive types as initial dialgebras and coinductive types as final dialgebras in fibres of a fibration. Moreover, we singled out functors that describe the domain and codomain of constructors and destructors of strictly positive, recursive types. This allowed us to construct models for these categorical types in terms of polynomial functors on the slices of certain locally Cartesian closed categories.

The goal of this final chapter is to find a syntactic type theory that matches the category theoretical setup of Chapter 6 and show that this theory is consistent. This dependent type theory is centred around recursive types that correspond to the initial and final dialgebras for strictly positive signatures. To show that the theory is consistent, we give proofs of subject reduction (types are preserved by reduction steps) and of strong normalisation. These are the main results of the present chapter. However, to demonstrate also the usefulness of the type system, we give plenty of examples in Section 7.2, including an encoding of the propositional connectives of first-order intuitionistic logic, plus an extensive application in Section 7.5. There we show the equivalence between the definition of the substream relation in terms of stream filtering (Example 5.1.13), and the direct definition in Example 6.2.14. We use this equivalence to prove that the substream relation is transitive. The full development of this is given in Section 7.5, which is compiled from Agda code.

Since the application in Section 7.5 requires an induction principle for inductive types, we need to extend the calculus that we developed in Section 7.1. The reason is that this calculus only features a non-dependent iteration scheme, which is, as we have seen in Section 6.4.2, not enough to establish a general dependent iteration scheme. This extension is straightforward, though somewhat tedious to carry out. It is expected that the strong normalisation proof for the basic calculus carries over to this extension, but this is a conjecture at this point. However, we extend the subject reduction proof, as this is only a small exercise in patience. The result of this effort is a calculus in which we can prove, under the propositions-as-types interpretation, propositions on inductive types by induction.

At this point, we should justify the need for another type theory. This is a pressing question, since Martin-Löf type theory (MLTT) [Mar75a] and the calculus of inductive constructions (CoIC) [BC04; Pau93; Wer94] are well-studied frameworks for intuitionistic logic. The main reason is that the existing type theories have no explicit dependent coinductive types, which prevents us from directly defining coinductive predicates and relations like the substream relation. Giménez [Gim95] discusses an extension of the CoIC with coinductive types and guarded recursive schemes, but he proves no properties about the conversion relation. On the other hand, Sacchini [Sac13] extended the Calculus

of Constructions with streams and sized-types, and proves subject reduction and strong normalisation. However, the problem of limited support for general coinductive types remains. Finally, we should also mention that general coinductive types are available in implementations like Coq [Coq12], which is based on [Gim95], Agda [Agd15] and Nuprl [Con97]. Yet, none of these has a formal justification, and Coq's coinductive types are even known to have problems, for example related to subject reduction.[59]

One might argue that (dependent) coinductive types can be encoded through inductive types, as we have seen in Section 6.3 and was demonstrated in the setting of univalent homotopy type theory by Ahrens et al. [ACS15]. However, such an encoding does not give rise to a useful computation principle in an intensional type theory such as MLTT or CoIC, see [cLa16] for details. This becomes an issue once we try to prove propositions about terms of coinductive type, as we cannot use computational rules like those in Definition 3.1.11. For example, in such an encoding not even the conversion $\mathrm{tl}\,\underline{1}^{\omega} \equiv \underline{1}^{\omega}$ holds, which prevents us from proving the simple identity in Example 5.1.11 directly.[60] Clearly, such a restriction is not desirable, hence an encoding of coinductive types is for practical reasons not an option.

Interesting is also the fact that all basic connectives of MLTT, like the (dependent) function space, can be described as recursive types. This means that we do not need to have separate type constructors and the corresponding terms formation rules for dependent sums and products, but rather can derive them from the rules for recursive types. This is well-known in category theory, see also Theorem 6.2.11, but we do not know of any treatment of this fact in syntactic type theories.

Chapter Outline

The remainder of this chapter is structured as follows. We start by introducing the calculus around which the chapter revolves. Due to the complicated interactions between types, terms and the reduction relation in a dependently typed calculus, we proceed in three steps. First, in Section 7.1.1 we give the raw syntax in form of a context-free grammar, in which types and terms are not distinguished. The second step is to separate types and terms into, what we call here, pre-types and pre-terms in Section 7.1.2. At this stage terms are, however, not yet well-typed. This separation allows us to define a reduction relation on terms, which we need in the typing rules in the last step. The third, and last, step is to give the rules for well-formed types and terms in Section 7.1.4.

Having defined the calculus, we provide in Section 7.2 a host of examples that illustrate the use of the calculus. In particular, we show how all connectives of intuitionistic first-order logic can be represented in the calculus. Section 7.3 is devoted to proving the central properties of the type theory that we are interested here. These are subject reduction in Section 7.3.2, and strong normalisation in Section 7.3.3. To be able to reason about inductive types, we introduce in Section 7.4 a dependent iteration principle for the calculus. This is a somewhat tedious, but straightforward, extension of the previous calculus. At this stage, there is no proof of strong normalisation, but it is expected that the proof in Section 7.3.3 can be extended accordingly. However, subject reduction is preserved under this extension, as we will see. With this extended calculus under our belt, we come in Section 7.5 to the final application in this thesis. There, we prove, yet again, that the substream relation is transitive. This section has been fully formalised in Agda. We end the chapter with a discussion of related work and future directions in Section 7.6.

Original Publication

The calculus that we develop in Section 7.1 was first presented in a paper [BG16a], for which the author did the principle technical development and writing. That paper also featured the examples in Section 7.2 and outlines of the proofs of subject reduction and strong normalisation, the full details of which were made available in an ArXiv version of said article [BG16c]. Both the discussion of the dependent iteration principle in Section 7.4 and the application in Section 7.5 are new.

7.1. The Calculus $\lambda P\mu$

Before we formally introduce the $\lambda P\mu$ of dependent recursive types, let us give an informal overview over some important concepts of that calculus. The calculus is based on several judgements that single out well-formed types and terms. We will now go through the three ways how types can be formed: by parameter abstraction and instantiation, and as recursive types. These concepts form the backbone of our calculus. The term formation rules follow then just what is dictated by the category theoretical development in Section 6.2.

The formation of types involves two kinds of variables: type constructor variables and term variables. Type constructor variables fulfil here the same role as the type variables that we used in Section 3.1.1 to form recursive types. Term variables are required in the formation of types here because $\lambda P\mu$ is a calculus of dependent types. We will then use well-formedness judgements of the form

$$\Theta \mid \Gamma \vdash A : *,$$

which states that A is a type in the type constructor context Θ and the term context Γ.

Since the calculus is centred around dependent recursive types, we need a way to annotate type constructor variables, through which recursive types are constructed, with dependencies. The way we chose to do this here is by introducing, what we call, *parameter contexts* and *parameter instantiations*.[61] This leads us to generalise the above judgement to

$$\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *,$$

which is to be read as follows. As before, A is a type in the combined context $\Theta \mid \Gamma_1$, but this time it has also parameters of the types as they are specified by the second context Γ_2. Importantly, the types in the parameter context Γ_2 my depend on variables in the context Γ_1. For instance, if we have constructed a type Vec_B of vectors over some type B with $n : \mathrm{Nat} \vdash \mathrm{Vec}_B : *$, then we may have types C with $\Theta \mid n : \mathrm{Nat} \vdash C : (x : \mathrm{Vec}_B) \multimap *$. Here, C has a free variable n of type Nat and a parameter x of type Vec_B.

Given a parameterised type A as above, we can instantiate its parameters with terms of the corresponding types as follows. Suppose that $\Gamma_2 = x_1 : B_1, \ldots, x_n : B_n$ and that we are given terms $t_1, \ldots, t_n$ with $\Gamma_1 \vdash t_k : B_k[t_1/x_1, \ldots, t_{k-1}/x_{k-1}]$. This judgement should be read as: in the term variable context Γ_1, the term t_k is of type $B_k[t_1/x_1, \ldots, t_{k-1}/x_{k-1}]$. Then we can instantiate parameters of the type A with these terms by forming the new type $A @ t_1 @ \cdots @ t_n$ with

$$\Theta \mid \Gamma_1 \vdash A @ t_1 @ \cdots @ t_n : *.$$

Since type constructor variables are placeholders for parameterised types, the variables in the context Θ are also allowed to have parameters. We illustrate this with a small example. Let Vec_B be again

the type of vectors over B and and let X be a type constructor variable. The type system will allow us to form the judgement

$$X : (n : \mathrm{Nat}, x : \mathrm{Vec}_B) \multimap * \mid \Gamma_1 \vdash X : (n : \mathrm{Nat}, x : \mathrm{Vec}_B) \multimap *$$

for any context Γ. If we are now given terms k and u with $\Gamma_1 \vdash k : \mathrm{Nat}$ and $\Gamma_1 \vdash u : \mathrm{Vec}_B[k/n]$, then we are able to instantiate X with these terms to obtain

$$X : (n : \mathrm{Nat}, x : \mathrm{Vec}_B) \multimap * \mid \Gamma_1 \vdash X \mathbin{@} k \mathbin{@} u : *.$$

Besides parameter instantiation, we also allow variables to be moved from the term variable context into the parameter context by parameter abstraction. Given a type A with $\Theta \mid \Gamma_1, x : B \vdash A : \Gamma_2 \multimap *$, we may form the abstraction $(z).A$ with $\Theta \mid \Gamma_1 \vdash (z).A : (x : B, \Gamma_2) \multimap *$. As an example, we will be able derive the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{X : (x : B, y : B) \multimap * \mid \emptyset \vdash X : (x : B, y : B) \multimap *}{X : (x : B, y : B) \multimap * \mid z : B \vdash X : (x : B, y : B) \multimap *}}{X : (x : B, y : B) \multimap * \mid z : B \vdash X \mathbin{@} z \mathbin{@} z : *}}{X : (x : B, y : B) \multimap * \mid \emptyset \vdash (z).X \mathbin{@} z \mathbin{@} z : (z : B) \multimap *}$$

The first judgement is thereby given by projecting the variable X out of the context, the second by weakening, the third by instantiating X, and the final one by abstracting over z. Through these two mechanisms of parameters instantiation and abstraction we can deal smoothly with type constructor variables in the formation of dependent recursive types.[62]

Similar to type constructors with parameters, part of the calculus are also *parameterised terms* and instantiations thereof. A parameterised term will be typed by a type with parameters of the shape $\Gamma_2 \multimap A$. Given a parameterised term s with $\Gamma_1 \vdash s : \Gamma_2 \multimap A$, we can instantiate s with arguments just like parameterised types: If $\Gamma_2 = x_1 : B_1, \ldots, x_n : B_n$ and $\Gamma_1 \vdash t_k : B_k[t_1/x_1, \ldots, t_{k-1}/x_{k-1}]$ for $1 \leq k \leq n$, then

$$\Gamma \vdash s \mathbin{@} t_1 \mathbin{@} \cdots \mathbin{@} t_n : A[\vec{t}\,/\vec{x}\,],$$

where $A[\vec{t}\,/\vec{x}\,]$ denotes the simultaneous substitution of all the terms t_k for the corresponding term variables x_k. In the case of terms, however, we do not allow parameter abstraction.

Having set up how we deal with type constructor variables, we come to the heart of the calculus: the type constructors for recursive types. Thes resemble the initial and final dialgebras for strictly positive signatures that we defined in Section 6.2. These type constructors are written as

$$\mu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A}) \qquad \text{and} \qquad \nu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A}),$$

where $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ are substitutions and $\vec{A} = A_1, \ldots, A_n$ are types with a free type constructor variable X. In view of the categorical development, the σ_k are the analogue of the morphisms u_k in the base category that we used there for reindexing, and the types A_k correspond to the components of the functor F. Thus pairs $(\vec{A}, \vec{\sigma})$ correspond to a strictly positive signatures.

Accordingly, we will associate constructors and an iteration scheme to inductive types, and destructors and a coiteration scheme to coinductive types. Suppose, for example, that Γ is a context with $\Gamma = x_1 : B_1, \ldots, x_n : B_n$, that A_k is a type with $X : \Gamma \multimap * \mid \Gamma_k \vdash A_k : *$, and σ_k is a substitution

with $\sigma_k = (t_1, \ldots, t_n)$. The kth constructor of $\mu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A})$ is then denoted by α_k and has the following type, where we use the shorthand $\mu = \mu(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A})$.

$$\vdash \alpha_k : (\Gamma_k, z : A_k[\mu/X]) \multimap (\mu \mathbin{@} t_1 \mathbin{@} \cdots \mathbin{@} t_m)\,.$$

The constructor α_k can now be instantiated according to the parameter context: Suppose, for simplicity, that $\Gamma_k = y : C$ for some type C that does not depend on X and that we are given a term $\Gamma \vdash s : C$. For a recursive argument $\Gamma \vdash u : A_k[\mu/X][s/y]$, we obtain thus

$$\Gamma \vdash \alpha_k \mathbin{@} s \mathbin{@} u : (\mu \mathbin{@} t_1 \mathbin{@} \cdots \mathbin{@} t_m)[s/y].$$

The rest of the type and term constructors are the standard structural rules, like weakening, one would expect. It should be noted that there is a strong similarity in the use of destructors for coinductive types to $\lambda\mu\nu$. Moreover, the definition scheme for generalised abstract data types in [HF11] describes the same *inductive* types. We will discuss this further in Section 7.6.

7.1.1. Raw Syntax

We now formally introduce the syntax of the calculus $\lambda\mathrm{P}\mu$. The first step is to give, what we call here, its *raw syntax*. This syntax has only one syntactic class for types and terms, which is given by the syntactic objects M and N in the following context free grammar. Part of the grammar are also type and term variable contexts, denoted by Θ and Γ, respectively, and substitutions σ.

Definition 7.1.1. Let Var and TyVar be two disjoint, countably infinite sets of term variables and type constructor variables. Term variables will be denoted by $x, y, z, \ldots$, whereas type constructor variables are denoted by capital letters $X, Y, Z, \ldots$. The raw contexts, substitutions and terms are given by the following grammar.

$$\begin{aligned}
\Gamma &::= \emptyset \mid \Gamma, x : M \quad x \in \mathrm{Var}\\
\Theta &::= \emptyset \mid \Theta, X : \Gamma \multimap *\\
\sigma &::= () \mid (\sigma, M)\\
M, N &::= \top \mid \langle\rangle \mid x \in \mathrm{Var} \mid M \mathbin{@} N \mid (x).\, M \mid X \in \mathrm{TyVar} \mid \rho(X : \Gamma \multimap * ; \vec{\sigma}; \vec{M}), \quad \rho \in \{\mu, \nu\}\\
&\quad \mid \alpha_k^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})} \mid \xi_k^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}, \quad k \in \mathbb{N}\\
&\quad \mid \mathrm{iter}^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{M})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)}\\
&\quad \mid \mathrm{coiter}^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{M})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)}
\end{aligned}$$

To disambiguate raw terms, we use the the convention that instantiation, that is the operator @, binds to the left, and that abstraction binds from the dot all the way to the right.

Let us explain the intention of the raw terms that we have not covered in the introduction above. In the calculus, we have an explicit type $\top$ with a single element $\langle\rangle$, thereby resembling a singleton set. We need this type to form closed terms, as we will explain further in Example 7.2.1. The term ξ_k is the destructor for coinductive types, analogous to the destructor for the greatest fixed point types in Section 3.1. Correspondingly, terms of the form $\mathrm{coiter}^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{M})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)}$ are

instances of the coiteration scheme for the coinductive type $\nu(X : \Gamma \multimap * ; \vec{\sigma} ; \vec{M})$. The role of the terms N_k and the binding construct $(\Gamma_k, y_k).\, N_k$ will become clearer once we introduce the typing rules for well-formed terms. Thus, we postpone the explanation of these to Section 7.1.4. Dually, $\mathrm{iter}^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{M})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)}$ is an instance of the iteration scheme for inductive types.

The type annotations on the constructors, destructors, and the iteration and coiteration schemes are necessary in the definition of the reduction relation and typing rules that we give later. However, whenever the annotated types are clear from the context, then we will usually leave them out.

7.1.2. Pre-Types and Pre-Terms

The next step is to separate the raw terms from Section 7.1.1 into, what we call here, pre-types and pre-terms. This is an intermediate step towards the definition of the typing rules, which has a two-fold purpose. First of all, the definition of the typing rules of the calculus $\lambda \mathrm{P}\mu$ require a reduction relation on terms. However, we cannot produce such a reduction relation for the raw syntax because essential to the definition of this relation is an action of types on terms, just like for the reduction relation of $\lambda\mu\nu$ in Section 3.1.2. Such an action on terms is not definable for the whole raw syntax though. Thus, we are lead to single out syntactic objects that allow us to define an action on terms for them. These are the pre-types that we are going to define in this section. The second need for pre-types and pre-terms arises in the proof of strong normalisation for $\lambda \mathrm{P}\mu$. This proof is based on the construction of a syntactic model in terms of so-called *saturated sets*. Usually, such a syntactic model is constructed from raw terms because the typing rules get in the way otherwise. But since we can define the reduction relation only on pre-terms, we also use pre-terms as a sweet spot in between raw and typed terms in the strong normalisation proof.

The major difference between pre-types and pre-terms, and the (well-formed) types and terms in Section 7.1.4 is that that pre-terms only get a single placeholder type assigned. This placeholder type is denoted by an empty box $\Box$. Having only a placeholder type breaks the dependency between the reduction relation and rules for pre-terms. Such a dependency occurs because among the rules for terms is a so-called *conversion rule*, which allows computations (reduction steps) during type-checking. Instead, we define first pre-terms and pre-types, for which we can define in a second step the reduction relation.

Pre-types and pre-terms are defined through the following two judgements.

$$\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : \Gamma_2 \multimap * \qquad \text{and} \qquad \Gamma_1 \vdash_{\mathrm{pre}} t : \Gamma_2 \multimap \Box.$$

The first judgement expresses that A is a pre-type in the type context Θ, the term context Γ_1 and has the parameter context Γ_2. Pre-terms are given by the second judgement, where the contexts Γ_1 and Γ_2 play the same role, but it should be noted that the type of t is just the placeholder $\Box$.

We will now just give the rules for pre-types and -terms without further commenting on them, since they are essentially the same as the rules in Section 7.1.4. The only difference is that the types of terms are erased and that there is no conversion rule for pre-terms. Moreover, the pre-types and -terms are only a technical tool to establish the reduction relation in Definition 7.1.6 and for the strong normalisation proof. Thus, the reader is advised to skip ahead and read the explanation of the reduction relation below and of the well-formedness rules in Section 7.1.4 first. The definition of pre-terms and pre-types is mostly here for completeness. However, due to the technical difficulty, their definition and that of the reduction relation below have been formalised in Agda [Bas18b].

$$\frac{}{\vdash_{\mathrm{pre}} \top : *}\ \textbf{(PT)-}\top$$

$$\frac{}{\Theta, X : \Gamma \multimap * \mid \emptyset \vdash_{\mathrm{pre}} X : \Gamma \multimap *}\ \textbf{(PT-TyVar)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : \Gamma_2 \multimap *}{\Theta, X : \Gamma \multimap * \mid \emptyset \vdash_{\mathrm{pre}} A : *}\ \textbf{(PT-TyWeak)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : \Gamma_2 \multimap * \qquad \Gamma_1 \vdash_{\mathrm{pre}} B : *}{\Theta \mid \Gamma_1, x : B \vdash_{\mathrm{pre}} A : \Gamma_2 \multimap *}\ \textbf{(PT-Weak)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : (x : B, \Gamma_2) \multimap * \qquad \Gamma_1 \vdash_{\mathrm{pre}} t : \square}{\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A \mathbin{@} t : \Gamma_2 \multimap *}\ \textbf{(PT-Inst)}$$

$$\frac{\Theta \mid \Gamma_1, x : A \vdash_{\mathrm{pre}} B : \Gamma_2 \multimap *}{\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} (x).B : (x : A, \Gamma_2) \multimap *}\ \textbf{(PT-Param-Abstr)}$$

$$\frac{\Theta, X : \Gamma \multimap * \mid \Gamma_k \vdash_{\mathrm{pre}} A_k : * \qquad \vdash_{\mathrm{pre}} \sigma_k : \Gamma_k \triangleright \Gamma \qquad 1 \leq k \leq |\vec{A}| \qquad \rho \in \{\mu, \nu\}}{\Theta \mid \emptyset \vdash_{\mathrm{pre}} \rho(X : \Gamma \multimap * ; \vec{\sigma}; \vec{A}) : \Gamma \multimap *}\ \textbf{(PT-FP-}\rho\textbf{)}$$

Figure 7.1.: Pre-Types

Definition 7.1.2. The *pre-types* and *pre-terms* of λPμ are defined inductively by the rules in Figure 7.1 and Figure 7.2, respectively. Whenever in any of the rules a recursive type occurs, it is implicitly assumed that this recursive type is a pre-type. In the rule **(PT-FP)** for recursive pre-types, substitutions by pre-terms are used. We will refer to such substitutions as *pre-context morphisms*. They are given, simultaneously with pre-types and pre-terms, by the following two rules.

$$\frac{}{\vdash_{\mathrm{pre}} () : \Gamma_1 \triangleright \emptyset} \qquad \frac{\vdash_{\mathrm{pre}} \sigma : \Gamma_1 \triangleright \Gamma_2 \qquad \Gamma_1 \vdash_{\mathrm{pre}} A : * \qquad \Gamma_1 \vdash_{\mathrm{pre}} t : \square}{\vdash_{\mathrm{pre}} (\sigma, t) : \Gamma_1 \triangleright (\Gamma_2, x : A)}$$

This concludes the definition of pre-types and pre-terms. ◀

Substitution of pre-types and pre-terms are defined as expected. It is also straightforward to show that being a pre-type, respectively a pre-term, is preserved under substitutions. This is proved in the Agda formalisation [Bas18b]. Given a pre-context morphism $\sigma = (s_1, \ldots, s_n)$, we denote by

$$A[\sigma] \coloneqq A[s_1/x_1, \ldots, s_n/x_n] \qquad \text{and} \qquad t[\sigma] \coloneqq t[s_1/x_1, \ldots, s_n/x_n]$$

the simultaneous substitution of all terms in σ for the corresponding variables in the pre-type A, respectively the pre-term t.

7.1.3. Reductions on Pre-Types and Pre-Terms

We now come to the reduction relation on pre-types and pre-terms. To define such a relation, we proceeds, analogously to Section 3.1.2, in four steps: First, we define an action of pre-types on

$$\frac{}{\vdash_{\mathrm{pre}} \langle\rangle : \square}\ \textbf{(PO-}\top\textbf{-I)} \qquad \frac{\Gamma_1 \vdash_{\mathrm{pre}} t : (x : A, \Gamma_2) \multimap \square \qquad \Gamma_1 \vdash_{\mathrm{pre}} s : \square}{\Gamma_1 \vdash_{\mathrm{pre}} t \mathbin{@} s : \Gamma_2 \multimap \square}\ \textbf{(PO-Inst)}$$

$$\frac{x \in \mathrm{Var} \qquad \Gamma \vdash_{\mathrm{pre}} A : *}{\Gamma, x : A \vdash_{\mathrm{pre}} x : \square}\ \textbf{(PO-Proj)} \qquad \frac{\Gamma \vdash_{\mathrm{pre}} A : * \qquad \Gamma_1 \vdash_{\mathrm{pre}} t : \Gamma_2 \multimap \square}{\Gamma_1, x : A \vdash_{\mathrm{pre}} t : \Gamma_2 \multimap \square}\ \textbf{(PO-Weak)}$$

$$\frac{1 \le k \le |\vec{A}|}{\vdash_{\mathrm{pre}} \alpha_k^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})} : (\Gamma_k, y : A_k[\mu/X]) \multimap \square}\ \textbf{(PO-Ind-I)}$$

$$\frac{1 \le k \le |\vec{A}|}{\vdash_{\mathrm{pre}} \xi_k^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})} : (\Gamma_k, y : \nu \mathbin{@} \sigma_k) \multimap \square}\ \textbf{(PO-Coind-E)}$$

$$\frac{\Delta \vdash_{\mathrm{pre}} C : \Gamma \multimap * \qquad \forall 1 \le k \le |\vec{A}|.\ (\Delta, \Gamma_k, y_k : (C \mathbin{@} \sigma_k) \vdash_{\mathrm{pre}} g_k : \square)}{\Delta \vdash_{\mathrm{pre}} \mathrm{iter}^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)} : (\Gamma, y : \mu \mathbin{@} \mathrm{id}_\Gamma) \multimap \square}\ \textbf{(PO-Ind-E)}$$

$$\frac{\Delta \vdash_{\mathrm{pre}} C : \Gamma \multimap * \qquad \forall 1 \le k \le |\vec{A}|.\ (\Delta, \Gamma_k, y_k : (C \mathbin{@} \sigma_k) \vdash_{\mathrm{pre}} g_k : \square)}{\Delta \vdash_{\mathrm{pre}} \mathrm{coiter}^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}\ \overrightarrow{(\Gamma_k, y_k.\, N_k)} : (\Gamma, y : C \mathbin{@} \mathrm{id}_\Gamma) \multimap \square}\ \textbf{(PO-Coind-I)}$$

Figure 7.2.: Pre-Terms

pre-terms. Then we use this action to define a simple contraction relation. From this contraction relation we obtain, as the third step, a reduction relation by taking its compatible closure. Finally, we use the resulting reduction relation on pre-terms to define a reduction on pre-types.

For convenience, let us introduce some notation for dealing with pre-context morphisms. These notations make it also easier to relate the present development to the category theoretical approach in Section 6.2.

Notation 7.1.3. First, for $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ we define the identity (pre-)context morphism id_Γ by $\mathrm{id}_\Gamma := (x_1, \ldots, x_n)$. Second, given a type A with $\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : \Gamma \multimap *$ and a pre-context morphism σ with $\vdash_{\mathrm{pre}} \sigma : \Gamma_1 \triangleright \Gamma$ and $\sigma = (t_1, \ldots, t_n)$, we denote by $A \mathbin{@} \sigma$ the instantiation $A \mathbin{@} t_1 \mathbin{@} \cdots \mathbin{@} t_n$. Finally, we will need to compose pre-context morphisms later. This composition is given for pre-context morphisms $\Gamma_3 \overset{\sigma}{\triangleright} \Gamma_2 \overset{\tau}{\triangleright} \Gamma$ by

$$\tau \bullet \sigma := (\tau_1[\sigma], \ldots, \tau_n[\sigma]), \tag{7.1}$$

where $\tau_k[\sigma]$ denotes the substitution of σ in all terms in τ_k. ◀

Since we will frequently deal with parameter abstractions of many variables at the same time, it is worth to also introduce some notation concerning multi-variable abstractions.

Notation 7.1.4. Let us agree on the following notations for parameter abstraction. Given a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and a pre-type $\Theta \mid \Gamma \vdash_{\mathrm{pre}} B : *$, we denote the full abstraction of B by $(\Gamma).\, B := (x_1).\, \cdots (x_n).\, B$, which gives us

$$\Theta \mid \emptyset \vdash_{\mathrm{pre}} (\Gamma).\, B : \Gamma \multimap *.$$

Moreover, if we are given a sequence $\vec{B}$ of such types, that is, if $\Theta \mid \Gamma_i \vdash_{\text{pre}} B_i : *$ for each B_i in that sequence, we denote by $\overrightarrow{(\Gamma_i).B}$ the sequence of types that arises by fully abstracting each type B_i separately. ◀

Having introduced all this notation, we can now define the action $\widehat{A}$ of a pre-type A on pre-types and -terms. Suppose that A and U are pre-types with $X : \Gamma \twoheadrightarrow * \mid \Gamma_1 \vdash_{\text{pre}} A : \Gamma_2 \twoheadrightarrow *$ and $\Gamma \vdash_{\text{pre}} U : *$. Then the action of A on U is given by

$$\widehat{A}(U) = A[(\Gamma).U/X] \mathbin{@} \mathrm{id}_{\Gamma_2}.$$

On terms, we will define $\widehat{A}$ so that the following rule holds.

$$\frac{X : \Gamma \twoheadrightarrow * \mid \Gamma_1 \vdash_{\text{pre}} A : \Gamma_2 \twoheadrightarrow * \qquad \Gamma, x : U \vdash_{\text{pre}} t : \Box}{\Gamma_1, \Gamma_2, y : \widehat{A}(U) \vdash_{\text{pre}} \widehat{A}(t) : \Box}$$

This action of pre-types is defined analogously to that of types on terms in the simply typed case in Section 3.1.2. As such, it follows in the case of recursive types the definition of functors from parameterised initial and final dialgebras in Section 6.2.2. In fact, $\widehat{A}$ has the type of a functor that turns types in context Γ into types in the context Γ_1, Γ_2. That will become clearer once we introduce well-formed types and terms in Section 7.1.4 and prove subject reduction in Section 7.3.2.

In the following definition of the action of pre-types, we need to generalise the above rules to an arbitrary number of free type constructor variables in A. That means that, instead of assuming $X : \Gamma \twoheadrightarrow * \mid \Gamma_1 \vdash_{\text{pre}} A : \Gamma_2 \twoheadrightarrow *$, we have $\Theta \mid \Delta_1 \vdash_{\text{pre}} A : \Delta_2 \twoheadrightarrow *$ for an arbitrary type context Θ.

Definition 7.1.5. Let $\Theta \mid \Delta_1 \vdash_{\text{pre}} A : \Delta_2 \twoheadrightarrow *$ be a pre-type with $\Theta = X_1 : \Gamma_1 \twoheadrightarrow *, \ldots, X_n : \Gamma_n \twoheadrightarrow *$, $\vec{U}$ and $\vec{V}$ be sequences of pre-types with $\Gamma_i \vdash_{\text{pre}} U_i : *$ and $\Gamma_i \vdash_{\text{pre}} V_i : *$ for all $1 \leq i \leq n$. The action of A on the sequence $\vec{U}$ of types is given by

$$\widehat{A}(\vec{U}) \coloneqq A[\overrightarrow{(\Gamma_i).U}/\vec{X}] \mathbin{@} \mathrm{id}_{\Delta_2}.$$

Note that we then have that $\Theta \mid \Delta_1, \Delta_2 \vdash_{\text{pre}} \widehat{A}(\vec{U}) : *$.

For a sequence $\vec{t}$ of pre-terms with $\Gamma_i, x : U_i \vdash_{\text{pre}} t : \Box$ for all $1 \leq i \leq n$, we define the action of A on $\vec{t}$, denoted by $\widehat{A}(\vec{t})$, so that the following rule holds.

$$\frac{\Theta \mid \Delta_1 \vdash_{\text{pre}} C : \Delta_2 \twoheadrightarrow * \qquad \forall 1 \leq i \leq n.\ \Gamma_i, x : U_i \vdash_{\text{pre}} t_i : \Box}{\Delta_1, \Delta_2, y : \widehat{C}(\vec{U}) \vdash_{\text{pre}} \widehat{C}(\vec{t}) : \Box}$$

If $n = 0$, we simply put $\widehat{A}(\varepsilon) = y$, which also covers the case **(PT-⊤)**. If $n > 0$, we define $\widehat{A}(\vec{t})$

by induction on the derivation of $\Theta \mid \Gamma_1 \vdash_{\mathrm{pre}} A : \Gamma_2 \multimap *$ as follows.

$$\widehat{X_i}(\vec{t}) = t_i[y/x] \qquad \textbf{(PT-TyVar)}$$

$$\widehat{A}(\vec{t}, t_{n+1}) = \widehat{A}(\vec{t}) \qquad \textbf{(PT-TyWeak)}$$

$$\widehat{A}(\vec{t}) = \widehat{A}(\vec{t}) \qquad \textbf{(PT-Weak)}$$

$$\widehat{A \mathbin{@} s}(\vec{t}) = \widehat{A}(\vec{t})[s/x] \qquad \textbf{(PT-Inst)}$$

$$\widehat{(y).A}(\vec{t}) = \widehat{A}(\vec{t}) \qquad \textbf{(PT-Param-Abstr)}$$

$$\widehat{\mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{A})}(\vec{t}) = \mathrm{iter}^{\mu_U}\ \overrightarrow{(\Delta_k, y'.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} y \qquad \textbf{(PT-FP-}\mu\textbf{)}$$

with $g_k = \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} \left(\widehat{A_k}\,(\vec{t}, x)\right)$

and $\mu_U = \mu(Y : \Gamma \multimap *; \vec{\sigma}; \overrightarrow{\widehat{A}(\vec{U})})$

$$\widehat{\nu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{A})}(\vec{t}) = \mathrm{coiter}^{\nu_V}\ \overrightarrow{(\Delta_k, z.\, g)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} y \qquad \textbf{(PT-FP-}\nu\textbf{)}$$

with $g_k = \widehat{A_k}\,(\vec{t}, x)\,[(\xi_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} z)/y']$

and $\nu_V = \nu(Y : \Gamma \multimap *; \vec{\sigma}; \overrightarrow{\widehat{A}(\vec{V})})$

In the case **(PT-FP-μ)** for inductive types, $\overrightarrow{\widehat{A}(\vec{U})}$ is the sequence of $\widehat{A_i}(\vec{U})$ that arises by applying all A_i to $\vec{U}$. Analogously, $\overrightarrow{\widehat{A}(\vec{V})}$ is given by applying all A_i to $\vec{V}$ in the case for coinductive types. ◀

To make it easier to follow this definition, let us show that the result in the case **(PT-FP-μ)** is indeed a pre-term. All the other cases are covered by the proof of subject reduction in Section 7.3.2. So suppose we are given an inductive type $\mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{A})$ with

$$\frac{\forall 1 \le k \le m.\ \left(\Theta, X : \Gamma \multimap * \mid \Delta_k \vdash_{\mathrm{pre}} A_k : *\right) \qquad \vdash_{\mathrm{pre}} \sigma_k : \Delta_k \triangleright \Gamma}{\mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{A})}$$

We refer to this type just by μ in what follows. Let $\mu_U \coloneqq \mu(Y : \Gamma \multimap *; \vec{\sigma}; \overrightarrow{\widehat{A}(\vec{U})})$, and note that $\mu_U \mathbin{@} \mathrm{id}_\Gamma = \widehat{\mu}(\vec{U})$. The intuition for the definition of the action of μ on terms follows that for the definition of a functor from parameterised initial dialgebras, see Definition 6.2.5, in the sense that $\widehat{\mu}(\vec{t})$ is defined as morphism $\Gamma, y : \widehat{\mu}(\vec{U}) \vdash_{\mathrm{pre}} h : \square$, as in the following diagrams for all k with $1 \le k \le n$. Note that we cannot say yet that the diagram commutes, as this requires the the reduction relation.

$$\begin{array}{ccc}
\Delta_k, y' : \widehat{A_k}\left(\vec{U}, \mu_U\right) & \xrightarrow{\widehat{A_k}\left(\overrightarrow{\mathrm{id}_{\Gamma_i}}, h\right)} & \Delta_k, y' : \widehat{A_k}\left(\vec{U}, \mu_V\right) \\
 & & \Big\downarrow{\scriptstyle \widehat{A_k}(\vec{t}, x)} \\
\Big\downarrow{\scriptstyle \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} y'} & & \Delta_k, z : \widehat{A_k}\left(\vec{V}, \mu_V\right) \\
 & & \Big\downarrow{\scriptstyle \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} z} \\
\Delta_k, y : \mu_U \mathbin{@} \sigma_k & \xrightarrow{h[\sigma_k]} & \Delta_k, z' : \mu_V \mathbin{@} \sigma_k
\end{array}$$

$$\begin{array}{c}\mathrm{iter}\,\overrightarrow{(\Gamma_k, y_k.\, g_k)} \mathbin{@} (\sigma_k \bullet \tau) \mathbin{@} (\alpha_k \mathbin{@} \tau \mathbin{@} u) \succ g_k\Big[\widehat{A_k}(\mathrm{iter}\,\overrightarrow{(\Gamma_k, y_k.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x)\Big/ y_k\Big][\tau, u] \\ \xi_k \mathbin{@} \tau \mathbin{@} (\mathrm{coiter}\,\overrightarrow{(\Gamma_k, y_k.\, g_k)} \mathbin{@} (\sigma_k \bullet \tau) \mathbin{@} u) \succ \widehat{A_k}\left(\mathrm{coiter}\,\overrightarrow{(\Gamma_k, y_k.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x\right)[g_k/y][\tau, u]\end{array}$$

Figure 7.3.: Contraction of Terms.

In Definition 7.1.5, the pre-term h was given by defining

$$\begin{aligned} g_k &= \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@}\Big(\widehat{A_k}(\vec{t}, x)\Big) \\ h &= \mathrm{iter}^{\mu_U}\,\overrightarrow{(\Delta_k, y'.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} y. \end{aligned}$$

Note that $g_k = (\alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} z)\Big[\Big(\widehat{A_k}(\vec{t}, x)\Big)\Big/ z\Big]$, which is the right-hand side of the above diagram. Let us now derive that all the g_k and h are correct pre-terms. First of all, we obtain from the induction hypothesis in the definition of the pre-type action the following derivation.

$$\frac{\Theta, X : \Gamma \to * \mid \Delta_k \vdash_{\mathrm{pre}} A_k : * \qquad \forall 1 \leq i \leq n.\ \big(\Gamma_i, x : U_i \vdash_{\mathrm{pre}} t : \square\big) \qquad \Gamma, x : \widehat{\mu}(\vec{V}) \vdash_{\mathrm{pre}} x : \square}{\Delta_k, y' : \widehat{A_k}(\vec{U}, \widehat{\mu}(\vec{V})) \vdash_{\mathrm{pre}} \widehat{A_k}(\vec{t}, x) : \square}$$

By the typing rules for the constructors of the inductive type μ_U, we also have the following.

$$\frac{\Delta_k, z : \widehat{A_k}(\vec{U}, \mu_U) \vdash_{\mathrm{pre}} z : \square}{\Delta_k, z : \widehat{A_k}(\vec{U}, \mu_U) \vdash_{\mathrm{pre}} \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} z : \square}$$

Putting these together, we have

$$\dfrac{\dfrac{\dfrac{\forall 1 \leq i \leq n.\ \big(\Gamma_i, x : U_i \vdash_{\mathrm{pre}} t : \square\big)}{\forall 1 \leq k \leq m.\ \Big(\Delta_k, y' : \widehat{A_k}(\vec{U}, \widehat{\mu}(\vec{V})) \vdash_{\mathrm{pre}} g_k : \square\Big)}}{\vdash_{\mathrm{pre}} \mathrm{iter}^{\mu_U}\,\overrightarrow{(\Delta_k, y.\, g_k)} : (\Gamma, y : \widehat{\mu}(\vec{U})) \to \square}}{\Gamma, y : \widehat{\mu}(\vec{U}) \vdash_{\mathrm{pre}} h : \square}$$

This shows that $\widehat{\mu}(\vec{t}) = h$ is indeed a pre-term in the expected context.

We now come to the definition of the reduction relations on pre-terms and pre-types. Analogously to the development in Section 3.1.2, we define the reduction relation on pre-terms by first giving a contraction relation, which carries out single-step computations, and then taking the compatible closure of this contraction relation is reduction relation. The reduction relation on pre-types reduces then parameters by means of the reduction relation on pre-terms. Moreover, it performs β-reduction for instantiations.

So let us start by providing the reduction on pre-terms by appealing to the action of pre-types.

Definition 7.1.6. The *reduction relation of* $\lambda\mathrm{P}\mu$ $\longrightarrow$ on pre-terms is defined as compatible closure[63] of the *contraction relation of* $\lambda\mathrm{P}\mu$ $\succ$ given in Figure 7.3. We introduce in the definition of contraction a fresh variable x, for which we immediately substitute (either u or g_k). This is necessary for the use of the action of types on terms, see Definition 7.1.5. ◀

On terms, the reduction relation for a destructor-coiterator pair essentially emulates the homomorphism diagram for coinductive extensions. So suppose we are given a coinductive type $\nu = \nu(Y : \Gamma \twoheadrightarrow *; \vec{\sigma}; \vec{A})$ with $\Gamma_k \vdash_{\text{pre}} A_k : *$ and a pre-type C with $\vdash_{\text{pre}} C : \Gamma \twoheadrightarrow *$. Then the contraction relation acts as in the following diagram, where composition is given by substitution.

$$\begin{array}{ccc} \Gamma_k, x_k : C \mathbin{@} \sigma_k & \xrightarrow{\text{coiter } \overrightarrow{(\Gamma_k, y.\, g_k)} \mathbin{@} \sigma_k \mathbin{@} x} & \Gamma_k, y : \nu \mathbin{@} \sigma_k \\ {\scriptstyle g_k}\downarrow & \prec & \downarrow{\scriptstyle \xi_k \mathbin{@} \mathrm{id}_{\Gamma_k} \mathbin{@} y} \\ \Gamma_k, y : A_k[C/X] & \xrightarrow[\widehat{A_k}(\text{coiter } \overrightarrow{(\Gamma_k, y.\, g)} \mathbin{@} \sigma_k \mathbin{@} x)]{} & \Gamma_k, z : A_k[\nu/X] \end{array}$$

To be precise, this means that we have

$$\begin{aligned} (\xi_k \mathbin{@} \mathrm{id}_{\Gamma_k} \mathbin{@} y)\Big[\text{coiter } \overrightarrow{(\Gamma_k, y.\, g_k)} \mathbin{@} \sigma_k \mathbin{@} x \Big/ y\Big] &= \xi_k \mathbin{@} \mathrm{id}_{\Gamma_k} \mathbin{@} \Big(\text{coiter } \overrightarrow{(\Gamma_k, y.\, g_k)} \mathbin{@} \sigma_k \mathbin{@} x\Big) y \\ &\succ \widehat{A_k}(\text{coiter } \overrightarrow{(\Gamma_k, y.\, g)} \mathbin{@} \sigma_k \mathbin{@} x)[g_k/y]. \end{aligned}$$

The reduction relation for a iterator-constructor pair emulates the dual of this diagram for homomorphisms out of initial dialgebras.

From the reduction relation on pre-terms, we can now define reductions on pre-types.

Definition 7.1.7. The reduction relation on pre-types consists of two types of reductions: First, β-reduction for parameter instantiations is given by *parameter reduction*[64]

$$((x).\, A) \mathbin{@} t \longrightarrow_p A[t/x].$$

Second, we lift the reduction relation on pre-terms to pre-types by taking the compatible closure of reduction of parameters, which is given by

$$\frac{t \longrightarrow t'}{A \mathbin{@} t \longrightarrow A \mathbin{@} t'} \tag{7.2}$$

We combine these relations into one reduction relation on pre-types:

$$\longrightarrow_T \coloneqq \longrightarrow_p \cup \longrightarrow.$$

One-step conversion of pre-types is then given by

$$A \longleftrightarrow_T B \iff A \longrightarrow_T B \text{ or } B \longrightarrow_T A. \qquad \blacktriangleleft$$

This concludes the definition of reductions on pre-types and -terms. So we can finally come to the definition well-formed types and terms.

7.1.4. Well-Formed Types and Terms

The goal of this section is carve out the well-formed types and terms of the calculus $\lambda \mathrm{P}\mu$ from the pre-terms and pre-types that we defined in the last section. We will do this through several judgements, each of which has its own set of derivations rules. It is understood that the derivability of these judgments is defined by simultaneous induction. So, Definitions 7.1.8, 7.1.9, 7.1.10 and 7.1.11 should be seen as one simultaneous definition. The judgements we are going to use are are the following.

- $\vdash \Theta$ **TyCtx** — The type constructor variable context Θ is well-formed.
- $\vdash \Gamma$ **Ctx** — The term variable context Γ is well-formed.
- $\sigma : \Gamma_1 \triangleright \Gamma_2$ – The context morphism σ is a well-formed substitution for Γ_2 with terms in context Γ_1.
- $\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \twoheadrightarrow * $ — The type constructor A is well-formed in the combined context $\Theta \mid \Gamma_1$ and can be *instantiated* with terms according to the *parameter context* Γ_2, where it is implicitly assumed that Θ, Γ_1 and Γ_2 are well-formed.
- $\Gamma_1 \vdash t : \Gamma_2 \twoheadrightarrow A$ — The term t is well-formed in the term variable context Γ_1 and, after instantiating it with arguments according to parameter context Γ_2, is of type A with the arguments substituted into A.

Definition 7.1.8. The judgements for singling out well-formed contexts (type variable contexts and term variable contexts) are given by the following rules.

$$\frac{}{\vdash \emptyset \ \textbf{TyCtx}} \qquad \frac{\vdash \Theta \ \textbf{TyCtx} \qquad \vdash \Gamma \ \textbf{Ctx}}{\vdash \Theta, X : \Gamma \twoheadrightarrow * \ \textbf{TyCtx}} \qquad\qquad \frac{}{\vdash \emptyset \ \textbf{Ctx}} \qquad \frac{\emptyset \mid \Gamma \vdash A : *}{\vdash \Gamma, x : A \ \textbf{Ctx}}$$ ◀

It is important to note that whenever a term variable declaration is added into the context, its type is not allowed to have any free type constructor variables, which ensures that all types are strictly positive. For example, we are not allowed to form the term context $\Gamma = x : X$ in which X occurs freely. This prevents us, as we will see in Ex. 7.2.4, from forming function spaces $X \to A$.

Definition 7.1.9 (Context Morphism)**.** We introduce the notion of context morphisms as a shorthand notation for typed substitutions. Let Γ_1 and Γ_2 be contexts. A *context morphism* $\sigma : \Gamma_1 \triangleright \Gamma_2$ is given by the following two rules.

$$\frac{}{() : \Gamma_1 \triangleright \emptyset} \qquad \frac{\sigma : \Gamma_1 \triangleright \Gamma_2 \qquad \Gamma_1 \vdash t : A[\sigma]}{(\sigma, t) : \Gamma_1 \triangleright (\Gamma_2, x : A)}$$

where $\emptyset \mid \Gamma_2 \vdash A : *$, and $A[\sigma]$ denotes the simultaneous substitution of the terms in σ for the corresponding variables, which is often also denoted by $A[\sigma] = A[\sigma/\vec{x}]$. ◀

Definition 7.1.10 (Well-formed Type Constructor of λPμ)**.** The judgement for type constructors is given inductively by the rules in Figure 7.4, where it is understood that all involved contexts are well-formed. where in the **(FP-Ty)**-rule $\vec{\sigma}$ and $\vec{A}$ are assumed to have the same length $|\vec{A}|$. ◀

Note that type constructor variables come with a parameter context. This context determines the parameters of an initial/final dialgebra, which essentially bundle the *local* context, the domain and the codomain of their constructors respectively destructors. In other words, the types implement precisely the closure rules for μP-complete categories.

This brings us finally to the rules for well-formed terms.

Definition 7.1.11 (Well-formed Terms of λPμ)**.** The judgement for terms is given by the rules in Fig. 7.5. To improve readability, we use the shorthand $\rho = \rho(X : \Gamma \twoheadrightarrow * ; \vec{\sigma} ; \vec{A})$, $\rho \in \{\mu, \nu\}$, and implicitly assume all involved types and contexts are well-formed. ◀

$$\frac{}{\vdash \top : *}\ \textbf{(}\top\textbf{-I)}$$

$$\frac{\vdash \Theta\ \ \textbf{TyCtx} \qquad \vdash \Gamma\ \textbf{Ctx}}{\Theta, X : \Gamma \multimap * \mid \emptyset \vdash X : \Gamma \multimap *}\ \textbf{(TyVar-I)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap * \qquad \vdash \Gamma\ \textbf{Ctx}}{\Theta, X : \Gamma \multimap * \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *}\ \textbf{(TyVar-Weak)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap * \qquad \Gamma_1 \vdash B : *}{\Theta \mid \Gamma_1, x : B \vdash A : \Gamma_2 \multimap *}\ \textbf{(Ty-Weak)}$$

$$\frac{\Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \multimap * \qquad \Gamma_1 \vdash t : B}{\Theta \mid \Gamma_1 \vdash A\,@\,t : \Gamma_2[t/x] \multimap *}\ \textbf{(Ty-Inst)}$$

$$\frac{\Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \multimap *}{\Theta \mid \Gamma_1 \vdash (x).\,B : (x : A, \Gamma_2) \multimap *}\ \textbf{(Param-Abstr)}$$

$$\frac{\forall 1 \le k \le \left|\overrightarrow{A}\right|. \qquad \sigma_k : \Gamma_k \triangleright \Gamma \qquad \Theta, X : \Gamma \multimap * \mid \Gamma_k \vdash A_k : * \qquad \rho \in \{\mu, \nu\}}{\Theta \mid \emptyset \vdash \rho(X : \Gamma \multimap *; \overrightarrow{\sigma}; \overrightarrow{A}) : \Gamma \multimap *}\ \textbf{(FP-Ty-}\rho\textbf{)}$$

Figure 7.4.: Judgements for the well-formed types of $\lambda \mathrm{P}\mu$

We will often leave out the type information in the superscript of constructors and destructors. The domain of a constructor $\alpha_k^{\mu(X:\Gamma\multimap *;\overrightarrow{\sigma};\overrightarrow{A})}$ is determined by A_k and its codomain by the instantiation σ_k. Dually, the domain of a destructor ξ_k is given by the instantiation σ_k and its codomain by A_k. It is important to note in that both the iteration and coiteration schemes variable binding is introduced: Given terms g_k with $\Delta, \Gamma_k, y_k : A_k[C/X] \vdash g_k : C\,@\,\sigma_k$, the iterator iter $\overrightarrow{(\Gamma_k, y_k.\, g_k)}$ binds the variables in Γ_k and the variable y_k of each g_k. For instance, this will give us the variable binding that happens in λ-abstraction, see Example 7.2.4.

This concludes the definition of our proposed calculus $\lambda \mathrm{P}\mu$. Note that there are no primitive type constructors for $\to$-, Π- or $\exists$-types, all of these are, together with the corresponding introduction and elimination principles, definable in the above calculus, as we will see in the next section.

As the alert reader might have noticed, our calculus does not have dependent recursion and corecursion, that is, the type C in **(Ind-E)** and **(Coind-I)** cannot depend on elements of the corresponding recursive type. This clearly makes the calculus weaker than if we had the dependent version: In the case of inductive types we do not have an induction principle, cf. Example 7.2.6. We will come back to this issue in Section 7.4. For coinductive types, on the other hand, one cannot even formulate a dependent version of **(Coind-I)**, rather one would expect a coinduction rule that turns a bisimulation into an equality proof. This would imply that we have an extensional function space, see Example 7.2.4. How to possibly add such a bisimulation proof principle will be discussed in the future work in Section 7.6.

$$\frac{}{\vdash \langle\rangle : \top}\ (\top\textbf{-I}) \qquad \frac{\Gamma_1 \vdash t : (x : A, \Gamma_2) \multimap B \qquad \Gamma_1 \vdash s : A}{\Gamma_1 \vdash t \mathbin{@} s : \Gamma_2[s/x] \multimap B[s/x]}\ \textbf{(Inst)}$$

$$\frac{\Gamma \vdash t : A \qquad A \longleftrightarrow_T B}{\Gamma \vdash t : B}\ \textbf{(Conv)}$$

$$\frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}\ \textbf{(Proj)} \qquad \frac{\Gamma_1 \vdash t : \Gamma_2 \multimap A \qquad \Gamma_1 \vdash B : *}{\Gamma_1, x : B \vdash t : \Gamma_2 \multimap A}\ \textbf{(Weak)}$$

$$\frac{\vdash \mu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap * \qquad 1 \le k \le |\vec{A}|}{\vdash \alpha_k^{\mu(X:\Gamma\multimap *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : A_k[\mu/X]) \multimap \mu \mathbin{@} \sigma_k}\ \textbf{(Ind-I)}$$

$$\frac{\vdash \nu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap * \qquad 1 \le k \le |\vec{A}|}{\vdash \xi_k^{\nu(X:\Gamma\multimap *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : \nu \mathbin{@} \sigma_k) \multimap A_k[\nu/X]}\ \textbf{(Coind-E)}$$

$$\frac{\Delta \vdash C : \Gamma \multimap * \qquad \Delta, \Gamma_k, y_k : A_k[C/X] \vdash g_k : (C \mathbin{@} \sigma_k) \qquad \forall k = 1, \dots, |\vec{A}|}{\Delta \vdash \mathrm{iter}\ \overrightarrow{(\Gamma_k, y_k . g_k)} : (\Gamma, y : \mu \mathbin{@} \mathrm{id}_\Gamma) \multimap C \mathbin{@} \mathrm{id}_\Gamma}\ \textbf{(Ind-E)}$$

$$\frac{\Delta \vdash C : \Gamma \multimap * \qquad \Delta, \Gamma_k, y_k : (C \mathbin{@} \sigma_k) \vdash g_k : A_k[C/X] \qquad \forall k = 1, \dots, |\vec{A}|}{\Delta \vdash \mathrm{coiter}\ \overrightarrow{(\Gamma_k, y_k . g_k)} : (\Gamma, y : C \mathbin{@} \mathrm{id}_\Gamma) \multimap \nu \mathbin{@} \mathrm{id}_\Gamma}\ \textbf{(Coind-I)}$$

Figure 7.5.: Judgements for well-formed terms of $\lambda \mathrm{P}\mu$

7.2. Examples

In this section, we illustrate the calculus given in Section 7.1 on a variety of examples. We begin with a few basic ones, then work our way through the encoding of logical operators, and finish with lists indexed by their length (vectors) as an actually recursive type.

Before we go through the examples, let us introduce a notation for sequences of empty context morphisms. We denote such a sequence of k empty context morphisms by

$$\varepsilon_k \coloneqq ((), \dots, ()).$$

Given contexts $\Gamma_1, \dots, \Gamma_k$, we then have that ε_k is a sequence of context morphisms $(\Gamma_1 \triangleright \emptyset, \dots, \Gamma_k \triangleright \emptyset)$.

Let us start with one of the most basic types: the singleton type. This first example also gives us the opportunity to explain the role of the base type $\top$.

Example 7.2.1 (Final Object)**.** We first note that, in principle, we can encode $\top$ as a coinductive type by $\mathbf{1} \coloneqq \nu(X : * ; \varepsilon_1 ; X)$:

$$\frac{X : * \mid \emptyset \vdash X : * \qquad \varepsilon_1 : \emptyset \triangleright \emptyset}{\vdash \nu(X : * ; \varepsilon_1 ; X) : *}\ \textbf{(FP-Ty)}$$

This gives us the destructor $\xi_1 : (x : \mathbf{1}) \multimap \mathbf{1}$ and the inference

$$\frac{\vdash C : * \qquad y : C \vdash y : C}{\vdash \mathrm{coiter}\,(y.\, y) : (y : C) \multimap \mathbf{1}}\ \textbf{(Coind-I)}$$

So the analogue of the categorical concept of the (unique) morphism into a final object is given by $!_C \coloneqq \mathrm{coiter}\,(y.\, y)$. Note that it is not possible to define a closed term of type $\mathbf{1}$ directly, rather we only get one with the help of $\top$ by $\langle\rangle' \coloneqq !_\top \mathbin{@} \langle\rangle$. Thus, the purpose of $\top$ is to allow the formation of closed terms. Now, these definitions and $\widehat{X}(t) = t$, see Def. 7.1.5, give us the following reduction.

$$\begin{aligned}
\xi_1 \mathbin{@} \langle\rangle' &= \xi_1 \mathbin{@} (\mathrm{coiter}\,(y.\, y) \mathbin{@} \langle\rangle) \\
&\longrightarrow \widehat{X}(\mathrm{coiter}\,(y.\, y) \mathbin{@} x)[\langle\rangle/y] \\
&= (\mathrm{coiter}\,(y.\, y) \mathbin{@} x)[y/x][\langle\rangle/y] \\
&= \mathrm{coiter}\,(y.\, y) \mathbin{@} \langle\rangle \\
&= \langle\rangle'
\end{aligned}$$

Hence, $\langle\rangle'$ is the canonical element of $\mathbf{1}$ with no observable behaviour. ◀

Dual to the final object $\mathbf{1}$, we can form an initial object.

Example 7.2.2. We put $\mathbf{0} \coloneqq \bot \coloneqq \mu(X : *; \varepsilon_1; X)$, dual to the definition of $\mathbf{1}$. For a given type C, we can define the usual elimination principle for falsum by $E_C^\bot \coloneqq \mathrm{iter}\,(y.\, y)$. As expected, we have

$$\frac{\Gamma \vdash C : *}{\Gamma \vdash E_C^\bot : (y : \bot) \multimap C}$$ ◀

Let us now move to more complex type formers, or logical connectives under the propositions-as-types interpretation.

Example 7.2.3 (Binary Product and Coproduct)**.** Suppose we are given types $\Gamma \vdash A_1, A_2 : *$, then their binary product is fully specified by the two projections and pairing. Thus, we can use the following coinductive type for $A_1 \times_\Gamma A_2$.

$$\frac{\Gamma \vdash A_1 : * \qquad \Gamma \vdash A_2 : *}{\Gamma \vdash \nu(X : \Gamma \multimap *; (\mathrm{id}_\Gamma, \mathrm{id}_\Gamma); (A_1, A_2)) \mathbin{@} \mathrm{id}_\Gamma : *}$$

Then the projections are given by $\pi_k \coloneqq \xi_k \mathbin{@} \mathrm{id}_\Gamma$, and pairing by $\langle t_1, t_2\rangle \coloneqq P_{t_1,t_2} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} \langle\rangle$, where we abbreviate $P_{t_1,t_2} \coloneqq \mathrm{coiter}\,((\Gamma, \_.\, t_1), (\Gamma, \_.\, t_2))$. For this definition of pairing, we have the following expected typing derivation.

$$\dfrac{\dfrac{\vdash (\Gamma).\, \top : \Gamma \multimap * \qquad \dfrac{\Gamma \vdash t_k : A_k}{\Gamma, \_ : \top \vdash t_k : A_k}}{\vdash P_{t_1,t_2} : (\Gamma, \_ : \top) \multimap A_1 \times_\Gamma A_2}}{\Gamma \vdash \langle t_1, t_2\rangle : A_1 \times_\Gamma A_2}$$

This setup gives us the usual reductions for $k \in \{1, 2\}$:

$$\begin{aligned}
\pi_k \mathbin{@} \langle t_1, t_2\rangle &= \xi_k \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} (P_{t_1,t_2} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} \langle\rangle) \\
&\longrightarrow \widehat{A_k}(P_{t_1,t_2} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x)[t_k/y][(\mathrm{id}_\Gamma, \langle\rangle)] \\
&= y[t_k/y][(\mathrm{id}_\Gamma, \langle\rangle)] \\
&= t_k,
\end{aligned}$$

where the third step is given by $\widehat{A_k} = y$, since A_k does not use type constructor variables, see Def. 7.1.5.

Dually, the binary coproduct of A_1 and A_2 is given by

$$A_1 +_\Gamma A_2 \coloneqq \mu(X : \Gamma \multimap * ; (\mathrm{id}_\Gamma, \mathrm{id}_\Gamma); (A_1, A_2)) \mathbin{@} \mathrm{id}_\Gamma,$$

the corresponding injections by $\kappa_i \coloneqq \alpha_i \mathbin{@} \mathrm{id}_\Gamma$, and we can form the case distinction

$$\{\kappa_1\, x_1 \mapsto t_1 \;;\; \kappa_2\, x_2 \mapsto t_2\}\, s \coloneqq \mathrm{iter}\,((\Gamma, x.\, t_1), (\Gamma, x.\, t_2)) \mathbin{@} s,$$

which is subject to the following typing rule.

$$\frac{\Gamma \vdash C : * \qquad \Gamma, x_k : A_k \vdash t_k : C \qquad \Gamma \vdash s : A_1 +_\Gamma A_2}{\Gamma \vdash \{\kappa_1\, x_1 \mapsto t_1 \;;\; \kappa_2\, x_2 \mapsto t_2\}\, s : C}$$

Moreover, we get the expected reduction:

$$\{\kappa_1\, x_1 \mapsto t_1 \;;\; \kappa_2\, x_2 \mapsto t_2\}\, (\kappa_k \mathbin{@} s) \longrightarrow t_k[s/x_k].$$

Thus, we have recovered binary products and coproducts with their introduction and elimination rules, and the corresponding reduction rules in $\lambda\mathrm{P}\mu$. These definitions correspond to our result in Theorem 6.2.11 and the table that follows this theorem. ◀

Next, we show how to recover the dependent function space as coinductive type and dependent sums as inductive types in $\lambda\mathrm{P}\mu$. This follows again Theorem 6.2.11.

Example 7.2.4 (Dependent Product, Universal Quantifier)**.** Given types A and B with $\vdash A : *$ and $x : A \vdash B : *$, we expect the dependent product of B to be a type with $\vdash \Pi x : A.\, B : *$. This leads us to use $\emptyset$ as global context for the dependent product and $\Gamma_1 = x : A$ as local context. Since we have

$$\textbf{(FP-Ty)}\ \frac{\textbf{(TyVar-Weak)}\ \dfrac{x : A \vdash B : *}{X : * \mid x : A \vdash B : *} \qquad \varepsilon_1 : \Gamma_1 \triangleright \emptyset}{\vdash \nu(X : * ; \varepsilon_1 ; B) : *}$$

we may define $\Pi x : A.B \coloneqq \nu(X : * ; \varepsilon_1 ; B)$. If t is a term with $\Gamma, x : A \vdash t : B$, we get its λ-abstraction by putting $\lambda x.t \coloneqq \mathrm{coiter}\,(x, u.\, t) \mathbin{@} \langle\rangle$ for some fresh variable u. This term is indeed correctly typed:

$$\cfrac{\cfrac{\cfrac{\Gamma, x : A \vdash t : B \qquad \Gamma, x : A \vdash \top : *}{\Gamma, x : A, u : \top \vdash t : B}\,\textbf{(Weak)}}{\Gamma \vdash \mathrm{coiter}\,(x, u.\, t) : (u : \top) \multimap \Pi x : A.B}\,\textbf{(Coind-I)}}{\Gamma \vdash \lambda x.t : \Pi x : A.B}\,\textbf{(Inst)}$$

Function application is given by destructor application: $s\ a \coloneqq \xi_1 \mathbin{@} a \mathbin{@} s$. Since we have that $B[\Pi x : A.B/X] = B$ and $(\Pi x : A.B)[\varepsilon_1] = \Pi x : A.B$, we can derive the typing rule for application.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash s : \Pi x : A.B}{\Gamma \vdash s\ a : B[a/x]}$$

In particular, we have that $(\lambda x.t)\ a$ is well-typed, and we can derive the usual β-reduction:

$$\begin{aligned}(\lambda x.t)\ a &= \xi_1 \mathbin{@} a \mathbin{@} (\mathrm{coiter}\ (x, u.\ t) \mathbin{@} \langle\rangle) \\ &\longrightarrow \widehat{B}(\mathrm{coiter}\ (x, u.\ t) \mathbin{@} x')[t/y][\langle\rangle/u, a/x] \\ &= x'[y/x'][t/y][\langle\rangle/u, a/x] \\ &= t[a/x],\end{aligned}$$

where we again use that $X \notin \mathrm{fv}(B)$. As customary, we can derive from the dependent function space also the non-dependent function space by putting

$$A \to B \coloneqq \Pi x : A.B \text{ if } x \notin \mathrm{fv}(B).$$

We can now extend the correspondence between variables in context and terms of function type to parameters as follows. First, from $\Gamma \vdash r : (x : A) \multimap B$, we get $\Gamma, x : A \vdash r \mathbin{@} x : B$. Next, for $\Gamma, x : A \vdash s : B$ we can form $\lambda x.s$, and finally a term $\Gamma \vdash t : A \to B$ gives rise to $\Gamma, x : A \vdash t\ x : B$. This situation can be summarised as follows.

$$\begin{array}{c}\Gamma \vdash r : (x : A) \multimap B \\ \hline \Gamma, x : A \vdash s : B \\ = = = = = = = = = \\ \Gamma \vdash t : \Pi x : A.B\end{array}$$

Here, a single line is a downwards correspondence, and the dashed double line signifies a two-way, though not one-to-one, correspondence. These correspondences allow us, for example, to give the product projections the function type $A_1 \times_\Gamma A_2 \to A_k$, and to write $\pi_k\ t$ instead of $\pi_k \mathbin{@} t$. ◂

Let us briefly divert here on the issue of non-positive types and non-strictly positive types.

Example 7.2.5. We claimed earlier that type system only allows the formation of strictly positive types. Let us show why this is indeed the case. Suppose we would want to form the type $X \to B$ for some variable X and type B. Recall that $X \to B = \nu(Y : * ; \varepsilon_1 ; B)$ with $\varepsilon_1 : (x : X) \triangleright \emptyset$. However, for this to be a valid context morphism, we would need to derive $\vdash x : X$ **Ctx** (note the empty context), which is not possible according to Def. 7.1.8. Hence, we cannot form $X \to B$ as a type in $\lambda \mathrm{P}\mu$, which prevents us from having non(-strictly) positive recursive types. ◂

After this interlude, let us continue with the dual of the dependent product type.

Example 7.2.6 (Dependent Coproduct, Existential Quantifier)**.** In this example we show how dependent coproducts/existential quantifier arise $\lambda \mathrm{P}\mu$. Recall that we do not have dependent iteration, hence no induction principle, in $\lambda \mathrm{P}\mu$, cf. Section 7.4. This means that we are not able to encode Σ-types à la Martin-Löf with projections, see Note 47 of Chapter 6 and the later Example 7.4.5. Instead, we can define intuitionistic existential quantifiers, see 11.4.4 and 10.8.2 in [TvD88]. In fact, $\exists$-types occur as the dual of dependent products (Ex. 7.2.4) as follows.

Let $x : A \vdash B : *$ and put $\exists x : A.\, B \coloneqq \mu(X : * ; \varepsilon_1 ; B)$ for some fresh type variable X. The pairing of terms t and s is given by $\langle t, s\rangle \coloneqq \alpha_1 \mathbin{@} t \mathbin{@} s$. One can easily derive that $\vdash \exists x : A.\, B : *$ and

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B[t/x]}{\Gamma \vdash \langle t, s\rangle : \exists x : A.\, B}$$

from **(Ind-I)** and **(Inst)**. Equally easy is also the derivation that the elimination principle for existential quantifiers, defined by

$$\textbf{unpack } t \textbf{ as } \langle x, y\rangle \textbf{ in } p \coloneqq \mathrm{iter}\,(x : A, y : B.\, p) \mathbin{@} t,$$

can be formed by the following rule.

$$\frac{\vdash C : * \qquad \Gamma, x : A, y : B \vdash p : C \qquad \Gamma \vdash t : \exists x : A.B}{\Gamma \vdash \textbf{unpack } t \textbf{ as } \langle x, y\rangle \textbf{ in } p : C}$$

Finally, we get the usual reduction rule

$$\textbf{unpack } \langle t, s\rangle \textbf{ as } \langle x, y\rangle \textbf{ in } p \longrightarrow p\,[t/x, s/y]\,.$$

Even though we do not have a strong enough elimination principle for $\exists$-types to define the second projection of pairs, we are still able to define the first projection by

$$\pi_1 \coloneqq \lambda z.\, \textbf{unpack } z \textbf{ as } \langle x, y\rangle \textbf{ in } x.$$

Indeed, one easily derives that $\vdash \pi_1 : (\exists x : A.\, B) \to A$. ◂

Example 7.2.7 (Generalised Dependent Product and Coproduct)**.** From a categorical perspective, it makes sense to not just consider product and coproducts that bind a variable in a type but also to allow the restriction of terms we allow as values for this variable. We can achieve this by replacing ε_1 in Ex. 7.2.4 and Ex. 7.2.6 by an arbitrary term $x : I \vdash f : J$. This gives us type constructors with

$$y : J \vdash \coprod_f A : * \quad \text{ and } \quad y : J \vdash \prod_f A : *$$

that are weakly adjoint $\coprod_f \dashv f^* \dashv \prod_f$, where f^* substitutes f. Similarly, propositional equality arises as left adjoint to contraction δ^*, where $\delta : (x : A) \triangleright (x : A, y : A)$ is the diagonal substitution $\delta \coloneqq (x, x)$, see Example 7.2.8 below. ◂

Example 7.2.8 (Propositional Equality)**.** In Ex. 7.2.7, we remarked that the propositional equality arises as left adjoint to the contraction δ^*, cf. [Jac99, Def. 10.5.1], hence can be represented in $\lambda\mathrm{P}\mu$ as a generalised dependent coproduct. Let us elaborate this a bit more. First, we represent the equality type in $\lambda\mathrm{P}\mu$ as an inductive type by

$$\mathrm{Eq}_A(s, t) \coloneqq \mu(X : (x : A, y : A) \multimap * ; \delta; \top) \mathbin{@} s \mathbin{@} t,$$

where $\delta : (x : A) \triangleright (x : A, y : A)$ is the diagonal $\delta = (x, x)$. This type has the usual constructor

$$\begin{aligned} &\mathrm{refl} : \Pi x : A.\, \mathrm{Eq}_A(x, x) \\ &\mathrm{refl} \coloneqq \lambda x.\, \alpha_1 : (x : A) \multimap \mathrm{Eq}_A(x, x) \end{aligned}$$

and the (weak) elimination rule

$$\frac{x : A, y : A \vdash C : * \qquad x : A \vdash p : C[x/x, x/y] \qquad \Gamma \vdash q : \mathrm{Eq}_A(s, t)}{\Gamma \vdash E^{\mathrm{Eq}}(p, q) : C\,[s/x, t/y]}$$

where $E^{\mathrm{Eq}}(p,q) \coloneqq \mathrm{iter}\,(x.\,p) \mathbin{@} s \mathbin{@} t \mathbin{@} q$. This is of course not the full J-rule (or path induction) but it is already strong enough to prove, for example, the following replacement (or substitution or transport) rule that allows us to replace equal terms in types.

$$\frac{\Gamma, x : A \vdash P : * \qquad \Gamma \vdash p : P[s/x] \qquad \Gamma \vdash q : \mathrm{Eq}_A(s,t)}{\Gamma \vdash \mathrm{repl}(p,q) : P[t/x]}$$

This can be derived by using $P[x/x] \to P[y/x]$ for C and putting

$$\mathrm{repl}(p,q) \coloneqq E^{\mathrm{Eq}}(\lambda r.\, r, q)\, p.$$

That this term is of the correct type can be seen by the following derivation.

$$\dfrac{\dfrac{\overline{\Gamma \vdash \lambda r.\, r : P[x/x] \to P[x/x]} \qquad \Gamma \vdash q : \mathrm{Eq}_A(s,t)}{\Gamma \vdash E^{\mathrm{Eq}}(\lambda r.\, r, q) : P[s/x] \to P[t/x]} \qquad \Gamma \vdash p : P[s/x]}{\Gamma \vdash E^{\mathrm{Eq}}(\lambda r.\, r, q)\, p : P[t/x]}$$

As such, this derivation is a typical example of higher-order iteration, similar to Example 3.1.7. ◂

After we have recovered all of basic Martin-Löf type theory, with the caveat that the existential quantifier does not have a second projection, it is time to move to truly recursive types. The first type is that of the natural numbers, which we will use in the definition of length-indexed lists.

Example 7.2.9. We can define the type of natural numbers in $\lambda \mathrm{P}\mu$ by

$$\mathrm{Nat} \coloneqq \mu(X : *\,; \varepsilon_2; (\top, X)),$$

with contexts $\Gamma = \Gamma_1 = \Gamma_2 = \emptyset$. We get the usual constructors for the zero and the successor:

$$0 = \alpha_1^{\mathrm{Nat}} \mathbin{@} \langle\rangle : \mathrm{Nat} \quad \text{and} \quad s = \alpha_2^{\mathrm{Nat}} : (y : \mathrm{Nat}) \rightarrowtriangle \mathrm{Nat}.$$

Moreover, we obtain the standard iteration principle:

$$\frac{\vdash t_0 : C \qquad y : C \vdash t_s : C}{\vdash \mathrm{iter}\,(t_0, (y.\, t_s)) : (y : \mathrm{Nat}) \rightarrowtriangle C}$$

◂

The reader will have noticed that this representation of the natural numbers is essentially the same as the definition that we gave in Example 3.1.2. In the calculus $\lambda \mathrm{P}\mu$ it is only easier to give the types of the two constructors separately. We could have just as well represented Nat as $\mu(X : *\,; \varepsilon_1; \top + X)$ by using the coproduct from Example 7.2.3. More generally, all types from Definition 3.1.1 together with the corresponding terms of the simple calculus $\lambda\mu\nu$ can be represented in $\lambda \mathrm{P}\mu$ by appealing to the above examples. ◂

Let us show now how length-indexed lists (vectors) arise as a dependent recursive in $\lambda \mathrm{P}\mu$.

Example 7.2.10 (Vectors)**.** We define vectors $\mathrm{Vec}\,A : (n : \mathrm{Nat}) \rightarrowtriangle *$, which are lists over A indexed by their length, as follows.

$$\begin{aligned}
&\mathrm{Vec}\,A \coloneqq \mu(X : \Gamma \rightarrowtriangle *\,; (\sigma_1, \sigma_2); (\top, A \times X \mathbin{@} k))\\
&\Gamma = n : \mathrm{Nat} \quad \text{and} \quad \Gamma_1 = \emptyset \quad \text{and} \quad \Gamma_2 = k : \mathrm{Nat}\\
&\sigma_1 = (0) : \Gamma_1 \triangleright (n : \mathrm{Nat}) \quad \text{and} \quad \sigma_2 = (s \mathbin{@} k) : \Gamma_2 \triangleright (n : \mathrm{Nat})\\
&X : (n : \mathrm{Nat}) \rightarrowtriangle * \mid \Gamma_1 \vdash \top : *\\
&X : (n : \mathrm{Nat}) \rightarrowtriangle * \mid \Gamma_2 \vdash A \times X \mathbin{@} k : *
\end{aligned}$$

This yields the usual constructors nil $\coloneqq \alpha_1 \mathbin{@} \langle\rangle$ and cons $\coloneqq \alpha_2$, which have the expected types:

$$\begin{aligned}&\alpha_1 : \mathrm{Vec}\, A \mathbin{@} 0\\ &\alpha_2 : (k : \mathrm{Nat}, y : A \times \mathrm{Vec}\, A \mathbin{@} k) \to \mathrm{Vec}\, A \mathbin{@} (s \mathbin{@} k).\end{aligned}$$

The induced iteration scheme for $B : (n : \mathrm{Nat}) \to *$ is then also the expected one:

$$\frac{y : \top \vdash g_1 : B \mathbin{@} 0 \qquad k : \mathrm{Nat}, y : A \times \mathrm{Vec}\, B \mathbin{@} k \vdash g_2 : B \mathbin{@} (s \mathbin{@} k)}{\vdash \mathrm{iter}\,((y.\, g_1), (k, y.\, g_2)) : (n : \mathrm{Nat}, y : \mathrm{Vec}\, A \mathbin{@} n) \to B \mathbin{@} n}$$

We can use this scheme to, for example, define the function map that applies a function entry-wise to vectors. Let $x : A \vdash f : B$ be a term, then we define map f as follows. First, we put $C = \mathrm{Vec}\, B$, $g_1 = \alpha_1 \mathbin{@} y$ and $g_2 = \mathrm{cons} \mathbin{@} k \mathbin{@} \langle f[\pi_1\ y/x], \pi_2\ y\rangle$. To define map f, we instantiate the above iteration scheme:

$$\frac{y : \top \vdash g_1 : \mathrm{Vec}\, B \mathbin{@} 0 \qquad k : \mathrm{Nat}, y : A \times \mathrm{Vec}\, B \mathbin{@} k \vdash g_2 : \mathrm{Vec}\, B \mathbin{@} (s \mathbin{@} k)}{\vdash \mathrm{map}\ f = \mathrm{iter}\,((y.\, g_1), (k, y.\, g_2)) : (n : \mathrm{Nat}, y : \mathrm{Vec}\, A \mathbin{@} n) \to \mathrm{Vec}\, B \mathbin{@} n}$$

It is straightforward to establish the expected reduction rules

$$\begin{gathered}\mathrm{map}\ f \mathbin{@} 0 \mathbin{@} \mathrm{nil} \longrightarrow \mathrm{nil} \text{ and}\\ \mathrm{map}\ f \mathbin{@} (k+1) \mathbin{@} (\mathrm{cons} \mathbin{@} k \mathbin{@} u) \longrightarrow \mathrm{cons} \mathbin{@} k \mathbin{@} \langle f[\pi_1\ u/x], \mathrm{map}\ f \mathbin{@} (\pi_2\ u)\rangle,\end{gathered}$$

by using that

$$\begin{aligned}&y : \top \vdash \widehat{\top}(\mathrm{map} f \mathbin{@} k \mathbin{@} x) : \top\\ &\widehat{\top}(\mathrm{map} f \mathbin{@} k \mathbin{@} x) = y\end{aligned}$$

and

$$\begin{aligned}&k : \mathrm{Nat}, y : A \times \mathrm{Vec}\ A \vdash \widehat{A \times X}(\mathrm{map} f \mathbin{@} k \mathbin{@} x) : A \times \mathrm{Vec}\ B\\ &\widehat{A \times X}(\mathrm{map} f \mathbin{@} k \mathbin{@} x) = \langle \pi_1 \mathbin{@} y, \mathrm{map} f \mathbin{@} k \mathbin{@} (\pi_2 \mathbin{@} y)\rangle,\end{aligned}$$

both of which can be derived from Definition 7.1.5. ◀

At this point, we could also recast the substream example from Example 6.2.13 in $\lambda \mathrm{P}\mu$. We, however, leave this to the applications in Section 7.5.3.

7.3. Meta Properties

To make the calculus $\lambda \mathrm{P}\mu$ useful as a logic we better ensured that the calculus is consistent as such. In particular, we have to show that types of terms are preserved by reductions and that all terms are strongly normalising. The former property ensures thereby that, for example, well-formedness of types does not break in the conversion rule. Strong normalisation, on the other hand, guarantees that types like $\bot$ from Example 7.2.2 are not inhabited, as we could otherwise use the elimination of $\bot$ to prove any proposition. Before we come to these more interesting properties, we need to prove some basic structural rules that allow us to manipulate variable contexts.

7.3.1. Derivable Structural Rules

Proposition 7.3.1. *The following rules holds for the calculus $\lambda P\mu$.*

- *Substitution*

$$\frac{\Theta \mid \Gamma_1, x : A, \Gamma_2 \vdash B : \Gamma_3 \multimap * \qquad \Gamma_1 \vdash t : A}{\Theta \mid \Gamma_1, \Gamma_2[t/x] \vdash B[t/x] : \Gamma_3[t/x] \multimap *}$$

$$\frac{\Gamma_1, x : A, \Gamma_2 \vdash s : B \qquad \Gamma_1 \vdash t : A}{\Gamma_1, \Gamma_2[t/x] \vdash s[t/x] : B[t/x]}$$

- *Exchange*

$$\frac{\Theta \mid \Gamma_1, x : A, y : B, \Gamma_2 \vdash C : \Gamma_3 \multimap * \qquad x \notin \mathrm{fv}(B)}{\Theta \mid \Gamma_1, y : B, x : A, \Gamma_2 \vdash C : \Gamma_3 \multimap *}$$

$$\frac{\Gamma_1, x : A, y : B, \Gamma_2 \vdash t : \Gamma_3 \multimap C \qquad x \notin \mathrm{fv}(B)}{\Gamma_1, y : B, x : A, \Gamma_2 \vdash t : \Gamma_3 \multimap C}$$

- *Contraction*

$$\frac{\Theta \mid \Gamma_1, x : A, y : A, \Gamma_2 \vdash C : \Gamma_3 \multimap *}{\Theta \mid \Gamma_1, x : A, \Gamma_2[x/y] \vdash C[x/y] : \Gamma_3[x/y] \multimap *}$$

$$\frac{\Gamma_1, x : A, y : A, \Gamma_2 \vdash t : \Gamma_3 \multimap C}{\Gamma_1, x : A, \Gamma_2[x/y] \vdash t[x/y] : \Gamma_3[x/y] \multimap C[x/y]}$$

Proof. In each case, the rules are straightforwardly proved by simultaneous induction over types and terms. It should be noted that for types only the instantiation and weakening rules appear as cases, since the other rules have only types without free variables in the conclusion. Similarly, only terms constructed by means of the the projection, weakening or the instantiation rule appear as cases in the proofs. □

Analogously, the substitution, exchange and contraction rules for type variables are valid in the calculus, as well. Also these are easily proved by induction on the derivation of the corresponding well-formedness proofs

Proposition 7.3.2. *The following rules for type variable contexts hold in the calculus $\lambda P\mu$.*

- *Substitution*

$$\frac{\Theta_1, X : \Delta \multimap *, \Theta_2 \mid \Gamma_1 \vdash A : \Gamma_2 \multimap * \qquad \Theta_1, \Theta_2 \mid \Gamma_1 \vdash B : \Delta \multimap *}{\Theta_1, \Theta_2 \mid \Gamma_1 \vdash A[B/X] : \Gamma_2 \multimap *}$$

- *Exchange*

$$\frac{\Theta_1, X : \Delta_1 \multimap *, Y : \Delta_2 \multimap *, \Theta_2 \mid \Gamma_1 \vdash C : \Gamma_2 \multimap *}{\Theta_1, Y : \Delta_2 \multimap *, X : \Delta_1 \multimap *, \Theta_2 \mid \Gamma_1 \vdash C : \Gamma_2 \multimap *}$$

- *Contraction*

$$\frac{\Theta_1, X : \Delta_1 \multimap * , Y : \Delta_2 \multimap * , \Theta_2 \mid \Gamma_1 \vdash C : \Gamma_2 \multimap *}{\Theta_1, X : \Delta_1 \multimap * , \Theta_2 \mid \Gamma_1 \vdash C[X/Y] : \Gamma_2 \multimap *}$$

It should be noted that the analogues of Proposition 7.3.1 and Proposition 7.3.2 are proved for pre-types and pre-terms in the Agda formalisation [Bas18b]. These are necessary there to construct the reduction relation.

7.3.2. Subject Reduction

Let us now come to a more interesting property: the preservation of types by the reduction relation. This result is stated in Theorem 7.3.4. Towards the proof of it, we need the the following key lemma first, which essentially states that the action of types on terms acts like a functor.

Lemma 7.3.3 (Type correctness of type action)**.** *Given the action of types on terms, see Def. 7.1.5, the following inference rule holds.*

$$\frac{X : \Gamma_1 \multimap * \mid \Gamma_2' \vdash C : \Gamma_2 \multimap * \qquad \Gamma_1, x : A \vdash t : B}{\Gamma_2', \Gamma_2, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

Proof. We want to prove this lemma by induction on the derivation of $X : \Gamma_1 \multimap * \mid \Gamma_2' \vdash C : \Gamma_2 \multimap *$, thus we need to generalise the statement to arbitrary type constructor contexts Θ. So assume that Θ is a context, such that $\Theta = X_1 : \Gamma_1 \multimap *, \dots, X_n : \Gamma_n \multimap *$, that $\Theta \mid \Gamma' \vdash C : \Gamma \multimap *$ and $\Gamma_i, x : A_i \vdash t_i : B_i$ terms for $i = 1, \dots, n$. Our goal is then to show that $\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})$, which we can do now by induction on the derivation of $\Theta \mid \Gamma' \vdash C : \Gamma \multimap *$.

The induction base has two cases. First, it is clear that if $\Theta = \emptyset$, then $\vec{A} = \varepsilon$ and $\widehat{C}(\vec{A}) = C$, thus the definition is thus well-typed. Second, if $C = X_i$ for some i, then we immediately have

$$\widehat{C}(\vec{A}) = C[\overrightarrow{(\Gamma_i).A_i}/\vec{X}]\,\mathrm{id}_{\Gamma_i} = ((\Gamma_i).A_i)\,\mathrm{id}_{\Gamma_i} \longrightarrow_p A_i,$$

thus, by **(Conv)** and the type of t_i, we have $\Gamma_i, x : \widehat{C}(\vec{A}) \vdash t_i : \widehat{C}(\vec{B})$ as required.

In the induction step, we have five cases for C.

- The type correctness for $\widehat{C}$ in case C has been constructed by Weakening for type and term variables is immediate by induction and the definition of F in these cases.
- $C = C' \mathbin{@} s$ and $\Gamma = \Delta[s/y]$ with

$$\frac{\Theta \mid \Gamma' \vdash C' : (y : D, \Delta) \multimap * \qquad \Gamma' \vdash s : D}{\Theta \mid \Gamma' \vdash C' \mathbin{@} s : \Delta[s/y] \multimap *}$$

  By induction we have then that $\Gamma', y : D, \Delta, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{A})$, thus, since

$$\begin{aligned}\widehat{C'}(\vec{A}) &= C'[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} \mathrm{id}_{y:D,\Delta} \\ &= C'[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} y \mathbin{@} \mathrm{id}_{\Delta},\end{aligned}$$

we get by Prop. 7.3.1

$$\Gamma', \Delta[s/y], x : C'[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} s \mathbin{@} \mathrm{id}_{\Delta[s/y]} \vdash \widehat{C'}(\vec{t})[s/y] : C'[\overrightarrow{(\Gamma_1).B}/\vec{X}] \mathbin{@} s \mathbin{@} \mathrm{id}_{\Delta[s/y]}.$$

As we now have

$$\begin{aligned}\widehat{C' \mathbin{@} s}(\vec{A}) &= (C' \mathbin{@} s)[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} \mathrm{id}_{\Delta[s/y]} \\ &= C'[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} s \mathbin{@} \mathrm{id}_{\Delta[s/y]}\end{aligned}$$

and $\widehat{C' \mathbin{@} s}(\vec{t}) = \widehat{C'}(\vec{t})[s/y]$, we find that

$$\Gamma', \Delta[s/y], x : \widehat{C' \mathbin{@} s}(\vec{A}) \vdash \widehat{C' \mathbin{@} s}(\vec{t}) : \widehat{C' \mathbin{@} s}(\vec{B})$$

as expected.

- $C = (y).C'$ with $\Theta \mid \Gamma', y : D \vdash C' : \Gamma \multimap *$. This gives us, by the induction hypothesis, $\Gamma', y : D, \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})$. Now we observe that

$$\begin{aligned}\widehat{C'}(\vec{A}) &= C'[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} \mathrm{id}_\Gamma \\ &\longleftarrow_p ((y).C'[\overrightarrow{(\Gamma_1).A}/\vec{X}]) \mathbin{@} y \mathbin{@} \mathrm{id}_\Gamma \\ &= C[\overrightarrow{(\Gamma_1).A}/\vec{X}] \mathbin{@} \mathrm{id}_{y:D,\Gamma} \\ &= \widehat{C}(\vec{A}),\end{aligned}$$

which gives us, by **(Conv)**, that $\Gamma', y : D, \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C}(\vec{B})$. Thus the definition $\widehat{(x).C'}(\vec{t}) = \widehat{C'}(\vec{t})$ is well-typed.

- $C = \mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{D})$ with

$$\frac{\Theta, Y : \Gamma \multimap * \mid \Delta_k \vdash D_k : * \qquad \sigma_k : \Delta_k \triangleright \Gamma}{\Theta \mid \emptyset \vdash \mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{D}) : \Gamma \multimap *}$$

For brevity, we define $R_{\vec{B}} = \mu(Y : \Gamma \multimap *; \vec{\sigma}; \vec{D}[\vec{B}/\vec{X}])$. Then, by induction, we have

$$\Gamma, x : \widehat{D_k}(\vec{A}, R_{\vec{B}}) \vdash \widehat{D_k}(\vec{t}, x) : \widehat{D_k}(\vec{B}, R_{\vec{B}})$$

Now we have that $\widehat{D_k}(\vec{A}, R_{\vec{B}}) = D_k[\overrightarrow{(\Gamma_i).A}/\vec{X}][R_{\vec{B}}/Y]$. Note that the second substitution does not contain a parameter abstraction, as $R_{\vec{B}}$ is closed. If we define

$$g_k = \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} \mathbin{@} \left(\widehat{D_k}\,(\vec{t}, \mathrm{id}_{R_{\vec{B}}})\right),$$

where α_k refers to $\alpha_k^{\mu(Y:\Gamma\to *;\vec{\sigma};\vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}])}$ (see the definition of F), then we can derive the following.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \alpha_k : \overline{\Delta_k}(\widehat{D_k}(\vec{A}, R_{\vec{B}}) \to R_{\vec{B}} \mathbin{@} \sigma_k)}{\Delta_k \vdash \alpha_k \mathbin{@} \mathrm{id}_{\Delta_k} : \widehat{D_k}(\vec{A}, R_{\vec{B}}) \to R_{\vec{B}} \mathbin{@} \sigma_k}\,\textbf{(Inst)}}{\Delta_k, x : D_k[\overrightarrow{(\Gamma_i).A}/\vec{X}][R_{\vec{B}}/Y] \vdash g_k : R_{\vec{B}}}\,\textbf{(Inst)}}{\Gamma \vdash \mathrm{iter}\,\overrightarrow{(\Delta_k, y'.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma : R_{\vec{A}} \mathbin{@} \mathrm{id}_\Gamma \to R_{\vec{B}} \mathbin{@} \mathrm{id}_\Gamma}\,\textbf{(Ind-E)}}{\Gamma, x : R_{\vec{A}} \mathbin{@} \mathrm{id}_\Gamma \vdash \mathrm{iter}\,\overrightarrow{(\Delta_k, y'.\, g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x : R_{\vec{B}} \mathbin{@} \mathrm{id}_\Gamma}\,\textbf{(Inst)}$$

Finally, we have

$$\widehat{C}(\vec{A}) = \mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D})[\overrightarrow{(\Gamma_i).A}/\vec{X}] \mathbin{@} \mathrm{id}_\Gamma = R_{\vec{A}} \mathbin{@} \mathrm{id}_\Gamma,$$

which implies, by the above derivitations, that we indeed have

$$\Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B}).$$

- $C = \nu(Y : \Gamma \to *; \vec{\sigma}; \vec{D})$. This case is treated analogously to that for inductive types.

This concludes the induction, thus (7.3.3) indeed holds for types C with one free variable. □

The following is now an easy consequence of Lemma 7.3.3.

Theorem 7.3.4 (Subject reduction)**.** *If* $\Gamma \vdash t_1 : A$ *and* $t_1 \longrightarrow t_2$*, then* $\Gamma \vdash t_2 : A$*.*

Proof. Lemma 7.3.3 gives us immediately subject reduction for the contraction relation. Since the reduction relation is the compatible closure of contraction, type preservation $\longrightarrow$ follows at once. □

7.3.3. Strong Normalisation

This section is devoted to show that all well-formed terms of $\lambda\mathrm{P}\mu$ are strongly normalising, which means that all reduction sequences in the calculus terminate. We would intuitively expect this, given that we introduced the reduction relation by following the homomorphism property of (co)recursion for initial and final dialgebras. But since the proof is not obvious, we fully present it here.

The proof of strong normalisation uses the saturated sets approach, see for example [Geu94]. In this approach, one defines a model of strongly normalising (pre-)terms for the types as follows. First, we define what it means for a set of pre-terms to be saturated, where, most importantly, all terms in a saturated set are strongly normalising. Next, we give an interpretation $[\![A]\!]$ of dependent types A as families of saturated sets. Finally, we show that if $\Gamma \vdash t : A$, then for all assignments ρ of terms to variables in Γ, we have $t \in [\![A]\!](\rho)$. This proof proceeds by induction on the derivation of $\Gamma \vdash t : A$, and the interpretation of terms as saturated sets give us precisely the necessary induction hypothesis for this proof to go through. Since $[\![A]\!](\rho) \subseteq \mathbf{SN}$, strong normalisation for all well-formed terms follows.

Towards this model, we need to define all of the above concepts, starting with a few basic notations.

Definition 7.3.5. We use the following notations.

- Λ is the set of pre-terms.
- **SN** is the set of strongly normalising pre-terms.
- $\lceil\Gamma\rceil$ is the set of variables in context Γ.

For simplicity, we identify context morphisms $\sigma : \Gamma_1 \triangleright \Gamma_2$ and valuations $\rho : \lceil\Gamma_2\rceil \to \Lambda$, if we know that the terms of ρ are typeable in Γ_1. This allows us to write $\sigma(x)$ for $x \in \lceil\Gamma_2\rceil$, and $M @ \rho$ for pre-terms M. It is helpful though to make explicit the action of context morphisms on valuations, essentially given by composition, and write

$$\begin{aligned} &[\![\sigma : \Gamma_1 \triangleright \Gamma_2]\!] : \Lambda^{\lceil\Gamma_1\rceil} \to \Lambda^{\lceil\Gamma_2\rceil} \\ &[\![\sigma : \Gamma_1 \triangleright \Gamma_2]\!](\gamma)(y) = \sigma(y)[\gamma]. \end{aligned} \tag{7.3}$$

Saturated sets are defined by containing certain open terms (base terms) and by being closed under key reductions. The idea of base terms is that they are neutral, i.e. they do not admit any reductions, but it is possible to substitute variables so that new reductions are possible. Moreover, they will ensure that all variables are in the interpretation of types as saturated sets, see Definition 7.3.8. Key redexes, on the other hand, will allow us to prove that saturated sets are backwards closed under constructors for inductive types and coiteration for coinductive types, see Lemma 7.3.22 and Lemma 7.3.24. We introduce these two notions in the following two definitions.

Definition 7.3.6 (Base Terms)**.** The set of *base terms* $\mathcal{B}$ is defined inductively by the following three closure rules.

- $\mathrm{Var} \subseteq \mathcal{B}$
- $\mathrm{iter}\,\overrightarrow{(\Gamma_k, x.\, N_k)} \mathbin{@} \sigma \mathbin{@} M \in \mathcal{B}$, provided that $M \in \mathcal{B}$, $N_k \in \mathbf{SN}$ and $\sigma \in \mathbf{SN}$.
- $\xi_k^{\nu(X:\Gamma\to *;\vec{\tau};\vec{A})} \mathbin{@} \sigma \mathbin{@} M \in \mathcal{B}$, provided that $M \in \mathcal{B}$, $\sigma \in \mathbf{SN}$ and $\exists\gamma.\, (\sigma = [\![\tau_k]\!](\gamma))$. ◀

Definition 7.3.7 (Key Redex)**.** A pre-term M is a *redex*, if there is a P with $M \succ P$. M is the *key redex*

1. of M itself, if M is a redex,
2. of $\mathrm{iter}\,\overrightarrow{(\Gamma_k, y_k.\, N_k)} \mathbin{@} \sigma \mathbin{@} N$, if M the key redex of N, or
3. of $\xi_k \mathbin{@} \sigma \mathbin{@} N$, if M the key redex of N.

We denote by $\mathrm{red}_k(M)$ the term that is obtained by contracting the key redex of M. ◀

Note that a key redex is given both for inductive and coinductive types by the corresponding elimination principles (iteration and destructors, respectively).

We are now in the position to define what a saturated set is.

Definition 7.3.8 (Saturated Sets)**.** A set $X \subseteq \Lambda$ is *saturated*, if

1. $X \subseteq \mathbf{SN}$

2. $\mathcal{B} \subseteq X$
3. If $\mathrm{red}_k(M) \in X$ and $M \in \mathbf{SN}$, then $M \in X$.

We denote by **SAT** the set of all saturated sets. ◂

It is easy to see that $\mathbf{SN} \in \mathbf{SAT}$, and that every saturated set is non-empty. Moreover, it is easy to show that **SAT** is a complete lattice with set inclusion as order. Besides these standard facts, we will use the following constructions on saturated sets.

Definition 7.3.9. Let Γ be a context. We define a semantical context extension (comprehension) of pairs (E, U) with $E \subseteq \Lambda^{\lceil\Gamma\rceil}$ and $U : E \to \mathbf{SAT}$ with respect to a given variable $x \notin \lceil\Gamma\rceil$ by

$$\{(E,U)\}_x = \big\{\rho[x \mapsto M] \;\big|\; \rho \in E \text{ and } M \in U(\rho)\big\}, \tag{7.4}$$

where $\rho[x \mapsto M] : \lceil\Gamma\rceil \cup \{x\} \to \Lambda$ extends ρ by mapping x to M. Moreover, we define a semantical version of the typing judgement:

$$E \Vdash U = \{M \mid \forall \gamma \in E.\ M[\gamma] \in U(\gamma)\}. \tag{7.5}$$

We now show that we can give a model of well-formed types by means of saturated sets. To achieve this, we define simultaneously an interpretation of contexts and the interpretation of types. The intention is that we have that

- if $\vdash \Gamma$ **Ctx**, then $[\![\Gamma]\!]$ is the set of valuations for variables in Γ with $[\![\Gamma]\!] \subseteq \Lambda^{\lceil\Gamma\rceil}$;
- if $\vdash \Theta$ **TyCtx**, then $[\![\Theta]\!](X) \colon [\![\Gamma]\!] \to \mathbf{SAT}$ for all $X : \Gamma \twoheadrightarrow *$ in Θ, and
- if $\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \twoheadrightarrow *$, then $[\![A]\!] : [\![\Theta]\!] \times [\![\Gamma_1, \Gamma_2]\!] \to \mathbf{SAT}$.

Definition 7.3.10 (Interpretations)**.** We interpret type variable contexts, term variable contexts and types simultaneously. First, we assign to each term context is a set of allowed possible valuations:

$$\begin{aligned} [\![\emptyset]\!] &\coloneqq \{! : \emptyset \to \Lambda\} \\ [\![\Gamma, x : A]\!] &\coloneqq \{([\![\Gamma]\!], [\![A]\!])\}_x = \{\rho[x \mapsto M] \mid \rho \in [\![\Gamma]\!] \text{ and } M \in [\![A]\!](\rho)\} \end{aligned}$$

For $\Theta = X_1 : \Gamma_1 \twoheadrightarrow *, \ldots, X_n : \Gamma_n \twoheadrightarrow *$ we define

$$[\![\Theta]\!] \coloneqq \prod_{X_i \in \lceil\Theta\rceil} I_{\Gamma_i},$$

where I_Γ is the set of valuations that respect convertibility:

$$I_\Gamma = \{U : [\![\Gamma]\!] \to \mathbf{SAT} \mid \forall \rho, \rho'.\ \rho \longrightarrow\!\!\!\!\!\rightarrow_T \rho' \Rightarrow U(\rho) = U(\rho')\}$$

Finally, we define in Fig. 7.6 the interpretation of types as families of term sets. In the clause for inductive types, A_k^Δ denotes the type that is obtained by weakening $\Gamma_k \vdash A_k : *$ to $\Delta, \Gamma_k \vdash A_k^\Delta : *$, $\pi : (\Gamma_k, y : A_k^\Delta) \triangleright \Gamma_k$ projects y away, and $[\![\sigma_k \bullet \pi]\!]^*(U) \coloneqq U \circ [\![\sigma_k \bullet \pi]\!]$ is the reindexing for set families. ◂

$$
\begin{aligned}
[\![\vdash \top : *]\!](\delta,\rho) &= \bigcap\{X \in \mathbf{SAT} \mid \langle\rangle \in X\}\\
[\![\Theta, X : \Gamma \multimap * \mid \emptyset \vdash X : \Gamma \multimap *]\!](\delta,\rho) &= \delta(X)(\rho)\\
[\![\Theta, X \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *]\!](\delta,\rho) &= [\![\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *]\!](\delta|_{\lceil\Theta\rceil},\rho)\\
[\![\Theta \mid \Gamma_1, x : B \vdash A : \Gamma_2 \multimap *]\!](\delta,\rho) &= [\![\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *]\!](\delta,\rho|_{\lceil\Gamma_1\rceil})\\
[\![\Theta \mid \Gamma_1 \vdash A\,@\,t : \Gamma_2[t/x] \multimap *]\!](\delta,\rho) &= [\![\Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \multimap *]\!](\delta,\rho[x \mapsto t[\rho]])\\
[\![\Theta \mid \Gamma_1 \vdash (x).A : (x : B, \Gamma_2) \multimap *]\!](\delta,\rho) &= [\![\Theta \mid \Gamma_1, x : B \vdash A : \Gamma_2 \multimap *]\!](\delta,\rho)\\
[\![\Theta \mid \emptyset \vdash \mu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap *]\!](\delta,\rho) &=\\
\big\{M \mid\ & \forall U \in I_\Gamma.\, \forall \Delta.\, \forall k.\, \forall N_k \in \big\{[\![\Gamma_k]\!], [\![A_k^{\Delta}]\!](\delta[X \mapsto U])\big\}_y \Vdash [\![\sigma_k \bullet \pi]\!]^*(U).\\
& \overrightarrow{\mathrm{iter}\,(\Gamma_k, y.\, N_k)}\,@\,\rho\,@\,M \in U(\rho)\big\}\\
[\![\Theta \mid \emptyset \vdash \nu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap *]\!](\delta,\rho) &=\\
\big\{M \mid\ & \exists U \in I_\Gamma.\, \forall k.\, \forall \gamma \in [\![\sigma_k]\!]^{-1}(\rho).\, \xi_k\,@\,\gamma\,@\,M \in [\![A_k]\!](\delta[X \mapsto U], \gamma)\big\}
\end{aligned}
$$

Figure 7.6.: Interpretation of types as families of saturated sets

Most of the clauses in the interpretation in Figure 7.6 are straightforward. The only two interesting cases are the recursive types. In both the inductive and the coinductive case we interpret the types as sets that are closed under the corresponding elimination principles. For inductive case we use thereby a least fixed point, impredicatively defined by the use of a universal quantification. On the other hand, coinductive types are interpreted as a greatest fixed point by means of existential quantification.

Before we continue stating the key results about this interpretation of types, let us briefly look at an example.

Example 7.3.11. Suppose A, B are closed types. Recall that the function space was defined by $A \to B = \nu(X : *; \varepsilon_1 : (x : A) \triangleright \emptyset; B)$, and the application by $t\,a = \xi_1\,@\,a\,@\,t$. Note that the conditition $\gamma \in [\![\varepsilon_1]\!]^{-1}(\rho)$ reduces to $\gamma(x) \in [\![A]\!]$ because $[\![\varepsilon_1]\!](\gamma)(y \in \emptyset) = \rho(y)[\varepsilon_1]$ holds for any $\gamma \in [\![(x : A)]\!]$. So we write N instead of $\gamma(x)$. We further note that, since A, B and thus $A \to B$ are closed, we can leave out the type variable valuation δ. Taking all of this into account, we obtain

$$
\begin{aligned}
[\![A \to B]\!](\gamma) &= \{M \mid \forall N \in [\![A]\!].\, \xi_1\,@\,\gamma\,@\,M \in [\![B]\!]\}\\
&= \{M \mid \forall N \in [\![A]\!].\, M\,N \in [\![B]\!]\},
\end{aligned}
$$

which is the usual definition definition, see [Geu94]. ◂

We just state here the key lemmas and the main result.[65] The interested reader may find the proofs and the supporting lemmas in Section 7.3.4 below.

Lemma 7.3.12 (Soundness of type action)**.** *Suppose C is a type with $\Theta \mid \Gamma \vdash C : \Gamma' \multimap *$ that Θ is a type variable context with $\Theta = X_1 : \Gamma_1 \multimap *, \ldots, X_n : \Gamma_n \multimap *$. Assume further that that for all parameters $\Delta \vdash r : C'$ occurring in C and $\tau : \Delta' \triangleright \Delta$, we have $r[\tau] \in [\![C']\!](\tau)$. Let $\delta_A, \delta_B \in [\![\Theta]\!]$ and $\Gamma_i, x : A_i \vdash t_i : B_i$, such that for all $\sigma \in [\![\Gamma_i]\!]$, $t_i \in \delta_B(X_i)(\tau)$. Then for all contexts Δ, all $\sigma : \Delta \triangleright \Gamma, \Gamma'$*

and all $s \in [\![C]\!](\delta_A, \sigma)$

$$\widehat{C}(\overrightarrow{t})[\sigma, s] \in [\![C]\!](\delta_B, \sigma). \tag{7.6}$$

Lemma 7.3.13. *The interpretation of types* $[\![-]\!]$ *given in Def. 7.3.10 is well-defined and* $[\![A]\!](\delta, \rho) \in \mathbf{SAT}$ *for all* A, δ, ρ.

Lemma 7.3.14 (Soundness)**.** *If* $\Gamma \vdash t : A$*, then for all* $\rho \in [\![\Gamma]\!]$ *we have* $t[\rho] \in [\![A]\!](\rho)$.

From the soundness, we immediately derive strong normalisation.

Theorem 7.3.15. *All well-typed terms are strongly normalising, that is, if* $\Gamma_1 \vdash t : \Gamma_2 \twoheadrightarrow A$ *then* $t \in \mathbf{SN}$.

Proof. We first note that terms only reduce if $\Gamma_2 = \emptyset$. In that case we can apply we can apply Lem. 7.3.14 with ρ being the identity, so that $t \in [\![A]\!](\rho)$. Thus, by Lem. 7.3.13 and the definition of saturated sets, we can conclude that $t \in \mathbf{SN}$. Since t does not reduce if Γ_2 is non-trivial, we also have in that case that $t \in \mathbf{SN}$. Hence every well-typed term is strongly normalising. □

7.3.4. Soundness proof for saturated sets model

In this section, we present the technical details of the proof of the soundness Lemma 7.3.14 and the results leading up to it. These details are presented here mainly for completeness and without much further comment, as it is not expected that the reader will dive into the details at first read.

This first lemma proves that the composition of context morphisms is the same as the composition of their interpretation.

Lemma 7.3.16. *For all* $\sigma : \Gamma_1 \triangleright \Gamma_2$ *and* $\tau : \Gamma_2 \triangleright \Gamma_3$ *we have* $[\![\tau \bullet \sigma]\!] = [\![\tau]\!] \circ [\![\sigma]\!]$.

Proof. For all $\rho : \lceil \Gamma_1 \rceil \to \Lambda$ and $x \in \lceil \Gamma_3 \rceil$ we have

$$\begin{aligned}[\![\tau \bullet \sigma]\!](\rho)(x) &= (\tau \bullet \sigma)(x)[\rho] = \tau(x)[\sigma][\rho] \\ &= [\![\tau]\!]([\![\sigma]\!](\rho))(x) = ([\![\tau]\!] \circ [\![\sigma]\!])(\rho)(x)\end{aligned}$$

as required. □

Next, we show that instantiation of types is given equivalently by updating valuations.

Lemma 7.3.17. *If* $\Gamma_1 \vdash A : \Gamma_2 \twoheadrightarrow *$, $\sigma : \Gamma_1 \triangleright \Gamma_2$ *and* $\rho \in [\![\Gamma_1]\!]$*, then* $[\![A \mathbin{@} \sigma]\!](\rho) = [\![A]\!]([\rho, [\![\sigma]\!](\rho)])$*, where* $[\rho, [\![\sigma]\!](\rho)] \in [\![\Gamma_1, \Gamma_2]\!]$ *is given by*

$$[\rho, [\![\sigma]\!](\rho)]\,(x) = \begin{cases} \rho(x), & x \in \lceil \Gamma_1 \rceil \\ [\![\sigma]\!](\rho)(x), & x \in \lceil \Gamma_2 \rceil \end{cases}.$$

Proof. Simply by repeatedly applying the case of the semantics of type instantiations. □

The following four lemmas 7.3.18-7.3.21 are easily proved by induction on the derivation of well-formedness of the corresponding type A.

Lemma 7.3.18. *If* $\Gamma_1, x : B, \Gamma_2 \vdash A : *$ *and* $\Gamma_1 \vdash t : B$*, then for all* $\rho \in [\![\Gamma_1, \Gamma_2[t/x]]\!]$ *we have* $[\![A[t/x]]\!] = [\![A]\!](\rho[x \mapsto t])$.

Lemma 7.3.18 shows that subsitution is, just as instantiation, is equally given by updating valuations. This is generalised in the following lemma to context morphisms.

Lemma 7.3.19. *If* $\Theta \mid \Gamma_2 \vdash A : *$ *and* $\sigma : \Gamma_1 \triangleright \Gamma_2$*, then for all* $\delta \in [\![\Theta]\!]$ *and* $\rho \in [\![\Gamma_1]\!]$ *we have* $[\![A[\sigma]]\!](\delta, \rho) = [\![A]\!](\delta, [\![\sigma]\!](\rho))$.

Analogously, also substitution of types for type variables is equally given by updating the corresponding type variable valuations.

Lemma 7.3.20. *If* $\Theta_1, X : \Gamma \to *, \Theta_2 \mid \Gamma_1 \vdash A : *$ *and* $\vdash B : \Gamma \to *$*, then we have for the type variable substitution that* $[\![A[B/X]]\!](\delta, \rho) = [\![A]\!](\delta[X \mapsto [\![B]\!]], \rho)$.

The last of these four straightforward lemmas gives us that the interpretation of types is monotone in the interpretation of type variables.

Lemma 7.3.21. *If* $\Theta \mid \Gamma \vdash A : *$ *and* $\delta, \delta' \in [\![\Theta]\!]$ *with* $\delta \sqsubseteq \delta'$ *(point-wise order), then for all* $\rho \in [\![\Theta]\!]$

$$[\![A]\!](\delta, \rho) \subseteq [\![A]\!](\delta', \rho).$$

Given these basic results, we can now show that the interpretation of inductive types contain all constructor terms.

Lemma 7.3.22. *Let* $\mu = \mu(X : \Gamma \to *; \vec{\sigma}; \vec{A})$ *where we have* $\Theta \mid \emptyset \vdash \mu : \Gamma \to *$*. If* $\delta \in [\![\Theta]\!]$, $\rho \in [\![\Gamma_k]\!]$ *and* $P \in [\![A_k]\!](\delta[X \mapsto [\![\mu]\!](\delta)], \rho)$*, then*

$$\alpha_k \mathbin{@} \rho \mathbin{@} P \in [\![\mu]\!](\delta, [\![\sigma_k]\!](\rho)).$$

Proof. Let δ, ρ and P be given as in the lemma, and put $M = \alpha_k \mathbin{@} \rho \mathbin{@} P$. We need to show for any choice of $U \in I_\Gamma$ and

$$N_k \in \{[\![\Gamma_k]\!], [\![A_k^\triangle]\!](\delta[X \mapsto U])\}_y \Vdash [\![\sigma_k \bullet \pi]\!]^*(U)$$

that

$$K = \mathrm{iter}\,\overrightarrow{(\Gamma_k, y.\, N_k)} \mathbin{@} (\sigma_k \bullet \rho) \mathbin{@} M$$

is in $U([\![\sigma_k]\!](\rho))$. Now we define $r = \mathrm{iter}\,\overrightarrow{(\Gamma_k, y.\, N_k)}$ and

$$K' = N_k\left[\widehat{A_k}(r \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x') \Big/ y\right][\rho, P]$$

so that $K \succ K'$. Let us furthermore put

$$V = U([\![\sigma_k]\!](\rho)).$$

By $V \in \mathbf{SAT}$, it suffices to prove that $K' \in V$. Note that we can rearrange the substitution in K' to get $K' = N_k[\rho, P']$ with $P' = \widehat{A_k}(r \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x')[\rho, P]$.

We get $K' \in V$ from $N_k \in \{[\![\Gamma_k]\!], [\![A_k]\!](\delta[X \mapsto U])\}_y \Vdash [\![\sigma_k \bullet \pi]\!]^*(U)$, provided that $\rho \in [\![\Gamma_k]\!]$ and $P' \in [\![A_k]\!](\delta[X \mapsto U], \rho)$. The former is given from the assumption of the lemma. The latter we get from Lem. 7.3.12, since we have assumed soundness for the components of μ and $P \in [\![\widehat{A_k}]\!](\rho)$. Thus we have $K' = N_k[\rho, P'] \in V$.

So by saturation we have $K \in V = U([\![\sigma_k]\!](\rho))$ for any choice of U and N_k, thus if follows that $M \in [\![\mu]\!](\delta, [\![\sigma_k]\!](\rho))$. □

Similarly, we wish to prove that the interpretation of a coinductive type contains all its coiteration scheme instances. Towards this, we first show that coinductive types give rise to greatest fixed points, as one would expect.

Lemma 7.3.23. *Let $\nu = \nu(X : \Gamma \to * ; \vec{\sigma}; \vec{A})$ where we have $\Theta \mid \emptyset \vdash \nu : \Gamma \to * $. If $U \in I_\Gamma$ and $\delta \in [\![\Theta]\!]$, such that for all $M \in U(\rho)$, all k, all $\rho \in [\![\Gamma]\!]$ and all $\gamma \in [\![\sigma_k]\!]^{-1}(\rho)$, $\xi_k \mathbin{@} \gamma \mathbin{@} M \in [\![A_k]\!](\delta[X \mapsto U], \gamma)$, then*

$$\forall \rho.\, U(\rho) \subseteq [\![\nu]\!](\delta, \rho).$$

Proof. This follows immediately from the definition of $[\![\nu]\!]$, just instantiate the definition with the given U. Then all all $M \in U(\rho)$ are in $[\![\nu]\!](\delta, \rho)$. □

Given the greatest fixed point property in Lemma 7.3.23, we can prove that the interpretation of a coinductive type contains all instances of the coiteration scheme.

Lemma 7.3.24. *Let $\nu = \nu(X : \Gamma \to * ; \vec{\sigma}; \vec{A})$ where we have $\Theta \mid \emptyset \vdash \nu : \Gamma \to * $. If $\delta \in [\![\Theta]\!]$, $U \in I_\Gamma$, $\rho \in [\![\Gamma_k]\!]$ and*

$$N_k \in \{[\![\Gamma_k]\!], [\![\sigma_k]\!]^*(U)\}_y \Vdash [\![A_k]\!](\delta[X \mapsto U])$$

for $k = 1, \ldots, n$, then

$$\mathrm{coiter}\, \overrightarrow{(\Gamma_k, y.\, N_k)} \mathbin{@} \rho \mathbin{@} M \in [\![\nu]\!](\delta, \rho).$$

Proof. Similar to the proof of Lem. 7.3.22 by using that the interpretation of ν-types is a largest fixed point Lem. 7.3.23 and that the interpretation is monotone Lem. 7.3.21. □

The last lemma that we need to prove that the action of types on terms is sound for the saturated sets interpretation is that the interpretation respects reduction steps.

Lemma 7.3.25. *Suppose C is a type with $\Theta \mid \Gamma_1 \vdash C : \Gamma_2 \to *$. If $\rho, \rho' \in [\![\Gamma_1, \Gamma_2]\!]$ with $\rho \longrightarrow \rho'$, then $\forall \delta.\, [\![C]\!](\delta, \rho) = [\![C]\!](\delta, \rho')$. Furthermore, if $C \longrightarrow C'$, then $[\![C]\!] = [\![C']\!]$.*

Proof. The first part follows by an easy induction, in which the only interesting case $C = X$ is. Here we have

$$[\![C]\!](\delta, \rho) = \delta(X)(\rho) = \delta(X)(\rho') = [\![C]\!](\delta, \rho'),$$

since $\delta(X) \in I_{\Gamma_i}$ and thus respects conversions.

For the second part, let D be given by replacing all terms in parameter position in C by variables, so that $C = D[\rho]$ for some substitution ρ. But then there is a ρ' with $\rho \longrightarrow \rho'$ and $C' = D[\rho']$, and the claim follows from the first part. □

We are now in the position to give the proofs that we left out before. First, we have to show that the action of types on terms is sound for the interpretation.

Proof of Lemma 7.3.12. We proceed by induction on the derivation of $\Theta \mid \Gamma \vdash C : \Gamma' \to *$.

- $\vdash \top : *$ by **($\top$-I)**. In this case we have that $\widehat{\top}(\varepsilon) = x \in \mathcal{B}$, thus $\widehat{\top}(\varepsilon) \in [\![C]\!](\delta)$ by by saturation.
- $\Theta, X_{n+1} : \Gamma_{n+1} \to * \mid \emptyset \vdash X : \Gamma_{n+1} \to *$ by **(TyVar-I)**. Note that $\widehat{X_{n+1}}(\vec{t}, t_{n+1}) = t_{n+1}$ and $[\![X_{n+1}]\!](\delta, \sigma) = \delta(X_{n+1})(\sigma) = [\![B_{n+1}]\!](\sigma)$. thus the claim follows directly from the assumption of the lemma.

- $\Theta, X : \Gamma'' \to * \mid \Gamma \vdash C : \Gamma' \to *$ by **(TyVar-Weak)**. Immediate by induction.
- $\Theta \mid \Gamma, y : D \vdash C : \Gamma' \to *$ by **(Ty-Weak)**. Again immediate by induction.
- $\Theta \mid \Gamma \vdash C\,@\,r : \Gamma'[s/x] \to *$ with $\Gamma \vdash r : C'$ by **(Ty-Inst)**. First, we note that $\sigma = (\sigma_1, \sigma_2)$ with $\sigma_1 : \Delta \triangleright \Gamma$ and $\sigma_2 : \Delta \triangleright \Gamma'[\sigma_1, r]$. Let us put $\tau = (\sigma_1, r[\sigma_1], \sigma_2)$, so that we have

  $$\widehat{(C\,@\,r)}(\vec{t})[\sigma, s] = \widehat{C}(\vec{t})[s/x][\sigma, s] = \widehat{C}(\vec{t})[\sigma_1, r[\sigma_1], \sigma_2, s] = \widehat{C}(\vec{t})[\tau, s].$$

  By the assumption of the lemma on parameters we have $r[\sigma_1] \in [\![C']\!](\sigma_1)$, and thus $\tau \in [\![\Gamma, x : C'[\sigma_1], \Gamma'[\sigma_1, r[\sigma_1]]]\!]$, which gives $[\![C\,@\,r]\!](\delta, \sigma) = [\![C]\!](\delta, \tau)$. By induction, we have $\widehat{C}(\vec{t})[\tau, s] \in [\![C]\!](\delta, \tau)$, and it follows that

  $$\widehat{(C\,@\,r)}(\vec{t})[\sigma, s] \in [\![C\,@\,r]\!](\delta, \sigma).$$

- $\Theta \mid \Gamma_1 \vdash (x).\, B : (x : A, \Gamma_2) \to *$ by **(Param-Abstr)**. Immediate by induction.
- $\Theta \mid \emptyset \vdash \mu(Y : \Gamma \to *; \vec{\tau}; \vec{D}) : \Gamma \to *$ by **(FP-Ty)**. We abbreviate, as before, this type just by μ. Recall the definition of $\widehat{\mu}$:

  $$\begin{aligned}\widehat{\mu}(\vec{t}) &= \mathrm{iter}^{R_A}\, \overrightarrow{(\Delta_k, x.\, g_k)}\, @\, \mathrm{id}_\Gamma\, @\, x \\ &\text{with } g_k = \alpha_k\, @\, \mathrm{id}_{\Delta_k}\, @\, \left(\widehat{D_k}\,(\vec{t}, y)\right) \\ &\text{and } R_A = \mu[\overrightarrow{(\Gamma_i).\, A_i}/\vec{X}] \\ &\text{for } \Theta, Y : \Gamma \to * \mid \Delta_k \vdash D_k : *\end{aligned}$$

  Now, put $\delta' = \delta[Y \mapsto [\![R_B]\!]]$, then we have by induction that $\widehat{D_k}(\vec{t}, y)[\mathrm{id}_{\Delta_k}, y] \in [\![D_k]\!](\delta', \mathrm{id}_{\Delta_k})$. Since $\mathrm{id}_{\Gamma_k} \in \mathbf{SN}$ we have by Lem. 7.3.22 for all $\rho \in [\![\Delta_k, \widehat{D_k}(\vec{A}, R_B)]\!]$ that $g_k[\rho] \in [\![\mu]\!](\delta, [\![\tau_k]\!](\rho))$. By assumption, we have $s \in [\![R_A]\!](\sigma)$, hence by choosing $U = [\![R_B]\!]$ in the definition of $[\![R_A]\!]$ we find $\widehat{\mu}(\vec{t})[\sigma, s] \in [\![R_B]\!](\sigma) = [\![\mu]\!](\delta, \sigma)$.
- $\Theta \mid \emptyset \vdash \nu(Y : \Gamma \to *; \vec{\tau}; \vec{D}) : \Gamma \to *$ by **(FP-Ty)**. Analogous to the case for inductive types, only that we use Lemma 7.3.24. □

Finally, we can use all of the above results to show that the interpretation of types as saturated sets is sound with respect to the typing rules of $\lambda\mathrm{P}\mu$.

Proof of Lemma 7.3.14. We proceed by induction on the type derivation for t. Since t does not have any parameters, we only have to deal with fully applied terms and will thus leave out the case for instantiation in the induction. Instead, we will have cases for fully instantiated α, ξ, etc. So let $\rho \in [\![\Gamma]\!]$ and proceed by the cases for t.

- $\langle\rangle \in [\![\top]\!](\rho)$ by definition.
- For $\Gamma, x : A \vdash x : A$ we have $x[\rho] = \rho(x)$. From the definition of $[\![\Gamma, x : A]\!]$, we obtain $\rho(x) \in [\![\Gamma \vdash A : *]\!](\rho|_{\lceil\Gamma\rceil}) = [\![\Gamma, x : A \vdash A : *]\!](\rho)$. Thus $x[\rho] \in [\![A]\!](\rho)$ as required.
- Weakening is dealt with immediately by induction.

- If t is of type B by **(Conv)**, then by induction $t \in [\![A]\!](\rho)$. Since by Lem. 7.3.25 $[\![B]\!](\rho) = [\![A]\!](\rho)$, we have $t \in [\![B]\!](\rho)$.

- Suppose we are given $\mu = \mu(X : \Gamma \to * ; \vec{\sigma} ; \vec{A})$ and $\Delta \vdash \alpha_k \mathbin{@} \tau \mathbin{@} t : \mu[\sigma_k \bullet \tau]$ with $\tau : \Delta \triangleright \Gamma$ and $\Delta \vdash t : A_k[\mu/X][\tau]$.

  Then, by induction, we have $t \in [\![A_k]\!](X \mapsto [\![\mu]\!], \tau)$ and soundness for the components of μ, thus by Lem. 7.3.22

$$\alpha_k \mathbin{@} \tau \mathbin{@} t \in [\![\mu]\!]([\![\sigma_k \bullet \tau]\!](\rho)) = [\![\mu[\sigma_k \bullet \tau]]\!](\rho).$$

- Suppose we have $\mu = \mu(X : \Gamma \to * ; \vec{\sigma} ; \vec{A})$ and $\Delta \vdash \mathrm{iter}^{\mu}\ \overrightarrow{(\Gamma_k, x.\, g_k)} \mathbin{@} \tau \mathbin{@} t : C \mathbin{@} \tau$. Then by induction we get from $\Delta \vdash t : \mu[\tau]$ that $t \in [\![\mu[\tau]]\!](\rho)$, hence if we chose $U = [\![C \mathbin{@} \tau]\!]$ and $N_k = g_k$ the definition of $[\![\mu]\!]$ yields $\mathrm{iter}^{\mu}\ \overrightarrow{(\Gamma_k, x.\, g_k)} \mathbin{@} \tau \mathbin{@} t \in [\![C \mathbin{@} \tau]\!](\rho)$.

- Suppose we have $\nu = \nu(X : \Gamma \to * ; \vec{\sigma} ; \vec{A})$, and $\Delta \vdash \xi_k \mathbin{@} \tau \mathbin{@} t : A_k[\nu/X][\tau]$ with $\tau : \Delta \triangleright \Gamma_k$ and with $\Delta \vdash t : \nu[\sigma_k \bullet \tau]$. By induction, $t \in [\![\nu[\sigma_k \bullet \tau]]\!](\rho)$ thus there is a U such that $\xi_k \mathbin{@} \tau \mathbin{@} t \in [\![A_k]\!](X \mapsto U, [\![\tau]\!](\rho))$. By Lem. 7.3.23 and Lem. 7.3.21 we thus obtain that $\xi_k \mathbin{@} \tau \mathbin{@} t \in [\![A_k]\!](X \mapsto [\![\nu]\!], [\![\tau]\!](\rho))$. Since $[\![A_k]\!](X \mapsto [\![\nu]\!], [\![\tau]\!](\rho)) = [\![A_k[\nu/X][\tau]]\!](\rho)$, the claim follows.

- For coiter-terms we just apply Lem. 7.3.24, similar to the α_k-case.

This concludes the induction, thus the interpretation of types is sound with respect to the typing judgement for terms. □

7.4. Dependent Iteration

In Section 7.1 we have introduced the calculus $\lambda \mathrm{P}\mu$ of dependent inductive and coinductive types. As we saw in Section 7.2, this calculus subsumes all of first-order intuitionistic logic or Martin-Löf type theory. However, we were not able to define the second projection for dependent coproducts or existential quantifiers. Similarly, we are also not able to carry out proofs by induction in $\lambda \mathrm{P}\mu$. Both of these problems are rectified once we extend the calculus with a dependent iteration scheme, cf. Section 6.4.2.

The goal of the present section is to propose an extension of $\lambda \mathrm{P}\mu$ by a dependent iteration principle for inductive types. Such a dependent iteration principle will be necessary to carry out the proofs in the application section below. For now it is left open whether the calculus that results from extending $\lambda \mathrm{P}\mu$ with dependent iteration is still strongly normalising. It seems, however, that the proof we gave in Section 7.3.3 can be adapted accordingly.

One may ask what the dual extension of $\lambda \mathrm{P}\mu$ for coinductive types is that corresponds to dependent iteration. It is clear that there is no such thing as dependent coiteration that generalises coiteration in the way dependent iteration extends iteration. As we discussed in Section 6.4.3 the dual concept of induction is the bisimulation proof principle, which we referred to as coinduction. We will not extend $\lambda \mathrm{P}\mu$ here with coinduction, rather we will discuss possible future work in this direction in Section 7.6.

$$
\begin{aligned}
(\overline{\vdash \top : *})(\theta) &= \top \\
(\overline{\Theta, X : \Gamma \multimap * \mid \emptyset \vdash X : \Gamma \multimap *})(\theta, B) &= Y_X \mathbin{@} \mathrm{id}_{\Gamma, y:B \mathbin{@} \mathrm{id}_\Gamma} \\
(\overline{\Theta, X \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *})(\theta, B) &= \overline{A}(\theta) \\
(\overline{\Theta \mid \Gamma_1, x : B \vdash A : \Gamma_2 \multimap *})(\theta) &= \overline{A}(\theta) \\
(\overline{\Theta \mid \Gamma_1 \vdash A \mathbin{@} t : \Gamma_2[t/x] \multimap *})(\theta) &= \overline{A}[t/x] \\
(\overline{\Theta \mid \Gamma_1 \vdash (x).A : (x : B, \Gamma_2) \multimap *})(\theta) &= \overline{A}(\theta) \\
(\overline{\Theta \mid \emptyset \vdash \rho(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap *})(\theta) &= \rho(X : \Gamma, z : R \mathbin{@} \mathrm{id} \multimap *; \vec{\sigma}'; \vec{A}') \mathbin{@} \mathrm{id}_{\Gamma, y} \\
\text{where} \quad R &= \rho(Y : \Gamma \multimap *; \vec{\sigma}; \vec{A}) \\
\sigma'_k &= (\sigma, \alpha_k \mathbin{@} \mathrm{id} \mathbin{@} y_k) \\
A'_k &= \overline{A_k}(\theta, R)[\theta]
\end{aligned}
$$

Figure 7.7.: Lifting of Dependent Types to Predicates

To be able to formulate the dependent iteration scheme, we first need to lift types to predicates. As in Section 6.4.1, a predicate is here meant to be a type that depends on terms of the type that it is a predicate for. More precisely, given a type B with $\vdash B : \Delta \multimap *$, a B-predicate is a type P with $\Gamma, x : B \mathbin{@} \mathrm{id}_\Delta \vdash P : *$. A lifting of an open type A with $X : \Delta \multimap * \mid \Gamma \vdash A : *$ to B-predicates is then a type $\overline{A}(B)$ with a free type variable that ranges over B-predicates, cf. Definition 2.5.7. In other words, the lifting should satisfy the following rule.

$$
\frac{X : \Delta \multimap * \mid \Gamma \vdash A : * \qquad \vdash B : \Delta \multimap *}{Y : (\Delta, x : B \mathbin{@} \mathrm{id}_\Delta) \multimap * \mid \Gamma, y : A[B/X] \vdash \overline{A}(B) : *}
$$

We give such a lifting in the following definition. That the above rule holds is proved below.

Definition 7.4.1. Let A be a type with $\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap *$ and θ be a sequence of types, such that $\theta = B_1 : \Delta_1 \multimap *, \ldots, B_n : \Delta_n \multimap *$ if $\Theta = X_1 : \Delta_1 \multimap *, \ldots, X_n : \Delta_n \multimap *$. We define then the lifting $\overline{A}(\theta)$ of A to predicates over θ. If Θ is empty, we just put $\overline{A}() = \top$. Otherwise, $\overline{A}$ is by induction on the well-formedness derivation of A in Figure 7.6. ◀

The following lemma shows that the predicate lifting results in well-formed types. This will enable us to prove subject reduction.

Lemma 7.4.2. *The following rule holds for the predicate lifting in Definition 7.4.1.*

$$
\frac{X : \Delta \multimap * \mid \Gamma \vdash A : * \qquad \vdash B : \Delta \multimap *}{Y : (\Delta, x : B \mathbin{@} \mathrm{id}_\Delta) \multimap * \mid \Gamma, y : A[B/X] \vdash \overline{A} : *}
$$

Proof. Let us write, analogously to the notation for context morphisms, $\theta : \Theta_1 \triangleright \Theta_2$, if θ is a sequence of types in context Θ_1 that matches Θ_2. More precisely, θ is a sequence constructed by the following two rules.

$$
\frac{}{() : \Theta_1 \triangleright \emptyset} \qquad \frac{\theta : \Theta_1 \triangleright \Theta_2 \qquad \vdash B : \Gamma \multimap *}{(\theta, B) : \Theta_1 \triangleright (\Theta_2, X : \Gamma \multimap *)}
$$

Given such a sequence $\theta : \emptyset \triangleright \Theta$ of types, we define a lifted context $\overline{\Theta}(\theta)$ as follows by induction.

$$\overline{\emptyset}(\theta) = \emptyset$$
$$(\overline{\Theta, X : \Gamma \multimap *})(\theta, B : \Gamma \multimap *) = \overline{\Theta}(\theta), Y : (\Gamma, B \mathbin{@} \mathrm{id}_\Gamma) \multimap *$$

This allows us to give a typing rule for the general lifting that we can use in a proof by induction.

$$\frac{\Theta \mid \Gamma_1 \vdash A : \Gamma_2 \multimap * \qquad \theta : \emptyset \triangleright \Theta}{\overline{\Theta}(\theta) \mid \Gamma_1, \Gamma_2, y : A[\theta] \mathbin{@} \mathrm{id}_{\Gamma_2} \vdash \overline{A} : *} \tag{7.7}$$

If we can show that this typing rule holds, then we are done because the typing rule of the lemma holds then by picking $\Theta = X : \Delta \multimap *$, $\Gamma_1 = \Gamma$ and $\Gamma_2 = \emptyset$. The rule (7.7) can now be shown by a straightforward induction on the typing derivation of A. Details are provided in the Agda formalisation [Bas18b]. □

Definition 7.4.3. We define an extension of $\lambda\mathrm{P}\mu$ with a dependent iteration principle as follows.[66] Suppose that $\mu(X : \Gamma \multimap *; \vec{f}; \vec{A})$ is an inductive type, which we refer to as μ here. *Dependent iteration* is a term $\mathrm{diter}\,\overrightarrow{(\Gamma_k, y_k, z_k . g_k)}$ that is subject to the following typing rule.

$$\frac{\Delta \vdash C : (\Gamma, x : \mu \mathbin{@} \mathrm{id}_\Gamma) \multimap * \qquad \Delta, \Gamma_k, y_k : A_k[\mu/X], z_k : \overline{A_k}(\mu)[C/X] \vdash g_k : (C \mathbin{@} \sigma_k \mathbin{@} (\alpha_k \mathbin{@} \mathrm{id}_{\Gamma_k} \mathbin{@} y_k)) \quad \forall 1 \le k \le \left|\vec{A}\right|}{\Delta \vdash \mathrm{diter}\,\overrightarrow{(\Gamma_k, y_k, z_k . g_k)} : (\Gamma, y : \mu \mathbin{@} \mathrm{id}_\Gamma) \multimap C \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} y} \textbf{(D-Rec)}$$

Additionally, we introduce an induction principle for the type $\top$. This induction principle is given by terms $\mathrm{diter}_\top\, g$, subject to the following rule.

$$\frac{\Gamma \vdash C : (x : \top) \multimap * \qquad \Gamma \vdash g : C \mathbin{@} \langle\rangle}{\Gamma \vdash \mathrm{diter}_\top\, g : (y : \top) \multimap C \mathbin{@} y} \textbf{(D-Rec-}\top\textbf{)}$$

We refer to the extension of $\lambda\mathrm{P}\mu$ by the above dependent iteration and induction principles by $\lambda\mathrm{P}\mu^+$. The contraction relation of $\lambda\mathrm{P}\mu^+$ on terms is given by that of $\lambda\mathrm{P}\mu$ with the additional following two clauses: one for general dependent iteration

$$\mathrm{diter}\,\overrightarrow{(\Gamma_k, y_k, z_k . g_k)} \mathbin{@} (\sigma_k \bullet \tau) \mathbin{@} (\alpha_k \mathbin{@} \tau \mathbin{@} u)$$
$$\succ g_k \left[\widehat{\overline{A_k}}(\mathrm{diter}\,\overrightarrow{(\Gamma_k, y_k . g_k)} \mathbin{@} \mathrm{id}_\Gamma \mathbin{@} x) \Big/ z_k\right][\tau, u]$$

and one for the induction principle for $\top$

$$\mathrm{diter}_\top\, g \,\langle\rangle \succ g.$$

The reduction relation of $\lambda\mathrm{P}\mu^+$ is then again the compatible closure of its contraction relation. ◀

Let us illustrate the usefulness of the extended calculus $\lambda\mathrm{P}\mu^+$ by means of two examples.

Example 7.4.4. Let us first derive that $\langle\rangle$ is the unique inhabitant of $\top$, which allows us to generally ignore variables of type $\top$. We write

$$s \doteq_A t \coloneqq \mathrm{Eq}_A(s,t)$$

and if A is understood from the context, we drop the subscript. The proposition we thus want to prove is

$$y : \top \vdash y \doteq \langle\rangle : *,$$

in other words, we need to establish a term $\mathrm{canon}_\top$ with

$$y : \top \vdash \mathrm{canon}_\top : y \doteq \langle\rangle.$$

This term is given by $\mathrm{canon}_\top \coloneqq \mathrm{diter}_\top\ \mathrm{refl} @ y$, as we show now. First, we put

$$C \coloneqq (y).\, y \doteq \langle\rangle$$

and thus have $C @ y \equiv y \doteq \langle\rangle$. Since $\mathrm{refl} : \langle\rangle \doteq \langle\rangle$, we then obtain

$$\dfrac{\dfrac{\dfrac{\vdash C : (y:\top) \multimap * \qquad \vdash \mathrm{refl} : C @ \langle\rangle}{\vdash \mathrm{diter}_\top\ \mathrm{refl} : (y:\top) \multimap C @ y}\ \textbf{(D-Rec-}\top\textbf{)}}{y : \top \vdash \mathrm{diter}_\top\ \mathrm{refl} @ y : C @ y}\ \textbf{(Inst)}}{y : \top \vdash \mathrm{canon}_\top : y \doteq \langle\rangle}\ \textbf{(Conv)}$$

This proves that $\langle\rangle$ is the unique inhabitant of $\top$, up to propositional equality. Our main use of this fact is to eliminate variables of type $\top$, that is, we consider terms t with $\Gamma, x : \top \vdash t : A$ to be essentially the same as terms s with $\Gamma \vdash s : A$. ◀

The following example shows that we can now define a second projection for coproduct types, we thereby show that coproducts are strong in $\lambda \mathrm{P}\mu^+$, cf. Definition 6.3.2 and Section 6.4.2.

Example 7.4.5. In Example 7.2.6 we have seen that the existential quantifier can be represented by

$$\exists x : A.\, B = \mu(X : * ; \varepsilon_1 ; B),$$

where $\varepsilon_1 : (x : A) \triangleright \emptyset$ is the empty context morphism. Moreover, we have also seen there that the first projection $\pi_1 : (\exists x : A.\, B) \to A$ can be defined by simple iteration. The goal of this example is to show that dependent iteration allows us to define also the second projection

$$\pi_2 : \Pi(z : \exists x : A.\, B).\, B[\pi_1\ z/x].$$

In other words, under the assumption of dependent iteration, the existential quantifier is becomes a strong sum[67] or coproduct type, cf. Section 6.4.2.

First, we note that, since B has no free type variable, the predicate lifting $\overline{B}$ is defined to be the type $\top$. Thus, the dependent iteration scheme for the existential quantifier amounts to the following rule.

$$\dfrac{\Gamma \vdash C : (\exists x : A.\, B) \multimap * \qquad \Gamma, x : A, y : B \vdash g : C @ \langle x, y\rangle}{\Gamma \vdash \mathrm{diter}\,(x, y.\, g) : (z : (\exists x : A.\, B)) \multimap C @ z}\ \textbf{(D-Rec-}\exists\textbf{)}$$

To define the second projection, we use the type C given by

$$C \coloneqq (z).\, B[\pi_1\ z/x].$$

Since

$$C\,@\langle x, y\rangle \equiv B[\pi_1\ \langle x, y\rangle/x] \equiv B[x/x] = B,$$

we can derive the following from **(D-Rec-∃)**.

$$\frac{\Gamma, x : A, y : B \vdash y : C\,@\langle x, y\rangle}{\Gamma \vdash \mathrm{diter}\,(x, y.\, y) : (z : (\exists x : A.\, B)) \multimap C\,@\,z}\ \textbf{(D-Rec-}\exists\textbf{)}$$

This allows us to define

$$\pi_2 \coloneqq \lambda z.\, \mathrm{diter}\,(x, y.\, y)\,@\,z,$$

so that we have $\pi_2 : \Pi(z : \exists x : A.\, B).\, B[\pi_1\ z/x]$, as required. ◂

This concludes the discussion of an extension of $\lambda\mathrm{P}\mu$ by dependent iteration principles for inductive types and $\top$. These principles will prove necessary in the next section, where we discuss an application of inductive and coinductive types in logical reasoning.

7.5. An Application: Transitivity of the Substream Relation

In this section, we put our dependent type theory to use. The aim of this section is to give a direct definition of the substream relation inside the type theory, see also Example 6.2.14, and then prove that this definition is equivalent to the definition based on stream filtering. This allows us to show that the substream relation is transitive.

As we have seen in Section 3.2, it can be very difficult to use iteration and coiteration schemes, especially whenever we mix inductive and coinductive types. This situation is not getting any better in the presence of dependent types and if we carry out mixed inductive-coinductive proofs. For this reason, it would be preferable to use also equational specification upon defining dependently typed terms. However, the problem with using recursive equations is that one needs to be careful not to introduce non-normalising terms. We discussed this at length in Section 4.1. Having non-normalising terms in a dependent type theory has two detrimental consequences. First, it usually makes the theory inconsistent under the propositions-as-types interpretation, if we are able to construct a (non-normalising) term of type $\bot$. Second, the conversion rule **(Conv)** of the type system requires us to decide whether two terms are convertible. This becomes troublesome in the presence of non-normalising terms, since conversion is usually decided by reducing terms to normal forms and comparing these normal forms. If no such normal forms exist or cannot be deterministically reached, then this naive approach fails. Both problems can be overcome but we will not discuss this here further. Rather, we will give the definitions and (proof) terms in Agda, which has mechanisms to ensure that equational specifications are well-defined and thus lead to normalising terms.

It is understood that all the proofs that are given in this section can be translated into our calculus $\lambda\mathrm{P}\mu^{+}$. In general, it is unknown at this stage whether all Agda terms can be encoded as (dependent) iteration and coiteration. We leave this for future work, though there is some good evidence that such an encoding should be possible as the work of Goguen et al. [GMM06] shows.

7.5.1. Some Preliminaries in Agda

This first section is devoted to the introduction of the propositions-as-types interpretation in Agda syntax. We will interleave the given Agda code with the necessary explanation. All the indented code is formally verified in Agda and will be explained as we go.

We start off by introducing the propositions-as-types interpretation in its own module called PropsAsTypes. Agda allows the separation of code into modules that can be imported in other files, as we will see later.

```
module PropsAsTypes where
```

Inside the module, we now introduce our first type, which implements the dependent sum that we encountered earlier. The following code declares a new inductive data type called Σ with two parameters I and X. So far, we denoted the universe of valid types by $*$, whereas Agda uses the notation Set instead. Thus, I is a type parameter. The parameter X, on the other hand, is a family of types. In our syntax, we would denote this by $X : (i : I) \rightarrowtriangle *$ and instantiation of X with a parameter t would be written as $X \mathbin{@} t$. Agda uses for types with parameters the same notation as for the function space,[68] so that the instantiation of X is written as $X\ t$.

Another mechanism of Agda that appears in the definition of Σ is that I is an implicit parameter, signified by the curly braces, and that X is an explicit parameter. The idea of the implicit parameter mechanism is that Agda ought to be able to infer the argument I from the context. For example, if we have a type U : Nat $\rightarrow$ Set, then Agda can infer from the instantiation $\Sigma\, U$ that I needs to be instantiated with Nat. Should this inference not be possible, then one can always resort to writing $\Sigma\, \{\text{Nat}\}\, U$ to explicitly give the parameter I. We will see this below in the definition of the binary product.

The type Σ itself has one constructor "," with two arguments, where the argument positions are marked by the underscores. This allows us to write (i, x) if $i : I$ and $x : X\ i$, thereby resembling the pairing notation that we used before.

```
data Σ {I : Set} (X : I → Set) : Set where
  _,_ : (i : I) → X i → Σ X
```

In Example 7.2.6, we denoted the type Σ by $\exists$ to emphasise its relation to existential quantification. This was done in the light of the fact that the iteration scheme for that type corresponded exactly to the usual elimination rule the existential quantifier. Later we introduced dependent iteration, which we could use in Example 7.4.5 to define both projections from the existential type. In this section, we will distinguish between Σ as a data type, which has both projections, and the existential quantifier as a proposition. The latter will only come with the non-dependent elimination principle.

We now give the dependent iteration scheme for the dependent sum type Σ. Let A be a type family that depends on $\Sigma\, X$, that is, $A : \Sigma\, X \rightarrow$ Set. Recall from Example 7.4.5 that, if we want to eliminate $\Sigma\, X$ into A, then we need to provide a term of type $A\,(i, x)$ with parameters $i : I$ and $x : X\ i$. Such a term can equivalently be given as a dependent function f of type $\Pi i : I.\, \Pi x : X\ i.\, A\,(i, x)$. In Agda, this function type is written as $(i : I) \rightarrow (x : X\ i) \rightarrow A\ (i\ ,\ x)$, in the style of de Bruijn's telescopes [dBru68]. Conveniently, we can define the dependent iteration scheme in Agda then by using pattern matching as follows.

```
drec−Σ : {I : Set} {X : I → Set} {A : Σ X → Set} →
         ((i : I) → (x : X i) → A (i , x)) →
         (u : Σ X) → A u
drec−Σ f (i , x) = f i x
```

It is also straightforward to define, as we did in Example 7.4.5, the two projections for the dependent sum by appealing to dependent iteration.

```
π₁ : {I : Set} {X : I → Set} → Σ X → I
π₁ = drec−Σ (λ i x → i)

π₂ : {I : Set} {X : I → Set} → (u : Σ X) → X (π₁ u)
π₂ = drec−Σ (λ i x → x)
```

From the dependent sum type we can also derive the binary product type. Note that we introduced this type before as a coinductive type. It saves us here, however, some work to define the binary product in terms of the dependent sum.

```
_×_ : Set → Set → Set
A × B = Σ {A} λ _ → B
```

In the propositions-as-types interpretation, one uses a type system to represent logical propositions. To make clear whenever we write a proof of a proposition that has been internalised into Agda, we will use Prop instead of Set. Thus, if φ is a proposition that we encoded in Agda, we will denote this as φ : Prop. This notation is introduced by the following definition.

```
Prop = Set
```

An important logical connective that we will need later is conjunction. Even though conjunction is defined in terms the binary product, we use separate notations to emphasise its logical nature.

```
_∧_ : Prop → Prop → Prop
A ∧ B = A × B

∧−elim₁ : {A₁ A₂ : Prop} → A₁ ∧ A₂ → A₁
∧−elim₁ = π₁

∧−elim₂ : {A₁ A₂ : Prop} → A₁ ∧ A₂ → A₂
∧−elim₂ = π₂

∧−intro : {A₁ A₂ : Prop} → A₁ → A₂ → A₁ ∧ A₂
∧−intro a₁ a₂ = (a₁ , a₂)
```

As already mentioned, we also introduce a separate notation for the existential quantifier.

```
∃ : {X : Set} → (A : X → Prop) → Prop
∃ = Σ
```

Agda allows us to provide a syntax for the existential quantifier that matches the usual notation with explicit quantification domain. It is not so relevant to precisely understand the following definition, the reader should just remember that we can write ∃[x ∈ X] A instead of $\exists x : X. A$ for propositions A with a free occurrence of x.

```
∃-syntax : ∀ X → (X → Prop) → Prop
∃-syntax X A = ∃ A

syntax ∃-syntax X (λ x → A) = ∃[ x ∈ X ] A
```

From the pairing constructor and the iteration scheme for the dependent sum type we obtain the usual introduction and elimination rules for the existential quantifier that we already discussed in Example 7.2.6. For convenience, we also provide a scheme for the simultaneous elimination of two existential quantifiers. None of these definitions should come at a surprise to the reader at this stage.

```
∃-intro : {X : Set} {A : X → Prop} → (x : X) → A x → ∃[ x ∈ X ] (A x)
∃-intro x a = (x , a)

∃-elim : {X : Set} {A : X → Prop} {B : Prop} →
         ((x : X) → A x → B) → ∃[ x ∈ X ] (A x) → B
∃-elim = drec-Σ

∃₂-elim : {X Y : Set} {A : X → Prop} {B : Y → Prop} {C : Prop} →
          ((x : X) (y : Y) → A x → B y → C) →
          ∃[ x ∈ X ] (A x) → ∃[ x ∈ Y ] (B x) → C
∃₂-elim f p q = ∃-elim (λ x p' → ∃-elim (λ y q' → f x y p' q') q) p
```

Finally, we define bi-implication between propositions, which is is of course just given as the conjunction of implications in both directions.

```
_⇔_ : Prop → Prop → Prop
A ⇔ B = (A → B) ∧ (B → A)

equivalence : {A B : Prop} → (A → B) → (B → A) → A ⇔ B
equivalence = ∧-intro
```

Having set up the necessary definitions of the propositions-as-types interpretation, we can now move to more interesting topics.

7.5.2. Streams and Bisimilarity

To define the substream relation, we first need to introduce streams. We have done this already in the simple type system in Example 3.1.2 but to have a complete example, we recast the definition of streams here again in Agda. This allows us to also explain the syntax of Agda's coinductive types. Moreover, we will introduce bisimilarity on streams as a coinductively defined relation and prove that it is an equivalence relation.

```
module Stream where
```

Before we come to the actual definitions, we import a few modules that we use throughout this section. The module Relation.Binary contains definitions surrounding binary relations. In particular, we can write R : Rel A to express that R is a type with two parameters of type A, that is, if R is of type A → A → Set. We will later show that bisimilarity of streams implies point-wise equality. Since point-wise equality is defined in terms of indexing into streams by natural numbers, we also need to import the module Data.Nat. Finally, the module PropsAsTypes was given in Section 7.5.1 above.

```
import Relation.Binary as BinRel

open import Data.Nat using (ℕ; zero; suc)
open import PropsAsTypes
```

Let us now come to the definition of streams and their corresponding coiteration principle. The following coinductive record type corresponds to what we have called so far codata types. Thus, instead of

```
codata Stream (A : Set) : Set where
  hd : A
  tl : Stream A
```

the type of streams is given in Agda by the following record type.

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : Stream A
```

A record type in Agda is a type that is defined by its destructors, which are called fields in Agda. If a record is recursive, that is, it uses itself in its own definition, then we need to specify whether it this recursion is to be interpreted inductively or coinductively. Since we never encounter inductive record types here, we stick to the codata terminology. The reader should thus just replace the verbose coinductive record syntax in her head by the simpler codata syntax.

The coiteration principle for streams is given in Agda by the following equational specification.

```
coiter : {X A : Set} → (X → A × X) → (X → Stream A)
coiter c x .hd = π₁ (c x)
coiter c x .tl = coiter c (π₂ (c x))
```

Note that we define the coiteration principle in terms of copattern equations, which are the same as in the calculus $\lambda\mu\nu{=}$. Agda ensures that this definition is well-formed by performing covering and guardedness checks.[69] These checks intuitively work for coinductive types as follows. First, if we specify an element of a coinductive type, like for example a stream, then we are required to specify the outcome of each destructor. In the case of streams, we need to give both head and tail

of a stream. Second, the guardedness condition requires that the element we are specifying does not occur in another term on the right-hand side of an equation. This ensures that the result of a destructor application does not depend on this very result itself, an instance of which we have seen in Example 4.1.18. Note that coiter, as given above, is thus well-defined because it appears directly and in another term on the right-hand side of the equation that specifies the tail.

It is possible, though very tedious, to translate all the following equational specification into instances of iteration and coiteration schemes. But this translation largely obscures the proof idea and so we refrain from carrying it out. However, we should also make clear that this translation from such Agda specifications into the given calculus has the status of a conjecture for the time being. Some evidence that such a translation should be possible is given by [Gim95] and [GMM06].

The following two functions are straightforward definitions of higher derivatives of streams, that is, repeated applications of the tail destructor, and indexing of stream positions by natural numbers. Since we have seen such definitions before, we will not explain these further here.

```
δ : ∀{A} → ℕ → Stream A → Stream A
δ 0       s = s
δ (suc n) s = δ n (s .tl)

_at_ : ∀{A} → Stream A → ℕ → A
s at n = (δ n s) .hd
```

We now come to the definition of bisimilarity for streams. In Example 6.2.13, we defined stream bisimilarity as a coinductive relation in the category theoretical setup of recursive type closed categories. The idea of that definition was that stream bisimilarity is given by two destructors: one that extracts from a proof of bisimilarity a proof that the heads of related streams are equivalent, and one that extracts a bisimilarity proof for their tails. To facilitate reuse, we define bisimilarity in terms of a so called *setoid*, which is a set A together with an equivalence relation $\approx$ on it. Such a pair is given by the term S of type Setoid. To avoid further complications, we directly introduce names for the carrier A of the setoid, the relation $\approx$ and for the proof that that $\approx$ is an equivalence relation.

```
module Bisim (S : Setoid) where
  open BinRel.Setoid S
    renaming (Carrier to A; _≈_ to _≈_; isEquivalence to S-equiv)
```

Given such a setoid, let us now define bisimilarity for streams over A. We define it as a binary relations that can be written in infix notation. This is signalled by the following declaration, which says that $\sim$ is a relation with two positions (marked by the underscores). The relation itself is then defined as a coinductive type in the expected way.

```
record _~_ (s t : Stream A) : Prop where
  coinductive
  field
    hd≈ : s .hd ≈ t .hd
    tl~ : s .tl ~ t .tl
```

Since bisimilarity is a coinductive type, it comes with a coiteration principle. This principle is, as we have explained in Example 6.2.13, by the usual bisimulation proof (or coinduction) principle. We briefly show how this principle can be derived from equational specifications. First, we define what it means for a relation R to be a bisimulation on streams. This is witnessed by the following predicate on relations, which should be intuitively clear: A relation R is a bisimulation if for all streams s and t that are related by R, their heads must be equivalent (in the setoid A) and their tails must again be related by R.

$$\begin{aligned}
&\mathsf{isBisim} : \mathsf{Rel}\ (\mathsf{Stream}\ \mathsf{A}) \longrightarrow \mathsf{Prop}\\
&\mathsf{isBisim}\ _R_ = \forall\ s\ t \longrightarrow s\ R\ t \longrightarrow (s\ .\mathsf{hd} \approx t\ .\mathsf{hd}) \wedge (s\ .\mathsf{tl}\ R\ t\ .\mathsf{tl})
\end{aligned}$$

Given this notion of bisimulation relation we can now derive the usual proof principle:

$$\begin{aligned}
&\exists\text{–}\mathsf{bisim}{\longrightarrow}{\sim} : \forall\ \{_R_\} \longrightarrow \mathsf{isBisim}\ _R_ \longrightarrow\\
&\qquad\qquad \forall\ (s\ t : \mathsf{Stream}\ \mathsf{A}) \longrightarrow s\ R\ t \longrightarrow s \sim t\\
&\exists\text{–}\mathsf{bisim}{\longrightarrow}{\sim}\ R\text{–}\mathit{isBisim}\ s\ t\ q\ .\mathsf{hd}{\approx} = \wedge\text{–}\mathsf{elim}_1\ (R\text{–}\mathit{isBisim}\ s\ t\ q)\\
&\exists\text{–}\mathsf{bisim}{\longrightarrow}{\sim}\ R\text{–}\mathit{isBisim}\ s\ t\ q\ .\mathsf{tl}{\sim} =\\
&\quad \exists\text{–}\mathsf{bisim}{\longrightarrow}{\sim}\ R\text{–}\mathit{isBisim}\ (s\ .\mathsf{tl})\ (t\ .\mathsf{tl})\ (\wedge\text{–}\mathsf{elim}_2\ (R\text{–}\mathit{isBisim}\ s\ t\ q))
\end{aligned}$$

Note the striking similarity to the coiteration principle for streams, only that in this case we construct a bisimilarity proof rather than a stream. Let us briefly go through the types that appear in the definition of $\exists\text{–}\mathsf{bisim}{\longrightarrow}$. First, we are given a binary relation R and the following arguments.

$$\begin{aligned}
&R\text{–}\mathit{isBisim} : \mathsf{isBisim}\ R\\
&s, t : \mathsf{Stream}\ \mathsf{A}\\
&q : s\ R\ t
\end{aligned}$$

To show that s and t are bisimilar, we need to provide now proofs that their heads are equivalent and that the tails are related. These are the two copattern cases of $\exists\text{–}\mathsf{bisim}{\longrightarrow}$. By the definition of isBisim, we have

$$R\text{–}\mathit{isBisim}\ s\ t : (s.\mathsf{hd} \approx t.\mathsf{hd}) \wedge (s.\mathsf{tl}\ R\ t.\mathsf{tl}).$$

Thus, the first conjunction elimination gives us the sought after proof for the head case. The tail case is given by recursively constructing a bisimilarity proof for the tails from the second part of the above conjunction. This should intuitively clarify how proofs of coinductive predicates are given in Agda, and it should also highlight the similarity to the definition of the coiteration principle for streams.

As a sanity check for the correctness of the definition of bisimilarity, we prove that bisimilar streams are point-wise equivalent.

$$\begin{aligned}
&\mathsf{bisim}{\longrightarrow}\mathsf{ext}{-}{\approx} : \forall\ \{s\ t\} \longrightarrow s \sim t \longrightarrow (\forall\ n \longrightarrow s\ \mathsf{at}\ n \approx t\ \mathsf{at}\ n)\\
&\mathsf{bisim}{\longrightarrow}\mathsf{ext}{-}{\approx}\ p\ \mathsf{zero} \quad = p\ .\mathsf{hd}{\approx}\\
&\mathsf{bisim}{\longrightarrow}\mathsf{ext}{-}{\approx}\ p\ (\mathsf{suc}\ n) = \mathsf{bisim}{\longrightarrow}\mathsf{ext}{-}{\approx}\ (p\ .\mathsf{tl}{\sim})\ n
\end{aligned}$$

We finish this section by showing that bisimilarity is an equivalence relation, where we appeal to the fact that $\approx$ is an equivalence relation. This allows us to show that streams form a setoid with

bisimilarity as equivalence relation on them. These proofs should speak for themselves, thus we will not go through them in detail.

```
module SE = IsEquivalence S-equiv

s-bisim-refl : ∀ {s : Stream A} → s ~ s
s-bisim-refl      .hd≈  = SE.refl
s-bisim-refl {s} .tl~   = s-bisim-refl {s .tl}

s-bisim-sym : ∀ {s t : Stream A} → s ~ t → t ~ s
s-bisim-sym p .hd≈ = SE.sym        (p .hd≈)
s-bisim-sym p .tl~ = s-bisim-sym (p .tl~)

s-bisim-trans : ∀ {r s t : Stream A} → r ~ s → s ~ t → r ~ t
s-bisim-trans p q .hd≈ = SE.trans        (p .hd≈) (q .hd≈)
s-bisim-trans p q .tl~ = s-bisim-trans (p .tl~)  (q .tl~)

stream-setoid : Setoid
stream-setoid = record
  { Carrier = Stream A
  ; _≈_ = _~_
  ; isEquivalence = record
    { refl  = s-bisim-refl
    ; sym   = s-bisim-sym
    ; trans = s-bisim-trans
    }
  }
```

This concludes the definition of streams and bisimilarity as their canonical notion of equivalence.

7.5.3. Stream-entry Selection and the Substream Relation

We come now to the definition of the substream relation and the proof that it is transitive. As in Section 5.1.3, first define the substream relation in terms of selection from streams, which we call here $\leq'$. Also we repeat, for the sake of completeness, in Agda code the proof that selecting distributes over composition of selectors, from which we derived that $\leq'$ is transitive in Section 5.1.3. In Example 6.2.14, we showed how the substream relation can be directly defined as an inductive-coinductive relation. The second part of this section is concerned with restating this definition in Agda and proving that it is equivalent to the definition in terms of selecting. We end by deriving transitivity of the directly defined substream relation from the distribution of selecting over selector composition.

```
module Substream (A : Set) where
```

In this section, it suffices to instantiate bisimilarity for streams over A so that we compare heads by propositional equality. Recall that we introduced in Example 7.2.8 for terms s and t a type $\mathrm{Eq}_A(s, t)$

with one constructor: refl : $(x : A) \rightarrowtriangle \mathrm{Eq}_A(x, x)$. This types is meant to model propositional equality, that it, the notion of equality on A internal to the type theory. In Agda, propositional equality is a binary relation _≡_, which we import from the module Relation.Binary. Since propositional equality is an equivalence relation, $(A, \equiv)$ is a setoid. Thus, we can instantiate Bisim that setoid to obtain that streams over A form a setoid with bisimilarity.

```
open import Relation.Binary
open import Relation.Binary.PropositionalEquality renaming (setoid to ≡-setoid)
open import PropsAsTypes
open import Stream

open Bisim (≡-setoid A)
```

Recall that we defined in Example 3.2.11 the type of stream selectors Sel to be the inductive-coinductive type $\nu X.\, \mu Y.\, X + Y$, and that the first unfolding Sel_μ to the finitary steps was given by $\mu Y.\, \mathrm{Sel} + Y$. In Agda, we declare these types mutually as follows, where we directly give readable names to the corresponding destructors and constructors.

```
mutual
  record Sel : Set where
    coinductive
    field out : Selμ

  data Selμ : Set where
    pres : Sel → Selμ
    drop : Selμ → Selμ
```

Given this definition of selectors, we can define selection from streams exactly as in Example 3.2.11 by the following equational specification.

```
selectμ : Selμ → Stream A → Stream A
select  : Sel → Stream A → Stream A

select x = selectμ (x .out)

selectμ (pres x) s .hd = s .hd
selectμ (pres x) s .tl = select x (s .tl)
selectμ (drop u) s     = selectμ u (s .tl)
```

Stream selecting allows us now to define the first version of the substream relation.[70] Since it will turn out to be useful to be explicit about the x selector that witnesses that s is a substream of t, we first define a ternary relation. This relation should be read as saying that if $s \preceq\mu[x]\ t$, then s is a substream of t witnessed by the selector x. In the later proofs, we also need to be able to witness the substream relation by elements of Sel_μ, hence we also introduce the corresponding version of the substream relation.

```
_≤[_]_ : Stream A → Sel → Stream A → Prop
s ≤[ x ] t = s ~ select x t

_≤μ[_]_ : Stream A → Selμ → Stream A → Prop
s ≤μ[ u ] t = s ~ selectμ u t
```

From the ternary substream relation we can recover the definition that we used in Example 5.1.13.

```
_≤′_ : Stream A → Stream A → Prop
s ≤′ t = ∃[ x ∈ Sel ] (s ≤[ x ] t)
```

To prove transitivity of $\leq'$, we defined in Section 5.1.3 also composition of stream selectors, the definition of which we repeat here now in Agda syntax.

```
_•_  : Sel → Sel → Sel
_•μ_ : Selμ → Selμ → Selμ

(x • y) .out = (x .out) •μ (y .out)

(pres x′) •μ (pres y′) = pres (x′ • y′)
(drop u′) •μ (pres y′) = drop (u′ •μ (y′ .out))
u         •μ (drop v′) = drop (u •μ v′)
```

The main result of Section 5.1.3, Proposition 5.1.19, was that selecting from streams by a composition of selectors is equally given by composition of select functions. We went in that section through great pain to prove this result. This was caused by the fact that we had to separate the induction principle for Sel_μ from the coinduction principle of Sel. Using recursive equational specifications, we can merge these two and thereby give now a very compact proof. Note that we use bisimilarity in the following propositions to compare streams, rather than the weaker notion of propositional equality.

```
select−hom  : ∀ x y s → select (y • x) s ~ select y (select x s)
selectμ−hom : ∀ u v s → selectμ (v •μ u) s ~ selectμ v (selectμ u s)

select−hom x y s = selectμ−hom (x .out) (y .out) s

selectμ−hom (pres x) (pres y) s .hd≈ = refl
selectμ−hom (pres x) (pres y) s .tl~ = select−hom x y (s .tl)
selectμ−hom (pres x) (drop v) s     = selectμ−hom (x .out) v (s .tl)
selectμ−hom (drop u) (pres x) s     = selectμ−hom u (pres x) (s .tl)
selectμ−hom (drop u) (drop v) s     = selectμ−hom u (drop v) (s .tl)
```

This is our first proof that proceeds by mixing induction and coinduction, so let us briefly explain what is going on here. First of all, we are simultaneously proving two propositions, namely select−hom and selectμ−hom. The first is the statement we are after and constitutes the coinductive part of the proof, whereas the second proposition encapsulates the induction. select−hom is fairly simple

and is only used in the case when we have to prove that the tails of of both outputs are bisimilar (second case of selectμ−hom). In the proof of the second proposition selectμ−hom, we proceed by induction on the two Sel_μ arguments. The cases that appear here are essentially those that appear in the definition of selector composition. Note, however, that in the last two cases of selectμ−hom we do the same. At the time of writing, they both need to be there for Agda to be able to reduce the definition of •μ while checking the correctness of the proof. This is a slight nuisance, which might be improved in the future though.

To use that selecting preserves composition for proving transitivity of $\leq'$, we will need the following lemma. This lemma states that selecting respects bisimilarity, that is, bisimilar streams are mapped to bisimilar streams by select.

```
select−resp~  : ∀{s t} (x : Sel) → s ~ t → select x s ~ select x t
selectμ−resp~ : ∀{s t} (u : Selμ) → s ~ t → selectμ u s ~ selectμ u t

select−resp~ x p = selectμ−resp~ (x .out) p

selectμ−resp~ (pres x) p .hd≈ = p .hd≈
selectμ−resp~ (pres x) p .tl~ = select−resp~ x (p .tl~)
selectμ−resp~ (drop u) p      = selectμ−resp~ u (p .tl~)
```

We are now in the position to show that the relation $\leq'$ is transitive. To do so, we first show transitivity for the substream relation with explicit witnesses. This proof proceeds by chaining together bisimilarity proofs that are given in the begin ... ■ block. Since the technicalities of how such blocks can be defined in Agda would lead us of track, we will skip these details and just explain the intuition of the proof. By definition of $r \leq[x]\ s$, p is a proof of $r \sim \mathrm{select}\ (x \bullet y)\ t$. Inside the block, we write this as $r \sim\langle p\rangle\ \mathrm{select}\ (x \bullet y)\ t$. Next, we use that selecting respects bisimilarity to replace s by $\mathrm{select}\ y\ t$. Finally, we turn the composition of selecting into composition of the witnessing selectors. This proves $r \sim \mathrm{select}\ (x \bullet y)\ t$, which is by definition $r \leq[x \bullet y]\ t$. The proof term ≤−select−trans could be given by using transitivity of bisimilarity (S.trans) twice, but the presented version is certainly preferable in terms of readability.

```
≤−select−trans : ∀{r s t} {x y} → r ≤[ x ] s → s ≤[ y ] t → r ≤[ x • y ] t
≤−select−trans {r} {s} {t} {x} {y} p q =
  begin
    r
  ~⟨ p ⟩
    select x s
  ~⟨ select−resp~ x q ⟩
    select x (select y t)
  ~⟨ S.sym (select−hom y x t) ⟩
    select (x • y) t
  ■
  where
    module S = Setoid stream−setoid
```

Now that we have proved transitivity with explicit witnesses, transitivity of $\leq'$ follows by an easy manipulation of existential quantifiers:

```
≤′–trans : ∀{r s t} → r ≤′ s → s ≤′ t → r ≤′ t
≤′–trans = ∃₂–elim (λ x y p q →
                      ∃–intro (x • y) (≤–select–trans {x = x} {y} p q))
```

This concludes the recasting of the proof of transitivity relation based of stream selecting in Agda. We come now to the direct definition of the substream relation as mixed inductive-coinductive relation. The intuition for this definition was explained in Example 6.2.14, we merely repeat the definition here in Agda syntax.

```
mutual
  record _≤_ (s t : Stream A) : Prop where
    coinductive
    field out≤ : s ≤μ t

  data _≤μ_ (s t : Stream A) : Prop where
    match : (s .hd ≡ t .hd) → (s .tl ≤ t .tl) → s ≤μ t
    skip : (s ≤μ t .tl) → s ≤μ t
```

In the remainder of the section we show that this direct definition is equivalent to the selecting-based one and derive transitivity of ≤ from there. The first step towards this is to extract from a proof of ≤ a selector witness.

```
witness  : {s t : Stream A} → s ≤ t → Sel
witnessμ : {s t : Stream A} → s ≤μ t → Selμ

witness p .out = witnessμ (p .out≤)

witnessμ (match _ t≤) = pres (witness t≤)
witnessμ (skip u)     = drop (witnessμ u)
```

We can now use the extracted witness to show that the substream relation is included in the witness-based substream relation and hence in the select-based substream relation.

```
≤–implies–select≤   : ∀{s t} → (p : s ≤ t) → s ≤[ witness p ] t
≤μ–implies–selectμ≤ : ∀{s t} → (p : s ≤μ t) → s ≤μ[ witnessμ p ] t

≤–implies–select≤ {s} {t} p = ≤μ–implies–selectμ≤ (p .out≤)

≤μ–implies–selectμ≤ (match h≡ t≤) .hd≈ = h≡
≤μ–implies–selectμ≤ (match h≡ t≤) .tl~ = ≤–implies–select≤ t≤
≤μ–implies–selectμ≤ (skip q)           = ≤μ–implies–selectμ≤ q

≤–implies–≤′ : ∀{s t} → s ≤ t → s ≤′ t
≤–implies–≤′ p = ∃–intro (witness p) (≤–implies–select≤ p)
```

Conversely, we can construct from a selector witness a proof for the substream relation. This allows us then to show that the select-based substream relation is in included in the substream relation.

```
select≤−implies−≤ : ∀{s t} (x : Sel) → s ≤[ x ] t → s ≤ t
selectμ≤⇒≤μ : ∀{s t} (u : Selμ) → s ≤μ[ u ] t → s ≤μ t

select≤−implies−≤ x p .out≤ = selectμ≤⇒≤μ (x .out) p

selectμ≤⇒≤μ (pres x) p  = match (p .hd≈) (select≤−implies−≤ x (p .tl~))
selectμ≤⇒≤μ (drop u) p = skip (selectμ≤⇒≤μ u p)

≤′−implies−≤ : ∀{s t} → s ≤′ t → s ≤ t
≤′−implies−≤ = ∃−elim select≤−implies−≤
```

Putting these to results together, we obtain a proof that the two definitions for the substream relation are equivalent.

```
≤−and−≤′−equiv : ∀ s t → s ≤ t ⇔ s ≤′ t
≤−and−≤′−equiv s t = equivalence ≤−implies−≤′ ≤′−implies−≤
```

Finally, using this equivalence, we can derive from the fact that $\leq'$ is transitive that also $\leq$ is transitive.

```
≤−trans : ∀{r s t} → r ≤ s → s ≤ t → r ≤ t
≤−trans p q = ≤′−implies−≤ (≤′−trans (≤−implies−≤′ p) (≤−implies−≤′ q))
```

7.6. Discussion

In this chapter we have have introduced a type theory $\lambda P\mu$ that is solely based on inductive and coinductive types, following the ideas of the category theoretical development in Chapter 6. This is in contrast to other type theories that usually have separate type constructors for, for example, the (dependent) function space or coproducts. The result is a theory with a small set of rules for its judgements and reduction relation. To justify the use of our type theory as logic, we also proved that the reduction relation preserves types and is strongly normalising on well-typed terms. Combining the present theory with that in [Nor07] would give us a justification for a large part Agda's current type system, especially including coinductive types.

Contributions

Let us briefly sum up the contributions made in this chapter. First of all, we introduced the dependent type theory $\lambda P\mu$ and showed how important logical operators and type formers can be represented in this theory. We also discussed how the, somewhat worn-out, standard example of vectors arises as a recursive (inductive) type in $\lambda P\mu$. Other examples, like the substream relation, were given as an application in Section 7.5. Second, we showed that computations of terms, given in form of a reduction relation, are meaningful, in the sense that the reduction relation preserves types (subject reduction) and that all computations are terminating (strong normalisation). Thus, under the propositions-as-types interpretation, our type theory can serve as formal framework

for intuitionistic reasoning. Finally, we have shown how the calculus $\lambda P\mu$ can be extended with dependent iteration to a calculus $\lambda P\mu^+$, so that an induction principle for inductive types is also available in the propositions-as-types interpretation.

Related Work

A major source of inspiration for the setup of our type theory is categorical logic. Especially, the use of fibrations, brought forward in [Jac99], helped a great deal in understanding how the work of Hagino [Hag87] can be extended to dependent types. Another source of inspiration is the view of type theories as internal language or even free models for classes of categories, see for example [LS88]. This view is especially important in topos theory, where final coalgebras have been used as foundation for predicative, constructive set theory [Acz88; vdBer06; vdBdM07]. In Chapter 6 we saw how these ideas can be extended to general strictly positive inductive and coinductive types, which form the category theoretical analogue of the type theory $\lambda P\mu$.

Let us briefly discuss other type theories that the present work relates to. Especially close is the copattern calculus introduced in [Abe+13] and Section 3.2, as there the coinductive types are also specified by the types of their destructors. However, said calculus does not have dependent types, and it is based on systems of equations to define terms, whereas the calculus in the present paper is based on iteration and coiteration schemes, similar to the calculus $\lambda\mu\nu$ from Section 3.1. There are other calculi that use iteration and coiteration as basis for defining operations on non-dependent inductive and coinductive types, respectively, see Section 3.4 for further discussion.

To ensure strong normalisation, the copatterns have been combined with size annotations in [AP13]. Due to the nature of the reduction relation in these copattern-based calculi, strong normalisation also ensures productivity for coinductive types or, more generally, observational normalisation, cf. Section 4.1. As another way to ensure productivity, guarded recursive types were proposed and in [Biz+16a] guarded iteration was extended to dependent types. Guarded recursive types go beyond strictly positive types, which is what we restricted ourselves to in this chapter, but also to positive and even negative types. However, it is not clear how one can include inductive types into such a type theory, which are, in the author's opinion, crucial to mathematics and computer science. Finally, in [Sac13] another type theory with type-based termination conditions and a type former for streams has been introduced. This type theory, however, lacks again dependent coinductive types.

Future Work

There are still some open questions, regarding the present type theory, that we wish to settle in the future. First of all, a basic property of the reduction relation that is still missing is confluence. Second, we have constructed the type theory with certain categorical structures in mind, and it is easy to interpret the types and terms in data type closed categories as we introduced them in Section 6.2. However, one has to be careful in showing the soundness of such an interpretation, in which two subtleties occur. First, if for types A and B we have that $A \longleftrightarrow_T B$, then we had better ensured that their interpretations agree, that is $[\![A]\!] = [\![B]\!]$, because of the conversion rule of $\lambda P\mu$. This property corresponds to using *split fibrations*, instead of merely cloven fibrations, in Section 6.2, cf. [Jac99, Sec. 10.3]. The second subtlety, which might be in the way of a sound interpretation, is the definition of substitution on recursive types. However, the Beck-Chevalley condition that we proposed in Section 6.5 is designed precisely to give us the soundness of substitution for the

interpretation of types. Though it seems that these conditions suffice to give an interpretation of $\lambda P\mu$, these conditions need to be carefully checked, as we have learned from previous attempts on category theoretical semantics for dependent type theories [Str91].

In Section 7.4, we mentioned already that the strong normalisation proof from Section 7.3.3 does not subsume dependent iteration. It is expected that the proof can be extended accordingly, but this certainly needs to be checked carefully. Moreover, one has to ask what the dual principle to dependent iteration is for coinductive types. The expectation would be coinduction, that is, that internally provable bisimilarity of terms implies their propositional equality. However, setting up a calculus in which coinduction holds is far from trivial, if we want to preserve decidability of type checking. In particular, this is because bisimilarity on function types corresponds to pointwise equality, so that the coinduction principle would give us therefore extensional function types, cf. Section 6.4.3 and Example 7.2.4. An idea in this direction would be to combine observational type theory [AMS07] with ideas of cubical/homotopy type theory [Coh+16; Uni13], which would result in more complex calculi, albeit with decidable type checking. This is inspired by an idea that was communicated by Conor McBride in a talk at the Agda Implementors' Meeting XXIII in Glasgow, and it was further developed by the author in talks given at the TYPES'16 meeting [BG16b] and the Chocola Seminar at the ENS Lyon in September 2016. Due to the complexity of this task, we leave the extension of $\lambda P\mu$ or $\lambda P\mu^+$ with coinduction open for now.

Finally, certain acceptable facts are not provable in $\lambda P\mu^+$, since, for instance, we do not have universes. Another common feature that is missing from the type theory, is predicative polymorphism, which is in fact justified and desirable from the categorical point of view. Both extensions go in the direction of turning the calculus into a foundation for Agda. It is expected that such extensions are straightforward but tedious to carry out.

Notes

[58] Noam Chomsky, "Language and Freedom". In "On Anarchism", Nathan Schneider (Ed.), 2014.

[59] Indeed, subject reduction is broken in Coq, which is caused by the fact that coinductive types are defined through constructors that can be pattern matched against. Consider the following example.

```
1  CoInductive Stream := Cons : Stream → Stream.
2  CoFixpoint c : Stream := Cons c.
3  Definition foo : c = Cons c :=
4    match c as t return match t with Cons t' ⇒ t = Cons t' end
5    with Cons _ ⇒ eq_refl end.
6  Eval compute in foo.
7  Definition bar : c = Cons c := Eval compute in foo.
```

In Coq 8.6, line 6 outputs

```
= eq_refl
: c = Cons c
```

but the definition in the last line leads to the error

```
Error
The term "eq_refl" has type
  "Cons (cofix c  : Stream := Cons c) =
   Cons (cofix c  : Stream := Cons c)"
while it is expected to have type "c = Cons c".
```

Thus, Coq accepts the definition of foo, which computes to eq_refl, but it cannot infer that eq_refl is of type c = Cons c. In other words, the computation of foo gives the resulting value another type than it actually has. Hence, subject reduction is broken in Coq with coinductive types.

This has in fact already been observed as an inherit limitation of the constructor-based approach to coinductive types by Giménez in his thesis [Gim96]. The problem is that one either only obtains a weak form of subject reduction or that constructor expansion for coinductive types needs to be added, where the latter would make type checking undecidable. At `https://sympa.inria.fr/sympa/arc/coq-club/2008-06/msg00022.html` and `https://sympa.inria.fr/sympa/arc/coq-club/2008-06/msg00045.html` one may find the corresponding discussions in the context of Coq.

[60] In the case of streams, which is not really representative of general coinductive types however, the usual representation in terms of inductive types is to take functions $\mathrm{Nat} \to A$ as streams over A. But even in this simple representation one only obtains a useful computation principle in presence of η-reduction. This can be seen as follows. First, one defines an alternative type of streams by $\mathrm{Str}_A \coloneqq \mathrm{Nat} \to A$. On this type, we can define coiteration for $c \colon X \to A \times X$ and $x : X$ by pattern matching:

$$\mathrm{coiter}'\ c\ x\ 0 = \pi_1\ (c\ x)$$
$$\mathrm{coiter}'\ c\ x\ (n+1) = \mathrm{coiter}'\ c\ (\pi_2\ (c\ x))\ n$$

Moreover, the tail observation is defined to be $\mathrm{tl}'\ s \coloneqq \lambda n.\ s\ (n+1)$. Then we have

$$\mathrm{tl}'\ (\mathrm{coiter}'\ c\ x) = \lambda n.\ \mathrm{coiter}'\ c\ x\ (n+1) \equiv \lambda n.\ \mathrm{coiter}'\ c\ (\pi_2\ (c\ x))\ n.$$

But this is only convertible to $\mathrm{coiter}'\ c\ (\pi_2\ (c\ x))$ in the presence of η-conversion. Adding η-conversion might not be considered so bad, but the reader is invited to derive the computational rules for more general coinductive types as they are encoded in [cLa16], which is only possible with general function extensionality.

[61] This mechanism of handling dependencies in type constructors is very similar to that for handling assumption valid relative to a context in [NPP08]. We thank an anonymous referee of [BG16a] for the pointer.

[62] It should be noted that the arrow $\twoheadrightarrow$ is *not* meant to be the function space in a higher universe, as one finds it in MLTT with (higher) universes or in Agda. Rather, parameter contexts are a syntactic tool to deal elegantly with parameters of type constructors. On terms, parameters and function spaces are of course not unrelated. We explain this in Example 7.2.4.

[63] The precise definition of the compatible closure can be found in the Agda formalisation [Bas18b].

[64] Essentially, parameter abstraction and instantiation for types corresponds to a simply typed λ-calculus on the type level.

[65] One interesting result, used to prove the following lemmas, is that the interpretation of types is monotone in δ, and that the interpretation of coinductive types is the largest set closed under destructors. This suggests that it might be possible to formulate the definition of the interpretation in categorical terms.

[66] We note that the calculus with dependent iteration does *not* satisfy the *uniqueness of identity proofs (UIP)* principle. The reason is that the dependent iteration scheme amounts for Eq_A to

$$\frac{\Gamma, x : A, y : \top, z : \top \vdash g : C \mathbin{@} x \mathbin{@} x \mathbin{@} (\alpha_1 \mathbin{@} x \mathbin{@} y)}{\Gamma \vdash \mathrm{diter}\,(x, y, z.\, g) : x : A, y : A, z : \mathrm{Eq}_A(x, y) \multimap C \mathbin{@} x \mathbin{@} y \mathbin{@} z}$$

Leaving out variables of type $\top$, cf. Example 7.4.4, and writing $\mathrm{refl}\ s$ for $\alpha_1 \mathbin{@} s \mathbin{@} \langle\rangle$, we obtain from there the following rule.

$$\frac{\Gamma, x : A \vdash g : C \mathbin{@} x \mathbin{@} x \mathbin{@} (\mathrm{refl}\ x)}{\Gamma \vdash \mathrm{diter}\,(x.\, g) : x : A, y : A, z : \mathrm{Eq}_A(x, y) \multimap C \mathbin{@} x \mathbin{@} y \mathbin{@} z}$$

If we now try to prove that all proofs of $\mathrm{Eq}_A(x, y)$ are given by reflexivity, we are obliged to find a type C with

$$C \mathbin{@} x \mathbin{@} x \mathbin{@} (\mathrm{refl}\ x) \equiv \mathrm{Eq}(\mathrm{refl}\ x, \mathrm{refl}\ x).$$

A natural candidate would thus be $C = (x, y, z).\, \mathrm{Eq}(z, \mathrm{refl}\ x)$. The problem with this is that z is of type $\mathrm{Eq}_A(x, y)$ but $\mathrm{refl}\ x$ is of type $\mathrm{Eq}_A(x, x)$. Hence, C is not a well-formed type.

This is of course not a formal argument that the calculus does not fulfil the UIP principle, but it gives some good evidence. For detailed discussions of the failure of UIP in Martin-Löf type theory, which should be adaptable to our calculus, see the work by Hofmann and Streicher [HS94].

[67] The commonly used definition of coproducts (Σ-types) in Agda is the following.

```
record Σ (A : Set) (B : A → Set) : Set where
  field
    proj₁ : A
    proj₂ : B proj₁
```

Note that the second destructor refers to the first, which is not allowed in our setup and leads to a strong elimination principle for Σ, which we obtained in Example 7.4.5 from dependent iteration.

[68] In fact, $X : I \to \mathrm{Set}$ is a function in Agda, only that it is a type in the universe Set_1. This is similar to the situation in category theory, where a family of sets can be given by a functor from I, seen as a discrete category, to the category of sets. Such a functor is then an object in the large category of functors between categories.

[69] Agda also supports more sophisticated termination checks by using sized types. We will not discuss these further here though.

[70] Bertot [Ber05] defines a different notion of stream selection that allows one to select from streams by productive predicates. This way of selecting from streams is essentially equivalent to our notion of selecting by positions, see.

Epilogue

Everything Faustian is far away from me. [...] Thousands of questions are silenced as if dissolved. There are neither doctrines nor heresies. The possibilities are endless. Only faith in them lives creatively in me.

— Paul Klee, 1916.

Towards the end of writing this thesis, in July 2017, I was invited to give a tutorial on the topics of Chapters 6 and 7 at the Université Savoie in the magnificent area of Lac de Bourget. This tutorial was part of a "Workshop on Coinduction in Type Theory", organised by Tom Hirschowitz. The workshop was a perfect illustration of the diversity that can be found in the views on induction and coinduction. There were people whose main interest was indeed type theory, others came from a category theoretical background, and still others came from logic or base their work mostly on classical set theory. Having all these different views leads to very stimulating but also intense discussions, in particular because everyone has seen or used inductive-coinductive objects in some form or even worked on them.

Trying to accommodate all these different views in one thesis is impossible, and even though I tried to reach as wide an audience as possible, I had to make a selection. In the end, I chose those topics that give a general account to inductive-coinductive objects, in the hope that the provided theory may serve as a foundation for reasoning about such objects. Unfortunately, selecting the more abstract theory as a focus means that many of the wonderful and interesting applications of inductive-coinductive reasoning could not be covered here. Though one has to say that the point of Mathematics is in any case not to generate examples but to construct general theories that account for and explain these examples.

But I'm digressing. We have seen some usages of inductive-coinductive reasoning in this thesis, like the substream relation or the correctness proof for μ-recursion. However, there are many more, some of which we discussed in the intro, and plenty of them are not explored. For instance, Kőnig's lemma and Brouwer's fan theorem mix inductive and coinductive reasoning, but only the latter of which has been studied so far from this perspective [NUB11]. Similarly, Cauchy sequences and weak bisimilarity also involve both induction and coinduction, which again is hardly ever made explicit. Finally, a recurring theme in this thesis is that already the very notion of function, on which most of Mathematics is built, is itself coinductive. A striking consequence of this is that the coinduction principle for function spaces, namely that bisimilarity implies equality, corresponds to the usual principle of function extensionality: two functions are equal precisely if they agree on all arguments. This is another insight that arose from the abstract treatment of inductive-coinductive objects, even though I was told that this is sort of "folklore". I hope that these examples show that it is worthwhile to study mixed inductive-coinductive reasoning and to take it seriously. So, I hope that this thesis may inspire others to study inductive-coinductive objects, and advance the theory provided here beyond its, now very basic, state.

The Way Onward

So where to go from here? There are many problems that this thesis leaves open and many directions for future research that it suggests. Out of those that we discussed throughout the chapters, let me highlight a few that I deem most important.

One of the observations that we made was that iteration and coiteration schemes require a lot of work and ingenuity to bring mixed inductive-coinductive specifications into a form to which these schemes can be applied. For this reason, we generally preferred specifications in terms of recursive equations. This leaves open the problem of reconciling iteration and coiteration with recursive equations by, for instance, expressing observationally normalising terms in $\lambda\mu\nu{=}$ as terms in $\lambda\mu\nu$. A similar direction is to explore how one can dualise the principle of well-founded induction to coiteration and coinduction. Well-founded induction can be used to implement a recursive function by establishing a well-founded relation on its arguments and showing that the recursion descends the relation. Since this principle allows for fairly general recursive specifications, one has to ask whether there is a similar principle to specify productive processes.

Another major obstacle to the usability of type theories as foundation for mathematical reasoning is, in my opinion, the lack of a proper coinduction principle. The trouble is that without a coinduction principle, one has to explicitly use hand-crafted equivalence relations (bisimilarity) everywhere. This is very irritating and puts an enormous load on the user of a type theory. As we sketched in Section 7.6, it should be possible to extend $\lambda\mathrm{P}\mu^+$ with a coinduction principle through a combination of ideas from observational type theory and cubical type theory. Together with type theoretical universes we would obtain a type theory that covers already a large part of Mathematics and Computer Science. What is missing then are only quotients and subobjects or, more generally, colimit and limit constructions. These arise by suitably extending the dependent inductive and coinductive types of $\lambda\mathrm{P}\mu^+$ — to higher inductive and coinductive types, if the reader is familiar with these concepts. If we are able to pull off these developments, then this would lead to a type theory that should cover, except for non-constructive proofs, most of Mathematics and Computer Science.

Apart from developments in the field of type theory, there are also other aspects of inductive-coinductive reasoning that I think need to be explored further. First of all, the logic $\mathbf{FOL}_{\blacktriangleright}$ that we developed in Section 5.2 needs to be fleshed out for mixed inductive-coinductive proofs. Moreover, the use of the later modality to ensure correctness of recursive proofs is not limited to bisimilarity, but should work for more general coinductive, and perhaps even for inductive-coinductive, predicates and relations. Therefore, it would in my opinion be fruitful to explore this logic further. Next, we found that up-to techniques allowed us to carry out the soundness proof for the logic $\mathbf{FOL}_{\blacktriangleright}$ more easily. This seems to be a general phenomenon that needs to be explored for other logics and also for type theories. Finally, in the light of the goal to develop type theories that feature general coinduction, it would be worthwhile to study also the category theoretical side of such type theories. To be more precise, I think that it would be interesting to study Universal Coalgebra, as it was coined by Rutten, in so-called $(\infty, 1)$-categories. Combining all these endeavours would result in a convergence of type theory and category theory.

References

[Abb03] Michael Abbott. 'Categories of Containers'. Leicester, 2003. URL: http://www.cs.le.ac.uk/people/ma139/docs/thesis.pdf (visited on 19/06/2016) (cit. on pp. 164, 190, 191, 194–197, 199, 217).

[AAG05] Michael Abbott, Thorsten Altenkirch and Neil Ghani. 'Containers: Constructing Strictly Positive Types'. In: *Theoretical Computer Science*. Applied Semantics: Selected Topics Applied Semantics: Selected Topics 342.1 (2005), pp. 3–27. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.06.002 (cit. on pp. 9, 164, 181, 188–191, 193, 217).

[Abe04] Andreas Abel. 'Termination Checking with Types'. In: *ITA* 38.4 (2004), pp. 277–319. DOI: 10.1051/ita:2004015 (cit. on p. 39).

[AA99] Andreas Abel and Thorsten Altenkirch. 'A Predicative Strong Normalisation Proof for a Lambda-Calculus with Interleaving Inductive Types'. In: *TYPES'99*. Vol. 1956. LNCS. Springer, 1999, pp. 21–40. DOI: 10.1007/3-540-44557-9_2 (cit. on pp. 69, 101, 103).

[AMU05] Andreas Abel, Ralph Matthes and Tarmo Uustalu. 'Iteration and Coiteration Schemes for Higher-Order and Nested Datatypes'. In: *Theor. Comput. Sci.* 333 (1-2 2005), pp. 3–66. DOI: 10.1016/j.tcs.2004.10.017 (cit. on p. 9).

[AP13] Andreas Abel and Brigitte Pientka. 'Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity'. In: *ICFP*. 2013, pp. 185–196. DOI: 10.1145/2500365.2500591 (cit. on pp. 9, 68, 77, 126, 272, 309).

[Abe+13] Andreas Abel, Brigitte Pientka, David Thibodeau and Anton Setzer. 'Copatterns: Programming Infinite Structures by Observations'. In: *Proc. of POPL*. Symposium on Principles of Programming Languages. ACM, 2013, pp. 27–38. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429075 (cit. on pp. 9, 37, 49, 63, 65, 68, 272, 309).

[Abr90] Samson Abramsky. 'The Lazy Lambda Calculus'. In: *Research Topics in Functional Programming*. Addison-Wesley, 1990, pp. 65–116 (cit. on pp. 102, 105, 152, 153).

[Acz88] Peter Aczel. *Non-Well-Founded Sets*. Lecture Notes 14. Center for the Study of Language and Information, Stanford University, 1988. ISBN: 0-937073-22-9 (cit. on pp. 7, 272).

[Acz+03] Peter Aczel, Jiří Adámek, Stefan Milius and Jiří Velebil. 'Infinite Trees and Completely Iterative Theories: A Coalgebraic View'. In: *Theoretical Computer Science* 300 (1–3 2003), pp. 1–45. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(02)00728-4 (cit. on p. 218).

[AM89] Peter Aczel and Nax Paul Mendler. 'A Final Coalgebra Theorem'. In: *Proceedings of Category Theory and Computer Science*. Vol. 389. LNCS. Springer, 1989, pp. 357–365. DOI: 10.1007/BFb0018361 (cit. on p. 7).

[AMV11] Jiří Adámek, Stefan Milius and Jiří Velebil. 'Semantics of Higher-Order Recursion Schemes'. In: *Logical Methods in Computer Science* 7.1 (2011), pp. 1–43. DOI: 10.2168/LMCS-7(1:15)2011 (cit. on p. 101).

[Agd15] Programming Logic group on Agda. *Agda Documentation*. Version 2.4.2.5. Chalmers and Gothenburg University, 2015. URL: http://wiki.portal.chalmers.se/agda/ (cit. on pp. 9, 154, 167, 224).

[AU79] Alfred V. Aho and Jeffrey D. Ullman. 'The Universality of Data Retrieval Languages'. In: *Conference Record of POPL 1979*. ACM Press, 1979, pp. 110–120. DOI: 10.1145/567752.567763 (cit. on p. 163).

[ACS15] Benedikt Ahrens, Paolo Capriotti and Régis Spadotti. 'Non-Wellfounded Trees in Homotopy Type Theory'. In: (2015). arXiv: 1504.02949 [cs, math] (cit. on p. 224).

[AMM05] Thorsten Altenkirch, Conor McBride and James McKinna. *Why Dependent Types Matter*. 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.8190 (cit. on p. 167).

[AMS07] Thorsten Altenkirch, Conor McBride and Wouter Swierstra. 'Observational Equality, Now!' In: *Proc. of PLPV '07*. Workshop on Programming Languages Meets Program Verification. ACM, 2007, pp. 57–68. ISBN: 978-1-59593-677-6. DOI: 10.1145/1292597.1292608 (cit. on pp. 101, 155, 273).

[AM09] Thorsten Altenkirch and Peter Morris. 'Indexed Containers'. In: *LICS*. IEEE Computer Society, 2009, pp. 277–285. DOI: 10.1109/LICS.2009.33 (cit. on pp. 181, 190).

[App+07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards and Jérôme Vouillon. 'A Very Modal Model of a Modern, Major, General Type System'. In: *POPL*. ACM, 2007, pp. 109–122. DOI: 10.1145/1190216.1190235 (cit. on pp. 126, 137, 154).

[AM13] Robert Atkey and Conor McBride. 'Productive Coprogramming with Guarded Recursion'. In: *ICFP*. ACM, 2013, pp. 197–208. DOI: 10.1145/2500365.2500597 (cit. on pp. 68, 126, 130, 132, 157).

[Awo10] Steve Awodey. *Category Theory*. 2nd ed. Oxford Logic Guides 52. Oxford University Press, 2010. ISBN: 978-0-19-923718-0 (cit. on pp. 22, 25).

[Bae09] David Baelde. 'On the Proof Theory of Regular Fixed Points'. In: *Proceedings of TABLEAUX 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 93–107. ISBN: 978-3-642-02716-1. DOI: 10.1007/978-3-642-02716-1_8 (cit. on p. 126).

[Bar85] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Revised edition. Vol. 103. Studies in Logic and the Foundations of Mathematics. Amsterdam; New York; Oxford: North Holland, 1985. 621 pp. ISBN: 978-0-444-87508-2 (cit. on p. 89).

[Bar91] Henk Barendregt. 'Introduction to Generalized Type Systems'. In: *J. Funct. Program.* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025 (cit. on p. 182).

[Bar13] Henk Barendregt. 'Foundations of Mathematics from the Perspective of Computer Verification'. In: *Mathematics, Computer Science and Logic - A Never Ending Story: The Bruno Buchberger Festschrift*. Springer, 2013, pp. 1–49. ISBN: 978-3-319-00966-7. DOI: 10.1007/978-3-319-00966-7_1 (cit. on p. 9).

[BDS13] Henk Barendregt, Wil Dekkers and Richard Statman. *Lambda Calculus with Types*. Cambridge ; New York: Cambridge University Press, 2013. 854 pp. ISBN: 978-0-521-76614-2 (cit. on pp. 39, 68, 69, 105, 168).

[Bar93] Michael Barr. 'Terminal Coalgebras in Well-Founded Set Theory'. In: *Theoretical Computer Science* 114.2 (1993), pp. 299–315. DOI: 10.1016/0304-3975(93)90076-6 (cit. on p. 193).

[Bar04] Falk Bartels. 'On Generalised Coinduction and Probabilistic Specification Formats'. PhD. Amsterdam: Vrije Universiteit, 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.4992&rep=rep1&type=pdf (visited on 12/12/2016) (cit. on pp. 8, 76).

[Bar+04] Gilles Barthe, Maria João Frade, Eduardo Giménez, Lúis Pinto and Tarmo Uustalu. 'Type-Based Termination of Recursive Definitions'. In: *Mathematical Structures in Computer Science* 14.1 (2004), pp. 97–141. DOI: 10.1017/S0960129503004122 (cit. on pp. 68, 126).

[BGM71] K. Jon Barwise, Robin O. Gandy and Yiannis N. Moschovakis. 'The Next Admissible Set'. In: *J. Symb. Log.* 36.1 (1971), pp. 108–120. DOI: 10.2307/2271519 (cit. on p. 7).

[BM96] K. Jon Barwise and Lawrence S. Moss. *Vicious Circles. On the Mathematics of Non-Wellfounded Phenomena*. CSLI Lecture Notes 60. Center for the Study of Language and Information, Chigaco University Press, 1996. ISBN: 1-57586-008-2 (cit. on p. 7).

[Bas15b] Henning Basold. *Higher Coinductive Types - cLab*. 2015. URL: http://coalg.org/clab/Higher_Coinductive_Types (visited on 25/11/2016) (cit. on p. 69).

[Bas16] Henning Basold. *Stream Differential Equations in Agda*. GitHub. 2016. URL: https://github.com/hbasold/Sandbox (visited on 12/12/2016) (cit. on p. 77).

[Bas18b] Henning Basold. *Code Repository*. 2018. URL: https://perso.ens-lyon.fr/henning.basold/code/ (cit. on pp. 8, 228, 229, 245, 257, 274).

[Bek99] Lev D. Beklemishev. 'Parameter Free Induction and Provably Total Computable Functions'. In: *Theor. Comput. Sci.* 224 (1-2 1999), pp. 13–33. DOI: 10.1016/S0304-3975(98)00305-3 (cit. on pp. 126, 154).

[Ber05] Yves Bertot. 'Filters on CoInductive Streams, an Application to Eratosthenes' Sieve'. In: *Proceedings of TLCA 2005*. Vol. 3461. LNCS. Springer, 2005, pp. 102–115. DOI: 10.1007/11417170_9 (cit. on pp. 8, 58, 275).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004 (cit. on p. 223).

[Bir+16] Lars Birkedal, Alěs Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters and Andrea Vezzosi. 'Guarded Cubical Type Theory: Path Equality for Guarded Recursion'. In: *CSL 2016*. Vol. 62. LIPIcs. Schloss Dagstuhl, 2016, 23:1–23:17. ISBN: 978-3-95977-022-4. DOI: 10.4230/LIPIcs.CSL.2016.23 (cit. on p. 155).

[BH99] Lars Birkedal and Robert Harper. 'Relational Interpretations of Recursive Types in an Operational Setting'. In: *Inf. Comput.* 155 (1-2 1999), pp. 3–63. DOI: 10.1006/inco.1999.2828 (cit. on p. 153).

[BM13] Lars Birkedal and Rasmus Ejlers Møgelberg. 'Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes'. In: *LICS*. IEEE Computer Society, 2013, pp. 213–222. DOI: 10.1109/LICS.2013.27 (cit. on pp. 143, 155).

[Bir+11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer and Kristian Støvring. 'First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees'. In: *Proceedings LICS 2011*. IEEE Computer Society, 2011, pp. 55–64. DOI: 10.1109/LICS.2011.16 (cit. on pp. 22, 23, 137, 155).

[Biz+16a] Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Rasmus Ejlers Møgelberg and Lars Birkedal. 'Guarded Dependent Type Theory with Coinductive Types'. In: *Proceedings of FOSSACS'16*. FOSSACS'16. 2016 (cit. on p. 272).

[Biz+16b] Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg and Lars Birkedal. 'Guarded Dependent Type Theory with Coinductive Types'. In: *FoSSaCS*. Vol. 9634. Lecture Notes in Computer Science. Springer, 2016, pp. 20–35. arXiv: 1601.01586 (cit. on pp. 126, 155).

[BRV01] Patrick Blackburn, Maarten de Rijke and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001. DOI: 10.1017/CBO9781107050884 (cit. on p. 85).

[BvB07] Patrick Blackburn and Johan van Benthem. 'Modal Logic: A Semantic Perspective'. In: Patrick Blackburn, Johan van Benthem and Frank Wolter. *Handbook of Modal Logic*. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, 2007, pp. 1–84. ISBN: 978-0-444-51690-9. DOI: 10.1016/S1570-2464(07)80004-8. URL: http://cgi.csc.liv.ac.uk/~frank/MLHandbook/ (cit. on p. 85).

[Bla+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu and Dmitriy Traytel. 'Truly Modular (Co)Datatypes for Isabelle/HOL'. In: *Proceedings of ITP 2014*. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 93–110. DOI: 10.1007/978-3-319-08970-6_7 (cit. on p. 125).

[Bon+13] Filippo Bonchi, Georgiana Caltais, Damien Pous and Alexandra Silva. 'Brzozowski's and Up-To Algorithms for Must Testing'. In: *Proceedings of APLAS 2013*. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 1–16. DOI: 10.1007/978-3-319-03542-0_1 (cit. on p. 156).

[Bon+14] Filippo Bonchi, Daniela Petrişan, Damien Pous and Jurriaan Rot. 'Coinduction Up-to in a Fibrational Setting'. In: *Proc. of CSL-LICS '14*. New York, USA: ACM, 2014, 20:1–20:9. ISBN: 978-1-4503-2886-9. DOI: 10.1145/2603088.2603149 (cit. on pp. 8, 27, 28, 118, 142, 154, 158).

[BP13] Filippo Bonchi and Damien Pous. 'Checking NFA Equivalence with Bisimulations up to Congruence'. In: *Proceedings of POPL '13*. ACM, 2013, pp. 457–468. DOI: 10.1145/2429069.2429124 (cit. on p. 156).

[BRS09] Marcello M. Bonsangue, Jan Rutten and Alexandra Silva. 'An Algebra for Kripke Polynomial Coalgebras'. In: *Proceedings of LICS 2009*. IEEE Computer Society, 2009, pp. 49–58. DOI: 10.1109/LICS.2009.18 (cit. on p. 217).

[Bor08] Francis Borceux. *Handbook of Categorical Algebra: Volume 1, Basic Category Theory*. Cambridge University Press, 2008. 364 pp. ISBN: 978-0-521-06119-3 (cit. on pp. 20, 21, 29–31, 33).

[BMH07] Edwin Brady, James McKinna and Kevin Hammond. 'Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types'. In: *Proceedings of TFP 2007*. Vol. 8. Trends in Functional Programming. Intellect, 2007, pp. 159–176 (cit. on p. 167).

[Bro05] James Brotherston. 'Cyclic Proofs for First-Order Logic with Inductive Definitions'. In: *Proceedings of TABLEAUX 2005*. Vol. 3702. Lecture Notes in Computer Science. Springer, 2005, pp. 78–92. DOI: 10.1007/11554554_8 (cit. on p. 126).

[BBC08] James Brotherston, Richard Bornat and Cristiano Calcagno. 'Cyclic Proofs of Program Termination in Separation Logic'. In: *Proceedings of POPL 2008*. ACM, 2008, pp. 101–112. DOI: 10.1145/1328438.1328453 (cit. on p. 126).

[BDP11] James Brotherston, Dino Distefano and Rasmus Lerchedahl Petersen. 'Automated Cyclic Entailment Proofs in Separation Logic'. In: *Proceedings of CADE-23*. Vol. 6803. LNCS. Springer, 2011, pp. 131–146. DOI: 10.1007/978-3-642-22438-6_12 (cit. on p. 126).

[BG14] James Brotherston and Nikos Gorogiannis. 'Cyclic Abduction of Inductively Defined Safety and Termination Preconditions'. In: *Proceedings of SAS 2014*. Vol. 8723. LNCS. Springer, 2014, pp. 68–84. DOI: 10.1007/978-3-319-10936-7_5 (cit. on p. 126).

[BGP12] James Brotherston, Nikos Gorogiannis and Rasmus Lerchedahl Petersen. 'A Generic Cyclic Theorem Prover'. In: *Proceedings of APLAS 2012*. Vol. 7705. LNCS. Springer, 2012, pp. 350–367. DOI: 10.1007/978-3-642-35182-2_25 (cit. on p. 126).

[BS07] James Brotherston and Alex Simpson. 'Complete Sequent Calculi for Induction and Infinite Descent'. In: *Proceedings of LICS 2007*. IEEE Computer Society, 2007, pp. 51–62. DOI: 10.1109/LICS.2007.16 (cit. on p. 126).

[BS11] James Brotherston and Alex Simpson. 'Sequent Calculi for Induction and Infinite Descent'. In: *J. Log. Comput.* 21.6 (2011), pp. 1177–1216. DOI: 10.1093/logcom/exq052 (cit. on p. 126).

[Brz64] Janusz A. Brzozowski. 'Derivatives of Regular Expressions'. In: *J. ACM* 11.4 (1964), pp. 481–494. DOI: 10.1145/321239.321249 (cit. on p. 8).

[Bur69] R. M. Burstall. 'Proving Properties of Programs by Structural Induction'. In: *The Computer Journal* 12.1 (1969), pp. 41–48. DOI: 10.1093/comjnl/12.1.41 (cit. on p. 7).

[Caj18] Florian Cajori. 'Origin of the Name "Mathematical Induction"'. In: *The American Mathematical Monthly* 25.5 (1918), pp. 197–201. ISSN: 00029890, 19300972. JSTOR: 2972638 (cit. on p. 7).

[Cap05] Venanzio Capretta. 'General Recursion via Coinductive Types'. In: *Logical Methods in Computer Science* 1.2 (2005). ISSN: 18605974. DOI: 10.2168/LMCS-1(2:1)2005. arXiv: cs/0505037 (cit. on pp. 8, 103, 104).

[Car78] John Cartmell. 'Generalised Algebraic Theories and Contextual Categories'. PhD Thesis. Oxford University, 1978 (cit. on p. 176).

[Car86] John Cartmell. 'Generalised Algebraic Theories and Contextual Categories'. In: *Ann. Pure Appl. Logic* 32 (1986), pp. 209–243. DOI: 10.1016/0168-0072(86)90053-9 (cit. on p. 176).

[Cha+10] James Chapman, Pierre-Évariste Dagand, Conor McBride and Peter Morris. 'The Gentle Art of Levitation'. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. New York, NY, USA: ACM, 2010, pp. 3–14. ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863547 (cit. on p. 181).

[CUV15] James Chapman, Tarmo Uustalu and Niccolò Veltri. 'Quotienting the Delay Monad by Weak Bisimilarity'. In: *ICTAC*. Vol. 9399. LNCS. Springer, 2015, pp. 110–125. DOI: 10.1007/978-3-319-25150-9_8 (cit. on p. 8).

[Chu32] Alonzo Church. 'A Set of Postulates for the Foundation of Logic'. In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X. DOI: 10.2307/1968337. JSTOR: 1968337 (cit. on pp. 168, 170).

[Chu36] Alonzo Church. 'An Unsolvable Problem of Elementary Number Theory'. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. ISSN: 00029327. DOI: 10.2307/2371045. JSTOR: 2371045 (cit. on p. 168).

[Chu40] Alonzo Church. 'A Formulation of the Simple Theory of Types'. In: *J. Symb. Log.* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170 (cit. on p. 168).

[CF07] Horatiu Cirstea and Germain Faure. 'Confluence of Pattern-Based Calculi'. In: *Proceedings of RTA '07*. Term Rewriting and Applications. LNCS. Springer Berlin Heidelberg, 2007, pp. 78–92. ISBN: 978-3-540-73447-5. DOI: 10.1007/978-3-540-73449-9_8 (cit. on pp. 310, 311).

[cLa16] cLab. *Type Theoretic Interpretation of the Final Chain*. 2016. URL: https://coalg.org/clab/Type_Theoretic_Interpretation_of_the_Final_Chain (visited on 14/01/2016) (cit. on pp. 224, 274).

[CH89] Rance Cleaveland and Matthew Hennessy. 'Testing Equivalence as a Bisimulation Equivalence'. In: *Automatic Verification Methods for Finite State Systems*. Vol. 407. Lecture Notes in Computer Science. Springer, 1989, pp. 11–23. DOI: 10.1007/3-540-52148-8_2 (cit. on p. 156).

[Coc01] J. Robin B. Cockett. 'Deforestation, Program Transformation, and Cut-Elimination'. In: *Electr. Notes Theor. Comput. Sci.* 44.1 (2001), pp. 88–127. DOI: 10.1016/S1571-0661(04)80904-6 (cit. on pp. 69, 126).

[CDP16] Jesper Cockx, Dominique Devriese and Frank Piessens. 'Eliminating Dependent Pattern Matching without K'. In: *J. Funct. Program.* 26 (2016). DOI: 10.1017/S0956796816000174 (cit. on p. 218).

[Coh+16] Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg. 'Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom'. In: *CoRR* abs/1611.02108 (2016). URL: http://arxiv.org/abs/1611.02108 (cit. on p. 273).

[Con97] Robert L. Constable. 'The Structure of Nuprl's Type Theory'. In: *Logic of Computation*. NATO ASI Series 157. Springer Berlin Heidelberg, 1997, pp. 123–155. ISBN: 978-3-642-63832-9. DOI: 10.1007/978-3-642-59048-1_4 (cit. on pp. 68, 168, 224).

[Con71] John H. Conway. *Regular Algebra and Finite Machines*. London: Chapman and Hall, 1971. 160 pp. ISBN: 978-0-412-10620-0 (cit. on p. 8).

[Coq12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.4. LogiCal Project, 2012. URL: http://coq.inria.fr (cit. on pp. 9, 168, 224).

[Coq89] Thierry Coquand. *Metamathematical Investigations of a Calculus of Constructions*. INRIA, 1989. URL: https://hal.inria.fr/inria-00075471 (cit. on p. 218).

[Coq93] Thierry Coquand. 'Infinite Objects in Type Theory'. In: *TYPES*. Vol. 806. Lecture Notes in Computer Science. Springer, 1993, pp. 62–78 (cit. on p. 68).

[CH88] Thierry Coquand and Gérard P. Huet. 'The Calculus of Constructions'. In: *Inf. Comput.* 76 (2/3 1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3 (cit. on p. 168).

[CC79] Patrick Cousot and Radhia Cousot. 'Constructive Versions of Tarski's Fixed Point Theorems'. In: *Pacific J. Math.* 82.1 (1979), pp. 43–57. URL: http://projecteuclid.org/euclid.pjm/1102785059 (cit. on pp. 27, 139).

[CH07] Karl Crary and Robert Harper. 'Syntactic Logical Relations for Polymorphic and Recursive Types'. In: *Electr. Notes Theor. Comput. Sci.* 172 (2007), pp. 259–299. DOI: 10.1016/j.entcs.2007.02.010 (cit. on p. 153).

[Cro75] J. N. Crossley. 'Reminiscences of Logicians'. In: *Algebra and Logic: Papers from the 1974 Summer Research Institute of the Australian Mathematical Society, Monash University, Australia*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 1–62. ISBN: 978-3-540-37480-0. DOI: 10.1007/BFb0062850 (cit. on p. 69).

[DM13] P.-E. Dagand and C. McBride. 'A Categorical Treatment of Ornaments'. In: *2013 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS)*. 2013 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS). 2013, pp. 530–539. DOI: 10.1109/LICS.2013.60 (cit. on p. 181).

[DHL06] Christian Dax, Martin Hofmann and Martin Lange. 'A Proof System for the Linear Time μ-Calculus'. In: *Proceedings of FSTTCS 2006*. Vol. 4337. LNCS. Springer, 2006, pp. 273–284. DOI: 10.1007/11944836_26 (cit. on pp. 9, 126).

[dBru66] Nicolaas Govert de Bruijn. 'Verification of Mathematical Proofs by a Computer. A Preparatory Study for a Project Automath'. In: Rob Nederpelt, Herman Geuvers and Roel de Vrijer. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics 133. North-Holland, 1966, pp. 57–72. ISBN: 0-444-89822-0 (cit. on p. 168).

[dBru68] Nicolaas Govert de Bruijn. 'The Mathematical Language Automath, Its Usage, and Some of Its Extensions.' In: Rob Nederpelt, Herman Geuvers and Roel de Vrijer. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics 133. North-Holland, 1968, pp. 73–100. ISBN: 0-444-89822-0 (cit. on pp. 168, 260).

[Ded88] Richard Dedekind. *Was Sind Und Was Sollen Die Zahlen?* 1st ed. Braunschweig: Vieweg, 1888. URL: http://resolver.sub.uni-goettingen.de/purl?PPN23569441X (cit. on p. 7).

[DAB11] Derek Dreyer, Amal Ahmed and Lars Birkedal. 'Logical Step-Indexed Logical Relations'. In: *Logical Methods in Computer Science* 7.2 (2011). DOI: 10.2168/LMCS-7(2:16)2011 (cit. on p. 155).

[Dyb95] Peter Dybjer. 'Internal Type Theory'. In: *Selected Papers of TYPES'95*. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 120–134. DOI: 10.1007/3-540-61780-9_6 (cit. on p. 176).

[EGH08] Jörg Endrullis, Clemens Grabmayer and Dimitri Hendriks. 'Data-Oblivious Stream Productivity'. In: *Proc.Conf.on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2008)*. LNCS. Springer, 2008, pp. 79–96. DOI: 10.1007/978-3-540-89439-1 (cit. on p. 101).

[End+10] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara and Jan Willem Klop. 'Productivity of Stream Definitions'. In: *TCS* 411 (4–5 2010), pp. 765–782. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2009.10.014 (cit. on pp. 84, 101).

[EH11] Jörg Endrullis and Dimitri Hendriks. 'Lazy Productivity via Termination'. In: *TCS* 412.28 (2011), pp. 3203–3225. DOI: 10.1016/j.tcs.2011.03.024 (cit. on pp. 84, 101).

[Fio12] Marcelo Fiore. 'Discrete Generalised Polynomial Functors'. In: *Proceedings of ICALP 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 214–226. ISBN: 978-3-642-31585-5. DOI: 10.1007/978-3-642-31585-5_22 (cit. on p. 217).

[Fle07] Peter Fletcher. 'Infinity'. In: *Philosophy of Logic*. Handbook of the Philosophy of Science. Amsterdam: North-Holland, 2007, pp. 523–585. ISBN: 0-444-51541-0. URL: http://eprints.keele.ac.uk/id/eprint/62 (cit. on p. 12).

[Fre90] Peter J. Freyd. 'Recursive Types Reduced to Inductive Types'. In: *Proceedings of LICS '90*. IEEE Computer Society, 1990, pp. 498–507. DOI: 10.1109/LICS.1990.113772 (cit. on p. 68).

[FGJ11] Clément Fumex, Neil Ghani and Patricia Johann. 'Indexed Induction and Coinduction, Fibrationally'. In: *Proc. of CALCO '11*. CALCO. Vol. 6859. Lecture Notes in Computer Science. Springer, 2011, pp. 176–191. DOI: 10.1007/978-3-642-22944-2_13 (cit. on pp. 164, 187, 199, 206, 210, 217).

[GH04] Nicola Gambino and Martin Hyland. 'Wellfounded Trees and Dependent Polynomial Functors'. In: *Types for Proofs and Programs*. Vol. 3085. Lecture Notes in Computer Science. Springer, 2004, pp. 210–225. DOI: 10.1007/978-3-540-24849-1_14 (cit. on pp. 191, 195, 196, 217).

[GK13] Nicola Gambino and Joachim Kock. 'Polynomial Functors and Polynomial Monads'. In: *Math. Proc. Cambridge Phil. Soc.* 154 (01 2013), pp. 153–192. ISSN: 0305-0041, 1469-8064. DOI: 10.1017/S0305004112000394. arXiv: 0906.4931 (cit. on pp. 164, 188–190, 217).

[Gen35] Gerhard Gentzen. 'Untersuchungen Über Das Logische Schließen. I'. In: *Mathematische Zeitschrift* 39.1 (1935), pp. 176–210. ISSN: 1432-1823. DOI: 10.1007/BF01201353 (cit. on p. 154).

[Geu92] Herman Geuvers. 'Inductive and Coinductive Types with Iteration and Recursion'. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Bastad*. 1992, pp. 193–217 (cit. on pp. 9, 25, 68, 69, 101, 102).

[Geu94] Herman Geuvers. 'A Short and Flexible Proof of Strong Normalization for the Calculus of Constructions'. In: *Types for Proofs and Programs*. LNCS 996. Springer Berlin Heidelberg, 1994, pp. 14–38. ISBN: 978-3-540-60579-9. DOI: 10.1007/3-540-60579-7_2 (cit. on pp. 247, 250).

[GN91] Herman Geuvers and Mark-Jan Nederhof. 'Modular Proof of Strong Normalization for the Calculus of Constructions'. In: *J. Funct. Program.* 1.2 (1991), pp. 155–189 (cit. on p. 168).

[GP07] Herman Geuvers and Erik Poll. 'Iteration and Primitive Recursion in Categorical Terms'. In: *Reflections on Type Theory, λ-Calculus, and the Mind. Essays Dedicated to Henk Barendregt on the Occasion of His 60th Birthday*. Radboud University Nijmegen, 2007, pp. 101–114. ISBN: 978-90-90-22446-6 (cit. on p. 102).

[GFO16] Neil Ghani, Fredrik Nordvall Forsberg and Federico Orsanigo. 'Proof-Relevant Parametricity'. In: *A List of Successes That Can Change the World*. Vol. 9600. Lecture Notes in Computer Science. Springer, 2016, pp. 109–131 (cit. on p. 101).

[GHP09a] Neil Ghani, Peter Hancock and Dirk Pattinson. 'Continuous Functions on Final Coalgebras'. In: *ENTCS* 249 (2009), pp. 3–18. DOI: 10.1016/j.entcs.2009.07.081 (cit. on p. 8).

[GHP09b] Neil Ghani, Peter Hancock and Dirk Pattinson. 'Representations of Stream Processors Using Nested Fixed Points'. In: *LMCS* 5.3 (2009). arXiv: 0905.4813 (cit. on p. 8).

[GJF12] Neil Ghani, Patricia Johann and Clement Fumex. 'Generic Fibrational Induction'. In: *Logical Methods in Computer Science* 8.2 (2012). ISSN: 18605974. DOI: 10.2168/LMCS-8(2:12)2012. arXiv: 1206.0357 (cit. on p. 206).

[Gim95] Eduardo Giménez. 'Codifying Guarded Definitions with Recursive Schemes'. In: *Selected Papers from the TYPES '94 Workshop*. London, UK: Springer-Verlag, 1995, pp. 39–59. ISBN: 3-540-60579-7. DOI: 10.1007/3-540-60579-7_3 (cit. on pp. 51, 68, 76, 125, 223, 224, 264).

[Gim96] Eduardo Giménez. 'Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants'. PhD Thesis. École Normale Supérieure de Lyon, 1996. URL: ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD96-11.ps.Z (cit. on p. 274).

[Gir71] Jean-Yves Girard. 'Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types'. In: *Studies in Logic and the Foundations of Mathematics* 63 (1971), pp. 63–92 (cit. on pp. 9, 168).

[Gir72] Jean-Yves Girard. 'Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur'. PhD. Université Paris VII, 1972 (cit. on pp. 153, 168).

[GTL89] Jean-Yves Girard, Paul Taylor and Yves Lafont. *Proofs and Types*. New York, NY, USA: Cambridge University Press, 1989. ISBN: 0-521-37181-3. URL: http://www.paultaylor.eu/stable/Proofs+Types.html (cit. on p. 176).

[Göd58] Kurt Gödel. 'Über Eine Bisher Noch Nicht Benützte Erweiterung Des Finiten Standpunktes'. In: *Dialectica* 12 (3-4 1958), pp. 280–287. ISSN: 1746-8361. DOI: 10.1111/j.1746-8361.1958.tb01464.x (cit. on p. 168).

[GMM06] Healfdene Goguen, Conor Mcbride and James Mckinna. 'Eliminating Dependent Pattern Matching'. In: *Of Lecture Notes in Computer Science*. Springer, 2006, pp. 521–540 (cit. on pp. 218, 259, 264).

[Gol84] Robert Goldblatt. *Topoi: The Categorial Analysis of Logic*. Vol. 98. Studies in Logic and the Foundations of Mathematics. Amsterdam: Elsevier, 1984. 551 pp. ISBN: 0-444-86711-2. URL: http://projecteuclid.org/euclid.bia/1403013939 (cit. on p. 101).

[Gol01] Robert Goldblatt. 'A Calculus of Terms for Coalgebras of Polynomial Functors'. In: *Electr. Notes Theor. Comput. Sci.* 44.1 (2001), pp. 161–184. DOI: 10.1016/S1571-0661(04)80907-1 (cit. on p. 217).

[GMS14] Sergey Goncharov, Stefan Milius and Alexandra Silva. 'Towards a Coalgebraic Chomsky Hierarchy'. In: *Theoretical Computer Science*. LNCS 8705. Springer, 2014, pp. 265–280. ISBN: 978-3-662-44601-0. DOI: 10.1007/978-3-662-44602-7_21 (cit. on p. 8).

[Gor95] Andrew D. Gordon. 'Bisimilarity as a Theory of Functional Programming'. In: *Electronic Notes in Theoretical Computer Science*. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference 1 (1995), pp. 232–252. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80013-6 (cit. on p. 102).

[Gor+11] Eugen-Ioan Goriac, Georgiana Caltais, Dorel Lucanu and Grigore Roşu. *CIRC Tutorial and User Manual Info*. 2011. URL: http://fsl.cs.illinois.edu/images/3/3b/CIRC_Tutorial.pdf (cit. on p. 154).

[Gre92] John Greiner. *Programming with Inductive and Co-Inductive Types*. Pittsburgh, PA, USA: Carnegie Mellon University, 1992. URL: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA249562 (cit. on pp. 9, 39, 68).

[GV92] Jan Friso Groote and Frits W. Vaandrager. 'Structured Operational Semantics and Bisimulation as a Congruence'. In: *Inf. Comput.* 100.2 (1992), pp. 202–260. DOI: 10.1016/0890-5401(92)90013-6 (cit. on p. 158).

[GS00] H. Peter Gumm and Tobias Schröder. 'Coalgebraic Structure from Weak Limit Preserving Functors'. In: *Electr. Notes Theor. Comput. Sci.* 33 (2000), pp. 111–131. DOI: 10.1016/S1571-0661(05)80346-9 (cit. on p. 142).

[GS86] Yuri Gurevich and Saharon Shelah. 'Fixed-Point Extensions of First-Order Logic'. In: *Annals of pure and applied logic* 32 (1986), pp. 265–280 (cit. on p. 163).

[Hag87] Tatsuya Hagino. 'A Typed Lambda Calculus with Categorical Type Constructors'. In: *Category Theory in Computer Science*. Lecture Notes in Computer Science. Springer, 1987, pp. 140–157. DOI: 10.1007/3-540-18508-9_24 (cit. on pp. 9, 15, 17, 25, 68, 164, 177, 178, 216, 217, 272).

[HF11] Makoto Hamana and Marcelo Fiore. 'A Foundation for GADTs and Inductive Families: Dependent Polynomial Functor Approach'. In: *Proceedings of the Seventh WGP*. WGP '11. New York, NY, USA: ACM, 2011, pp. 59–70. ISBN: 978-1-4503-0861-8. DOI: 10.1145/2036918.2036927 (cit. on pp. 181, 217, 227).

[HDL90] F. Keith Hanna, Neil Daeche and Mark Longley. 'Specification and Verification Using Dependent Types'. In: *IEEE Trans. Software Eng.* 16.9 (1990), pp. 949–964. DOI: 10.1109/32.58783 (cit. on p. 167).

[HKR14] Helle Hvid Hansen, Clemens Kupke and Jan Rutten. *Stream Differential Equations: Specification Formats and Solution Methods*. Technical Report FM-1404. CWI Amsterdam, 2014 (cit. on p. 101).

[HKR17] Helle Hvid Hansen, Clemens Kupke and Jan Rutten. 'Stream Differential Equations: Specification Formats and Solution Methods'. In: *LMCS* 13.1 (2017). DOI: 10.23638/LMCS-13(1:3)2017. arXiv: 1609.08367 (cit. on pp. 8, 56, 68, 76, 84).

[Han+14] Helle Hvid Hansen, Clemens Kupke, Jan Rutten and Joost Winter. 'A Final Coalgebra for K-Regular Sequences'. In: *Horizons of the Mind. A Tribute to Prakash Panangaden - Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday*. Vol. 8464. LNCS. Springer, 2014, pp. 363–383. DOI: 10.1007/978-3-319-06880-0_19 (cit. on p. 8).

[Has94] Ryu Hasegawa. 'Categorical Data Types in Parametric Polymorphism'. In: *Mathematical Structures in Computer Science* 4 (01 1994), pp. 71–109. DOI: 10.1017/S0960129500000372 (cit. on p. 101).

[Has+13] Ichiro Hasuo, Kenta Cho, Toshiki Kataoka and Bart Jacobs. 'Coinductive Predicates and Final Sequences in a Fibration'. In: *Electronic Notes in Theoretical Computer Science* 298 (2013), pp. 197–214. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2013.09.014 (cit. on pp. 27, 154).

[Hay85] Susumu Hayashi. 'Adjunction of Semifunctors: Categorical Structures in Nonextensional Lambda Calculus'. In: *Theor. Comput. Sci.* 41 (1985), pp. 95–104. DOI: 10.1016/0304-3975(85)90062-3 (cit. on pp. 92, 105).

[Her05] Hugo Herbelin. 'On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic'. In: *Proceedings of TLCA 2005*. Vol. 3461. Lecture Notes in Computer Science. Springer, 2005, pp. 209–220. DOI: 10.1007/11417170_16 (cit. on p. 219).

[HJ97] Claudio Hermida and Bart Jacobs. 'Structural Induction and Coinduction in a Fibrational Setting'. In: *Information and Computation* 145 (1997), pp. 107–152. DOI: 10.1006/inco.1998.2725 (cit. on pp. 10, 27, 146, 154, 164, 184, 199, 209, 210, 217).

[Hof97] Martin Hofmann. 'Syntax and Semantics of Dependent Types'. In: *Semantics and Logics of Computation*. Cambridge University Press, 1997, pp. 79–130 (cit. on p. 176).

[HS94] Martin Hofmann and Thomas Streicher. 'The Groupoid Model Refutes Uniqueness of Identity Proofs'. In: *Proceedings of LICS '94*. IEEE Computer Society, 1994, pp. 208–212. DOI: 10.1109/LICS.1994.316071 (cit. on p. 275).

[How92] Brian T. Howard. 'Fixed Points and Extensionality in Typed Functional Programming Languages'. Published as Stanford Computer Science Department Technical Report STAN-CS-92-1455. Stanford, CA, USA: Stanford University / Stanford University, 1992 (cit. on pp. 68, 69).

[How95] Brian T. Howard. *Lemon: A Functional Language with Inductive and Coinductive Types*. 1995. URL: http://people.cs.ksu.edu/~bhoward/lemon.html (cit. on p. 68).

[How96a] Brian T. Howard. 'Inductive, Coinductive, and Pointed Types'. In: *Proceedings of ICFP '96*. ACM, 1996, pp. 102–109. DOI: 10.1145/232627.232640 (cit. on pp. 9, 68, 102).

[How89] Douglas J. Howe. 'Equality in Lazy Computation Systems'. In: *, Fourth Annual Symposium on Logic in Computer Science, 1989. LICS '89, Proceedings.* , Fourth Annual Symposium on Logic in Computer Science, 1989. LICS '89, Proceedings. 1989, pp. 198–203. DOI: 10.1109/LICS.1989.39174 (cit. on p. 102).

[How96b] Douglas J. Howe. 'Proving Congruence of Bisimulation in Functional Programming Languages'. In: *Information and Computation* 124 (1996), pp. 103–112 (cit. on p. 102).

[Hur+13] Chung-Kil Hur, Georg Neis, Derek Dreyer and Viktor Vafeiadis. 'The Power of Parameterization in Coinductive Proof'. In: *Proceedings of POPL 2013*. POPL '13. New York, NY, USA: ACM, 2013, pp. 193–206. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429093 (cit. on pp. 125, 126).

[Idr17] The Idris Development Team. *Idris – A Language with Dependent Types*. 2017. URL: https://www.idris-lang.org/ (visited on 23/04/2017) (cit. on p. 167).

[Jac99] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999 (cit. on pp. 24, 25, 102, 105, 129, 145, 166, 176, 184, 187, 189, 190, 192, 200, 204, 206, 217, 220, 241, 272).

[Jac91] Bart Jacobs. 'Categorical Type Theory'. PhD Thesis. University of Nijmegen, 1991. URL: http://www.cs.ru.nl/B.Jacobs/PAPERS/PhD.ps (cit. on p. 204).

[Jac93] Bart Jacobs. 'Comprehension Categories and the Semantics of Type Dependency'. In: *Theor. Comput. Sci.* 107.2 (1993), pp. 169–207. DOI: 10.1016/0304-3975(93)90169-T (cit. on pp. 176, 217).

[Jac01] Bart Jacobs. 'Many-Sorted Coalgebraic Modal Logic: A Model-Theoretic Study'. In: *RAIRO - Theoretical Informatics and Applications* 35 (01 2001), pp. 31–59. ISSN: 1290-385X. DOI: 10.1051/ita:2001108 (cit. on p. 73).

[Jac16] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science 59. Cambridge University Press, 2016. ISBN: 978-1-107-17789-5. DOI: 10.1017/CBO9781316823187 (cit. on pp. 7, 25, 217).

[JR97] Bart Jacobs and Jan Rutten. 'A Tutorial on (Co)Algebras and (Co)Induction'. In: *EATCS Bulletin* 62 (1997), pp. 62–222 (cit. on p. 7).

[JR11] Bart Jacobs and Jan Rutten. 'An Introduction to (Co)Algebras and (Co)Induction'. In: *Advanced Topics in Bisimulation and Coinduction*. Vol. 52. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011 (cit. on pp. 7, 10, 25).

[Joh02] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Vol. 1. Oxford Logic Guides. Oxford University Press, 2002. ISBN: 978-0-19-853425-9 (cit. on p. 22).

[KLN04] Fairouz D. Kamareddine, Twan Laan and Rob Nederpelt. 'Pure Type Systems'. In: *A Modern Perspective on Type Theory: From Its Origins until Today*. Applied Logic Series 29. Springer, 2004, p. 116. ISBN: 1-4020-2334-0 (cit. on p. 168).

[KL16] Chris Kapulkin and Peter LeFanu Lumsdaine. 'The Simplicial Model of Univalent Foundations (after Voevodsky)'. In: *arXiv:1211.2851 [math]* (2016). arXiv: 1211.2851 (cit. on p. 176).

[Kim10] Jiho Kim. 'Higher-Order Algebras and Coalgebras from Parameterized Endofunctors'. In: *Electronic Notes in Theoretical Computer Science*. Electronic Notes in Theoretical Computer Science 264.2 (2010), pp. 141–154. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2010.07.018 (cit. on pp. 9, 98, 179).

[Kli07] Bartek Klin. 'Coalgebraic Modal Logic Beyond Sets'. In: *Electr. Notes Theor. Comput. Sci.* 173 (2007), pp. 177–201. DOI: 10.1016/j.entcs.2007.02.034 (cit. on p. 158).

[Kli11] Bartek Klin. 'Bialgebras for Structural Operational Semantics: An Introduction'. In: *TCS* 412.38 (2011), pp. 5043–5069. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2011.03.023 (cit. on pp. 8, 101, 158).

[KN14] Bartek Klin and Beata Nachyla. 'Distributive Laws and Decidable Properties of SOS Specifications'. In: *Proceedings EXPRESS 2014 and SOS 2014*. Vol. 160. EPTCS. 2014, pp. 79–93. DOI: 10.4204/EPTCS.160.8 (cit. on p. 158).

[Klo92] Jan Willem Klop. *Term Rewriting Systems*. 1992 (cit. on p. 19).

[KI13] Naoki Kobayashi and Atsushi Igarashi. 'Model-Checking Higher-Order Programs with Recursive Types'. In: *Proceedings of ESOP 2013*. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 431–450. DOI: 10.1007/978-3-642-37036-6_24 (cit. on p. 157).

[Koc11] Joachim Kock. 'Polynomial Functors and Trees'. In: *International Mathematics Research Notices* 2011.3 (2011), pp. 609–673. DOI: 10.1093/imrn/rnq068. arXiv: 0807.2874 (cit. on p. 217).

[KL17] Ekaterina Komendantskaya and Yue Li. 'Productive Corecursion in Logic Programming'. In: *TPLP* 17 (5-6 2017), pp. 906–923. DOI: 10.1017/S147106841700028X (cit. on p. 8).

[KP11] Ekaterina Komendantskaya and John Power. 'Coalgebraic Derivations in Logic Programming'. In: *CSL*. Vol. 12. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 352–366. DOI: 10.4230/LIPIcs.CSL.2011.352 (cit. on p. 8).

[Koz83] Dexter Kozen. 'Results on the Propositional μ-Calculus'. In: *Theor. Comput. Sci.* 27 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6 (cit. on p. 9).

[Kup06] Clemens Kupke. 'Finitary Coalgebraic Logics'. PhD Thesis. Amsterdam: Institute for Logic, Language and Computation, 2006. URL: http://www.illc.uva.nl/Publications/Dissertations/DS-2006-03.text.pdf (cit. on pp. 73, 106).

[KNR11] Clemens Kupke, Milad Niqui and Jan Rutten. *Stream Differential Equations: Concrete Formats for Coinductive Definitions*. To appear as a book chapter RR-11-10. University of Oxford, 2011 (cit. on p. 8).

[KR08] Clemens Kupke and Jan Rutten. 'Observational Coalgebras and Complete Sets of Co-Operations'. In: *Electron. Notes Theor. Comput. Sci.* 203.5 (2008), pp. 153–174. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.05.024 (cit. on pp. 8, 68, 69).

[KR12] Clemens Kupke and Jan Rutten. 'On the Final Coalgebra of Automatic Sequences'. In: *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*. Vol. 7230. LNCS. Springer, 2012, pp. 149–164. DOI: 10.1007/978-3-642-29485-3_10 (cit. on p. 8).

[Lac10] Stephen Lack. 'A 2-Categories Companion'. In: John C. Baez and J. Peter May. *Towards Higher Categories*. Vol. 152. The IMA Volumes in Mathematics and its Applications. New York, NY: Springer New York, 2010. ISBN: 978-1-4419-1523-8. DOI: 10.1007/978-1-4419-1524-5_4. arXiv: math/0702535 (cit. on pp. 31, 106).

[LS88] Joachim Lambek and Philip J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, 1988. 308 pp. ISBN: 978-0-521-35653-4 (cit. on pp. 12, 20, 101, 102, 104, 145, 172, 173, 176, 272).

[LJB01] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram. 'The Size-Change Principle for Program Termination'. In: *Conference Record of POPL 2001*. ACM, 2001, pp. 81–92. DOI: 10.1145/360204.360210 (cit. on p. 126).

[Lei04] Tom Leinster. *Higher Operads, Higher Categories*. Vol. 298. Cambridge University Press, 2004. ISBN: 0-521-53215-9. arXiv: math.CT/0305049 (cit. on pp. 29, 36).

[Lei89] Daniel Leivant. *Contracting Proofs to Programs*. CMU-CS-89-170. Carnegie Mellon University, 1989 (cit. on p. 69).

[Luo89] Zhaohui Luo. 'ECC, an Extended Calculus of Constructions'. In: *Proceedings of LICS '89*. IEEE Computer Society, 1989, pp. 386–395. DOI: 10.1109/LICS.1989.39193 (cit. on p. 168).

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. 2nd ed. Graduate Texts in Mathematics 5. Springer, 1998. ISBN: 0-387-98403-8 (cit. on p. 21).

[Mar75a] Per Martin-Löf. 'About Models for Intuitionistic Type Theories and the Notion of Definitional Equality'. In: *3rd Scandinavian Logic Symposium*. Scandinavian Logic Symposium. North Holland and American Elsevier, 1975, pp. 81–109 (cit. on pp. 128, 223).

[Mar75b] Per Martin-Löf. 'An Intuitionistic Theory of Types: Predicative Part'. In: *Proceedings of the Logic Colloquium '73*. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1 (cit. on pp. 168, 176).

[Mat99] Ralph Matthes. 'Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types'. PhD. Ludwig Maximilian University of Munich, Germany, 1999. URL: http://d-nb.info/956895891 (cit. on pp. 9, 39, 68, 69).

[Men87] Nex Paul Mendler. 'Recursive Types and Type Constraints in Second-Order Lambda Calculus'. In: *Proceedings of LICS '87*. IEEE Computer Society, 1987, pp. 30–36 (cit. on pp. 9, 68).

[Men91] Nex Paul Mendler. 'Inductive Types and Type Constraints in the Second-Order Lambda Calculus'. In: *Ann. Pure Appl. Logic* 51 (1-2 1991), pp. 159–172. DOI: 10.1016/0168-0072(91)90069-X (cit. on pp. 9, 39, 68).

[Mil77] Robin Milner. 'Fully Abstract Models of Typed λ-Calculi'. In: *Theoretical Computer Science* 4.1 (1977), pp. 1–22. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90053-6 (cit. on p. 102).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3 (cit. on p. 7).

[Mil89] Robin Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989 (cit. on pp. 8, 28).

[MP00] Ieke Moerdijk and Erik Palmgren. 'Wellfounded Trees in Categories'. In: *Annals of Pure and Applied Logic* 104 (1–3 2000), pp. 189–218. ISSN: 0168-0072. DOI: 10.1016/S0168-0072(00)00012-9 (cit. on p. 191).

[Møg14] Rasmus Ejlers Møgelberg. 'A Type Theory for Productive Coprogramming via Guarded Recursion'. In: *CSL-LICS*. ACM, 2014, 71:1–71:10. DOI: 10.1145/2603088.2603132 (cit. on pp. 68, 126, 155).

[Mog89] Eugenio Moggi. 'A Category-Theoretic Account of Program Modules'. In: *Proceedings of Category Theory and Computer Science*. Vol. 389. LNCS. Springer, 1989, pp. 101–117 (cit. on p. 176).

[Nak00] Hiroshi Nakano. 'A Modality for Recursion'. In: *LICS*. IEEE Computer Society, 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774 (cit. on pp. 107, 126, 154).

[NU10] Keiko Nakata and Tarmo Uustalu. 'Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction'. In: *Electronic Proceedings in Theoretical Computer Science* 32 (2010), pp. 57–75. ISSN: 2075-2180. DOI: 10.4204/EPTCS.32.5. arXiv: 1008.2112 (cit. on p. 8).

[NUB11] Keiko Nakata, Tarmo Uustalu and Marc Bezem. 'A Proof Pearl with the Fan Theorem and Bar Induction - Walking through Infinite Trees with Mixed Induction and Coinduction'. In: *Proceedings of APLAS 2011*. Vol. 7078. LNCS. Springer, 2011, pp. 353–368. DOI: 10.1007/978-3-642-25318-8_26 (cit. on pp. 8, 277).

[NPP08] Aleksandar Nanevski, Frank Pfenning and Brigitte Pientka. 'Contextual Modal Type Theory'. In: *ACM Trans. Comput. Log.* 9.3 (2008) (cit. on p. 274).

[NGdV94] Rob Nederpelt, Herman Geuvers and Roel de Vrijer. *Selected Papers on Automath*. Studies in Logic and the Foundations of Mathematics 133. North-Holland, 1994. ISBN: 0-444-89822-0 (cit. on pp. 168, 176).

[NG14] Rob Nederpelt and Professor Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. 1st. New York, NY, USA: Cambridge University Press, 2014. ISBN: 1-107-03650-X (cit. on pp. 168, 176, 182, 219).

[NR11] Milad Niqui and Jan Rutten. 'A Proof of Moessner's Theorem by Coinduction'. In: *Higher-Order and Symbolic Computation* 24.3 (2011), pp. 191–206 (cit. on p. 43).

[NW96] Damian Niwinski and Igor Walukiewicz. 'Games for the μ-Calculus'. In: *Theor. Comput. Sci.* 163 (1&2 1996), pp. 99–116. DOI: 10.1016/0304-3975(95)00136-0 (cit. on pp. 9, 126).

[nLa16] nLab. *Principle of Equivalence*. nLab. 2016. URL: https://ncatlab.org/nlab/show/principle+of+equivalence (visited on 31/08/2016) (cit. on p. 90).

[nLa17] nLab. *Categorical Models of Dependent Types*. In: 2017. URL: https://ncatlab.org/nlab/show/categorical+model+of+dependent+types (visited on 23/04/2017) (cit. on p. 176).

[Nor07] Ulf Norell. 'Towards a Practical Programming Language Based on Dependent Type Theory'. PhD thesis. Göteborg, Sweden: Chalmers University of Technology, 2007 (cit. on pp. 168, 271).

[Ong15] Luke Ong. 'Higher-Order Model Checking: An Overview'. In: *LICS 2015*. IEEE Computer Society, 2015, pp. 1–15. DOI: 10.1109/LICS.2015.9 (cit. on p. 157).

[Par79] David Michael Ritchie Park. 'On the Semantics of Fair Parallelism'. In: *Proceedings of Abstract Software Specifications, 1979 Copenhagen Winter School*. Vol. 86. Lecture Notes in Computer Science. Springer, 1979, pp. 504–526. DOI: 10.1007/3-540-10007-5_47 (cit. on p. 9).

[Par81] David Michael Ritchie Park. 'Concurrency and Automata on Infinite Sequences'. In: *Proceedings of TCS'81*. Vol. 104. LNCS. Springer, 1981, pp. 167–183. DOI: 10.1007/BFb0017309 (cit. on p. 7).

[PSW76] David Lorge Parnas, John E. Shore and David M. Weiss. 'Abstract Types Defined as Classes of Variables'. In: *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition and Structure*. ACM, 1976, pp. 149–153. DOI: 10.1145/942574.807133 (cit. on p. 176).

[Pau93] Christine Paulin-Mohring. 'Inductive Definitions in the System Coq - Rules and Properties'. In: *International Conference on Typed Lambda Calculi and Applications, TLCA, Proceedings*. Vol. 664. LNCS. Springer, 1993, pp. 328–345 (cit. on p. 223).

[Pau15] Christine Paulin-Mohring. 'Introduction to the Calculus of Inductive Constructions'. In: *All about Proofs, Proofs for All*. Studies in Logic (Mathematical Logic and Foundations) 55. College Publications, 2015. ISBN: 978-1-84890-166-7. URL: https://hal.inria.fr/hal-01094195/file/CIC.pdf (cit. on p. 168).

[PE98] Dusko Pavlovic and Marłın Hötzel Escardó. 'Calculus in Coinductive Form'. In: *Proceedings LICS 1998*. IEEE Computer Society, 1998, pp. 408–417. DOI: 10.1109/LICS.1998.705675 (cit. on p. 8).

[Pea89] Guiseppe Peano. *Arithmetices Principia: Nova Methodo Exposita*. Fratres Bocca, 1889. URL: https://archive.org/details/arithmeticespri00peangoog (cit. on p. 7).

[Pét61] Rózsa Péter. 'Über Die Verallgemeinerung Der Theorie Der Rekursiven Funktionen Für Abstrakte Mengen Geeigneter Struktur Als Definitionsbereiche'. In: *Acta Mathematica Hungarica* 12 (3-4 1961), pp. 271–314. DOI: 10.1007/BF02023919 (cit. on p. 7).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN: 0-262-16209-1 (cit. on p. 166).

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004. ISBN: 0-262-16228-8 (cit. on p. 176).

[Pit04] Andrew Pitts. 'Typed Operational Reasoning'. In: Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004, pp. 245–289. ISBN: 0-262-16228-8 (cit. on p. 88).

[Pit00] Andrew M. Pitts. 'Parametric Polymorphism and Operational Equivalence'. In: *Mathematical Structures in Computer Science* 10 (03 2000), pp. 321–359 (cit. on pp. 102, 153).

[Pit01] Andrew M. Pitts. 'Categorical Logic'. In: Samson Abramsky, Dov M. Gabbay and Thomas S. E. Maibaum. *Handbook of Logic in Computer Science: Algebraic and Logical Structures*. Vol. 5. Oxford University Press, 2001, pp. 39–128. ISBN: 978-0-19-853781-6 (cit. on p. 176).

[PS15] João Paulo Pizani Flor and Wouter Swierstra. 'Π-Ware: An Embedded Hardware Description Language Using Dependent Types'. In: *Extended Abstracts for International Conference on Types for Proofs and Programs (TYPES)*. TYPES'15. 2015. ISBN: 978-9949-430-86-4. URL: http://cs.ioc.ee/types15/abstracts-book/ (cit. on p. 167).

[Plo77] Gordon D. Plotkin. 'LCF Considered as a Programming Language'. In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90044-5 (cit. on p. 102).

[PA93] Gordon D. Plotkin and Martín Abadi. 'A Logic for Parametric Polymorphism'. In: *Typed Lambda Calculi and Applications*. International Conference on Typed Lambda Calculi and Applications TLCA '93. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 361–375. ISBN: 978-3-540-56517-8. DOI: 10.1007/BFb0037118 (cit. on p. 101).

[Pnu77] Amir Pnueli. 'The Temporal Logic of Programs'. In: *18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on p. 9).

[Pou07] Damien Pous. 'Complete Lattices and Up-To Techniques'. In: *Proceedings of APLAS 2007*. Vol. 4807. LNCS. Springer, 2007, pp. 351–366. DOI: 10.1007/978-3-540-76637-7_24 (cit. on pp. 8, 28, 154).

[Pou16] Damien Pous. 'Coinduction All the Way Up'. In: *Proceedings of LICS '16*. ACM, 2016, pp. 307–316. DOI: 10.1145/2933575.2934564 (cit. on pp. 29, 154).

[PR17] Damien Pous and Jurriaan Rot. 'Companions, Codensity, and Causality'. In: *Proceedings of FOSSACS 2017*. 2017. DOI: 10.1007/978-3-662-54458-7_7 (cit. on pp. 29, 154).

[PS11] Damien Pous and Davide Sangiorgi. 'Enhancements of the Coinductive Proof Method'. In: *Advanced Topics in Bisimulation and Coinduction*. New York, NY, USA: Cambridge University Press, 2011 (cit. on pp. 28, 154).

[Rey74] John C. Reynolds. 'Towards a Theory of Type Structure'. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. Vol. 19. LNCS. Springer, 1974, pp. 408–423. DOI: 10.1007/3-540-06859-7_148 (cit. on p. 153).

[RL09] Grigore Roşu and Dorel Lucanu. 'Circular Coinduction: A Proof Theoretical Foundation'. In: *CALCO*. Vol. 5728. LNCS. Springer, 2009, pp. 127–144. DOI: 10.1007/978-3-642-03741-2_10 (cit. on pp. 126, 154).

[Rot15] Jurriaan Rot. 'Enhanced Coinduction'. PhD. Leiden: University Leiden, 2015 (cit. on pp. 8, 28).

[Rot+17] Jurriaan Rot, Filippo Bonchi, Marcello Bonsangue, Damien Pous, Jan Rutten and Alexandra Silva. 'Enhanced Coalgebraic Bisimulation'. In: *MSCS* 27.7 (2017), pp. 1236–1264. DOI: 10.1017/S0960129515000523 (cit. on pp. 28, 154).

[RB17] Reuben N. S. Rowe and James Brotherston. 'Automatic Cyclic Termination Proofs for Recursive Procedures in Separation Logic'. In: *Proceedings of CPP 2017*. ACM, 2017, pp. 53–65. DOI: 10.1145/3018610.3018623 (cit. on p. 126).

[Rut99] Jan Rutten. 'A Note on Coinduction and Weak Bisimilarity for While Programs'. In: *ITA* 33 (4/5 1999), pp. 393–400. DOI: 10.1051/ita:1999125 (cit. on p. 8).

[Rut00] Jan Rutten. 'Universal Coalgebra: A Theory of Systems'. In: *Theor. Comput. Sci.* 249.1 (2000), pp. 3–80. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00056-6 (cit. on pp. 7, 10, 25, 154).

[Rut03] Jan Rutten. 'Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series'. In: *TCS*. Theor. Comput. Sci. 308 (1-3 2003), pp. 1–53. DOI: 10.1016/S0304-3975(02)00895-2 (cit. on pp. 1, 8, 49, 68).

[Rut05] Jan Rutten. 'A Coinductive Calculus of Streams'. In: *Mathematical Structures in Computer Science* 15.1 (2005), pp. 93–147. DOI: 10.1017/S0960129504004517 (cit. on pp. 1, 8).

[Sac13] Jorge Luis Sacchini. 'Type-Based Productivity of Stream Definitions in the Calculus of Constructions'. In: *2013 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS)*. 2013 28th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS). 2013, pp. 233–242. DOI: 10.1109/LICS.2013.29 (cit. on pp. 68, 223, 272).

[SV82] Giovanni Sambin and Silvio Valentini. 'The Modal Logic of Provability. The Sequential Approach'. In: *Journal of Philosophical Logic* 11.3 (1982), pp. 311–342. ISSN: 1573-0433. DOI: 10.1007/BF00293433 (cit. on p. 154).

[San98] Davide Sangiorgi. 'On the Bisimulation Proof Method'. In: *Mathematical Structures in Computer Science* 8.5 (1998), pp. 447–479 (cit. on pp. 8, 28).

[San09] Davide Sangiorgi. 'On the Origins of Bisimulation and Coinduction'. In: *ACM Trans. Program. Lang. Syst.* 31.4 (2009), 15:1–15:41. DOI: 10.1145/1516507.1516510 (cit. on pp. 7, 139).

[San11] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. New York, NY, USA: Cambridge University Press, 2011. ISBN: 978-1-107-00363-7 (cit. on pp. 8, 10, 16, 28, 73, 154, 157).

[SKS11] Davide Sangiorgi, Naoki Kobayashi and Eijiro Sumii. 'Environmental Bisimulations for Higher-Order Languages'. In: *ACM Trans. Program. Lang. Syst.* 33.1 (2011), 5:1–5:69. DOI: 10.1145/1889997.1890002 (cit. on pp. 153, 156).

[San02a] Luigi Santocanale. 'A Calculus of Circular Proofs and Its Categorical Semantics'. In: *FoSSaCS*. 2002, pp. 357–371. DOI: 10.1007/3-540-45931-6_25 (cit. on pp. 69, 126).

[San02b] Luigi Santocanale. 'μ-Bicomplete Categories and Parity Games'. In: 36.2 (2002), pp. 195–227. URL: https://doi.org/10.1051/ita:2002010 (cit. on pp. 9, 69, 182).

[SR10] Alexandra Silva and Jan Rutten. 'A Coinductive Calculus of Binary Trees'. In: *Inf. Comput.* 208.5 (2010), pp. 578–593. DOI: 10.1016/j.ic.2008.08.006 (cit. on p. 8).

[Sim17] Alex Simpson. 'Cyclic Arithmetic Is Equivalent to Peano Arithmetic'. In: *Proceedings of FoSSaCS'17*. FoSSaCS. LNCS. 2017. DOI: 10.1007/978-3-662-54458-7_17 (cit. on p. 126).

[Smo85] Craig Smoryński. *Self-Reference and Modal Logic*. Universitext. Springer-Verlag, 1985. ISBN: 0-387-96209-3 (cit. on pp. 154, 159).

[Smy92] Michael B. Smyth. 'Topology'. In: Samson Abramsky and Thomas S. E. Maibaum. *Handbook of Logic in Computer Science: Background: Mathematical Structures*. Vol. 1. New York, NY, USA: Oxford University Press, Inc., 1992, pp. 641–761. ISBN: 0-19-853735-2. URL: http://dl.acm.org/citation.cfm?id=162573.162536 (cit. on p. 58).

[Sol76] Robert M. Solovay. 'Provability Interpretations of Modal Logic'. In: *Israel Journal of Mathematics* 25.3 (1976), pp. 287–304. ISSN: 1565-8511. DOI: 10.1007/BF02757006 (cit. on pp. 126, 130, 154, 159).

[SD03] Christoph Sprenger and Mads Dam. 'On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus'. In: *Proceedings of FOSSACS 2003*. Vol. 2620. LNCS. Springer, 2003, pp. 425–440. DOI: 10.1007/3-540-36576-1_27 (cit. on p. 126).

[Sta08] Sam Staton. 'General Structural Operational Semantics through Categorical Logic'. In: *Proceedings of LICS 2008*. IEEE Computer Society, 2008, pp. 166–177. DOI: 10.1109/LICS.2008.43 (cit. on p. 158).

[Sta11] Sam Staton. 'Relating Coalgebraic Notions of Bisimulation'. In: *Logical Methods in Computer Science* 7.1 (2011), pp. 1–21. DOI: 10.2168/LMCS-7(1:13)2011 (cit. on pp. 10, 27, 114, 154, 221).

[Str89] Thomas Streicher. 'Independence Results for Calculi of Dependent Types'. In: *Proceedings of Category Theory and Computer Science*. Vol. 389. Lecture Notes in Computer Science. Springer, 1989, pp. 141–154. DOI: 10.1007/BFb0018350 (cit. on p. 219).

[Str91] Thomas Streicher. *Semantics of Type Theory - Correctness, Completeness and Independence Results*. Progress in Theoretical Computer Science. Birkhäuser Basel, 1991. XII, 299. ISBN: 978-1-4612-6757-7. DOI: 10.1007/978-1-4612-0433-6 (cit. on pp. 176, 273).

[Stu08] Thomas Studer. 'On the Proof Theory of the Modal Mu-Calculus'. In: *Studia Logica* 89.3 (2008), pp. 343–363. DOI: 10.1007/s11225-008-9133-6 (cit. on p. 126).

[SP07] Eijiro Sumii and Benjamin C. Pierce. 'A Bisimulation for Type Abstraction and Recursion'. In: *J. ACM* 54.5 (2007), p. 26. DOI: 10.1145/1284320.1284325 (cit. on pp. 153, 156).

[Tay99] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics 59. Cambridge University Press, 1999. 588 pp. ISBN: 978-0-521-63107-5. URL: http://paultaylor.eu/prafm/ (cit. on p. 176).

[Tro11] Anne Sjerp Troelstra. 'History of Constructivism in the 20th Century'. In: *Set Theory, Arithmetic, and Foundations of Mathematics*. Cambridge University Press, 2011, pp. 150–179. ISBN: 978-0-511-91061-6. URL: http://dx.doi.org/10.1017/CBO9780511910616.009 (cit. on p. 168).

[TvD88] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics: An Introduction*. Vol. 1 & 2. Studies in Logic and the Foundations of Mathematics 121 & 123. North-Holland, 1988. 384 pp. ISBN: 978-0-444-70266-1 (cit. on pp. 8, 154, 160, 168, 219, 240).

[TP97] Daniele Turi and Gordon D. Plotkin. 'Towards a Mathematical Operational Semantics'. In: *Proceedings LICS'97*. IEEE Computer Society Press, 1997, pp. 280–291. DOI: 10.1109/LICS.1997.614955 (cit. on pp. 8, 101, 158).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: http://homotopytypetheory.org/book (cit. on pp. 155, 273).

[UV96] Tarmo Uustalu and Varmo Vene. 'A Cube of Proof Systems for the Intuitionistic Predicate μ,ν-Logic'. In: *Selected Papers from the 8th Nordic Workshop on Programming Theory, NWPT*. Vol. 96. 1996, pp. 237–246 (cit. on pp. 39, 68).

[UV99a] Tarmo Uustalu and Varmo Vene. 'Mendler-Style Inductive Types, Categorically'. In: *Nordic J. of Computing* 6.3 (1999), pp. 343–361. ISSN: 1236-6064. URL: http://dl.acm.org/citation.cfm?id=774455.774462 (cit. on pp. 68, 102).

[UV99b] Tarmo Uustalu and Varmo Vene. 'Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically'. In: *Informatica* 10 (1999), pp. 5–26 (cit. on p. 9).

[UV99c] Tarmo Uustalu and Varmo Vene. 'Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically'. In: *Informatica, Lith. Acad. Sci.* 10.1 (1999), pp. 5–26 (cit. on p. 69).

[UV02] Tarmo Uustalu and Varmo Vene. 'Least and Greatest Fixed Points in Intuitionistic Natural Deduction'. In: *Theor. Comput. Sci.* 272 (1-2 2002), pp. 315–339. DOI: 10.1016/S0304-3975(00)00355-8 (cit. on pp. 68, 69).

[vdBer06] Benno van den Berg. 'Predicative Topos Theory and Models for Constructive Set Theory'. PhD. University of Utrecht, 2006 (cit. on p. 272).

[vdBdM04] Benno van den Berg and Federico de Marchi. 'Non-Well-Founded Trees in Categories'. In: (2004). arXiv: math/0409158 (cit. on pp. 9, 191).

[vdBdM07] Benno van den Berg and Federico de Marchi. 'Non-Well-Founded Trees in Categories'. In: *Annals of Pure and Applied Logic* 146.1 (2007), pp. 40–59. ISSN: 0168-0072. DOI: 10.1016/j.apal.2006.12.001 (cit. on pp. 191, 193, 217, 272).

[Ven00] Varmo Vene. 'Categorical Programming with Inductive and Coinductive Types'. PhD Thesis. University of Tartu, 2000 (cit. on pp. 68, 69, 102).

[VU98] Varmo Vene and Tarmo Uustalu. 'Functional Programming with Apomorphisms (Corecursion)'. In: *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*. 1998, pp. 147–161. URL: http://www.cs.ioc.ee/~tarmo/papers/nwpt97-peas.pdf (cit. on p. 69).

[Wal93] Igor Walukiewicz. 'On Completeness of the Mu-Calculus'. In: *Proceedings of LICS '93*. IEEE Computer Society, 1993, pp. 136–146. DOI: 10.1109/LICS.1993.287593 (cit. on p. 9).

[Wer94] Benjamin Werner. 'Une Théorie Des Constructions Inductives'. PhD. Université Paris VII, 1994 (cit. on pp. 68, 223).

[Wet14] Linda Wetzel. *Types and Tokens*. The Stanford Encyclopedia of Philosophy. Ed. by Edward N. Zalta. 2014. URL: https://plato.stanford.edu/archives/spr2014/entries/types-tokens/ (cit. on p. 16).

[WBR11] Joost Winter, Marcello M. Bonsangue and Jan Rutten. 'Context-Free Languages, Coalgebraically'. In: *Proceedings of CALCO'11*. Vol. 6859. LNCS. Springer, 2011, pp. 359–376. DOI: 10.1007/978-3-642-22944-2_25 (cit. on p. 8).

[Wir04] Claus-Peter Wirth. 'Descente Infinie + Deduction'. In: *Logic Journal of the IGPL* 12.1 (2004), pp. 1–96. ISSN: 1367-0751. DOI: 10.1093/jigpal/12.1.1 (cit. on pp. 125, 126).

[Wor05] James Worrell. 'On the Final Sequence of a Finitary Set Functor'. In: *Theor. Comput. Sci.* 338 (1-3 2005), pp. 184–199. DOI: 10.1016/j.tcs.2004.12.009 (cit. on p. 27).

[Xi01] Hongwei Xi. 'Dependent Types for Program Termination Verification'. In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pp. 231–242. DOI: 10.1109/LICS.2001.932500 (cit. on pp. 68, 126).

[Zei09] Noam Zeilberger. 'The Logical Basis of Evaluation Order and Pattern-Matching'. PhD. Pittsburgh: Carnegie Mellon, 2009 (cit. on p. 102).

Own Publications

[Bas15a] Henning Basold. 'Dependent Inductive and Coinductive Types Are Fibrational Dialgebras'. In: *Proceedings of FICS '15*. Vol. 191. EPTCS. Open Publishing Association, 2015, pp. 3–17. DOI: 10.4204/EPTCS.191.3 (cit. on pp. 14, 165, 191).

[Bas18a] Henning Basold. 'Breaking the Loop: Recursive Proofs for Coinductive Predicates in Fibrations'. In: *arXiv* (2018). arXiv: 1802.07143 (cit. on pp. 14, 108).

[Bas+14a] Henning Basold, Marcello Bonsangue, Helle Hvid Hansen and Jan Rutten. '(Co)Algebraic Characterizations of Signal Flow Graphs'. In: *Horizons of the Mind – Prakash Panangaden Festschrift*. 2014, pp. 124–145. DOI: 10.1007/978-3-319-06880-0_6 (cit. on p. 15).

[BG16a] Henning Basold and Herman Geuvers. 'Type Theory Based on Dependent Inductive and Coinductive Types'. In: *Proceedings of LICS '16*. Logic In Computer Science. ACM, 2016, pp. 327–336. DOI: 10.1145/2933575.2934514 (cit. on pp. 15, 225, 274).

[BG16b] Henning Basold and Herman Geuvers. 'Type Theory Based on Dependent Inductive and Coinductive Types'. In: *Extended Abstracts for International Conference on Types for Proofs and Programs (TYPES)*. TYPES'16. 2016. DOI: 10.5281/zenodo.1175868 (cit. on p. 273).

[BG16c] Henning Basold and Herman Geuvers. 'Type Theory Based on Dependent Inductive and Coinductive Types'. In: *CoRR* abs/1605.02206 (2016). URL: http://arxiv.org/abs/1605.02206 (cit. on pp. 15, 225).

[BGvdW17] Henning Basold, Herman Geuvers and Niels van der Weide. 'Higher Inductive Types in Programming'. In: *J.UCS* David Turner's Festschrift – Functional Programming: Past, Present, and Future (2017). URL: http://www.jucs.org/jucs_23_1/higher_inductive_types_in (cit. on p. 15).

[Bas+14b] Henning Basold, Henning Günther, Michaela Huhn and Stefan Milius. 'An Open Alternative for SMT-Based Verification of Scade Models'. In: *Proceedings of Formal Methods for Industrial Critical Systems, FMICS 2014*. 2014, pp. 124–139. DOI: 10.1007/978-3-319-10702-8_9 (cit. on p. 15).

[BH16] Henning Basold and Helle Hvid Hansen. 'Well-Definedness and Observational Equivalence for Inductive-Coinductive Programs'. In: *J. Log. Comput.* (2016). DOI: 10.1093/logcom/exv091 (cit. on pp. 13, 14, 37, 49, 62, 72, 88, 105, 108, 309).

[Bas+15] Henning Basold, Helle Hvid Hansen, Jean-Éric Pin and Jan Rutten. 'Newton Series, Coinductively'. In: *Proceedings of ICTAC '15*. 2015, pp. 91–109. DOI: 10.1007/978-3-319-25150-9_7 (cit. on p. 15).

[Bas+17] Henning Basold, Helle Hvid Hansen, Jean-Éric Pin and Jan Rutten. 'Newton Series, Coinductively: A Comparative Study of Composition'. In: *MSCS* (2017), pp. 1–29. DOI: 10.1017/S0960129517000159 (cit. on p. 15).

[BK16] Henning Basold and Ekaterina Komendantskaya. 'Models of Inductive-Coinductive Logic Programs'. In: *Pre-Proceedings of the Workshop on Coalgebra, Horn Clause Logic Programming and Types*. 2016. arXiv: 1612.03032 (cit. on p. 15).

[BPR17] Henning Basold, Damien Pous and Jurriaan Rot. 'Monoidal Company for Accessible Functors'. In: *CALCO 2017*. Vol. 72. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. DOI: 10.4230/LIPIcs.CALCO.2017.5 (cit. on p. 15).

Subject Index

Notation Index

Appendices

Confluence for $\lambda\mu\nu=$

In this appendix, we develop the details of the proof that $\longrightarrow$ is confluent on $\Lambda^{=}(A)$ for all types A (Proposition 3.2.32). This proof was was given in [BH16].

Recall that we have introduced in Section 3.2.2 a reduction relation $\longrightarrow_\Sigma$ for terms of $\lambda\mu\nu=$ as the compatible closure of contraction. To be able to prove that this reduction relation is confluent, we have to be careful with nested **rlet**-bindings. For this reason, we define $\longrightarrow_\Sigma$ as the union of reduction relations $\longrightarrow_\Sigma^k$, $k \in \mathbb{N}$, where $\longrightarrow_\Sigma^k$ is a relation on terms of **rlet**-nesting depth k. More precisely, as in [AP13; Abe+13], $\longrightarrow_\Sigma^k$ is the compatible closure of $\succ_\Sigma^k$ outside of declaration blocks, $\succ_\Sigma^0 = \succ_\Sigma$, and $\succ_\Sigma^{k+1}$ is defined inductively by

$$\frac{t \longrightarrow^k_{\Sigma_1,\Sigma_2} t'}{\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t \succ^{k+1}_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t'}$$

$$\frac{}{\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (\alpha\ t) \succ^{k+1}_{\Sigma_1} \alpha\ (\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t)} \qquad \frac{}{\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (\kappa_i\ t) \succ^{k+1}_{\Sigma_1} \kappa_i\ (\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t)}$$

$$\frac{}{(\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t).\mathrm{out} \succ^{k+1}_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (t.\mathrm{out})} \qquad \frac{}{(\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t).\mathrm{pr}_i \succ^{k+1}_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (t.\mathrm{pr}_i)}$$

$$\frac{}{(\mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ t)\ s \succ^{k+1}_{\Sigma_1} \mathbf{rlet}\ \Sigma_2\ \mathbf{in}\ (t\ s)}$$

Terms of the form $\mathbf{rlet}\,\Sigma\ \mathbf{in}\ t$ without further declarations in t can be translated into the calculus of [Abe+13], while preserving reduction steps. Therefore, we inherit that $\longrightarrow_\Sigma^0$ preserves types.

Recall from Definition 3.2.29 that $\Lambda^{=}(A)$ contains only well-covering terms, that is, terms in which all declaration bodies are well-covering with respect to $\lhd|$. We show now that the reduction relation is confluent on $\Lambda^{=}$ in three steps. First, we prove that (co)pattern matching is deterministic on well-covering terms. This allows us, in a second step, to prove that $\longrightarrow_\Sigma^0$ that is confluent. Finally, we show, by induction, that $\longrightarrow_\Sigma^n$ is confluent for all n, thus $\longrightarrow = \bigcup_{n\in\mathbb{N}} \longrightarrow_\emptyset^n$ is confluent as all $\longrightarrow_\emptyset^n$ are disjoint.

We start by showing that copattern matching is deterministic.

Lemma A.1. *Let A be a type, Q a copattern sequence with $A \lhd| Q$ and e an evaluation context on A. If there exists a copattern q with $\Gamma \vdash_{\mathfrak{cop}} q : A \Rightarrow B$ in Q and contexts e_1, e_2 with $e = e_1[e_2]$, such that $q[\sigma] = e_2$, then q, e_1 and e_2 are unique with this property.*

Proof. Since any copattern sequence Q covering A is constructed using the rules in Definition 3.2.24 starting at $Q_0 = (\emptyset \vdash_{\mathfrak{cop}} \cdot : A \Rightarrow A)$, we can proceed by induction on the application of said rules.

In the base case $Q = Q_0$, there is only one choice, namely $q = \cdot$ and $e_1 = e$ and $e_2 = \cdot$.

So assume that for any Q we have a unique choice of q and $e = e_1[e_2]$. We make a case distinction on the rule used to construct Q' from Q. Note that we can distinguish two types of rules: C_{Prod},

C_{App} and C_{Out} increase the size of copatterns, whereas C_{Incl} and C_{In} increase the size of patterns. We only prove the induction step for C_{App} and C_{Incl} as exemplary cases.

- Assume that Q' is constructed from $Q = Q''\ ;\ (\Gamma \vdash_{\mathfrak{cop}} q : A \Rightarrow (B \to C))$ by an application of C_{App}, so that $Q' = Q''\ ;\ (\Gamma, x : B \vdash_{\mathfrak{cop}} q\, x : A \Rightarrow C)$. Moreover, assume that q_0 in Q' matches e_2 for some splitting $e = e_1[e_2]$. If $q_0 \neq q\, x$, then $q_0 \in Q''$ and uniqueness of e_1, e_2 and q_0 follows by induction. Otherwise, if $q_0 = q\, x$, then by the typing of e we must have $e_2 = e_3\ t$ for some term $t : B$ and context e_3. Hence, we have that $q \in Q$ matches e_3 and, by induction, the splitting $e = e_1[e_3\ t]$ is unique, as the choice of q is. Combining these cases, we have that the splitting $e = e_1[e_2]$ and the choice of q_0 is still unique in Q'.

- Assume that Q' is constructed from $Q = Q''\ ;\ (\Gamma, x : B_1 + B_2 \vdash_{\mathfrak{cop}} q : A \Rightarrow C)$ using the rule C_{Incl}, resulting in $Q' = Q''\ ;\ \big(\Gamma, x' : B_i \vdash_{\mathfrak{cop}} q[\kappa_i\, x'/x] : A \Rightarrow C\big)_{i=1,2}$. If we now have a splitting $e = e_1[e_2]$ and a match $q[\kappa_i\, x'/x][\sigma] = e_2$, then we can define a substitution τ such that $q[\tau] = e_2$ by putting

$$\tau(y) = \begin{cases} \kappa_i\, \sigma(x'), & y = x \\ \sigma(y), & \text{otherwise} \end{cases}.$$

  By the induction hypothesis, we now have that the splitting and q are unique for this match, thus the splitting and the choice of $q[\kappa_i\, x'/x]$ is unique. □

The next step is to prove confluence of the reduction in the base case, that is, of $\longrightarrow^0_\Sigma$. To do so, we invoke a result by Cirstea and Faure [CF07], which proves confluence of a reduction relation induced by a pattern matching algorithm for the so-called dynamic pattern λ-calculus. This calculus is an extension of (untyped) λ-calculus, in which λ-abstraction is allowed to have arbitrary terms in the abstraction, not just variable, that is to say, abstractions are of the for $\lambda M.\, N$ for arbitrary terms M and N. To interpret such an abstraction, we need to provide a *pattern matching algorithm*, which is a partial map from pairs of terms and sets of variables to substitutions, and is written as $\mathrm{Sol}(M \ll N)$. Such a pattern matching algorithm induces a reduction relation by taking the parallel reduction closure of

$$(\lambda M.\, N)P \longrightarrow N[\sigma], \qquad \text{if } \sigma = \mathrm{Sol}(M \ll P).$$

For more details, the reader should consult [CF07].

The idea of how to encode the calculus we study in this paper into the dynamic pattern λ-calculus and the (co)pattern matching into a pattern matching algorithm is very simple. Since we are allowed to use arbitrary constants, we can encode all term constructors of our calculus directly, only that we need to turn the projections $.\mathrm{pr}_i$ and .out into function arguments. For instance, $t.\mathrm{out}.\mathrm{pr}_2$ becomes $M\ .\mathrm{out}\ .\mathrm{pr}_2$, where M is the encoding of M. For a fixed declaration block Σ, the pattern matching algorithm is then the adaption of the matching with respect to evaluation context we used in Definition 3.2.13. This is similar to the case branching example given in [CF07].

The induced parallel reduction is not exactly the same as the compatible closure of contraction because they differ in the reduction of applications. However, the reduction relations can simulate each other, in the sense that if we can reduce a term, then the other relation can simulate this reduction in one or more steps. This is good enough to prove confluence: if parallel reduction is confluent for this encoding, then our reduction is confluent as well.

The heart of the matter are now the following three conditions from [CF07], which are sufficient to ensure that parallel reduction is confluent. Let us denote by $\mathrm{fv}(M)$ the free variables of M and by $\mathrm{dom}(\sigma)$ the domain of σ. Then the conditions are given by.

- H_0: If $\mathrm{Sol}(M \ll N) = \sigma$, then $\mathrm{fv}(M) = \mathrm{dom}(\sigma)$ and for all $x \in \mathrm{dom}(\sigma)$, $\mathrm{fv}(\sigma(x)) \subseteq \mathrm{fv}(N)$.
- H_1: Pattern matching commutes with substitution of variables not bound by the pattern: If $\mathrm{Sol}(M \ll N) = \sigma$ and $\mathrm{dom}(\tau) \cap \mathrm{fv}(M) = \emptyset$, then $\mathrm{Sol}(M \ll N[\tau]) = \tau \circ \sigma$.
- H_2: Pattern matching commutes with one-step reduction: If $\mathrm{Sol}(M \ll N) = \sigma$ and $N \longrightarrow N'$, then $\mathrm{Sol}(M \ll N) = \sigma'$ where σ' is the point-wise reduction of σ.

It is now straightforward to prove that the (co)pattern matching we used to define contraction fulfils these conditions.

Lemma A.2. *The (co)pattern matching on well-covering copattern sequences fulfils the conditions H_0, H_1 and H_2.*

Proof. Let A and Q be so that $A \lhd| Q$, and let $q[\sigma] = e$ for an evaluation context e and $q \in Q$. Note that q is unique by Lem. A.1.

H_0 Clearly, σ binds all variables in q and does not introduce fresh variables.

H_1 The condition H_1 requires, given a substitution τ with $\mathrm{dom}(\tau) \cap \mathrm{fv}(q) = \emptyset$, that $q[\sigma][\tau] = e[\tau]$. This is clearly the case, as no variable in q are substituted.

H_2 Assume we have $e \longrightarrow e'$, we need to show that $q[\sigma'] = e'$ with $\sigma \longrightarrow \sigma'$, where we can use the same q by uniqueness. Here we use the obvious lifting of $\longrightarrow$ to evaluation contexts and substitutions. If e was closed e' is still closed and by Lem. A.1 we still have a match $q[\sigma'] = e'$. The only interesting case to show that $\sigma \longrightarrow \sigma'$ is $q = x$, i.e. $x[t'/x] = t'$, but since $t \longrightarrow t'$ we clearly have $\sigma = [t/x] \longrightarrow [t'/x] = \sigma'$. The rest follows by induction on q. □

By the above discussion, confluence follows on terms without **rlet**-bindings from [CF07].

Lemma A.3. *On $\Lambda^{=}_{\Sigma}(A)$, $\longrightarrow^{0}_{\Sigma}$ is confluent for all well-covering Σ.*

Lemma A.3 is the base case for the main result, the confluence of $\longrightarrow^{k}_{\Sigma}$ for any k, which we prove by induction. To justify that this induction is actually well-formed, we need the following technical lemma. Let us denote the **rlet**-nesting depth of any syntactic entity S by $d(S)$, where S can be a term, a declaration block etc.

Lemma A.4. *For all terms t, t' with $t \longrightarrow_{\Sigma} t'$, we have $d(t') \leq \max\{d(t), d(\Sigma)\}$.*

Proof. Let t and t' with $t \longrightarrow_{\Sigma} t'$, and note that we then have that $t \longrightarrow^{d(t)}_{\Sigma} t'$, by definition of $\longrightarrow_{\Sigma}$. We observe that the only possibility to change the **rlet**-nesting is a use of contraction $e[f] \succ_{\Sigma'} r$, where $e[f]$ is a subterm of t.

There are now two possibilities: Either there there is a declaration block $\Sigma'' \subseteq \Sigma'$ that contains f and a term s that contains $e[f]$ as a subterm, such that **rlet** Σ'' **in** s is a subterm of t, or f is already contained in Σ.

In the first case, we reduce the subterm $\mathbf{rlet}\,\Sigma''\,\mathbf{in}\,s$ to $\mathbf{rlet}\,\Sigma''\,\mathbf{in}\,s'$, which induces the reduction $t \longrightarrow_\Sigma t'$ by the compatible closure and the **rlet**-rules. In turn, this reduction of subterms must be given by a reduction $s \longrightarrow_{\Sigma'} s'$, where $\Sigma'' \subseteq \Sigma'$, which is then induced by a contraction $e[f] \succ_{\Sigma'} r$ with $(f : A = D) \in \Sigma''$. By definition, we now have $d(\mathbf{rlet}\,\Sigma''\,\mathbf{in}\,s) = \max\{d(\Sigma'') + 1, d(s)\}$, thus $d(r) \leq d(D)$ and $d(s') \leq \max\{d(r), d(s)\} \leq \max\{d(D), d(s)\}$. This implies that $d(\mathbf{rlet}\,\Sigma''\,\mathbf{in}\,s') = \max\{d(\Sigma'') + 1, d(s')\} \leq \max\{d(\Sigma'') + 1, d(D), d(s)\} = d(\mathbf{rlet}\,\Sigma''\,\mathbf{in}\,s)$, where the last step follows from $(f : A = D) \in \Sigma''$. Since the only change caused by the reduction $t \longrightarrow t'$ happens in s, we have $d(t') \leq d(t)$.

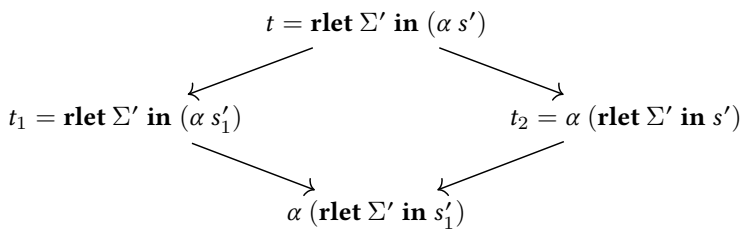
In the second case, we similarly get that $d(r) \leq d(\Sigma)$. Together with the first case, we have that $d(r) \leq \max\{d(t), d(\Sigma)\}$. □

This result allows us to prove that $\longrightarrow^k_\Sigma$ is confluent using induction on k, as the nesting depth cannot be increased by reduction steps.

Proof of Proposition 3.2.32. We show that $\longrightarrow^k_\Sigma$ is confluent by induction on k, which implies that $\longrightarrow_\Sigma$ is confluent because every term has a unique **rlet**-depth. This induction is well-founded by Lem. A.4. The base case is dealt with in Lem. A.3, so we immediately continue with the induction step.

Assume that $\longrightarrow^k_\Sigma$ is confluent for any well-covering Σ, we show that for any well-covering Σ the relation $\longrightarrow^{k+1}_\Sigma$ confluent. As usual, we show that for terms t, t_1 and t_2 with $t \longrightarrow^{k+1}_\Sigma t_i$ there is a term t_3 with $t_i \longrightarrow\!\!\!\!\rightarrow^{k+1}_\Sigma t_3$, which is called weak confluence and implies confluence. First, we note that if the reductions to t_1 and t_2 both come from the compatible closure, then we can find t_3 by induction on the definition of the compatible closure. The base case of this induction requires the existence of t_3 for the case where $t \longrightarrow^{k+1}_\Sigma t_1$ and $t \succ^{k+1}_\Sigma t_2$, which we prove in the following.

i) If $t = \mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s$, then we have the following cases.
   a) $t_i = \mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s_i$ with $s \longrightarrow^k_{\Sigma,\Sigma'} s_i$. Since $\longrightarrow^k_{\Sigma,\Sigma'}$ is confluent by induction, there is an s_3 with $s_i \longrightarrow^k_{\Sigma,\Sigma'} s_3$ and we can join t_1 and t_2 with $\mathbf{rlet}\,\Sigma\,\mathbf{in}\,s_3$.
   b) $t_1 = \mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s_1$ with $s \longrightarrow^k_{\Sigma,\Sigma'} s_1$, $s = \alpha\,s'$ and $t_2 = \alpha\,(\mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s')$. Then we must have that $s_1 = \alpha\,s_1'$ with $s_1 \longrightarrow^k_{\Sigma,\Sigma'} s_1'$, hence we can t_1 and t_2 by



   c) We proceed analogously if $s = \kappa_i\,s'$.
   d) The other cases follow by symmetry.

ii) If $t = (\mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s).\mathrm{out}$, $t_2 = \mathbf{rlet}\,\Sigma'\,\mathbf{in}\,(s.\mathrm{out})$ and $t_1 = (\mathbf{rlet}\,\Sigma'\,\mathbf{in}\,s_1).\mathrm{out}$ with $s \longrightarrow^k_{\Sigma,\Sigma'} s_1$, then we can reduce t_1 to $\mathbf{rlet}\,\Sigma'\,\mathbf{in}\,(s_1.\mathrm{out})$ and s to s_1. Thus the joining term is $\mathbf{rlet}\,\Sigma'\,\mathbf{in}\,(s_1.\mathrm{out})$.

iii) We proceed analogously if t is an **rlet** in context of an application or π_i.

iv) The remaining cases are either trivial because the same reduction happens on both t_1 and t_2, they follow by symmetry, or combinations of reductions by $\succ_\Sigma^{k+1}$ are excluded by the types of t_1 and t_2.

This proves that, if $t \longrightarrow_\Sigma^{k+1} t_1$ and $t \succ_\Sigma^{k+1} t_2$, then there exists t_3 with $t_i \twoheadrightarrow_\Sigma^{k+1} t_3$. It is straightforward to extend this by induction to the compatible closure, hence to arbitrary reductions towards t_2. This shows that $\longrightarrow_\Sigma^{k+1}$ is confluent for any well-covering Σ, provided $\longrightarrow_{\Sigma'}^{k}$ is for any well-covering Σ'. Thus, by induction on k, $\longrightarrow_\Sigma$ is confluent. □ □

Proofs of Section 6.3

We need the following technical tool.

Lemma B.1 (Primitive corecursion)**.** *Let* $\mathbf{C}$ *be a category with binary coproducts and* $F\colon \mathbf{C} \to \mathbf{C}$ *an endofunctor on* $\mathbf{C}$ *with a final coalgebra* $(M, \xi : M \to FM)$*. For every morphism* $c : X \to F(X + M)$ *in* $\mathbf{C}$*, there is a unique map* $h : X + M \to M$*, such that* $h \circ \kappa_2 = \mathrm{id}_M$ *and the following diagram commutes.*

$$\begin{array}{ccc} X & \xrightarrow{h \circ \kappa_1} & M \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle \xi} \\ F(X+M) & \xrightarrow{Fh} & FM \end{array}$$

Proof. We define h as the coinductive extension as in the following diagram.

$$\begin{array}{ccccc} X & \xrightarrow{\kappa_1} & X+M & \xrightarrow{h} & M \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle [c, F\kappa_2 \circ \xi]} & & \downarrow{\scriptstyle \xi} \\ F(X+M) & = & F(X+M) & \xrightarrow{Fh} & FM \end{array}$$

It is easily checked that the rectangle on the right commutes if and only if the above identities hold. Thus uniqueness of h follows from uniqueness of coinductive extensions. □

Primitive corecursion allows us to define one-step behaviour as follows.

Lemma B.2 (One-step extension)**.** *Let* F *and* (M, ξ) *as above, and let* $f : M \to FM$ *be a morphism. Then there exists a unique* $g : M \to M$*, such that* $\xi \circ g = f$*.*

Proof. We define $g = h \circ \kappa_1$, where h arises by primitive corecursion of $F\kappa_2 \circ f$. It is then straightforward to show that $\xi \circ g = f$ if and only the identities of primitive corecursion hold. Thus g is the unique morphism for which this identity holds. □

Using the definition of V as equaliser of u_1 and u_2, we can characterise elements of V as follows. First we note that V is indexed over I by $q = V \xrightarrow{g} M_f \xrightarrow{\rho} A \xrightarrow{t} I$, where ρ is the root map given by composing ξ_f with projection for coproducts. Abusing notation, we will use V instead of q, and write $x : V_i$ if $x : V$ and $q\,x = i$.

Let X be an object in $\mathbf{B}$. An object $R \in \mathbf{B}/X^2$ is called a relation, and we say that elements $x, y : X$ are related by are, denoted $(x, y) : R$, if there is a $z : R$, such that $\pi_1(R\,z) = x$ and $\pi_2(R\,z) = y$.

Lemma B.3 (Internal bisimulations)**.** *Let* $f : B \to A$ *be a polynomial and* $R \in \mathbf{B}/M_f^2$ *a relation over* M_f *such that*

$$\begin{aligned} \forall (x_1, x_2) : R.\ & \textit{if } \xi_f(x_k) = (a_k, v_k) \\ & \textit{then } a_1 = a_2 = a \\ & \textit{and } (\forall b : B. f\,b = a \Rightarrow (v_1\,b, v_2\,b) : R). \end{aligned}$$

Then for all $(x_1, x_2) : R$, we have that $x_1 = x_2$.

Proof. It is easy to see that this allows us to define a coalgebra structure on $R : U \to M_f^2$ such that $\pi_k \circ R : U \to M_2$ are homomorphism for $k = 1, 2$, which implies by finality of M_f that $\pi_1 \circ R = \pi_2 \circ R$. □

In the following lemmas we use the notation introduced in the proof of Thm. 6.3.6.

Lemma B.4. *If $y : M_f$ and $b : B$ such that $\phi(u_1\, y, b) = u_1\, y$, then $q\, y = s\, b$ and $u_1\, y = u_2\, y$.*

Proof. We let $\xi_f\, y = (a, v)$ and then find that

$$\begin{aligned}\xi_{f\times I}\,(\phi(u_1\, y, b)) &= (a, s\, b, \lambda b'.\phi(u_1\,(v\, b'), b')) \\ &= (a, t\, a, \lambda b'.u_1\,(v\, b')) && \text{by assumption} \\ &= \xi_{f\times I}\,(u_1\, y).\end{aligned}$$

Thus $s\, b = t\, a = q\, y$ and $\phi(u_1\,(v\, b'), b') = u_1\,(v\, b')$ for all $b' : B$ with $f\, b = a$. This gives us

$$\begin{aligned}\xi_{f\times I}\,(u_1\, y) &= (a, t\, a, \lambda b'.u_1\,(v\, b')) \\ &= (a, t\, a, \lambda b'.\phi(u_1\,(v\, b'), b')) && \text{see above} \\ &= \xi_{f\times I}\,(u_2\, y)\end{aligned}$$

as required. □

Lemma B.5. *Let $i : I$ and $x : M_f$, then the following are equivalent*

1. $x : V_i$
2. $u_1\, x = u_2\, x$ *and* $q\, x = i$
3. $\xi_f\, x = (a : A, v : \Pi_f M_f)$, $t\, a = i$ *and* $(\forall b : B. f\, b = a \Rightarrow v\, b : V_{s\, b})$
4. $\xi_f\, x = (a : A, v : \Pi_f M_f)$, $t\, a = i$ *and* $v : \Pi_f(s^* V)$

Proof. The equivalences 1 $\iff$ 2 and 3 $\iff$ 4 are the definitions, so let us prove 2 $\iff$ 3.

We begin by proving 2 $\Rightarrow$ 3. Let $x : M_f$ with $u_1\, x = u_2\, x$ and $q\, x = i$. Then we have for $x_f\, x = (a, v)$ that $t\, a = q\, x = i$,

$$\xi_{f\times I}\,(u_1\, x) = [\![f \times I]\!](u_1)\,(p_{M_f}\,(\xi_f\, x)) = (a, t\, a, \lambda b.u_1(v\, b))$$

and

$$\begin{aligned}\xi_{f\times I}\,(u_2\, x) &= [\![f \times I]\!](\phi)\,(\Sigma_{A\times I} K\,(\xi_{f\times I}\,(u_1\, x))) \\ &= [\![f \times I]\!](\phi)\,(\Sigma_{A\times I} K\,(a, t\, a, \lambda b.u_1\,(v\, b)) \\ &= (a, t\, a, \lambda b.\phi(u_1\,(v\, b), b)).\end{aligned}$$

By these calculations and Since $u_1\, x = u_2\, x$, we also have for all $b : B$ with $f\, b = a$ that $u_1(v\, b) = \phi(u_1\,(v\, b), b)$. Applying Lem. B.4 to $y = v\, b$ we get that $q\,(v\, b) = s\, b$ and $u_1\,(v\, b) = u_2\,(v\, b)$, thus $v\, b : V_{s\, b}$ and 3 holds.

For the other direction, assume that $\xi_f\, x = (a : A, v : \Pi_f M_f)$, $t\, a = i$ and $(\forall b : B. f\, b = a \Rightarrow v\, b : V_{s\, b})$. We show that $u_1\, x = u_2\, x$ by giving a bisimulation R that relates $u_1\, x$ and $u_2\, x$. We put

$$\begin{aligned} X &= \mathbf{1} + \Sigma_B.\, s^*\, V \\ R &: X \to M_f \times M_f \\ R(*) &= (u_1\, x, u_2\, x) \\ R(b, y) &= (u_1\, y, \phi(y, b)) \end{aligned}$$

which is a relation over M_f. To prove that R is a bisimulation, there are two cases to consider. First, we have $(u_1\, x, u_2\, x) : R$. Note that

$$\xi_{f \times I}\, (u_1\, x) = (a, t\, a, \lambda b. u_1\, (v\, b))$$

and

$$\xi_{f \times I}\, (u_2\, x) = (a, t\, a, \lambda b. \phi(u_1\, (v\, b), b))$$

so that $\rho_{f \times I}\, (u_1\, x) = (a, t\, a) = \rho_{f \times I}(u_1\, x)$. Moreover, we have for all $b : B$ that $u_1\, (v\, b)$ and $\phi(u_1\, (v\, b), b)$ are related by R. For the second case, let $b : B$ and $y : V_{s\, b}$. Then for $x_f y = (a', v')$ we have

$$\xi_{f \times I}\, (u_1\, y) = (a', t\, a', \lambda b'. u_1\, (v'\, b'))$$

and

$$\xi_{f \times I}\, \phi(y, b) = (a', s\, b, \lambda b'. \phi(u_1\, (v'\, b'), b')).$$

Since $y : V_{s\, b}$, we have, by definition, that $s\, b = q\, y = t\, a'$, thus $(a', t\, a') = (a', s\, b)$. Moreover, $u_1\, (v'\, b')$ and $u_1(v'\, b', b')$ are again related by R. Hence, we can conclude that R is a bisimulation, and so $u_1\, x = u_2\, x$. □

Theorem 6.3.6. By Lemma B.5.4, we immediately have that $\xi_f : M_f \to [\![f]\!](M_f)$ restricts to $\xi' : V \to [\![P]\!](V)$. To prove that ξ' is also final we need another ingredient. We define a natural transformation $\iota : \Sigma_I [\![P]\!] \Rightarrow [\![f]\!] \Sigma_I$ (where $\Sigma_I : \mathbf{B}/_I \to \mathbf{B}$) for each $k : X \to I$ by $\iota_k \left(i : I, a : A, v : \Pi_f (s^*\, k)\right) = (a, \lambda b. (s\, b, v\, b))$ where $t(a) = i$.

Now, let $k : X \to I$ be in $\mathbf{B}/_I$ and $c : k \to [\![P]\!](k)$ be a coalgebra on k. Using ι, we can define a morphism h as in the following diagram.

$$\begin{array}{ccc} \Sigma_I k & \xrightarrow{\quad h \quad} & M_f \\ \Big\downarrow{\scriptstyle \iota_k \circ \Sigma_I c} & & \Big\downarrow{\scriptstyle \xi_f} \\ [\![f]\!](\Sigma_I k) & \xrightarrow{[\![f]\!](h)} & [\![f]\!](M_f) \end{array}$$

Thus for $i : I$ and $x : X$ with $k(x) = i$, and $c(x) = (a, v)$, we have

$$\xi_f(h(i, x)) = [\![f]\!](h)(\iota_k(i, a, v)) = (a, \lambda b. h(s\, b, v\, b)). \tag{B.1}$$

Using (B.1), we can now show that $h(k\,x, x) : V_{k\,x}$ for $x : X$. For brevity, we put $i \coloneqq k\,x$. By Lemma B.5.3, we need to show that for $\xi_f(h(i,x)) = (a, \lambda b.h(s\,b, v\,b))$ with $c\,x = (a, v)$ we have $t\,a = i$ and $h(s\,b, v\,b) : V_{s\,b}$. The first is immediate, since $(a, v) : [\![P]\!](k)$, thus $t\,a = i$ by definition of the extension $[\![P]\!]$ of P. The second follows by coinduction, as $k\,(v\,b) = s\,b$.

This allows us to define the coinductive extension $\widetilde{c} : X \to V$ of c by $\widetilde{c}\,x = h(k\,x, x)$ as a morphism $k \to q$ in $\mathbf{B}/I$. That $\widetilde{c}$ is a homomorphism $c \to \xi'$ is easily checked as follows.

$$\begin{aligned}
\xi'(\widetilde{c}\,x) &= \xi'(h(k\,x, x)) \\
&= \xi_f(h(k\,x, x)) \\
&= (a, \lambda b.h(s\,b, v\,b)) && (a, v) = c\,x \\
&= (a, \lambda b.h(k\,(v\,b), v\,b)) && k\,(v\,b) = s\,b \\
&= (a, \lambda b.\widetilde{c}\,v\,b) \\
&= ([\![P]\!]\,\widetilde{c})(c\,x)
\end{aligned}$$

Finally, we show how uniqueness of $\widetilde{c}$ follows from uniqueness of h. Let $g : (k, c) \to (q, \xi')$ be a $[\![P]\!]$-homomorphism, and define $g' : \Sigma_I k \to M_f$ by $g'(i : I, x : X_i) = g\,x$. It is easy to see that g' is a $[\![f]\!]$-homomorphism from $\iota_k \circ \Sigma_I c$ to ξ_f:

$$\begin{aligned}
\xi_f\,(g'\,(i, x)) &= \xi_f\,(g\,x) \\
&= ([\![P]\!]\,g)(c\,x) && g \text{ homomorphism} \\
&= (a, \lambda b.(s^*\,g)(v\,b)) && (a, v) = c\,x \\
&= (a, \lambda b.g'(s\,b, v\,b)) && (*) \\
&= ([\![f]\!]\,g')(\iota_k((\Sigma_I c)(i, x)))
\end{aligned}$$

where $(*)$ follows since $v\,b : V_{s\,b}$ and $q(g(v\,b)) = k(v\,b) = s\,b$ by g being a morphism from k to q. Thus, by finality of ξ_f, $g' = h$ and so $g = \widetilde{c}$. □

Summary

Induction and coinduction are threads that cross the landscape of Mathematics and Computer Science as methods to define objects and reason about them. Of these two, induction is by far the better known technique, although coinduction has always been around in disguise. It was only in recent years that we began to see through this disguise and developed coinduction as a technique in its own right. This led to some remarkable theory under the umbrella of coalgebra and to striking applications of coinduction.

As it turns out, induction and coinduction are complementary techniques, they are *dual* in a precise sense. Being complementary, it is often necessary to use both techniques or even intertwine them. In this thesis, we show that combined induction-coinduction is often used implicitly, just like induction and coinduction used to be before they were studied systematically. Thus, the purpose of this thesis is to carefully study the combination of induction and coinduction, which hopefully, if anything, inspires others to work on and use inductive-coinductive techniques.

One example, which pervades the thesis, illustrates the combination particularly well: the so-called substream relation. A stream s, that is an infinite sequence, is a substream of stream t if all the entries of s occur in order in t. Intuitively, one has to find *for all entries* in s an entry in t with the same value in *finitely many steps*. The fact that we may only use a finite number of steps to find each entry is an iterative process, while its repetition for all entries is a coiterative process. Since these two processes are interleaved, the substream relation is a mixed inductive-coinductive relation.

To be able to deal with such examples, we aim in this thesis to *find and study languages for inductive-coinductive definitions and reasoning*, which *lend themselves to being automatically verifiable*, can be *equipped with formal semantics*, and *allow human-readable specifications and proofs*. Put in general terms, the intention of studying languages for inductive-coinductive definitions and reasoning is to provide a framework that is sufficiently rich to accommodate category theory and set theory. Moreover, this framework should allow for semantics that are, in principle, independent of those theories, and for proofs that can be formalised and automatically verified. This is of course an ambitious goal that will not be fully attained in this thesis, but we will nonetheless contribute to it.

Towards these aims, we proceed in several steps. The first step is to have some objects to reason about, which means specifically for this thesis that we provide two programming languages for inductive-coinductive types. In the first language, one can only write terminating programs, while the second allows arbitrary recursive specification. Such recursive specifications ease programming with mixed inductive-coinductive types dramatically over programming with the iteration and coiteration schemes in the first language. However, this increased expressiveness also comes at the price that we give up simple syntactic conditions for well-defined programs (programs that terminate under any observation) and have to characterise such programs in a different way.

To this end, we establish notions of observationally normalising programs, the well-defined programs, and an observational program equivalence in the presence of inductive-coinductive types and arbitrary recursive specifications. As it turns out, the characterisation of observationally normalising programs is itself an inductive-coinductive predicate. In contrast, the program equivalence is introduced through a modal logic and turns out to be purely coinductive. Given such a program equivalence, we have to ask whether it is well-motivated. An important property of the equivalence

is that is allows us to construct 2-categories of types and terms, in which equality captures computations, while 2-cells represent program equivalences. In particular, least fixed point types and greatest fixed point types carry, respectively, pseudo-initial algebras and pseudo-final coalgebras in these 2-categories.

Even though the 2-category theoretical results give us some principles to reason about programs, these principles are not always the most convenient ones to work with. For this reason, we develop in the next step more convenient reasoning techniques: a bisimulation proof method, a syntactic first-order logic that is itself recursive, and an algorithm that operates on fragments of the programming languages. The bisimulation proof method can be enhanced by using so-called up-to techniques, which we demonstrate by showing that the substream relation is transitive. In particular, we use an up-to technique that allows us to use induction inside a bisimulation proof.

This setup becomes, unfortunately, rather complex because implementing induction as up-to technique forces a stratification of a mixed inductive-coinductive proof into a coinduction and two inductions. What is worse, the induction proofs must be proven independently of the coinduction, which makes finding the correct induction hypothesis a highly non-trivial task. To overcome such problems, we construe a recursive logic, in which inductive and coinductive proofs are incrementally constructed together. Recursion in proofs of this logic is thereby controlled through the so-called later modality. This modality allows us to ensure the correctness of a proof through each proof rule separately, which makes both proof checking and the soundness proof easy to implement.

Up to this point, the thesis is about programming with simple types, and reasoning about a very specific inductive-coinductive predicate (observational normalisation) and a very specific coinductive relation (observational equivalence). Both are defined in naive set theory, which means that neither are given in a principled manner nor that proofs about them can be directly formally expressed. This clearly violates our aims and is what caused us to invent, for example, a new logic from scratch. The remainder of the thesis deals with this issue in that we develop a category theoretic and a type theoretic approach that allow us to formally express results about simple inductive-coinductive types, and inductive-coinductive predicates and relations. It turns out that inductive-coinductive predicates and relations are best expressed as certain dialgebras in fibrations. This approach subsumes simple types and general dependent types. What is more, we are able to define a class of strictly positive dependent types and reduce them to initial algebras and final coalgebras of polynomial functors. This reduction to minimal requirement allows us to interpret dependent types in well-studied models. The category theoretical approach to dependent types is concluded by an analysis of the logical principles that are available in a fibration that is closed under strictly positive dependent types.

Following the guiding principles of the category theoretical development, we construct also a (syntactic) dependent type theory. This results in a small type theory that is merely based on inductive-coinductive dependent types, yet it admits all basic logical operators like conjunction, implication, quantification etc. Since this type theory is built on iteration and coiteration principles, we are able to show that all terms in that theory are strongly normalising. The thesis is concluded by giving a general induction principle for this type theory and an elaborate example of inductive-coinductive reasoning in Agda.

Samenvatting

Inductie en coinductie vormen twee technieken in het landschap van Wiskunde en Informatica die gebruikt worden om objecten te definiëren en om eigenschappen van deze objecten te laten zien. Inductie is bekender dan coinductie, maar coinductie was er op de achtergrond ook altijd al. In de laatste jaren is men begonnen coinductie als een op zich zelf staande techniek te ontwikkelen. In het kader van coalgebra is daar een uitgebreide theorie met opmerkelijke toepassingen uitgekomen.

Het blijkt dat inductie en coinductie complementair zijn: ze zijn op een bepaalde manier *duaal*. Omdat dat ze complementair zijn, moeten beide technieken vaak samen gebruikt worden. In dit proefschrift laten wij zien dat dit vaak impliciet gebeurt, op dezelfde manier als inductie en coinductie vroeger impliciet gebruikt werden voordat ze systematisch bestudeerd werden. Het doel van dit proefschrift is dus de combinatie van inductie en coinductie in detail te bestuderen, wat hopelijk inspiratie geeft om inductieve/coinductieve technieken verder te ontwikkelen en te gebruiken.

Een voorbeeld, die als een rode draad door dit proefschrift loopt en de combinatie verduidelijkt, is de zogenaamde deelstroomrelatie. Die relateert stromen, dus oneindige rijen, s and t dan en slechts dan als alle elementen in s in dezelfde volgorde in t voorkomen. De intuïtie is dat we *voor iedere positie* in s een element met dezelfde waarde in t moeten vinden, *in eindig veel stappen*. Het feit dat wij slechts eindig veel stappen mogen gebruiken om de positie te vinden, maakt het een iteratief proces, terwijl de herhaling een coiteratief proces is. Omdat deze twee processen van elkaar afhankelijk zijn, is de deelstroomrelatie een gemengde inductieve/coinductieve relatie.

Om met dit soort voorbeelden om te kunnen gaan, bestuderen en zoeken we in dit proefschrift *talen voor inductieve/coinductieve definities en eigenschappen*, die *als basis voor de automatische verificatie van bewijzen gebruikt kunnen worden*, die *een formele semantiek hebben*, en die *makkelijk inzetbaar zijn door anderen*. Vanuit een abstract perspectief is het doel van deze studie van inductieve/coinductieve definities en bewijsprincipes het geven van een raamwerk dat als logische basis voor categorieëntheorie een verzamelingstheorie kan dienen. Verder moet het mogelijk zijn een semantiek voor dit raamwerk te geven, die in beginsel onafhankelijk van deze theorieën is, en moet het mogelijk zijn bewijzen uit dit raamwerk te formaliseren en automatisch te verifiëren. Dit is een ambitieus doel dat niet geheel gehaald kan worden in dit proefschrift, maar waartoe we zeker een bijdrage leveren.

We gaan in meerdere stappen te werk om in richting van de bovengenoemde doelen te gaan. De eerste stap is dat we objecten nodig hebben waarvan we eigenschappen willen beschrijven. In dit proefschrift betekent dit specifiek dat wij twee talen voor het programmeren met inductieve en coinductieve typen introduceren. In de eerste taal is het mogelijk alleen convergerende programma's op te schrijven, terwijl de tweede taal willekeurige, recursieve specificaties toestaat. Zulke recursieve specificaties maken het programmeren met gemengde inductieve/coinductieve typen veel makkelijker, vergeleken met het programmeren met iteratie- en coiteratieschema's in de eerste taal. Het probleem met algemene recursie is dat wij eenvoudige, syntactische condities voor welgedefinieerde programma's—die bij het toepassen van iedere observatie termineren—verliezen en dat we deze programma's op een andere manier moeten karakteriseren.

Om dit te bereiken, introduceren we een begrip van welgedefinieerde programma's, dat we "observationally normalising" noemen, en een programma-equivalentie in de context van inductieve/coinductieve typen en algemene recursieve specificaties. Het blijkt dat de karakterisering van

welgedefinieerde programma's zelf een inductief/coinductief predicaat is. Daarentegen wordt de programmaequivalentie als een modale logica geïntroduceerd, en is deze equivalentie puur coinductief. Om een goede motivatie voor deze programmaequivalentie te geven, introduceren we een 2-categorie van typen, termen en equivalenties als 2-cellen. In deze 2-categorie komen kleinste fixpunten overeen met pseudo-initiële algebra's, en grootste fixpunten met pseudo-finale coalgebra's.

Hoewel we bepaalde technieken uit 2-categorieëntheoretische resultaten kunnen halen, zijn deze technieken niet altijd het meest geschikt. Daarom ontwikkelen wij in een volgende stap praktischere bewijsmethoden: een methode gebaseerd op bisimulaties, een syntactische, recursieve eerste-orde logica, en een algoritme voor een fragment van de programmeertalen. De bisimulatieaanpak kan door zogenaamde up-to technieken verbeterd worden. Wij demonstreren dit door een voorbeeld, waarin we laten zien dat de deelstroomrelatie transitief is. Daarbij maken maken wij vooral gebruik van een up-to techniek waarmee we inductie in een bisimulatiebewijs kunnen gebruiken.

Deze opzet wordt helaas best ingewikkeld, omdat door de implementatie van inductie als up-techniek een stratificatie van een gemengd inductief/coinductief bewijs in een coinductie en twee inducties gebeurt. Sterker nog, de inductieve bewijzen moeten onafhankelijk van de coinductieve stap bewezen worden, waardoor het vinden van de juiste inductiehypotheses zelf een ingewikkelde opgave wordt. Om dit probleem op te lossen, ontwikkelen we een recursieve logica waarin inductieve en coinductieve bewijzen samen en incrementeel geconstrueerd worden. Om te voorkomen dat de recursie in bewijzen misgaat, gebruiken wij de zogenaamde "later modality". Omdat het door deze modaliteit mogelijk is de correctheid van bewijzen op het niveau van regels te waarborgen, worden ook het controleren van bewijzen en het correctheidsbewijs vereenvoudigd.

Tot dit punt gaat het in het proefschrift alleen maar over het programmeren met eenvoudige typen, en over eigenschappen van een specifiek inductief/coinductief predicaat (observational normalisation) en een specifieke coinductieve relatie (observational equivalence). Beide zijn in een naïeve verzamelingstheorie gedefinieerd, dus zijn deze definities niet op fundamentele principes gebaseerd, en kunnen bewijzen over dit predicaat/deze relatie niet direct formeel uitgedrukt worden. Dit gaat tegen onze doelen in, en is een reden om de recursieve logica te ontwikkelen. In het resterende deel van het proefschrift lossen we dit op door categorieëntheoretische en typentheoretische aanpakken te ontwikkelen die het mogelijk maken resultaten over eenvoudige inductieve/coinductieve typen, predicaten en relaties formeel uit te drukken. Het blijkt dat dialgebra's in vezelingen de beste manier zijn om inductieve/coinductieve predicaten en relaties uit te drukken. Door deze aanpak wordt het ook mogelijk met eenvoudige typen en algemene afhankelijke typen om te gaan. Bovendien kunnen wij een klasse van strikt positieve, afhankelijke typen definiëren, waarvoor wij een semantiek in vorm van initiële algebra's en finale coalgebra's van polynomiale functoren krijgen. Door deze reductie naar minimale eisen wordt het mogelijk afhankelijke typen over welbekende modellen te interpreteren. We sluiten de categorieëntheoretische aanpak af met een analyse van logische principes die gelden in een vezeling die gesloten is onder strikt positieve, afhankelijke typen.

De laatste stap is de constructie van een (syntactische) afhankelijke typentheorie, waarbij wij de principes van de categorieëntheoretische aanpak volgen. Daardoor krijgen wij een typentheorie met weinig regels die alleen maar op inductieve/coinductieve, afhankelijke typen gebaseerd is, maar nog steeds logische operatoren zoals conjunctie, implicatie en kwantificering toestaat. Omdat deze typentheorie alleen op iteratie en coiteratie gebaseerd is kunnen wij laten zien dat alle termen in deze theorie sterk-normaliserend zijn. We sluiten het proefschrift met een algemeen inductieprincipe voor deze typentheorie en een uitgebreid voorbeeld van een inductief/coinductief bewijs in Agda.

Zusammenfassung

Induktion und Koinduktion ziehen sich durch die mathematische und informatische Landschaft wie zwei Ströme, die genutzt werden um Objekte zu definieren und Eigenschaften dieser Objekte herzuleiten. Von den beiden Techniken ist Induktion die besser bekannte, obwohl Koinduktion schon immer, wenn auch implizit, genutzt worden ist. Lediglich in den letzten Jahren wurde Koinduktion als eigenständige Technik entwickelt, wobei sich das Gebiet der sogenannten Koalgebren herausgebildet hat. Dieses Gebiet umfasst mittlerweile eine weitläufige Theorie und bemerkenswerte Anwendungen.

Es hat sich herausgestellt, dass Induktion und Koinduktion komplementäre Techniken sind: um genau zu sein, sind sie *dual*. Da sie komplementär sind, ist es häufig notwendig beide Techniken zusammen zu gebrauchen. In dieser Dissertation zeigen wir, dass dies zumeist implizit passiert, genauso wie früher Induktion und Koinduktion implizit genutzt worden sind, bevor sie systematisch untersucht worden sind. Der Zweck dieser Dissertation ist daher systematisch und im Detail die Kombination von Induktion und Koinduktion zu untersuchen. Falls irgendetwas aus dieser Dissertation hervorgeht, dann dass sie hoffentlich andere inspiriert induktiv-koinduktive Techniken weiter zu entwickeln und zu gebrauchen.

Ein Beispiel, das sich als roter Faden durch diese Arbeit zieht und das die Kombination von Induktion und Koinduktion verdeutlicht, ist die sogenannte Teilstromrelation. In dieser Relation stehen Ströme (unendliche Folgen) s und t genau dann in Beziehung, wenn alle Einträge in s in der gleichen Reihenfolge in t auftauchen. Die Intuition dieser Beziehung ist, dass wir *für jede Position* in s mit Gebrauch von *endlich vielen Schritten* einen Eintrag in t mit dem gleichen Wert finden müssen. Tatsächlich ist die Suche in endlich vielen Schritten ein iterativer Prozess, wohingegen ihre Wiederholung für jede Position ein koiterativer Prozess ist. Da diese beiden Prozessen verzahnt sind, ist die Teilstromrelation eine gemischt induktiv-koinduktive Relation.

Um mit derartigen Beispielen umgehen zu können, untersuchen wir in dieser Arbeit *Sprachen für induktiv-koinduktive Definitionen und Eigenschaften*, welche als Basis für die *automatische Verifikation von Beweisen* genutzt werden können, welche eine *formale Semantik haben*, und welche es ermöglichen *Spezifikationen und Beweise derart zu formulieren, dass diese für Menschen verständlich sind.* Aus der Vogelperspektive ist das Ziel unserer Untersuchungen von induktiv-koinduktiven Definitionen und Beweisen das Schaffen eines Rahmens, welcher als logisches Fundament für Kategorien- und Mengentheorie fungieren kann. Außerdem muss es möglich sein, diesem Rahmen eine Interpretation zu geben, die im Prinzip unabhängig von diesen Theorien ist und es muss möglich sein die Beweise, die in diesem Rahmen gegeben werden, zu formalisieren. Das ist ein ehrgeiziges Ziel, das wir in dieser Arbeit nicht erreichen, und doch werden wir sicher einen Beitrag in diese Richtung leisten.

Um die genannten Ziele zu erreichen, werden wir in mehreren Schritten vorgehen. Zunächst benötigen wir Objekte, über die wir Aussagen treffen können. In dieser Arbeit bedeutet das im Speziellen, dass wir zwei Programmiersprachen zum Programmieren mit induktiven und koinduktiven Typen entwickeln. In der ersten Sprache können nur terminierende Programme beschrieben werden, während die zweite Sprache beliebige rekursive Programme zulässt. Im Vergleich zu den Iterations- und Koiterationsschemata der ersten Sprache, wird das Programmieren durch das Erlauben beliebiger Rekursionen immens vereinfacht. Allerdings ist der Preis, den wir für die stärkere Ausdruckskraft zahlen müssen, dass es nicht mehr möglich ist wohldefinierte Programme, die unter

jeder Beobachtung terminieren, durch einfache syntaktische Kriterien zu charakterisieren.

Um eine solche Charakterisierung dennoch zu erreichen und um Aussagen über induktiv-koinduktive Programme mit beliebiger Rekursion treffen zu können, führen wir einen Begriff von wohldefinierten Programmen, die wir "observationally normalising" nennen, und eine Programmäquivalenz ein. Es stellt sich heraus, dass die Charakterisierung von wohldefinierten Programmen selbst ein induktiv-koinduktives Prädikat ist. Dahingegen ist die Programmäquivalenz durch eine Modallogik gegeben und rein koinduktiv. Um die Programmäquivalenz zu motivieren, definieren wir eine 2-Kategorie, bestehend aus Typen als Objekte, Programmen als Morphismen und Programmäquivalenzen als 2-Zellen. In dieser 2-Kategorie werden Berechnungen durch Gleichungen zwischen Morphismen repräsentiert, während herleitbare Programmäquivalenzen Isomorphismen von Morphismen sind. Im Speziellen zeigen wir, dass kleinste Fixpunkttypen mit pseudo-initialen Algebren und größte Fixpunkttypen mit pseudo-finalen Koalgebren zusammenfallen.

Auch wenn die Resultate über 2-Kategorien uns bereits einige Prinzipien geben, um Beweise über Programme zu führen, ist dieser Ansatz nicht immer der geeignetste. Daher entwickeln wir im Folgenden praktischere Beweistechniken: eine auf Bisimulationen basierte Methode, eine syntaktische Prädikatenlogik erster Stufe mit rekursiven Beweisen, und einen Algorithmus, der die Äquivalenz für ein Fragment der Programmiersprachen entscheiden kann. Die Bisimulationenmethode kann durch sogenannte up-to-Techniken verbessert werden. Wir führen dies anhand eines Beispiels vor, in welchem wir zeigen, dass die Teilstromrelation transitiv ist. Hierbei benutzen wir vor allem eine up-to-Technik, durch welche wir Induktion innerhalb eines Bisimulationsbeweis gebrauchen können.

Leider wird dieser Ansatz ziemlich komplex, da die Umsetzung von Induktion als up-to-Technik einen gemischt induktiv-koinduktiven Beweis in eine Koinduktion und zwei Induktionen schichtet. Schwerer wiegt aber, dass die induktiven Beweise unabhängig von dem koinduktiven Schritt geführt werden müssen, wodurch das Finden der richtigen Induktionsannahme selbst zu einer schwierigen Aufgabe wird. Um dieses Problem zu lösen, entwickeln wir eine rekursive Logik, in der induktive und koinduktive Beweise zusammen und inkrementell konstruiert werden. Rekursion wird in dieser Logik durch die sogenannte "later modality" kontrolliert. Mit Hilfe dieser Modalität können wir die Korrektheit von formalen Beweisen in jedem Beweisschritt einzeln sicherstellen, wodurch das Prüfen von Beweisen und der Korrektheitsbeweis für die Logik relativ einfach werden.

Bis hierin geht es in der Dissertation allein um das Programmieren mit einfachen Typen, und um Eigenschaften eines spezifischen induktiv-koinduktiven Prädikates (observational normalisation) und einer spezifischen koinduktiven Relation (observational equivalence). Beide sind in naiver Mengenlehre definiert, das heißt, dass ihre Definitionen weder auf fundamentalen Prinzipien basieren noch, dass Aussagen über sie direkt formalisiert werden können. Dies widerspricht unseren Zielen und ist einer der Gründe, warum wir die rekursive Logik entwickeln mussten. Im übrigen Teil der Dissertation lösen wir dieses Problem auf, indem wir kategorientheoretische und typentheoretische Ansätze entwickeln, welche es uns erlauben Aussagen über induktiv-koinduktive Typen, Prädikate und Relationen formal auszudrücken. Dabei stellt sich heraus, dass induktiv-koinduktive Prädikate und Relationen am besten als sogenannte Dialgebren in Faserkategorien präsentiert werden. Dieser Ansatz erlaubt es uns mit einfachen und abhängigen Typen umzugehen. Darüber hinaus können wir in diesem Aufbau eine Klasse von strikt positiven, abhängigen Typen definieren, für die wir eine Interpretation durch initiale Algebren und finale Koalgebren von polynomiellen Funktoren angeben können. Durch dies Reduktion auf minimale Anforderungen wird es möglich, unsere abhängigen Typen in wohlbekannten Modellen zu interpretieren. Wir schließen den kategorientheoretischen

Ansatz mit einer Analyse der logischen Prinzipien, die aus einer, unter strikt positiven abhängigen Typen abgeschlossenen, Faserkategorie hervorgehen.

Im letzten Schritt konstruieren wir, den Prinzipien des kategorientheoretischen Ansatzes folgend, eine (syntaktische) abhängige Typentheorie. Daraus resultiert eine Typentheorie, die allein auf induktiv-koinduktiven Typen basiert ist und die mit wenigen Schlussregeln auskommt, in der aber trotzdem alle Operatoren (Konjunktion, Implikation, Quantoren etc.) einer Prädikatenlogik erster Stufe ausgedrückt werden können. Da diese Typentheorie ausschließlich auf Iteration und Koiteration basiert, können wir zeigen, dass jede Reduktion zu einer Normalform führt. Dies stellt, zusammen mit Konfluenz und kanonischen Normalformen, die Konsistenz der Typentheorie als Logik sicher. Wir schließen die Dissertation mit einem allgemeinen Induktionsprinzip für diese Typentheorie und einem ausführlichen Beispiel eines induktiv-koinduktiven Beweises in Agda.

Curriculum Vitae

2000 – 2006 *Abitur (university entrance qualification)*, Grammar school Gaußschule, Braunschweig.

2006 – 2007 *Alternative Civilian Service*, Umweltinformations- und Innovationszentrum (Centre for environment information and innovation), Städtisches Klinikum Braunschweig.

2007 – 2011 *Software development for train control and communication system*, BBR Verkehrstechnik, Braunschweig.

2007 – 2010 *Bachelor Computer Science*, TU Braunschweig, Bachelor's Thesis: Parallelism investigation for elliptic curve key exchange, Institute of Computer and Network Engineering.

2011 – 2012 *Research Assistant: Project VerSyKo* Institute of Theoretical Computer Science, TU Braunschweig, Supervised by Stefan Milius.

2010 – 2012 *Master Computer Science*, TU Braunschweig, Master's Thesis: Transformation of SCADE models for SMT based verification, Institute of Theoretical Computer Science.

2012 – 2013 *Internship: Research on coalgebra-based descriptions of systems*, CWI (Centre for Mathematics and Computer Science), Amsterdam, Supervised by Marcello Bonsague and Jan Rutten.

2013 – 2017 *PhD Computer Science*, Radboud University, Nijmegen. Thesis Title: Mixed Inductive-Coinductive Reasoning – Types, Programs and Logic.

2017 – *Post-Doc*, PLUME Team of the LIP, ENS Lyon, CNRS.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2018-06