A programming logic based on type theory

Citation for published version (APA):

DOI:
10.6100/IR423044

Document status and date:
Published: 01/01/1994

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 17. Sep. 2020
A Programming Logic Based on Type Theory

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op woensdag 5 oktober 1994 om 16.00 uur

door

Erik Poll

geboren te Haarlemmermeer
Dit proefschrift is goedgekeurd door de promotoren

prof dr. F. E. J. Kruseman Aretz
prof dr. H P Barendrecht

en de co-promotor

dr.ir. C Hemerik
Acknowledgements

The research on which this thesis is based was carried out while I was employed by the Dutch organisation for scientific research NWO, on the SION-project "Typed Lambda Calculi".

I would like to thank Kees Hemerik for his guidance and stimulation over the past four years, and for always insisting on more text between the formulas.

Thanks also go to the other members of the manuscript committee – Henk Barendregt, Marc Bezem, Frans Kruseman Aretz and Rob Nederpelt – for their valuable suggestions and comments on draft versions of this thesis.

Finally, I want to thank all the people I've enjoyed working with over the past four years, in particular Bert van Benthem Jutting, Tijn Borghuis, Huub ten Eikelder, Herman Geuvers, and Paula Severi.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Summary</td>
<td>iii</td>
</tr>
<tr>
<td></td>
<td>Samenvatting</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Aims and Motivations</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Overview of this thesis</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Pure Type Systems</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>PTSs</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>DPTSs</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Relation between PTSs and DPTSs</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>The Basic System: $\lambda\omega_L$</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>$\lambda\omega$ as a programming language: $\lambda\omega_\pi$</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>$\lambda\omega$ as a logic: $\lambda\omega_\mu$</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>The programming logic $\lambda\omega_L$</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Program and Proof Development</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Coupled derivation rules for function types</td>
<td>45</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Coupled derivation rules for polymorphic types</td>
<td>46</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Other datatype-constructors</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison with Program Extraction in $\lambda C$</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>Simple Extensions</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Extending PTSs with $+, \times$ and $\Sigma$</td>
<td>58</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Cartesian Products</td>
<td>59</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Disjoint Sums</td>
<td>60</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Dependent Sums</td>
<td>62</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Labeled Products</td>
<td>65</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Labeled Sums</td>
<td>67</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Properties of (D)PTSs extended with $+, \times$ and $\Sigma$</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Extension of the Programming Language</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Extension of the Programming Logic</td>
<td>76</td>
</tr>
</tbody>
</table>
Summary

Typed lambda calculi can be seen as functional programming languages or — by the Curry-Howard-de Bruijn isomorphism — as logics. The first view provides a solid basis for richly typed programming languages. The second view is the basis of several systems for the machine-assisted construction and verification of proofs. Together, the two points of view offer the possibility of using type theory as the basis for programming logics, i.e., logics in which we can reason about programs and datatypes, and in which we can prove that a program satisfies a certain specification.

One approach, made possible by the expressive power of type systems, is to identify the notions of datatype and specification, and obtain programs from constructive proofs. This approach is for instance taken in the system Coq, where programs are obtained from constructive proofs in the Calculus of Constructions, via some extraction operation [PM89a]. From a theoretical point of view this approach seems very elegant, but from a practical point of view it has the disadvantage that a program under construction can only be manipulated indirectly via its proof. Ways to avoid the need for an extraction operation have been considered in Martin-Löf's type theory [NPS90] and in Nuprl [Con86], using so-called subset types, but these have the very serious disadvantage of making typechecking undecidable.

From a practical point of view, it seems preferable to have a programming logic in which programs and correctness proofs are separate objects, and which allows the simultaneous development of programs and their correctness proofs, somewhat similar to the method advocated by Hoare and Dijkstra. Such a programming logic is the subject of this thesis.

Starting point of this thesis are the Pure Type Systems (PTSs) [Bai92], a large class of type systems which can be described in a uniform way. Apart from PTSs we also use DPTSs, which are extensions of PTSs with a definition mechanism. The extension with definitions does not increase the overall expressive power of a PTS, but it is indispensable for practical use.

We design a PTS $\lambda\omega_L$, which consists of a typed functional programming language and an associated logic in which properties of programs can be stated and proved. This programming language is the system $\lambda\omega$, which is the PTS corresponding to Girard’s system $F^\omega$. The system $\lambda\omega_L$ provides a single language for programs, datatypes, proofs, propositions and specifications, and integrates a strong type system for programs with a specification system for programs. Verification of proofs is a special case of typechecking, and is easy to implement.

Programs and correctness proof are separate, but related, objects in $\lambda\omega_L$. They can be constructed hand in hand, in a compositional manner, by stepwise refinement. It is also possible to prove the correctness of program transformation rules in the system, and use such rules to develop programs and their correctness proofs.

An implementation of the system would offer a programming environment combining a
syntax-directed editor for well-typed programs and a goal-directed proof assistant.

The basic system $\lambda \omega_L$ is extended with different language constructs that are useful for programs and datatypes. First it is extended with labelled products and sums — i.e. records and variants — and with abstract datatypes. The second extension is the inclusion of recursion, i.e. of recursive programs and recursive datatypes. This extension with recursion changes the nature of the programming language. The programs are no longer all terminating, which results in partial objects. To reason about recursive programs, the logic is extended in the style of LCF, with axioms stating that recursive programs are least fixed points.

The syntax of $\lambda \omega_L$ and its extensions is discussed in the first part of this thesis, along with the underlying design considerations, and some simple examples to illustrate the intended use of the systems. The main syntactical properties of the systems are given. These establish the soundness of the type systems, and some of them, notably strong normalisation, are important for the decidability of typechecking.

The semantics of $\lambda \omega_L$ and its extensions is treated in the second part of this thesis. Of all the systems that have been introduced we give models, which provide the justification (and the inspiration) of the different extensions.

These models provide denotational semantics for the parts of the systems used as programming languages. First, the general structure of models for these programming languages is described, by extending the Bruce-Meyer-Mitchell models [BMM90]. Then we give instantiations of these general model definitions. For the languages without recursion, these instantiations are the standard PER-models. For the language with recursion, this instantiation is a CPO-model, in which datatypes are interpreted as cpos and recursive programs as least fixpoints. This domain-theoretic model provides the basis for reasoning about recursive programs and partial objects.

For the parts of the systems used as logics, proof-irrelevance models are given, in which propositions are interpreted as truth values. These simple interpretations are all that is needed to prove consistency of the logics, and consistency of any axioms that we may want to include.
Samenvatting

Getypeerde lambda-calculi kunnen enerzijds beschouwd worden als functionele programmeertalen, en anderzijds - dankzij het Curry-Howard-de Bruijn isomorfisme - als logica's. Het eerste gezichtspunt biedt een solide basis voor programmeertalen met krachtige typesystemen. Het tweede is de basis van verschillende systemen voor computer-ondersteunde constructie en verificatie van bewijzen. Samen bieden de twee gezichtspunten de mogelijkheid om typetheorie te gebruiken als basis voor programmalogica's d.w.z. logica's voor het redeneren over programma's en datatypen, waarin bewezen kan worden dat een programma aan een bepaalde specificatie voldoet.

Een aanpak, die mogelijk gemaakt door de expressiviteit van typesystemen, is om de begrippen datatype en specifiekheid te identificeren, en programma's te verkrijgen uit constructieve bewijzen. Deze benadering wordt bijvoorbeeld gebruikt in het systeem Coq, waar programma's worden verkregen uit constructieve bewijzen in de Calculus van Constructions, via een extractie-operatie [PM89a]. Vanuit theoretisch oogpunt is deze aanpak erg elegant, maar praktisch gezien heeft het als nadeel dat een programma tijdens constructie alleen indirect gemanipuleerd kan worden via het bewijs ervan.

In Martin-Löf's typetheorie [NPS90] en in Nuprl [Con86], is gekeken naar de mogelijkheid om zo'n extractie-operatie overbodig te maken, door zgn. subset typen te gebruiken. Deze hebben echter het grote nadeel dat typering onbeslisbaar wordt.

Voor praktische programmaconstructie lijkt het beter om een programmalogica te hebben waarin programma's en correctheidsbewijzen aparte objecten zijn, die echter wel tegelijkertijd kunnen worden geconstrueerd, op een vergelijkbare manier als voorgesteld door Hoare en Dijkstra. Zó'n programmalogica is het onderwerp van dit proefschrift.

Uitgangspunt van dit proefschrift zijn de Pure Type Systems (PTSen) [Bar92], een grote klasse van typesystemen die op uniforme manier beschreven kunnen worden. Behalve PTSen worden ook DPTSen gebruikt, uitbreidingen van PTSen met een definitiemechanisme. De uitbreiding van een PTS met definities verhoogt de totale expressieve kracht van een PTS niet, maar is onmisbaar in praktisch gebruik.

We ontwerpen een PTS $\lambda_{awl}$, dat bestaat uit een getypeerde functionele programmeertaal, en een bijbehorende logica voor uitspraken over programma's en datatypen. De programmeertaal is het systeem $\lambda w$, het PTS dat overeenkomt met Girard's systeem $F^\omega$. Het systeem $\lambda_{awl}$ biedt één taal taal voor programma's, datatypen, bewijzen, propostities en specificaties, en integreert een krachtig typesysteem voor programma's met een specificatietaal voor programma's. Bewijzerverificatie is een bijzonder geval van typecontrole, en eenvoudig te implementeren.

Programma's en correctheidsbewijzen zijn aparte, maar verwante, objecten in $\lambda_{awl}$. Ze kunnen hand in hand geconstrueerd worden, op een compositionalle manier, door stapsgewijze verfijning. Ook is het mogelijk om de correctheid van programmamatrixformaties aan te tonen.
in $\omega_L$, en zulke transformaties te gebruiken bij het ontwikkelen van programma's en hun correctheidsbewijzen.

Een implementatie van het systeem zou een programmeeromgeving bieden, waarin een syntax-gestuurde editor voor goedgetypeerde programma's wordt gekoppeld met een doel-gestuurde bewijs-assistent.

Het kale systeem $\omega_L$ wordt uitgebreid met verschillende taakconstructies die nuttig zijn voor programma's en datatypes. Eerst wordt het uitgebreid met gelabelde product- en somtypen - d.w.z. records en variants - en met abstracte datatypes. Daarna wordt het systeem uitgebreid met recursieve programma's en recursieve datatypes. De programmeertaal verandert nogal met de de uitbreiding met recursie. Het is niet langer het geval dat alle programma's niet-termineren. Om over recursieve programma's en mogelijk niet-terminerende berekeningen te redeneren wordt de logica uitgebreid op de stijl van LCF, met axioma's die uitdrukken dat recursieve programma's kleinste dekpunten zijn.

The syntax van $\omega_L$ en haar uitbreidingen wordt behandeld in het eerste deel van dit proefschrift, met de overwegingen die hieraan ten grondslag liggen, en enkele eenvoudige voorbeelden om het beoogde gebruik te verduidelijken. De belangrijkste syntactische eigenschappen van de systemen komen aan de orde. Deze tonen de gezondheid van de type-systemen aan, en sommige, in het bijzonder sterke normalisatie, zijn belangrijk voor beseifbaarheid van typening.

The semantiek van $\omega_L$ en de uitbreidingen ervan komt aan de orde in het tweede deel van dit proefschrift. Van alle systemen die gedefinieerd zijn worden modellen gegeven, die de rechtvaardiging (en soms ook de inspiratie) van de verschillende uitbreidingen geven.

Voor die delen van de systemen die als programmeertaal dienen, bestaan deze modellen uit denotatief de semi-antiek. Eerst wordt de algemene structuur van modellen voor deze programmeertaal beschreven, door de Bruce-Meyer-Mitchell models [BMM90] uit te breiden. Daarna worden instantiaties van deze algemene model definities gegeven. Deze instantiaties zijn de gebruikelijke PER-modellen voor de talen zonder recursie. Voor de programmeertaal met recursie wordt een CPO-model als instantiatie gegeven, waarin datatypes als cpo's en recursieve programma's als kleinste dekpunten worden geïnterpreteerd. Dit model biedt de basis het redeneren over recursieve programma's en niet-terminatie.

Voor die delen van de systemen die als logica's dienen, worden proof-irrelevance modellen gegeven, waarin propositions als waarheidswaarden getransferd worden. Deze eenvoudige interpretaties volstaan om consistentie van logica's aan te tonen, en consistentie van axioma's die we zouden willen toewijzen.
Chapter 1

Introduction

1.1 Background

There has been a steady evolution in the type systems used in programming languages. The first programming languages, such as assembly languages and LISP, were untyped. The first higher-order languages provided a few base types, such as integers and reals. Later, programming languages offered increasingly powerful constructions with which programmers could build their own types. To meet the need for more flexible type systems, features such as subtyping and polymorphism were included, and, more recently, object-oriented languages were introduced.

Typed lambda calculi can be viewed as rudimentary, but very pure and expressive programming languages. This view provides a solid basis for richly typed (functional) programming languages. It offers a framework for the description and classification of type systems of programming languages, and often suggests improvements and generalisations (see for example [Rey85], [Car89], and [BH90]).

Typed lambda calculi can also be viewed as (constructive) logics, by the so-called Curry-Howard-de Bruijn isomorphism. Types are then regarded as propositions, and terms as their proofs. This is the view taken in AUTOMATH [dB80], Martin-Löf’s Type Theory [ML79], Edinburgh Logical Framework (LF) [HHP93], and the Calculus of Constructions [CH88]. Implementations of AUTOMATH provided mechanical verification of mathematical proofs. The other systems have also been implemented, namely in Nuprl [Con86], LEGO [LP92], and Coq [Dow91]. These implementations not only verify proofs, but also provide some computer-assistance for the construction of proofs.

The two possible interpretations of typed lambda calculi - as typed programming languages and as logics - make type theory a suitable basis for programming logics, i.e. logics in which specifications can be expressed and in which it can be proved (or disproved) that a program meets a certain specification.

The best-known example of a programming logic is Hoare’s logic. However, Hoare’s logic suffers from the mismatch between the object language for programs and the logical language, in particular where it concerns the status of variables, and from the very rigid format of the judgement (see [Apt81]). Also, the scopes of variables and assumption are not always clear. These problems makes it difficult to reason about functions and procedures.
CHAPTER 1. INTRODUCTION

The advantage of a single formalism that provides both the notions of program and datatype and the notions of proof and propositions, is that the programming language and the logic are more in tune. This means we can hope to avoid problems like the ones mentioned above.

1.2 Aims and Motivations

Our aim is the design of a programming logic, that is based on type theory, but whose language constructs and deduction rules are geared towards practical program construction. The system should offer the possibility for the mechanical verification of proofs. The ultimate goal is an implementation of the system that provides a programming environment for constructing programs and their correctness proofs.

There are several reasons for integrating the programming language and logic in one formalism.

One is that it may help to obtain a compact and homogeneous system that is as elegant and general as possible. Programs, datatypes, proofs, propositions, and specifications can all be handled in a uniform way.

Also, certain primitives in the programming language will require matching primitives in the logic. For instance, to reason about programs of the form \((\lambda x. \text{nat} \ldots)\), we probably need universal quantifications of the form \((\forall x. \text{nat} \ldots)\). By working in one formalism, we can hope that the primitives in programming language and logic are well-matched.

An added bonus is that proof checking is the same as type checking, so that the same algorithms can be used for type checking and proof checking.

The most important design consideration is the expressivity of the type system for programs. The remainder of this section is dedicated to the motivation of our choice. We compare some of the alternatives, and consider the different kinds of programming logics they result in.

There is a whole spectrum of type systems for programming languages. At one end of the spectrum are the untyped programming languages. Here we effectively have just one type, which is the type of all programs. At the other end of the spectrum are very expressive type systems, such as Martin-Löf's Type Theory and the Calculus of Constructions. Here types are complete specifications, and the type of a program provides all the relevant information about its (functional) behaviour. Most of the conventional programming languages lie somewhere between these two extremes. The type of a program gives some information about the program, but not all of it.

In all but the most expressive type systems, a type alone does not suffice as the specification of a program. Apart from the type, some additional information is needed to tell which programs of that type have all the required properties. This information can be regarded as a predicate with that type as domain, which characterises a subset of that type.

Regardless of how expressive the type system of the programming language is, we can say that a specification is a pair that consists of

- a type, and
- a predicate on that type.
A solution of a specification is a pair that consists of

- a program of the right type, and
- a proof that this program satisfies the predicate.

It depends on the strength of the type system how much of a specification is expressed by the type, and how much this leaves to be expressed by the predicate. In very strong type theories all information we may want to express in a specification can be expressed in the type, so no predicate is needed. In untyped languages no information can be expressed in the type, and the whole specification is given by the predicate. And in most conventional programming languages, some information can be expressed by the type, but the larger part of a specification is expressed by the predicate. In fact, there may be some overlap between the information given by the type and the predicate.

So we can make a (rough) distinction between three kinds of programming logics:

(i) An untyped programming language with all information about programs expressed in predicates.

(ii) A typed programming language, with some information about programs expressed in types, and the rest in predicates.

(iii) A typed programming language with a very expressive type system, where all information about programs is expressed in their types.

In [Gir86] the first two are called external programming logics and the last kind is called an integrated programming logic. We use the term internal instead of integrated. In external programming logics, specifications and correctness proofs live outside of the programming language. In internal programming logics, these notions are internal to the programming language; the type of a program is its specification, and type checking amounts to verification.

A comparison of these different kinds of programming logics is given in [Dyb90].

Regardless of how the specification is divided over type and predicate, the total amount of work it takes to establish correctness of an algorithm is roughly the same. From a pragmatic point of view, however, there are big differences between the different approaches. The programming logics in this thesis are all of the kind (ii). Basically, in types as much information is expressed as can be mechanically verified. Below we motivate our preference for (ii) over (i) and (iii).

Why types?

To explain why we do not favour approach (i), we briefly review the reasons for having types in a programming language.

The essential property of types is that they impose restrictions on the interaction between objects. The type of an object controls the access to that object and the ways it can be used. As a result, some of its properties can be hidden from the outside world. For this it is vital that all type constraints can be verified mechanically by a type checking algorithm.

One advantage of types concerns the implementation of programming languages. In a typed language, we can take advantage of efficient machine representations for certain base
int, ver)

The other advantages of types concern the use of the programming language. The type system impose restrictions on the combinatorial freedom. These restrictions can prevent simple mistakes, e.g. multiplying a boolean with a natural number.

More important for the construction of programs is the way in which data-structuring helps to organise programs. The classification of programs and data that is provided and enforced by the type system helps to structure programs, and makes programs easier to read.

Finally, there is the possibility of data-abstraction. In a typed programming language abstract datatypes can be used to hide information about one part of a program form the rest of the program.

The type of a program can be seen as an incomplete specification of a program. For example, if \( f : \texttt{int} \rightarrow \texttt{int} \), then \( f \) is a function that maps integers to integers. This is of course a very incomplete specification - we still have no clue as to what \( f \) is , but such incomplete specifications have the advantage of being very easy to write and to read. Moreover, they can easily be verified, and a type-checker can guarantee that these incomplete specifications are met.

**Why not specifications as types?**

Most research on programming logics based on type theory concerns systems of kind \((iii)\). Here the notions of type and specification are identified, as are the notions of program and correctness proof.

This idea dates back to Heyting's semantics of constructive proofs: a constructive proof \( f \) of \( \forall x \in A. \, \text{pre}(x) \Rightarrow \exists y \in B. \, \text{post}(x, y) \) contains an algorithm which, given a term \( x \) and a proof of \( \text{pre}(x) \), returns a term \( y \) and a proof of \( \text{post}(x, y) \). The proposition gives all the relevant information about this algorithm it is its specification.

This approach to program construction is attractive at first sight. However, the problem is that constructive proofs have taken the place of programs, and it is difficult to separate the algorithmic part of these constructive proofs from the correctness part. Typically, the larger part of the constructive proof \( f \) above will be devoted not to the construction of the witness \( y \), but to the proof of \( \text{post}(x, y) \). This proof is important for the correctness, but not for the algorithm.

It is possible to obtain programs from proofs by an "extraction" operation. In [PM89a] such an extraction procedure is defined for the Calculus of Constructions. However, this leaves the problem that the program that will eventually be extracted from a proof is not visible during the construction of that proof. Consequently, design choices can only be made on the basis of the specification, and not on the basis of an operational understanding of the algorithm or efficiency considerations.

In Martin-Löf's Type Theory there is an alternative to program extraction. So-called subset-types can be used to discard the correctness parts of a proof (see [NPS90]). However, this has the - very high - price of making typing undecidable. This should not come as a surprise: we cannot expect that it is decidable whether a program satisfies a specification. The undecidability of type-checking means that there is little practical difference between this approach and approach \((i)\). In fact, it can be argued that these systems are no longer
internal programming logics, because, although specifications live inside the system, proofs that programs have certain types — correctness proofs — live outside it.

There are an important differences between (ii) and (iii). When types are complete specifications, some of the advantages of having types that were mentioned earlier are lost. Types are no longer easy to read and write, and type constraints are no longer easy to verify.

One practical difference occurs in situations where we do not have a complete specification for a program. In practice, such situations occur frequently, because for large programs complete correctness proofs are not feasible, and not even all that interesting. Typically, our only concern is the correctness of relatively small "algorithmic" parts of the program. Approach (ii) is then much more convenient than (iii). It gives us the freedom to choose for which parts of a program we give complete specifications and correctness proofs. For the rest of the program we use the incomplete specifications provided by the type system.

During the construction of a program the specification may change. An advantage of the distinction between type information and the logical information — given by the predicate — in approach (ii) is that any changes in the specification are more likely to be in the predicate than in the type. This also affects the re-usability of programs and types. The same type can be used in different situations, with different predicates. For example, a type of trees can be used with different predicates, for instance predicates expressing that the trees are sorted or balanced.

1.3 Overview of this thesis

As starting point we use Pure Type Systems (PTSs), as described for instance in [Bar92]. In particular, we use the PTSs in Barendregt's λ-cube, which give the fine-structure of the Calculus of Constructions. The nice thing about PTSs is that many type systems can be defined as PTSs, and that these definitions are very compact. This makes it easy to compare systems, and is also useful to design a type system for a particular purpose. Another advantage of using PTSs is that their syntactic theory is well-developed.

Apart from PTSs, we also use DPTSs. A DPTS is the extension of a PTS with a definition-mechanism. This extension does not increase the overall expressive power of a PTS, but it is indispensable for practical use.

We design a PTS λωL that can be used as a programming logic. In this PTS both programs and their correctness proofs can be handled as related but distinguishable objects. There is a strict separation between the programming language and the logic; expressions in the programming language cannot depend on logical expressions. Of course, logical expressions can depend on expressions from the programming language.

Two extensions of this basic system are given. The reason for these extensions is pragmatic: we want to include more language constructs that are useful for programs and datatypes in the programming language. First the system is extended with more type-constructors (namely +, × and Σ), and then it is extended with recursion. For the logic associated extensions are needed for reasoning about these new language constructs.

Three aspects of the basic system and its extensions can be distinguished: syntax, pragmatics and semantics.
CHAPTER 1. INTRODUCTION

Syntax

Of all the systems the most important syntactic properties are given, such as subject reduction and uniqueness of types. These properties are the basic "soundness" criteria for any type system. For PTSs most of these properties are well-known, and for the different extensions they can be proved in the same way. Of vital importance for the decidability of typechecking is the property of strong normalisation of type expressions.

Pragmatics

Although programs and correctness proof are separate objects, we want to be able to construct programs and their correctness proofs together. For this, syntax-directed (or compositional) rules can be used. The programming language provides typing rules that deduce the type of a program from the types of its component parts. In the logic matching proof rules can be derived, that deduce a property of that program from properties of its component parts. With these coupled derivation rules, programs and their correctness proofs can be constructed hand in hand.

Instead of using these compositional rules, properties of programs can also be proved using equalities between programs, i.e. using correctness-preserving transformations. The powerful quantifications that are available make it possible to express and prove such transformation rules inside the formal systems.

Semantics

The semantics of the systems is treated in the second part of the thesis, in the chapters 6, 7 and 8. It is always done using the following three steps:

For each of the programming languages we first give a general model definition, which describes the general structure of a model for the programming language. These general model definitions are extensions of the BMM-environment models given in [BMM90].

Then an instantiation of this general model definition is given. For this, two kinds of models will be used, PER-models and CPO-models. For the programming languages without recursion the more or less standard PER-models are used. For the programming language with recursion a CPO-model is used, in which all datatypes are interpreted as cpos and recursive programs are interpreted as least fixed-points.

Based on the general model definitions for the programming languages, simple models for the associated logics are given. These are so-called proof-irrelevance models, in which propositions are simply interpreted as truth values. These simple models for the logics suffice to prove the consistency of the logics, and of any axioms that are added.

Short description of the chapters

We conclude this introduction with a short description of the individual chapters.

In chapter 2 we give a short introduction to Pure Type Systems (PTSs) and their most important properties. We also define DPTSs, PTSs extended with a definition-mechanism, and give their main properties.

The rest of the thesis is divided in two parts. The first part concerns the syntax and pragmatics of the different systems, the second part the semantics.
1.3. OVERVIEW OF THIS THESIS

In chapter 3 the basic system, the PTS $\lambda\omega_L$, is defined. It is a PTS that comprises a programming language and an associated logic in which properties of programs and datatypes can be expressed and proved. The programming language is $\lambda\omega$, which is the PTS that correspond to Girard’s system $F''$ $[\text{Gir72}]$. It is a higher-order extension of the Girard-Reynolds second-order (or polymorphic) lambda calculus. The whole system $\lambda\omega_L$ can be regarded a refinement of the Calculus of Constructions.

In the next two chapters, extensions of the programming language and associated extensions of the programming logic are given.

The first extension, given in chapter 4, is the inclusion of so-called $+$, $\times$ and $\Sigma$-types. In the programming language these can be used as labelled products and sums – i.e. records and variants – and abstract datatypes. In the logic they can be used for conjunction, disjunction and existential quantification.

The second extension, given in chapter 5, is the extension of the programming language with recursive datatypes and recursive programs. Programs are then no longer guaranteed to terminate. Domain theory is used as the basis for reasoning about recursive programs and non-termination. So, axioms are introduced in the logic to reason about recursive programs in the style of LCF $[\text{GMW79}]$.

In the next three chapters, we consider the semantics of the different systems that have been introduced.

Chapter 6 treats the semantics of the basic system introduced in chapter 3. A general model definition for the programming language is given, which describes the general structure of a $\lambda\omega$-model. It is shown that the standard PER-model is indeed an instantiation of this general model definition. For any instantiation of the general model definition, a simple proof irrelevance model suffices for the logic. This means that propositions are interpreted as truth values, and all proofs are identified. This interpretation of the logic can be used to show consistency of the logic and of axioms that are added to the logic, e.g. the axiom for classical logic.

In chapter 7 the general model definition is extended to incorporate the extensions of the programming language given in chapter 4, i.e the type constructors $+$, $\times$ and $\Sigma$. Again, it is shown that the standard PER-model provides an instantiation of the general model definition, and a proof irrelevance semantics of the logic is used to prove consistency of the logic and of axioms that have been added.

In chapter 8 the general model definition is extended to incorporate the extensions of the programming language given in chapter 5, i.e. recursion. We now give a completely different instantiation of the general model definition, namely a CPO-model, in which datatypes are interpreted as cpos, and the fixed-point operator as the least fixed-point operator on cpos. As before, a proof irrelevance semantics of the logic is used to prove consistency of the logic and of axioms that are added.

Finally, in chapter 9, we give a comparison with related work, and suggest some directions for further research.
Chapter 2

Pure Type Systems

Pure Type Systems (PTSs) provide a way of describing a large class of type systems in a uniform way. They were introduced by S. Berardi [Ber88] and J. Terlouw [Ter89] as a generalisation of the systems in Barendregt's lambda cube [Bar92]. The notion of PTS is useful for two reasons. Firstly, some meta-theoretical properties can be proved for all PTSs or for large classes of PTSs at the same time. Secondly, many type systems can be described as PTSs, and these compact and very precise descriptions make it possible to compare and classify them. They also make it easy to design a type system for a particular purpose, as will be done in chapter 3.

The "pureness" of PTSs lies in the fact that there is only one type constructor, namely the dependent product \( \Pi \) (which generalises the function space \( \rightarrow \)), and one reduction rule, namely \( \beta \)-reduction. As a result, PTSs are very bare type systems. In the course of this thesis we will extend PTSs with more type constructors and associated reduction rules, but the essence of these systems will already be contained in their underlying PTS.

One shortcoming of PTSs is that they do not provide a way to introduce definitions, i.e. abbreviations for terms. Such a facility does not increase the overall expressive power, but it is essential for practical use. Indeed, all implementations of PTSs, such as Coq [Dow91], LEGO [LP92], and CONSTRUCTOR [Hel91], do provide a definition mechanism, even though the formal definitions of the type systems they implement do not. This in contrast to the AUTOMATH systems [dB80], where the definition mechanism is explicitly considered as part of the formal system.

For this reason DPTSs - extensions of PTSs with a definition mechanism - were introduced in [SP93] [SP94]. For every PTS there is a corresponding DPTS. In some respects, these are the same type system, and depending on the situation it may be preferable to look at one or the other. The PTS provides a more abstract view, which is useful for talking about a system. But for working in a system the definition mechanism provided by the DPTS is needed.

The extension of a PTS with definitions is not as harmless as it may appear at first sight. In particular, it is an open problem whether the extension always preserves the property of strong normalisation. For the PTSs used in this thesis, we do prove that the extension with definition preserve strong normalisation.

In section 2.1, the definition of PTSs and their most important properties are given. In section 2.2, the definition of DPTSs and their most important properties are given. Finally, in section 2.3, the relation between PTSs and DPTSs is discussed.
2.1 PTSs

In this section we define Pure Type Systems and list their most important properties. For a more comprehensive discussion of PTSs we refer to [BH90], [Bar91] or [Bar92].

A PTS is a typed lambda calculus. It can be defined as a 4-tuple consisting of a set of pseudoterms, a set of pseudontexts, a reduction relation on pseudoterms, and most importantly - a typing relation. This typing relation is defined by a set of inference rules for deriving judgements of the form

\[ \Gamma \vdash a : A \]

which is read as "a has type A in context \( \Gamma \)" or "a is an inhabitant of A in context \( \Gamma \)"

2.1 Definition. A specification of a PTS is a triple \((S, A, R)\) with

- \( S \) is a set of symbols called the sorts,
- \( A \subseteq S \times S \), a set of axioms of the form \( s : s' \),
- \( R \subseteq S \times S \times S \), a set of rules of the form \((s_1, s_2, s_3)\).

We write \((s_1, s_2)\) for a rule \((s_1, s_2, s_3) \in R\) if \( s_2 = s_3 \). All PTSs mentioned in this thesis will only have rules of the form \((s_1, s_2)\).

The sorts in \( S \) are the universes of the type system. In specifications given later these may include a sort \( * \) for the universe of all types, a sort \( \ast_0 \) for the universe of all datatypes, or a sort \( \ast_p \) for the universe of all propositions. The axioms in \( A \) establish a hierarchy between the universes. The rules in \( R \) control the dependencies between inhabitants of the different universes, by controlling the abstractions and quantifications that are allowed.

The PTS specified by \( S = (S, A, R) \) is denoted by \( \lambda S \). The pseudoterms of a PTS are defined below. There is just one collection of pseudoterms, so there is no a priori distinction between terms and types (or other "levels" of expressions). This is an important difference between the definition of a type system as a PTS and a more ad-hoc definition. When a particular PTS is defined, we will generally distinguish different sets of pseudoterms for the different levels of expressions.

2.2 Definition (Pseudoterms of a PTS) The set of pseudoterms \( T_{\lambda S} \) of a PTS \( \lambda S = \lambda (S, A, R) \) is defined by (we usually just write \( T \))

\[ T := \text{Var} \mid S \mid (TT) \mid (\lambda \text{Var}. T) \mid (\Pi \text{Var}. T) \]

where \( \text{Var} \) is the set of variables. We assume there are disjoint subsets \( \text{Var}_{\{s\}} \) of \( \text{Var} \) for all \( s \in S \).

Both \( \lambda \) and \( \Pi \) bind variables in \((\lambda x : A \ b)\) and \((\Pi x : A \ b)\) occurrences of \( x \) in \( b \) are bound. Free and bound variables are defined as usual. \( \text{FV}(A) \) denotes the set of variables occurring free in a term \( A \). We take the usual " sloppy" approach to bound variables: terms that are equal up to the renaming of bound variables are identified, and we assume that in all expressions the bound variables are distinct from the free variables. We write \( A = B \) if \( A \) and \( B \) are equal up to renaming of bound variables, and \( A[x = B] \) for \( A \) with \( B \) substituted for the free occurrences of \( x \).
2.3 Definition (Pseudocontexts of a PTS)
The set of pseudocontexts $C_{\lambda S}$ of a PTS $\lambda S = \lambda(S, A, R)$ is defined by (we just write $C$)

- $\epsilon \in C$,
- $\Gamma, x : A \in C$ if $\Gamma \in C$, $A \in T$, $x \in \text{Var}$ is $\Gamma$-fresh and $x \not\in \text{FV}(A)$.

Here $\epsilon$ denotes the empty context, and a variable $x$ is called $\Gamma$-fresh if $x \not\in \{y\} \cup \text{FV}(B)$ for all $y : B$ occurring in $\Gamma$.

The prefix "pseudo" is used because only the pseudoterms and pseudocontexts that are in the typing relation (i.e. those that are well-typed) will ultimately be of interest. These pseudoterms and pseudocontexts will be called the terms and the contexts.

2.4 Convention.

- $s, s', s_1, s_2, \ldots$ range over $S$.
- $\tau, y, z$ range over $\text{Var}$
- $a, b, \ldots$ and $A, B, \ldots$ range over $T$.
- $\Gamma, \Gamma', \Gamma_1, \ldots$ range over $C$

- The usual conventions for omitting parentheses are used:
  - Application associates to the left, so $a_1 a_2 \ldots a_n$ is $((\ldots((a_1 a_2) a_3) \ldots) a_n)$
  - The scope of a $\lambda$- or $\Pi$-abstraction extends to the right as far as the first unmatched closing parenthesis. So $(\lambda x : A \ b \ c)$ is $(\lambda x : A \ (b \ c))$, and not $((\lambda x : A \ b) \ c)$.
  - Outermost parentheses may be omitted.

2.5 Definition. The reduction relations $\triangleright_{\beta} \subseteq T \times T$ and $\triangleright_{\eta} \subseteq T \times T$ - $\beta$ and $\eta$-reduction are defined as usual by

$$
(\lambda x : A \ b) \ a \ \triangleright_{\beta} \ b[x = a]
$$

and all the compatibility rules

$\triangleright_{\beta}$ and $\triangleright_{\eta}$ are the reflexive and transitive closures of $\triangleright_{\beta}$ and $\triangleright_{\eta}$. $\simeq_{\beta}$ and $\simeq_{\eta}$ are the reflexive, transitive and symmetric closures of $\triangleright_{\beta}$ and $\triangleright_{\eta}$.

In the rest of this chapter $\eta$-reduction does not play a role. We only discuss the theory of PTSs with just $\beta$-reduction. The theory of PTSs with $\beta$ and $\eta$-reduction, treated in [Gen93], is considerably more complicated. This is because unlike $\beta$-reduction, $\eta$-reduction is not Church-Rosser on the set of pseudoterms.

2.6 Theorem (CR$_{\beta}$) $\beta$-reduction is Church-Rosser (or confluent), i.e.

$$
\forall A, B \in T. \ [A \ \simeq_{\beta} B \Rightarrow \exists C \in T. \ [A \ \triangleright_{\beta} C \land B \ \triangleright_{\beta} C]]
$$
2.7 Definition (Typing relation of a PTS)
The typing relation $\Gamma \vdash \lambda S \vdash \cdot \subseteq C \times T \times T$ of a PTS $\lambda S = \lambda(S, A, R)$ is the smallest relation closed under the following type inference rules (we just write $\vdash$ if it is clear which $\lambda S$ is meant).

- **Axiom**
  \[
  \epsilon \vdash s_1 : s_2 \quad \text{if } s_1, s_2 \in A
  \]
  \[
  \Gamma \vdash A : s \quad \text{if } x \in \text{Var}^B
  \]
  \[
  \Gamma, x : A \vdash x : A
  \]
  \[
  \Gamma \vdash b : B \quad \Gamma \vdash A : s \quad \text{if } x \in \text{Var}^A
  \]

- **Scan**
  \[
  \Gamma \vdash \Pi(A) \vdash (\Pi \times A) : B \quad \text{if } (s_1, s_2, s_3) \in R
  \]

- **Introduction**
  \[
  \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi \times A) : B
  \]

- **Elimination**
  \[
  \Gamma \vdash b : (\Pi \times A) \quad \Gamma \vdash a : A
  \]

- **Beta Conversion**
  \[
  \Gamma \vdash b : B \quad \Gamma \vdash B' : s \quad B \approx B' \quad \text{if } (s_1, s_2, s_3) \in R
  \]
  \[
  \Gamma \vdash b' : B
  \]

where $s$ ranges over sorts, i.e., $s \in S$. Note that by the definition of the set of pseudocontexts $C$ the variable $x$ is $\Gamma$-fresh in the rules (var), (weaken), (Scan) and (Introduction).

$\Gamma \vdash A : B : C$ is short for $\Gamma \vdash A \quad B$ and $\Gamma \vdash B \quad C$.

If $x$ does not occur free in $B$, $(\Pi \times A) : B$ is written as $A \rightarrow B$. It is easy to see that the following rules are derivable:

- **Introduction**
  \[
  \Gamma, x : A \vdash b : B \quad \Gamma \vdash A \rightarrow B : s \quad \text{if } (s_1, s_2, s_3) \in R
  \]

- **Elimination**
  \[
  \Gamma \vdash b : (\Pi \times A) \quad \Gamma \vdash a : A
  \]

In a given PTS there may be rules $(s_1, s_2, s_3) \in R$ for which the rule (Scan) is more general than needed, and just the rule (Introduction) would suffice.

As already mentioned, the typing relation selects the terms and contexts from the pseudoterms and pseudocontexts.

2.8 Definition (Terms, contexts, well-typedness, $\Gamma \vdash A$, $\Gamma \vdash a : A$).

1. A pseudoterm $A$ is a $(\lambda S)$-term if $\Gamma \vdash A \quad B$ or $\Gamma \vdash B : A$ for some $\Gamma$ and $B$.
2. A pseudcontext $\Gamma$ is a $(\lambda S)$-context if $\Gamma \vdash A \quad B$ for some $A$ and $B$.
3. A term $A$ is well-typed in context $\Gamma$ if $\Gamma \vdash A \quad B$ or $\Gamma \vdash B : A$ for some $B \in T$.
4. A term $A$ is inhabited in context $\Gamma$ if $\Gamma \vdash a : A$ for some $a \in T$. 

A term \(a\) is *typable* in context \(\Gamma\) written \(\Gamma \vdash a : -\) if \(\Gamma \vdash a : A\) for some \(A \in T\). □

So a pseudoterm is a term if it is well-typed in some context, and a term is well-typed in some context if it is typable or inhabited in that context. There can be well-typed terms that are not typable, namely sorts which are at the highest level in the hierarchy given by the axioms \(A\), i.e., those \(s \in S\) for which \(\neg(\exists s' \in S. (s : s') \in A)\). The distinction between well-typed and typable only plays a role in some technical proofs about PTSs.

We will sometimes be interested in the sub-context containing only the declarations of variables that belong to certain sorts.

2.9 **DEFINITION.** For a context \(\Gamma\) and \(s_1, \ldots, s_n \in S\), the pseudoterm \(\Gamma^{s_1, \ldots, s_n}\) is defined by
\[
\varepsilon^{s_1, \ldots, s_n} \equiv \varepsilon \\
(\Gamma, x : A)^{s_1, \ldots, s_n} \equiv \begin{cases} 
\Gamma^{s_1, \ldots, s_n}, x : A & \text{if } x \in \operatorname{Var}^{s_i} \text{ for some } i \in \{1, \ldots, n\} \\
\Gamma^{s_1, \ldots, s_n} & \text{otherwise}
\end{cases}
\]

**Properties of PTSs**

Below we list some of the basic properties of PTSs. Proofs of these properties can be found in [GN91] or [Bar92].

The first two lemmas state that the context behaves as expected.

2.10 **LEMMA (Start).** Let \(\Gamma\) be a context. Then
\[
1. \text{If } s : s' \in A, \text{ then } \Gamma \vdash s : s'.
2. \text{If } x : A \in \Gamma, \text{ then } \Gamma \vdash x : A
\]

2.11 **LEMMA (Weakening).** If \(\Gamma\) is a context and \(\Gamma' \vdash a : A\) for some \(\Gamma' \subseteq \Gamma - \text{i.e. } x : B \in \Gamma\) for all \(x : B \in \Gamma'\), then \(\Gamma \vdash a : A\)

The next lemma is used to prove properties by induction on the structure of terms.

2.12 **LEMMA (Generation)**
\[
1. \text{If } \Gamma \vdash s \ C \text{ then } C \equiv s' \text{ for some } (s : s') \in A.
2. \text{If } \Gamma \vdash x : C \text{ then } C \equiv A \text{ for some } x : A \text{ in } \Gamma.
3. \text{If } \Gamma \vdash (\Pi x : A \ B) : C \text{ then } C \equiv (s_1, s_2, s_3), \Gamma \vdash A = s_1 \text{ and } \Gamma, x : A \vdash B \equiv s_2 \text{ for some } (s_1, s_2, s_3) \in R.
4. \text{If } \Gamma \vdash (\lambda x : A \ b) : C \text{ then } C \equiv (\Pi x : A \ B), \Gamma \vdash A = s_1, \Gamma, x : A \vdash b : B \text{ and } \Gamma, x : A \vdash B : s_2 \text{ for some } B \in T \text{ and } (s_1, s_2, s_3) \in R.
5. \text{If } \Gamma \vdash ba : C \text{ then } C \equiv B[x := a], \Gamma \vdash b : (\Pi x : A \ B) \text{ and } \Gamma \vdash a : A \text{ for some } x \in \operatorname{Var} \text{ and } A, B \in T
\]

If a term is inhabited then it is a sort or it is typable with a sort.
2.13 Lemma (Correctness of Types) If \( \Gamma \vdash a : A \) then \( A \in S \) or \( \Gamma \vdash A : s \) for some \( s \in S \).

The collection of inhabitants of a type is closed under reduction:

2.14 Lemma (Subject Reduction : \( \text{SR}_\beta \)) If \( \Gamma \vdash b : B \) and \( b \triangleright_b \beta' \) then \( \Gamma \vdash \beta' : B \).

It is possible to restrict the places in a type derivation where the rule (weaken) can be applied.

2.15 Lemma (Restricted Weakening). If \( \Gamma \vdash a : A \), then there is a derivation of this in which all applications of the rule (weaken) are of the form

\[
\frac{\Gamma \vdash b : B \quad \Gamma \vdash A : s \quad x \in \text{Var}^* \text{ and } b \in S \cup \text{Var}}{\Gamma, x \vdash A \vdash b : B}
\]

i.e. (weaken) is only used if the subject \( b \) is a sort or a variable.

Proof. See [Geu93].

This lemma will be useful later, when interpretations are defined by induction on type derivations, as it allows us to restrict the possible form of derivations.

Most - if not all - PTSs that are of interest are functional PTSs.

2.16 Definition. A (specification of a) PTS is called functional if

\[
(s : s') \in A \land (s : s'') \in A \Rightarrow s' = s''
\]

\[
(s_1, s_2, s_3) \in R \land (s_1, s_2, s_3') \in R \Rightarrow s_3 = s_3'
\]

The distinguishing property of the functional PTSs is that the type of a term is unique up to \( \beta \)-conversion.

2.17 Lemma (Uniqueness of Types - \( \text{UT}_\beta \) - for functional PTS). If \( \Gamma \vdash b : B \) and \( \Gamma \vdash b : B' \) then \( B \equiv_\beta B' \).

Proof. See [GN91] or [Bar92].

All PTSs mentioned in this thesis are functional. By UT we can speak of the type of a term, if we only care about its type modulo \( \beta \)-conversion.

2.18 Definition. A PTS is \( \beta \)-strongly normalising, written \( \text{SN}_\beta \) - if all its terms are, i.e. if \( \Gamma \vdash a : A \) implies that \( a \) and \( A \) are \( \beta \)-strongly normalising, which means that all \( \beta \)-reduction sequences starting in \( a \) and \( A \) terminate.

Together, CR and SN imply that convertibility of types is decidable, which is essential for decidability of type checking. Not all PTSs are SN, and SN cannot be proved for large classes of PTSs, unlike CR, SR and UT. For SN we have to consider particular PTSs.
2.19 Definition $(\lambda \rightarrow, \lambda 2, \lambda \omega, \lambda C)$. 

1. $\lambda \rightarrow$ is the PTS specified by

$$\begin{align*}
S &= \{*, \Box\}, \quad A = \{* : \Box\}, \quad R = \{(\ast, \ast)\}.
\end{align*}$$

$\lambda \rightarrow$ is Church's *simply typed* lambda calculus, introduced in [Chu40].

2. $\lambda 2$ is the PTS specified by

$$\begin{align*}
S &= \{*, \Box\}, \quad A = \{* : \Box\}, \quad R = \{(\ast, \ast), (\Box, \ast)\}.
\end{align*}$$

$\lambda 2$ is the *second order* or *polymorphic* typed lambda calculus, also known as system $F$, introduced in [Gir72] and [Rey74].

3. $\lambda \omega$ is the PTS specified by

$$\begin{align*}
S &= \{*, \Box\}, \quad A = \{* : \Box\}, \quad R = \{(\ast, \ast), (\Box, \ast), (\Box, \Box)\}.
\end{align*}$$

$\lambda \omega$ is the *higher order* typed lambda calculus, and is essentially the system $F^\omega$, introduced in [Gir72].

4. $\lambda C$ is the PTS specified by

$$\begin{align*}
S &= \{*, \Box\}, \quad A = \{* : \Box\}, \quad R = S^2.
\end{align*}$$

$\lambda C$ is the Calculus of Constructions, introduced in [CH88].

The relation between the systems is obvious from their specifications: $\lambda \rightarrow$ is a subsystem of $\lambda 2$, $\lambda 2$ is a subsystem of $\lambda \omega$, and $\lambda \omega$ is a subsystem of $\lambda C$. A disadvantage of these compact specifications is that some work is needed to get some idea of what these type systems are, e.g. what types can be formed and which abstractions are allowed. In chapter 3 the system $\lambda \omega$ and a refinement of the system $\lambda C$ will be discussed in detail.

The PTSs defined above are all strongly normalising, because $\lambda C$ -- the largest one -- is.

2.20 Theorem $\lambda C$ is SN$_{\beta}$.

This was first proved by Coquand in [Coq85]. An alternative proof is given in [GN91].

The type checking problem -- $\Gamma \vdash \alpha : A$? -- is the problem of deciding whether $\Gamma \vdash \alpha : A$ is derivable for given $\Gamma$, $\alpha$ and $A$. The type inference problem -- $\Gamma \vdash \alpha$? -- is the problem of deciding if a term $\alpha$ is typable in a context $\Gamma$ and, if so, computing its type. In [BJ93] it is proved that type checking is decidable and type inference is computable for all strongly normalising PTSs with a finite set of sorts. These proofs only provide algorithms that are far too inefficient for practical use. Efficient algorithms for some classes of PTSs can be found in [BJMP] and [Pol83].
2.2 DPTSs

DPTSs, introduced in [SP94], are PTSs extended with a definition mechanism. In a DPTS, terms and contexts can contain definitions of the form \( x = a : A \). A definition \( x = a : A \) introduces \( x \) as an abbreviation for the term \( a \) of type \( A \). A new reduction relation, called \( \delta \)-reduction, is defined to capture the notion of unfolding definitions.

Like a PTS, a DPTS is a 4-tuple consisting of a set of pseudotermso, a set of pseudocontexts, a reduction relation, and a typing relation. And like a PTS, a DPTS is specified by a triple \( S = (S, A, R) \). The DPTS specified by \( S \) is denoted \( \lambda S \).

2.21 DEFINITION (Pseudotermso of a DPTS)
The set of pseudotermso \( T_{\lambda S} \) of a DPTS \( \lambda S = \lambda (S, A, R)_\delta \) is defined by (we just write \( T \))

\[
T ::= \text{Var} \mid S \mid (T T) \mid (\lambda \text{Var} T) \mid (\Pi \text{Var} T) \mid (\text{Var} = T \; T \; \text{in} \; T)
\]

Like \( \lambda \) and \( \Pi \), \( = \) binds a variable \( x \) in \( (x = a : A \; \text{in} \; b) \) occurrences of \( x \) in \( b \) are bound. The notions of free variables, substitution, and \( \delta \)- and \( \eta \)-reduction are extended to this larger collection of pseudotermso in the obvious way. For definitions the same conventions for omitting parentheses apply as for \( \lambda \) - and \( \Pi \)-abstractions. So \( (x = a : A \; \text{in} \; b : c) \) is \( (x = a : A \; \text{in} \; (b : c)) \), and not \( ((x = a : A \; \text{in} \; b) : c) \).

2.22 DEFINITION (Pseudocontextso of a DPTS)
The set of pseudocoutextso \( C_{\lambda S} \) of a DPTS \( \lambda S = \lambda (S, A, R)_\delta \) is defined by (we usually just write \( C \))

- \( \epsilon \in C \),
- \( \Gamma, x : A \in C \) if \( \Gamma \in C \), \( A \in T \), and \( x \in \text{Var} \) is \( \Gamma \)-fresh and \( x \notin \text{FV}(A) \)
- \( \Gamma, x = a : A \in C \) if \( \Gamma \in C \), \( a \in T \), and \( A \in T \), and \( x \in \text{Var} \) is \( \Gamma \)-fresh and \( x \notin \text{FV}(a) \cup \text{FV}(A) \)

Here a variable \( x \) is called \( \Gamma \)-fresh if \( x \notin \{y\} \cup \text{FV}(B) \) for all \( y, B \) occurring in \( \Gamma \) and \( x \notin \{y\} \cup \text{FV}(b) \cup \text{FV}(B) \) for all \( y = b : B \) occurring in \( \Gamma \).

Definitions in pseudocoutextso, e.g. \( \Gamma, x = a : A, \Gamma' \), are called global definitions. Definitions in pseudotermso, e.g. \( (x = a : A \; \text{in} \; b) \), are called local definitions, sometimes called let-expressions.

2.23 DEFINITION (\( \delta \)-reduction)
The relation \( \Gamma, x = a : A, \Gamma_2 \vdash x \Delta \delta a \) - called \( \delta \)-reduction - is the smallest relation such that

- \( \Gamma, x = a : A, \Gamma_2 \vdash x \Delta \delta a \)
- \( \Gamma \vdash (x = a : A \; \text{in} \; b) \Delta \delta b \) if \( x \notin \text{FV}(b) \)

and that is closed under the following compatibility rules

- if \( \Gamma, x = a : A \vdash b \Delta \delta b' \) then \( \Gamma \vdash (x = a : A \; \text{in} \; b) \Delta \delta (x = a : A \; \text{in} \; b') \)
- if \( \Gamma, x : A \vdash b \Delta \delta b' \) then \( \Gamma \vdash (\Pi x : A \; b) \Delta \delta (\Pi x : A \; b') \)
  and \( \Gamma \vdash (\lambda x : A \; b) \Delta \delta (\lambda x : A \; b') \)
• if $\Gamma \vdash a \vartriangleright_\delta a'$ then $\Gamma \vdash (x = a : A \text{ in } b) \vartriangleright_\delta (x = a' : A \text{ in } b)$,
  $\Gamma \vdash (x = A : a \text{ in } b) \vartriangleright_\delta (x = A : a' \text{ in } b)$,
  $\Gamma \vdash (a b) \vartriangleright_\delta (a' b)$,
  $\Gamma \vdash (\Pi x : a \text{ in } b) \vartriangleright_\delta (\Pi x : a' \text{ in } b)$,
  and $\Gamma \vdash (\lambda x : a \text{ in } b) \vartriangleright_\delta (\lambda x : a' \text{ in } b)$.

We write $\Gamma \vdash a \vartriangleright_\delta b$ if $\Gamma \vdash a \vartriangleright_\delta b$ or $a \vartriangleright_\rho b$. For $\rho \in \{\delta, \beta\}$, $\Gamma \vdash - \vartriangleright_\rho -$ is the reflexive and transitive closure of $\Gamma \vdash - \vartriangleright_\rho -$, and $\Gamma \vdash - \cong_\rho -$ is the reflexive, transitive and symmetric closure of $\Gamma \vdash - \vartriangleright_\rho -$.

So, for example, $(x = a : A \text{ in } b) \vartriangleright_\delta (x = a : A \text{ in } b[x := a]) \vartriangleright_\delta b[x := a]$. The number of steps needed to $\delta$-reduce $(x = a : A \text{ in } b)$ to $(x = a : A \text{ in } b[x := a])$ is exactly the number of occurrences of $x$ in $b$.

In a $\delta$-reduction

$$\Gamma_1, x = a : A, \Gamma_2 \vdash x \vartriangleright_\delta a$$

none of the free variables of $a$ can be captured by definitions or declarations in $\Gamma_2$. This is because if $\Gamma_1, x = a : A, \Gamma_2 \in C$, then by the definition of $C$ there are no definitions $y = b : B$ or declarations $y : B$ in $\Gamma_2$ for which $y \in \text{FV}(a)$.

Expanding all definitions in a $\lambda S_\delta$-pseudoterm produces a $\lambda S$-pseudoterm:

2.24 Definition ($\deltanf$). For a pseudoterm $a \in T_{\lambda S_\delta}$ and a pseudocontext $\Gamma \in C_{\lambda S_\delta}$ the pseudoterm $\deltanf_{\Gamma}(a) \in T_{\lambda S}$ is defined as follows:

$$\deltanf_{\Gamma}(x) = \begin{cases} \deltanf_{\Gamma_1}(a) & \text{if } \Gamma \equiv \Gamma_1, x = a : A, \Gamma_2 \\ x & \text{otherwise} \end{cases}$$

$$\deltanf_{\Gamma}(s) = s$$

$$\deltanf_{\Gamma}(a b) = \deltanf_{\Gamma}(a) \deltanf_{\Gamma}(b)$$

$$\deltanf_{\Gamma}(\lambda x : A \text{ in } b) = \lambda x : \deltanf_{\Gamma}(A) \cdot \deltanf_{\Gamma,x : A}(b)$$

$$\deltanf_{\Gamma}(\Pi x : A \text{ in } B) = \Pi x : \deltanf_{\Gamma}(A) \cdot \deltanf_{\Gamma,x : A}(B)$$

$$\deltanf_{\Gamma}(x = a : A \text{ in } b) = \deltanf_{\Gamma,x = a : a}(b)$$

The definition of $\deltanf_{\Gamma}(a)$ is by induction on the number of characters in $a$ plus the number of characters in $\Gamma$ (not counting brackets).

It is not difficult to show that.

2.25 Lemma. $\deltanf_{\Gamma}(a)$ is the unique $\delta$-normal form of $a$ in $\Gamma$.

Proof. (Sketch) First we prove that $\deltanf_{\Gamma}(a)$ is a $\delta$-normal form of $a$ in $\Gamma$, by proving that $\Gamma \vdash a \vartriangleright_\delta \deltanf_{\Gamma}(a)$ and that $\deltanf_{\Gamma}(a)$ is in $\delta$-normal form. It then follows from $\Gamma \vdash a \vartriangleright_\delta a$ that $\deltanf_{\Gamma}(a)$ is the unique $\delta$-normal form of $a$ in $\Gamma$.

From the existence of unique $\delta$-normal forms it follows that $\delta$-reduction is Church-Rosser and weakly normalising. In fact, $\delta$-reduction has even stronger properties, as given by the next lemma.
2.26 Lemma.

1. \( \text{CR}_{\beta} \). \( \beta \)-reduction is Church-Rosser, i.e.
\[
\forall \Gamma \in \mathbb{C}, A, B \in \mathbb{T} \ [\Gamma \vdash A \Rightarrow_{\beta} B \Rightarrow \exists C \in \mathbb{T}. [\Gamma \vdash A \Rightarrow_{\beta} C \wedge \Gamma \vdash B \Rightarrow_{\beta} C]]
\]

2. \( \text{SN}_{\delta} \). \( \delta \)-reduction is strongly normalising

Proof. (Sketch)

1. To prove \( \text{CR}_{\beta} \) we use the fact that unique \( \delta \)-normal forms exist.

Suppose \( \Gamma \vdash a \Rightarrow_{\delta} a_1 \) and \( \Gamma' \vdash a \Rightarrow_{\delta} a_2 \). Then \( a, a_1 \) and \( a_2 \) reduce to their \( \delta \)-normal forms \( \delta \text{nf}_{\Gamma}(a), \delta \text{nf}_{\Gamma}(a_1) \) and \( \delta \text{nf}_{\Gamma}(a_2) \), respectively. By the following two properties

(i) if \( \Gamma \vdash a \Rightarrow_{\delta} a' \) then \( \delta \text{nf}_{\Gamma}(a) \equiv \delta \text{nf}_{\Gamma}(a') \)

(ii) if \( \Gamma \vdash a' \Rightarrow_{\delta} \) then \( \delta \text{nf}_{\Gamma}(a) \Rightarrow_{\delta} \delta \text{nf}_{\Gamma}(a') \)

it follows that \( \Gamma \vdash \delta \text{nf}_{\Gamma}(a) \Rightarrow_{\delta} \delta \text{nf}_{\Gamma}(a_1) \) and \( \Gamma \vdash \delta \text{nf}_{\Gamma}(a) \Rightarrow_{\delta} \delta \text{nf}_{\Gamma}(a_2) \). Finally, by \( \text{CR}_{\beta} \) there then is a common \( \beta \)-reduct of \( \delta \text{nf}_{\Gamma}(a_1) \) and \( \delta \text{nf}_{\Gamma}(a_2) \). The whole proof is best illustrated by the following diagram:

2. \( \text{SN}_{\delta} \) is proved by showing that \( \text{weight}_{\delta}(a) \in \mathbb{N} \), defined below, decreases with \( \delta \)-reduction

\[
\begin{align*}
\text{weight}_{\delta}(x) &= \left\{ \begin{array}{ll}
\text{weight}_{\Gamma_1}(a) + 1 & \text{if } \Gamma \equiv \Gamma_1, x = a, \Gamma_2 \\
0 & \text{otherwise}
\end{array} \right. \\
\text{weight}_{\delta}(s) &= 0 \quad \text{if } s \in S \\
\text{weight}_{\delta}(x = a \text{ in } b) &= \text{weight}_{\delta}(a) + \text{weight}_{\delta}(b) + \text{weight}_{\delta}(x = a, b) + 1 \\
\text{weight}_{\delta}(\lambda x. A) &= \text{weight}_{\delta}(a) + \text{weight}_{\delta}(b) \\
\text{weight}_{\delta}(\lambda x. A) &= \text{weight}_{\delta}(x = a, b) + \text{weight}_{\delta}(A) \\
\text{weight}_{\delta}(\text{let } x. A, B) &= \text{weight}_{\delta}(x = a, b) + \text{weight}_{\delta}(A) \\
\text{weight}_{\delta}(\text{let } x. A, B) &= \text{weight}_{\delta}(x = a, b) + \text{weight}_{\delta}(A)
\end{align*}
\]

Note that \( \delta \)-reduction is strongly normalising on the pseudoterm. This in contrast to \( \beta \)-reduction, which is not strongly normalising on the pseudoterm since for instance \( (\lambda x. y. x)(\lambda x. y. x) \) is a pseudoterm, but can only be strongly normalising on a subset of the pseudoterm (e.g. the \( \lambda C \)-terms).
2.27 Definition (Typing relation of a DPTS).

The typing relation $\Gamma \vdash \lambda S \vdash S \subseteq \mathbb{C} \times T \times T$ of a DPTS $\lambda S = \lambda(S, A, R)$ is the smallest relation closed under the type inference rules listed in definition 2.7 plus the following rules:

\[
\begin{align*}
(\delta \text{var}) & \quad \Gamma \vdash a : A \\
(\delta \text{weaken}) & \quad \Gamma, x = a:A \vdash x : A \\
(\delta \text{form}) & \quad \Gamma, x = a:A \vdash b : B \\
(\delta \text{intro}) & \quad \Gamma, x = a:A \vdash \lambda x.B : (\lambda x.A \in B) \\
(\delta \text{conv}) & \quad \Gamma \vdash B : B'
\end{align*}
\]

where $S$ ranges over sorts, i.e. $S \in S$.

The $(\lambda S)$-terms, the $(\lambda S)$-contexts and the notions of being well-typed, typable and inhabited in a context are defined for the DPTSs as for PTSs (see definition 2.8).

Properties of DPTSs

It is not difficult to show that most the properties that hold for arbitrary PTSs also hold for DPTSs. Proofs of the following properties can be found in [SP93].

2.28 Lemma (Start). Let $\Gamma$ be a context. Then

1. If $s : s' \in A$, then $\Gamma \vdash s \vdash s'$
2. If $x : A \in \Gamma$, then $\Gamma \vdash x : A$.
3. If $x = a : A \in \Gamma$ then $\Gamma \vdash \lambda x.A$ (and, by SR$_{\beta_6}$, also $\Gamma \vdash a : A$).

2.29 Lemma (Weakening).

If $\Gamma$ is a context and $\Gamma' \vdash a : A$ for some $\Gamma' \subseteq \Gamma$, then $\Gamma \vdash a : A$.

2.30 Lemma (Correctness of Types).

If $\Gamma \vdash a : A$ then $A \in S$ or $\Gamma \vdash a : s$ for some $s \in S$.

2.31 Lemma (Subject Reduction SR$_{\beta_6}$)

If $\Gamma \vdash B : B'$ and $\Gamma \vdash b : B \beta_6 b'$ then $\Gamma \vdash b' : B'$.

Like a PTS, a DPTS is called functional if its specification is functional (see definition 2.16).

2.32 Lemma (Uniqueness of Types - UT$_{\beta_6}$ - for functional DPTS).

For functional DPTSs if $\Gamma \vdash b : B$ and $\Gamma \vdash b : B'$ then $\Gamma \vdash B \equiv_{\beta_6} B'$. 

2.33 Definition. A DPTS is \( \beta\delta\)-strongly normalising, written \( SN_{\beta\delta} \) if all its terms are, i.e. if \( \Gamma \vdash a \cdot A \) implies that \( a \) and \( A \) are \( \beta\delta\)-strongly normalising in context \( \Gamma \). \( \square \)

Whether extending a PTS with definitions preserves strong normalisation, i.e. whether

\[ \lambda S \text{ is } SN_{\beta} \Rightarrow \lambda S_{b} \text{ is } SN_{\beta\delta} \]

is an open problem. Here the local definitions, i.e. definitions in terms (as opposed to the definitions in contexts), are the main complication. In [SP93] \( SN_{\beta\delta} \) is proved for many DPTSs, including \( \lambda C_{b} \), the Calculus of Constructions extended with definitions. Here we just give a sketch of the proof.

2.34 Theorem. \( \lambda C_{b} \) is \( SN_{\beta\delta} \)

Proof. (Sketch) This is proved by showing there is a function which maps an infinite \( \beta\delta \)-reduction sequence in \( \lambda C_{b} \) to an infinite \( \beta \)-reduction sequence in Luo's extended Calculus of Constructions, the system ECC. Strong normalisation of ECC is proved in [Luo89].

The functions \( \{ \cdot \}_f \in T_{\lambda S_b} \times \mathcal{L}_{\lambda S_b} \to T_{\lambda S_b} \) and \( \{ \cdot \}_c \in \mathcal{L}_{\lambda S_b} \to \mathcal{L}_{\lambda S_b} \) are defined as follows:

\[
\begin{align*}
\{a \}^c_r & = \begin{cases}
\{a\}_c & \text{if } r \\
\{x\}_c & \text{otherwise}
\end{cases} \\
\{a \}_f & = \begin{cases}
\{a\}_f & \text{if } c \in S \\
\{a\}_f & \text{otherwise}.
\end{cases} \\
\{a \ b\}_r & = \{a\}_c \{b\}_c \\
\{\lambda x . A . b\}_r & = \lambda x \{A\}_c \{b\}_c x A \\
\{\Pi x . A . B\}_r & = \Pi x \{A\}_c \{B\}_c x A \\
\{x = a . A \ in \ b\}_r & = \{\lambda x . A\}_c \{b\}_c x a \{a\}_c \\
\{c\} & = \{c\}_c \\
\{\Gamma , x : A\}_r & = \{\Gamma\}_c , x \{A\}_c \\
\{\Gamma , x = a . A\}_r & = \{\Gamma\}_c , x \{A\}_c \\
\end{align*}
\]

The only difference between \{ \cdot \}_c and \{ \cdot \}_f is in the clause for terms of the form \( (x = a . A \ in \ b) \). This difference is crucial for property (i) below.

For this mapping we prove the following properties.

(i) If \( a \triangleright_{\beta\delta} b \) then \( \{a\}_r \triangleright_{\beta\delta} \{b\}_r \) for all \( \Gamma \), where \( \triangleright_{\beta\delta} \) is the transitive closure of \( \triangleright_{\beta\delta} \).

(ii) If \( \Gamma \vdash a \triangleright_{\beta\delta} b \) then \( \{a\}_r \triangleright_{\beta\delta} \{b\}_r \), where \( \triangleright_{\beta\delta} \) is the reflexive closure of \( \triangleright_{\beta\delta} \).

(iii) If \( \Gamma \vdash a \cdot A \) in \( \lambda C_{b} \) then \( \Gamma \vdash \{a\}_f \{A\}_r \) in ECC.

It follows from \( SN_{\beta} \) that any infinite \( \beta\delta \)-reduction sequence has an infinite number of \( \beta \)-steps. So, by (i) and (ii), \( \{ \cdot \}_c \) maps an infinite \( \beta\delta \)-reduction sequence to an infinite \( \beta \)-reduction sequence. By (iii), \( \{ \cdot \}_c \) maps \( \lambda C_{b} \)-terms to ECC-terms. \( SN_{\beta\delta} \) of \( \lambda C_{b} \) then follows from \( SN_{\beta} \) of ECC. \( \square \)
2.3 Relation between PTSs and DPTSs

Apart from the question whether the DPTS $\lambda S_6$ has all the meta-theoretical properties that the PTS $\lambda S$ has, it is important to understand the relationship between the two systems. In other words, what is the relation between statements of the form $\Gamma \vdash a : A$, $\Gamma \vdash a : \_ \_ \or \_ \vdash a : A$ that hold in $\lambda S$ and $\lambda S_6$? To answer these questions, the definition of $\delta$-normal form is extended to contexts.

2.35 Definition. The $\delta$-normal form $\delta nf(\Gamma) \in C_{\lambda S}$ of a pseudocontext $\Gamma \in C_{\lambda S_6}$ is defined as follows

\[
\delta nf(\epsilon) \equiv \epsilon \\
\delta nf(\Gamma, x : A) \equiv \delta nf(\Gamma), x : \delta nf_f(A) \\
\delta nf(\Gamma, x = a : A) \equiv \delta nf(\Gamma)
\]

Definitions can be eliminated in $\lambda S_6$-type derivations to produce $\lambda S$-type derivations:

2.36 Theorem (Elimination of definitions).
If $\Gamma \vdash_{\lambda S_6} a : A$ then $\delta nf(\Gamma) \vdash_{\lambda S} \delta nf_f(a) : \delta nf_f(A)$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\lambda S_6} a : A$.

This means that introducing definitions does not increase the expressive power of the system. An immediate consequence is that the extension of a PTS with definitions is conservative. In fact, there are three possible notions of conservativity. Apart from conservativity for statements of the form $\Gamma \vdash a : A$, we also have conservativity for statements of the form $\Gamma \vdash a : \_ \_ \or _ \vdash a : A$, as defined in definition 2.8.

2.37 Corollary (Conservativity of $\lambda S_6$ over $\lambda S$).
Let $\Gamma$ be a $\lambda S$-context and $a$ and $A$ be $\lambda S$-terms. Then

1. $\Gamma \vdash_{\lambda S_6} a : A \iff \Gamma \vdash_{\lambda S} a : A$
2. $\Gamma \vdash_{\lambda S_6} a : \_ \_ \iff \Gamma \vdash_{\lambda S} a : \_ \_ \_\_ \_\_\_$
3. $\Gamma \vdash_{\lambda S_6} a : \_ \_ \iff \Gamma \vdash_{\lambda S} a : \_ \_ \_\_\_\_\_$

Part 3 of this corollary is the usual notion of conservativity if we think of types as propositions and their inhabitants as proofs. It states that a $\lambda S$-term $A$ is inhabited - i.e. provable - in a $\lambda S$-context $\Gamma$ in $\lambda S_6$ if $A$ is inhabited in $\Gamma$ in $\lambda S$.

For functional systems, there is even a stronger relation between $\lambda S$ and $\lambda S_6$. Theorem 2.36 says that, modulo $\delta$-conversion, a $\lambda S_6$-term does not have more types or inhabitants in $\lambda S_6$ than its $\delta$-normal form has in $\lambda S$. The following theorem says that, modulo $\delta$-conversion, a $\lambda S_6$-term does not have fewer types or inhabitants in $\lambda S_6$ than its $\delta$-normal form has in $\lambda S$, provided $S$ is functional.

2.38 Theorem (Introduction of definitions).
Let $S$ be a functional specification, and let $a$ and $A$ be well-typed in context $\Gamma$ in $\lambda S_6$ (and so $\Gamma$ is a $\lambda S_6$-context).

If $\delta nf(\Gamma) \vdash_{\lambda S} \delta nf_f(a) : \delta nf_f(A)$ then $\Gamma \vdash_{\lambda S_6} a : A$.

(In example 2.40 it is shown why the restriction to functional systems is needed.)
Proof. First we show that the rule

\[(\beta\delta\text{RED}) \quad \Gamma \vdash b : B \quad \Gamma \vdash B \bowtie_{\delta\delta} B' \quad \frac{}{\Gamma \vdash b : B'}\]

is admissible in \(\lambda S_\delta\).

Suppose \(\Gamma \vdash b : B\) and \(\Gamma \vdash B \bowtie_{\delta\delta} B'\). To prove \(\Gamma \vdash b : B'\) we only have to prove that \(\Gamma \vdash B' : s\) for some sort \(s\), because we have the rule \((\beta\delta\text{conv})\). By correctness of types \((2.30)\) it follows from \(\Gamma \vdash b : B\) that \(\Gamma \vdash B : s\) or \(B \in S\), but \(\Gamma \vdash B \bowtie_{\delta\delta} B'\), and so \(B \notin S\). By \(\text{SR}_{\delta\delta}\) it then follows that \(\Gamma \vdash B' : s\).

Then we show that the rule

\[(\beta\delta\text{CONV}) \quad \Gamma \vdash b : B \quad \Gamma \vdash B \bowtie_{\delta\delta} B' \quad B' \text{ is well-typed in } \Gamma \quad \frac{}{\Gamma \vdash b : B'}\]

is admissible in \(\lambda S_\delta\).

Suppose \(\Gamma \vdash b : B\), \(\Gamma \vdash B \bowtie_{\delta\delta} B'\) and that \(B'\) is well-typed in \(\Gamma\). Again, it suffices to prove that \(\Gamma \vdash B' : s\) for some sort \(s\). We distinguish two cases:

- Case 1: \(\Gamma \vdash B' : C\). \(\Gamma \vdash b : B\), so by correctness of types we can distinguish two cases:
  - Case 2.1: \(\Gamma \vdash B : s\). By \(\text{SR}_{\delta\delta}\) the common reduct of \(B\) and \(B'\) has both type \(s\) and type \(C\), so by \(\text{UT}_{\delta\delta}\) \(\Gamma \vdash C \bowtie_{\delta\delta} s\). But then \(\Gamma \vdash C \bowtie_{\delta\delta} s\), and by the rule \((\beta\delta\text{RED})\) it follows from \(\Gamma \vdash B' : C\) that \(\Gamma \vdash B' : s\).
  - Case 2.2: \(B \equiv s' \in S\). Then \(\Gamma \vdash B' \bowtie_{\delta\delta} s'\), so by \(\text{SR}_{\delta\delta}\) \(\Gamma \vdash C \bowtie_{\delta\delta} s\). This means that there is some \((s' : s) \in A\) such that \(\Gamma \vdash C \bowtie_{\delta\delta} s\). By the rule \((\beta\delta\text{RED})\) it follows from \(\Gamma \vdash B' : C\) that \(\Gamma \vdash B' : s\).

We are now ready to prove the theorem. Suppose that \(a\) and \(A\) are well-typed in context \(\Gamma\) in \(\lambda S_\delta\) and \(\text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(a) \cdot \text{dnf}(A)\).

To prove \(\Gamma \vdash_{\lambda S} a : A\) we only have to prove \(\Gamma \vdash_{\lambda S} a\). Because if \(\Gamma \vdash_{\lambda S} a : B\) for some \(B\), then \(\text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(a) \cdot \text{dnf}(B)\), by \(\text{UT}_{\delta\delta} \text{dnf}(B) \bowtie_{\delta\delta} \text{dnf}(A)\), so \(\Gamma \vdash B \bowtie_{\delta\delta} A\), and by \((\beta\delta\text{CONV})\) \(\Gamma \vdash_{\lambda S} A\).

It is not difficult to prove \(\Gamma \vdash_{\lambda S} a : a\) is well-typed in \(\Gamma\). \(\Gamma \vdash_{\lambda S} a : B\), or \(\Gamma \vdash_{\lambda S} a : a\). If \(\Gamma \vdash_{\lambda S} a : a\) then by correctness of types \(\Gamma \vdash_{\lambda S} a : B\) or \(a \in S\). If \(a \in S\), then it follows from \(\text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(a) \cdot \text{dnf}(A)\) by the generation lemma \((2.12)\) that \((a : s') \in A\) for some \(s' \in S\), and hence by the start lemma \((2.10)\) \(\Gamma \vdash_{\lambda S} a : s\).

An immediate consequence of theorems \(2.36\) and \(2.38\) is that for functional systems a typing statement holds in \(\lambda S_{\delta}\) if its \(\delta\)-normal form holds in \(\lambda S\).

2.39 Corollary (Equivalence of \(\lambda S\) and \(\lambda S_{\delta}\).)

Let \(S\) be a functional specification, and let \(a\) and \(A\) be well-typed in context \(\Gamma\) in \(\lambda S_{\delta}\). Then

1. \(\Gamma \vdash_{\lambda S_{\delta}} a : A \iff \text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(a) \cdot \text{dnf}(A)\)
2. \(\Gamma \vdash_{\lambda S_{\delta}} a : A \iff \text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(a) \cdot \text{dnf}(A)\)
3. \(\Gamma \vdash_{\lambda S_{\delta}} a : A \iff \text{dnf}(\Gamma) \vdash_{\lambda S} \text{dnf}(A)\).
2.3.

So, modulo $\delta$-conversion, the DPTS $\lambda S_\delta$ and the PTS $\lambda S$ are the same type system.

Theorem 2.38 does not hold for all (D)PTSs. It is easy to give a counterexample for a system that is not functional:

2.40 COUNTEREXAMPLE. Let $S$ be the specification

\[ S = \{ \ast, \square, \square' \}, \quad A = \{ \ast, \square, \ast, \square' \}, \quad R = \emptyset. \]

Then in the context $x = \ast \square$ the term $x$ has type $\square$, but not type $\square'$,

\[ x = \ast \square \vdash_{\lambda S_\delta} x : \square \]
\[ x = \ast \square \vdash_{\lambda S_\delta} x : \square' \]

whereas the $\delta$-normal form of $x = \ast -$ does have type $\square'$.

The problem is that we record types in definitions. For functional systems this is harmless, since the types are unique up to conversion. But for non-functional systems the choice of a particular type for a definition (e.g. $\square$ as the type of $x$ in the example above) imposes restrictions. Theorem 2.38 could be extended to all systems by having definitions of the form $x = a$ instead of $x = a.A$. However, in a typechecker the types of definitions will be recorded for the sake of efficiency. Otherwise to typecheck a term we have to typecheck its $\delta$-normal form, which can be much longer than the original term. Another reason to include type information in definitions is that in a definition $x = a : A$ the term $a$ may be a proof of a proposition $A$. In this case the definition plays the role of a lemma or theorem, and the type $A$ is then typically of more interest than the proof term $a$.

The definition mechanism of a DPTS leaves something to be desired. Many definitions involve parameters. In a DPTS, lambda abstractions have to be used to parameterise definitions. For example, an abbreviation for the composition of functions can be introduced by the definition

\[ \text{comp} = \lambda A, B, C. \lambda f : A \rightarrow B. \lambda g : B \rightarrow C. \lambda x : A. g(f(x)) \]
\[ \Pi A, B, C. (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C). \]

Then for $f : A \rightarrow B$ and $g : B \rightarrow C$ we can write $(\text{comp} A B C f g)$ for the composition of $f$ and $g$. Using lambda abstractions to parameterise definitions has two disadvantages.

A potential problem is that in a given PTS the required lambda abstractions may not be allowed. For instance, in the simply typed lambda calculus $\lambda \rightarrow$ abstractions over types, e.g. $\lambda A, B, C \ast \ldots$ in the example above, are not allowed.

A more practical problem is that terms such as $(\text{comp} A B C f g)$ are cluttered with type information. This problem does not only arise with definitions, but it is typical of explicitly typed languages. Often one would want to leave some parameters implicit, to be inferred by the type checker. We cannot expect a type checker to be able to reconstruct all arguments that are omitted. However, in many cases reconstruction of missing parameters is computable. For example, it is not difficult to see how the type parameters $A$, $B$ and $C$ in $(\text{comp} A B C f g)$ could be reconstructed if they were omitted. For the sake of readability we will sometimes omit parameters (and use infix notation).
Part I

Syntax
Chapter 3

The Basic System: $\lambda \omega_L$

In this chapter the type system $\lambda \omega_L$ is introduced. It consists of

- a programming language, which provides programs and their datatypes.
- a logic, which provides everything needed to reason about these programs. Properties of programs (e.g. specifications of programs) can be expressed in it, and it can be proved that a certain program has a certain property (e.g. meets a certain specification) or not.

$\lambda \omega_L$ can be concisely described as a PTS. It is a refinement of the Calculus of Constructions. Instead of having just one universe of types, we distinguish datatype and propositions, and restrict the dependencies between them. This strict separation between programming language and logic makes it easier to extend the programming language without disturbing the logic, and vice versa. This is exploited in later chapters, where possible extensions of the programming language and of the logic are considered.

The programming language is $\lambda \omega$, the PTS which is essentially Girard's system $\lambda^\omega$. So $\lambda \omega_L$ is a logic for reasoning about $\lambda \omega$. The system $\lambda \omega$ is also used as the basis of the programming languages QUEST [Car89] and LEAP [PL89]. Like all PTSs, $\lambda \omega$ is a very bare type system, providing only a few primitives. However, these are very general and powerful ones, making $\lambda \omega$ a very powerful language. In the following chapters, more typing constructors will be introduced as primitives.

In fact, $\lambda \omega_L$ has two subsystems that are copies of $\lambda \omega$. One of these serves as the programming language: terms are programs and types are datatypes. The other serves as a logic: terms are proofs and types are propositions. These two copies of $\lambda \omega$ are called $\lambda \omega_s$ and $\lambda \omega_p$. The subscripts refer to the fact that types are interpreted as sets in $\lambda \omega_s$, and as propositions in $\lambda \omega_p$.

Of course, for every PTS we introduce there is an associated DPTS. Depending on which is the best suited, we use one or the other. The PTS is used to explain the expressiveness of the system, the DPTS is used whenever we need the definition mechanism.

We begin by discussing the two interpretations of $\lambda \omega$ - as a programming language and as a logic - in sections 3.1 and 3.2, and then in section 3.3 the programming logic $\lambda \omega_L$ is defined. In section 3.4 we discuss how programs can be constructed together with their correctness proofs. Section 3.5 contains a comparison with the approach to program construction in the Calculus of Constructions using program extraction.
3.1 \( \lambda \omega \) as a programming language: \( \lambda \omega_2 \)

In this section we discuss the system \( \lambda \omega \) considered as a programming language. The following copy of \( \lambda \omega \) is reserved for the interpretation of \( \lambda \omega \) as a programming language.

3.1 Definition. \( \lambda \omega_{(d)} \) is the (D)PTS specified by

\[
\begin{align*}
S &= \{ *_s, \square_s \}, \\
A &= \{ *_s : \square_s \}, \\
R &= \{ (*_s, *_s), (\square_s, *_s), (\square_s, \square_s) \}.
\end{align*}
\]

This very compact definition of \( \lambda \omega_{(d)} \) is not very enlightening. Below we discuss the different kinds of terms that can be distinguished in \( \lambda \omega_2 \), and the different forms of abstractions that are allowed.

In a PTS there is just one collection of (pseudo)terms, but to understand \( \lambda \omega_2 \) we will distinguish different subsets for the different "levels" of \( \lambda \omega_2 \).

1. Programs. This includes data such as booleans, natural numbers or lists, and functions that manipulate these.

2. Datatypes and datatype-constructors. Datatypes are the types of programs. They form a subset of the datatype-constructors. The other datatype-constructors are functions that can be used to construct datatypes, such as a function \( x \) that maps two datatypes to their product type, or a function list that maps a datatype \( \sigma \) to the datatype of \( \sigma \)-lists.

An important difference between the datatypes and the other datatype-constructors is that the former can be inhabited by programs \( \sigma \), whereas the latter cannot. For instance, a function list from datatypes to datatypes cannot be the type of something.

3. Kinds. The kinds are the types of datatype-constructors. The kinds are generated by

\[
\mathcal{K} : = *_s \mid \mathcal{K} \to \mathcal{K}.
\]

\( *_s \) is the type of all datatypes, \( *_s \to *_s \) is the type of all functions from datatypes to datatypes, etc. For example, list has type \( *_s \to *_s \), and \( x \) has type \( *_s \to *_s \to *_s \).

In practice, only the simplest kind-expressions will ever be used. For a complex kind, for instance \( (*_s \to *_s) \to (*_s \to *_s) \), it is difficult to think of a datatype-constructor of that type.

There is a fourth level in \( \lambda \omega_2 \), which consists just of the symbol \( \square_s \). \( \square_s \) is the type of all the kinds, so \( *_s, *_s \to *_s, \) etc. all have type \( \square_s \). It plays an important role in the definition of \( \lambda \omega_2 \) as a PTS, as it allows a uniform description of the programs, datatype-constructors, and the kinds. Apart from this it is never used.

The kinds, datatype-constructors, datatypes, and programs, are formally defined as follows:

3.2 Definition.

1. If \( \Gamma \vdash \mathcal{K} : \square_s \) for some context \( \Gamma \), then \( \mathcal{K} \) is a kind.

2. If \( \Gamma \vdash \sigma : \mathcal{K} \) for some kind \( \mathcal{K} \) and context \( \Gamma \), then \( \sigma \) is a datatype-constructor.
3.1. $\lambda \omega$ AS A PROGRAMMING LANGUAGE : $\lambda \omega_S$

3. If $\Gamma \vdash \sigma : *_\alpha$ for some context $\Gamma$, then $\sigma$ is a datatype (and a datatype-constructor).

4. If $\Gamma \vdash M : \sigma$ for some datatype $\sigma$ and context $\Gamma$, then $M$ is a program

Later, in definition 3.4, it is shown that the distinction between these levels can already be made for the pseudoterm.

Contexts can declare variables from Var*$^\omega$ as datatype-constructors and variables from Var*$^\omega$ as programs. For example, the context

\[
\text{list} : *_\alpha \rightarrow *_\alpha, \ \text{n} : \text{nat}, \ f : \text{nat} \rightarrow \text{nat}, \ \alpha : *_\alpha, \ \alpha
\]
declares a datatype-constructor list, datatypes nat and $\alpha$, and programs (data) $n$, $f$ and $x$.

Here list, nat, $\alpha \in$ Var*$^\omega$ and $n, f, \alpha \in$ Var*$^\omega$.

3.3 CONVENTION. $K, K_1, K_2$ range over kinds. $\rho, \sigma, \tau$ range over datatypes and the other datatype-constructors. $\alpha, \beta$ range over Var*$^\omega$, i.e. variables that are datatype-constructors.

$M, N, f$ range over programs $x, y, z$ range over Var*$^\omega$, i.e. program-variables.

Each of the PTS-rules enables a certain kind of abstraction, by allowing certain product types to be formed. There are three different forms of abstraction in $\lambda \omega_S$, one for each of its rules. The rule $(\square, \square)$ provides a form of abstraction in datatype-constructors. The rules $(*, *)$ and $(\square, *)$ provide two forms of abstraction in programs.

The rule $(*, *)$ makes it possible to abstract over a datatype in a program. This results in programs of the form $(\lambda x : \sigma. M)$, where $\sigma$ is a datatype. Such a program expects another program (of type $\sigma$) as argument. It is a program that maps programs to programs.

The type of such an abstraction is a function type of the form $\sigma \rightarrow \tau$. The rule $(*, *)$ allows the formation of these function types

\[
(*) \frac{\Gamma \vdash \sigma : *_\alpha, \Gamma \vdash \tau : *_\alpha}{\Gamma \vdash \sigma \rightarrow \tau : *_\alpha}
\]

A simple example of this form of abstraction is $(\lambda x : \sigma. x)$, the identity on a datatype $\sigma$, which has type $\sigma \rightarrow \sigma$.

The rule $(\square, *)$ makes it possible to abstract over a kind in a program. This results in programs of the form $(\lambda \alpha : K. M)$ where $K$ is a kind. Such a program expects a datatype-constructor (of type $K$) as argument.

Most abstractions over a kind will be abstractions over the kind $*_\alpha$. This results in programs of the form $(\lambda \alpha : *_\alpha. M)$, Such a program expects a datatype as argument. It is a program that maps datatypes to programs.

For example, in the program $(\lambda x : \alpha. x)$ – the identity on a datatype $\alpha$ – we can abstract over all possible datatypes $\alpha$. This result in $(\lambda \alpha : *_\alpha. \lambda x : \alpha. x)$, the polymorphic identity. Depending on the datatype this function gets as argument, it produces the identity on that datatype.

Programs of the form $(\lambda \alpha : K. M)$ are called parametric polymorphic programs. Depending on the datatypeconstructor such a program gets as argument, it can have many different types. The types of these programs are polymorphic types of the form $(\Pi \alpha : K. \sigma)$. For
example, $(\lambda \alpha : \ast_s . \lambda x : \alpha . x) : (\Pi \alpha : \ast_s . \alpha \to \alpha)$. The rule $(\Box_s , \ast_s)$ allows the formation of these polymorphic types

$$(\Box_s , \ast_s) \frac{\Gamma \vdash K : \Box_s \quad \Gamma , \alpha : K \to \tau \vdash \ast_s}{\Gamma \vdash (\Pi \alpha : K . \tau) : \ast_s}$$

This kind of abstraction in programs can be used to define so-called generic programs – programs that are parameterised by datatypes – that behave uniformly for a whole class of types. An example of such a program is polymorphic identity defined above. A more interesting example is the program `reverse : $\Pi \alpha : \ast_s . (\text{list} \alpha) \to (\text{list} \alpha)$ which reverses a list, and can be used for lists of any datatype. Another use of this kind of abstraction in programs involves abstract datatypes. This is treated in more detail in chapter 4.

Finally, the rule $(\Box_s , \Box_s)$ allows the formation of kinds of the form $\Box_1 \to \Box_2$.

$$(\Box_s , \Box_s) \frac{\Gamma \vdash \Box_1 : \Box_s \quad \Gamma \vdash \Box_2 : \Box_s}{\Gamma \vdash \Box_1 \to \Box_2 : \Box_s}$$

For example, this rule allows the formation of $\ast_s \to \ast_s$, which is the type of functions from datatypes to datatypes such as list. These kinds make it possible to abstract over a kind in a datatype-constructor. This results in datatype-constructors of the form $(\lambda \alpha : K . \sigma)$, where $K$ is a kind. Such a datatype-constructor expects another datatype-constructor (of type $K$) as argument.

An example of such an abstraction is $(\lambda \alpha : \ast_s . \alpha \to \alpha) : \ast_s \to \ast_s$, the datatype-constructor that maps a datatype $\alpha$ to the datatype $\alpha \to \alpha$.

Another way to understand the effect of an individual PTS-rule, is to consider the dependency between programs and datatypes it introduces. This is used in Barendregt's $\lambda$-cube [Bar92], to characterise the different axes.

- $(\ast_s , \ast_s)$ gives programs that depend on programs
  
  For example, the program $(\lambda \tau \sigma . M)N$ depends on the program $N$

- $(\Box_s , \ast_s)$ gives programs that depend on datatypes
  
  For example, the program $(\lambda \alpha : \ast_s . M)\sigma$ depends on the datatype $\sigma$

- $(\Box_s , \Box_s)$ gives datatypes that depend on datatypes
  
  For example, the datatype list $\sigma$ depends on the datatype $\sigma$

In definition 3.2 the different levels of $\lambda \omega_s$ were defined in terms of the typing relation. This distinction can already be made for the pseudoterms.

3.4 Definition The sets of pseudoterms Kind, Cons and Prog are defined by

- **Kind** = $\ast_s | \text{Kind} \to \text{Kind}$
- **Cons** = $\text{Var}^{\Box_s} \mid \lambda \text{Var}^{\Box_s} \text{Kind} \mid \text{Cons} \mid \text{Cons} \to \text{Cons} | \Pi \text{Var}^{\Box_s} \text{Kind}$
- **Prog** = $\text{Var}^{\ast_s} \mid \lambda \text{Var}^{\ast_s} \text{Cons} \mid \text{Prog} \mid \text{Prog} \to \text{Prog} | \lambda \text{Var}^{\Pi_s} \text{Kind} \mid \text{Prog} \mid \text{Prog} \to \text{Prog} \text{Cons}$
This definition refines the context-free syntax given for arbitrary PTSs in chapter 2. Note that these three sets of pseudoterms are disjoint. The kinds, datatype-constructors and programs, as defined in definition 3.2 are elements of Kind, Cons and Prog, respectively:

3.5 Lemma (Classification for λω₁).
1. If \( \Gamma \vdash K : \Box_s \) then \( K \in \text{Kind} \).
2. If \( \Gamma \vdash \sigma : K : \Box_s \) then \( \sigma \in \text{Cons} \).
3. If \( \Gamma \vdash M : \sigma : \star_s \) then \( M \in \text{Prog} \).

Proof. By induction on the derivation of \( \Gamma \vdash a : A \) we prove

\[
\Gamma \vdash a : A \Rightarrow \begin{cases} A \equiv \Box_s & \Rightarrow a \in \text{Kind} \\ \Gamma \vdash A : \Box_s & \Rightarrow a \in \text{Cons} \\ \Gamma \vdash A : \star_s & \Rightarrow a \in \text{Prog} \end{cases}
\]

The only non-trivial cases are (\( \Pi \text{form} \)), (\( \Pi \text{intro} \)) and (\( \Pi \text{elim} \)). Here we have to distinguish the different possibilities for the subterms of \( a \), i.e. whether these are kinds, datatype-constructors or programs.

We show just the case that the last step is the rule (\( \Pi \text{form} \)), i.e. the last step in the derivation is

\[
(\Pi \text{form}) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x. A \ B) : s_2}
\]

for some \( (s_1, s_2) \in R \). There are three possibilities for \( (s_1, s_2) \):

1. \( (s_1, s_2) \equiv (\Box_s, \Box_s) \). Then, by the IH, \( A, B \in \text{Kind} \). By the definition of Kind then also \( A \rightarrow B \in \text{Kind} \), so we have to show that \( (\Pi x. A \ B) \) is \( A \rightarrow B \), i.e. that \( x \notin \text{FV}(B) \). It follows from the definition of Kind that all elements of Kind are closed, so \( x \notin \text{FV}(B) \).

2. \( (s_1, s_2) \equiv (\star_s, \star_s) \). Then, by the IH, \( A, B \in \text{Cons} \), and so \( A \rightarrow B \in \text{Cons} \). Again we have to show that \( (\Pi x. A \ B) \) is \( A \rightarrow B \), i.e. that \( x \notin \text{FV}(B) \). \( \Gamma \vdash A : \star_s \), so \( x \in \text{Var}^* \). It follows from the definition of Cons that elements of Cons do not contain variables from \( \text{Var}^* \), so \( x \notin \text{FV}(B) \).

3. \( (s_1, s_2) \equiv (\Box_s, \star_s) \). Then, by the IH, \( A \in \text{Kind} \) and \( B \in \text{Cons} \) \( \Gamma \vdash A : \Box_s \), so \( x \in \text{Var}^{\Box_s} \), and hence \( (\Pi x. A \ B) \in \text{Cons} \).

Kind, Cons and Prog provide a clear hierarchy between the different levels of (pseudo)terms. Kinds can occur in datatype-constructors and datatype-constructors can occur in programs, but programs cannot occur in datatype-constructors or kinds, and datatype-constructors cannot occur in kinds.

\( \lambda \omega \), and its extension with definitions \( \lambda \omega_3 \), are strongly normalising:

3.6 Theorem. \( \lambda \omega_3 \) is SN₃₆.

Proof. \( \lambda \omega_3 \) is a subsystem of \( \lambda C_3 \), and \( \lambda C_3 \) is SN₃₆ (theorem 2.34).
Basic datatypes and datatype-constructors

To use $\lambda \omega_s$ as a programming language, some basic datatypes and primitive operations on them are needed. There are representations for many datatypes in $\lambda \omega_s$, for example.

3.7 Example (Polymorphic Church-numerals). The context

$$
\begin{align*}
\text{polynat} & = \Pi \alpha \cdot (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) : \#_s, \\
0 & = \lambda \alpha \cdot \#_s \cdot \lambda f.\alpha \rightarrow \alpha : \lambda x:\alpha. \alpha \\
S & = \lambda n.\text{polynat} \cdot \lambda \alpha \cdot \#_s \cdot \lambda f.\alpha \rightarrow \alpha : \lambda x:\alpha. f(n \cdot \alpha \cdot x) \rightarrow \text{polynat} \rightarrow \text{polynat}
\end{align*}
$$

defines a representation for the natural numbers, the so-called polymorphic Church-numerals. These representations behave as natural numbers from a computational point of view, i.e. if we consider their reduction behaviour. For example, a function plus: $\text{polynat} \rightarrow \text{polynat} \rightarrow \text{polynat}$ can be defined such that $\text{plus}(S^0 0) (S^0 0) \approx_{\beta \delta} (S^0 S^0 0)$.

In $\lambda \omega_s$ there are representations for products, sums and existential types (or abstract datatypes), as well as many "base" types, such as the empty type, a unit type, and booleans. In fact, there are representations for all free term algebras, as shown in [BB85]. As representations of datatypes these encodings have well-known limitations (see for instance [CP90]). First of all, their reduction behaviour can be very inefficient. For example, for a predecessor function $\text{pred} : \text{polynat} \rightarrow \text{polynat}$ the reduction $\text{pred}(S^0 1) \approx_{\beta \delta} S^0 0$ requires $O(n)$ steps, and it is not possible to define a function $\text{pred}$ such that $\text{pred}(S x) \approx_{\beta \delta} x$ for variables $x$. Moreover, in $\lambda \omega_L$ none of the properties we need, for example the induction principle for polynat, are provable. So, if we want to reason about programs, it is not helpful to use these unnatural encodings.

In the following chapters, different datatypes and datatype-constructors will be added as primitives, even though there may be encodings for them in $\lambda \omega_s$. This gives us direct control over their interpretation in the model. For example, product types $\sigma \times \tau$ can then be directly interpreted as products in the model, and not as an interpretations of their $\lambda \omega_s$-encodings. This makes it easier to extend the programming language with certain operations, or the logic with certain axioms, on the basis of the model. Another advantage is that a direct implementation of these primitives can be more efficient that encodings.
3.2 \(\lambda w\) as a logic: \(\lambda w_p\)

In this section we discuss the system \(\lambda w\) considered as a logic. The types are now interpreted as propositions instead of datatypes, and the terms as proofs instead of programs. In comparison with the discussion of \(\lambda w_s\), the emphasis is now more on the types than on the terms.

The following copy of \(\lambda w\) is reserved for the interpretation of \(\lambda w\) as a logic:

3.8 Definition. \(\lambda w_{p(d)}\) is the (D)PTS specified by

\[
S = \{*p, \Box_p\}, \quad A = \{*p : \Box_p\}, \quad R = \{(\Box_p, *p), (\Box_p, \Box_p), (\Box_p, \Box_p)\}.
\]

The different levels of \(\lambda w_p\)-terms are called

1. "Proofs". The proofs are sometimes called proof terms to stress that they are \(\lambda\)-terms that represent natural deduction proofs. For human readers, the usefulness of proof terms is limited, because they become quickly too long to be readable. They are useful representations of proofs for machine manipulation. Verifying correctness of proofs amounts to type-checking them.

2. "Propositions and prop-constructors". The propositions are the types of the proofs. If a proof \(p\) has as type the proposition \(P\), this is read as "\(p\) is a proof of \(P\)".

The propositions form a subset of the prop-constructors. The other prop-constructors are functions that can be used to construct propositions. This includes the logical connectives, for example a function \(\land\) that maps two propositions to their conjunction, or a function \(\neg\) that maps a proposition to its negation.

3. "Prop-kinds". The prop-kinds are the type of prop-constructors. The prop-kinds are generated by

\[
\mathcal{P} : = *p \mid \mathcal{P} \rightarrow \mathcal{P}.
\]

*\(p\) is the type of all propositions. *\(p \rightarrow *p\) is the type of all functions from propositions to propositions, etc. For example, \(\neg\) has type *\(p \rightarrow *p\), and \(\land\) has type *\(p \rightarrow *p\rightarrow *p\).

The fourth level in \(\lambda w_p\) consists of just the symbol \(\Box_p\). \(\Box_p\) is the type of all the prop-kinds, e.g., *\(p\), *\(p \rightarrow *p\), etc. all have type \(\Box_p\).

3.9 Definition

1. If \(\Gamma \vdash \mathcal{P} : \Box_p\) for some context \(\Gamma\), then \(\mathcal{P}\) is a prop-kind.
2. If \(\Gamma \vdash P : \mathcal{P}\) for some prop-kind \(\mathcal{P}\) and some context \(\Gamma\), then \(P\) is a prop-constructor.
3. If \(\Gamma \vdash P : *p\) for some context \(\Gamma\), then \(P\) is a proposition (and a prop-constructor).
4. If \(\Gamma \vdash p : P\) for some proposition \(P\) for some context \(\Gamma\), then \(p\) is a proof.
The distinction between these different levels can already be made for the pseudoterms, exactly as is done for \( \lambda \omega \) in definition 3.4. This will be done later for the whole of \( \lambda \omega L \).

Contexts can declare prop-constructors and proofs. The former are variables in \( \text{Var}^\square_p \), the latter are variables in \( \text{Var}^* \). For example, the context

\[
* \in \square_p, \quad P : \star_p, \quad p : \neg P
\]
decares prop-constructors \( \neg \) and \( P \) and a proof \( p \). Here \( \neg, P \in \text{Var}^\square_p \) and \( p \in \text{Var}^* \).

The declaration \( \neg P \) in the context is an assumption (or an axiom), via the assumption that there is a proof \( p \) of the proposition \( \neg P \).

3.10 CONVENTION. \( \square_p, \square_1, \square_2 \) range over prop-kinds. \( P, Q, R, S, T \) range over propositions and the other prop-constructors. \( p, q \) range over proofs.

We do not reserve special variables for ranging over \( \text{Var}^\square_p \) and \( \text{Var}^* \). \( P, Q \) can be elements of \( \text{Var}^\square_p \), and \( p, q \) can be elements of \( \text{Var}^* \).

The rules \( * \in \square_p \) and \( \square_\phi, \star_\phi \) provide two ways of forming propositions. The rule \( \star_\phi \) allows the formation of implication

\[
\frac{\Gamma \vdash P \quad \Gamma \vdash Q : *_p}{\Gamma \vdash P \Rightarrow Q} \quad \text{(} \Rightarrow \text{ intro)}
\]

Here \( \Rightarrow \) is just another notation for \( \neg \) that reflects its intended interpretation. The associated introduction and elimination rules are

\[
\frac{\Gamma, p \vdash q \quad \Gamma \vdash q}{\Gamma \vdash (\lambda p : P \cdot q) : P \Rightarrow q} \quad \text{(} \Rightarrow \text{ elim)}
\]

A proof of \( P \Rightarrow Q \) is a function which maps proofs of \( P \) to proofs of \( Q \), which is the Brouwer-Heyting-Kolmogorov interpretation of implication. Omitting the proof terms \( p, q, (\lambda p : P \cdot q) \) and \( qp \) leaves the usual introduction and elimination rules for implication in natural deduction.

The rule \( \square_\phi, \star_\phi \) allows higher order universal quantification

\[
\frac{\Gamma \vdash \square_\phi : \square_p \quad \Gamma \vdash P : \square_p \quad \Gamma \vdash Q : *_p}{\Gamma \vdash (\forall P : \square_p \cdot \square_p) : \square_q} \quad \text{\( (\forall) \text{ intro}\)}
\]

Here \( \forall \) is just another notation for \( \sqcap \). If \( \square_p \) is \( \star_\phi \) this results in quantification over all propositions, as in second order logic. This makes it possible for example to form the proposition \( (\forall P : \star_\phi \cdot \neg P \Rightarrow P) \). Finally, the rule \( \forall_\phi, \forall_\phi \) allows the formation of prop-kinds of the form \( \square_1 \Rightarrow \square_2 \)

\[
\frac{\Gamma \vdash \forall_\phi : \forall_\phi \quad \Gamma \vdash P : \square_1 \quad \Gamma \vdash Q : \forall_\phi}{\Gamma \vdash \forall_\phi \Rightarrow \forall_\phi} \quad \text{\( \forall \text{ elim}\)}
\]

For example, it provides the prop-kind \( \star_\phi \Rightarrow \star_\phi \), the type of \( \neg \), and \( \star_\phi \Rightarrow \star_\phi \Rightarrow \star_\phi \), the type of \( \land \) and \( \lor \). It also makes it possible to abstract over a prop-kind in propositions and prop-constructors. For example, the propositional connective \( \neg \) can be defined as

\[
(\lambda P : \star_\phi \cdot P \Rightarrow \text{False}) : \star_\phi \Rightarrow \star_\phi,
\]
where \( \text{False} \) is the proposition representing falsehood.
Basic propositions and prop-constructors.

To use $\lambda\omega_p$ as a logic, some basic logical connectives and their properties are needed. These could be declared in the context, but it is well-known that all connectives are definable in terms of implication and higher order universal quantification.

3.1 DEFINITION. $\Gamma_{\text{LOGIC}}$ is the context

\[
\begin{align*}
\text{False} & = (\forall P : *_p, P : *_p, \\
\text{True} & = (\forall P : *_p, P \Rightarrow P : *_p), \\
\neg & = \lambda P, *_p, P \Rightarrow \text{False} : *_p \rightarrow *_p, \\
\land & = \lambda P, Q, *_p, (\forall R : *_p, (P \Rightarrow Q \Rightarrow R) : *_p \rightarrow *_p \rightarrow *_p), \\
\lor & = \lambda P, Q, *_p, (\forall R : *_p, (P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R) : *_p \rightarrow *_p \rightarrow *_p, \\
\iff & = \lambda P, Q, *_p, (P \Rightarrow Q) \land (Q \Rightarrow P) : *_p \rightarrow *_p \rightarrow *_p
\end{align*}
\]

This context defines the propositional constants truth (True) and falsehood (False) and the logical connectives negation ($\neg$), conjunction ($\land$), disjunction ($\lor$), and bi-implication ($\iff$).

For a propkind $\mathcal{P}$ we can define

\[
\exists_{\mathcal{P}} = \lambda P : \mathcal{P} \rightarrow *_p, (\forall R : *_p, (\forall X : \mathcal{P}, (PX) \Rightarrow R) \Rightarrow R).
\]

Then for a term $(\lambda X : \mathcal{P}, P) : \mathcal{P} \rightarrow *_p$, the term $\exists_{\mathcal{P}}(\lambda X : \mathcal{P}, P)$ represent the higher order existential quantification "($\exists X : \mathcal{P}, P$)". The definition of $\exists_{\mathcal{P}}$ is parameterised by a propkind $\mathcal{P}$. We cannot abstract over $\mathcal{P}$ and define $\exists$ as follows

\[
\exists = \lambda P ; \square_p. \exists_{\mathcal{P}}.
\]

because the abstraction $\lambda P ; \square_p$ is not allowed in $\lambda\omega_p$.

The $\lambda\omega_s$-counterparts of some of the $\lambda\omega_p$-encodings given above – i.e. these terms with all subscripts $p$ replaced by the subscript $s$ – can be used to represent certain datatype-constructors. For instance, the encoding of $\land$ can be used to represent product types, and the encoding of $\lor$ can be used to represent disjoint sum types. At the end of section 3.1 it was already mentioned that such encodings of datatype-constructors in $\lambda\omega_s$ have some shortcomings. The encodings of the logical connectives in $\lambda\omega_p$ is less problematic. Firstly, all their intutionistic properties, i.e. their introduction and elimination principles, are provable in $\lambda\omega_p$. For example, in $\lambda\omega_p$ we can prove $(\forall P : *_p, \text{False} \Rightarrow P)$, $(\forall P, Q, *_p, P \Rightarrow (P \lor Q))$, and $(\forall P, Q, *_p, (P \land Q) \Rightarrow P)$. Note that the quantification over propositions makes it possible to express these introduction and elimination principles inside the system $\lambda\omega_p$. Secondly, it is easy to show that these representations have the right interpretation in the proof-irrelevance model given in chapter 6 (see example 6.27).

Consistency

When we use $\lambda\omega_p$ as a logic, we are interested in the question whether it is consistent. $\lambda\omega_p$ is consistent, in the sense that not all propositions are provable. This is equivalent with saying that False, as defined in 3.11, is not provable. The definition mechanism does not affect consistency: by corollary 2.39 $\lambda\omega_{p,s}$ is consistent if $\lambda\omega_p$ is consistent.
Of course, which propositions are provable depends on the assumptions in the context. For example, in a context containing the assumption \( p : \text{False} \) all propositions are provable. A context is called consistent if not all propositions are provable in it. Using corollary 2.39 we can relate consistency of a context in \( \lambda\omega_p \) and \( \lambda\omega_{p\delta} \): a context is consistent in \( \lambda\omega_{p\delta} \) iff its \( \delta \)-normal form is consistent in \( \lambda\omega_p \).

3.12 Theorem (Consistency of \( \lambda\omega_p \)) False is not provable in \( \lambda\omega_{p\delta} \) in the context \( \Gamma_{\text{LOGIC}} \).

By corollary 2.39 this is equivalent with saying that \( (\forall P \cdot p \cdot P) \) is not provable in \( \lambda\omega_p \) in the empty context. There are two ways to prove this theorem. The syntactic way is to use the fact that \( \lambda\omega_p \) is SN. From SN it follows that False is not provable: if it were, it would have a proof in normal-form, and a simple syntactic argument shows that no inhabitant of False can be in normal form. A semantic proof can be given using the proof-irrelevance model of \( \lambda\omega_p \) (see lemma 6.31.) The advantage of the second proof is that it can easily be extended to prove consistency of certain contexts, for example:

3.13 Theorem (Consistency of classical logic). False is not provable in the context \( \Gamma_{\text{LOGIC}} \) extended with the assumption classical \( \cdot (\forall P \cdot p \cdot (\neg p) \Rightarrow p) \)

Proof. See lemma 6.31
3.3 The programming logic $\lambda\omega_L$

In this section we introduce the programming logic $\lambda\omega_L$, which contains both $\lambda\omega_s$ and $\lambda\omega_p$. As mentioned earlier, $\lambda\omega_L$ can be divided in two parts: a programming language and a logic for reasoning about this programming language. The relationship between the two parts is asymmetric. We want to be able to say things about the programming language in the logic, but we do not want to be able to say things about the logic in the programming language. Because programs and datatypes will not be allowed to depend on proofs or propositions, $\lambda\omega_L$ will be a conservative extension of the programming language.

$\lambda\omega_s$ serves as the programming language and $\lambda\omega_p$ as part of the logic. $\lambda\omega_p$ alone does not suffice as logic. To reason about $\lambda\omega_s$-constructs 4 more PTS-rules will be added. These rules will allow the formation of proofs and propositions that depend on programs and datatypes.

$\lambda\omega_L$ can be embedded in the Calculus of Constructions, the system $\lambda C$ defined in definition 2.19. In fact, it is a subsystem of one of the variants of the Calculus of Constructions defined in [PM89b], as we will see in section 3.5.

Before defining $\lambda\omega_L$, we consider which PTS-rules are required for reasoning about $\lambda\omega_s$-programs and datatypes.

To express properties of programs we need predicates and relations on datatypes. Predicates can be seen as propositional-valued functions, i.e., the type of a predicate on a datatype $\sigma$ is $\sigma \rightarrow *_p$. Similarly, a relation or binary predicate on a datatype $\sigma$ is of type $\sigma \rightarrow \sigma \rightarrow *_p$. For instance, the ordering $\leq$ on the type of natural numbers $\text{nat}$ has type $\text{nat} \rightarrow \text{nat} \rightarrow *_p$.

Equality of programs of type $\sigma$ is a relation $=_\sigma \, \sigma \rightarrow \sigma \rightarrow *_p$. Like $*_p$, the terms $*_p$ and $\sigma \rightarrow \sigma \rightarrow *_p$ are prop-kinds, i.e., they have type $\square_p$. Their formation requires the PTS-rule $(*, \square_p)$:

$$
\frac{\Gamma \vdash \sigma \quad \Gamma \vdash P : \square_p}{\Gamma \vdash \sigma \rightarrow P : \square_p}
$$

For every datatype $\alpha$ a relation of type $\alpha \rightarrow \alpha \rightarrow *_p$ is needed that is the notion of equality for that datatype. This can be achieved by having a single polymorphic equality relation $-_L$ of type $(\Pi \alpha : *_p. \alpha \rightarrow \alpha \rightarrow *_p)$. Then for all datatypes $\sigma$, $(=_L \sigma) : \sigma \rightarrow \sigma \rightarrow *_p$, i.e., $(=_L \sigma)$ is a relation on $\sigma$. For all terms $M$ and $N$ of type $\sigma$, $(=_L \sigma) M N : *_p$, i.e., $(=_L \sigma) M N$ is a proposition. Like $\sigma \rightarrow *_p$ and $\sigma \rightarrow \sigma \rightarrow *_p$, the term $(\Pi \alpha : *_p. \alpha \rightarrow \alpha \rightarrow *_p)$ is a prop-kind, i.e., it has type $\square_p$. The formation of $(\Pi \alpha : *_p. \alpha \rightarrow \alpha \rightarrow *_p)$ requires the PTS-rule $(\square_p, \square_p)$:

$$
\frac{\Gamma \vdash K : \square_p \quad \Gamma, \alpha, K \vdash P : \square_p}{\Gamma \vdash (\Pi \alpha : *_p. \alpha \rightarrow K \rightarrow P) : \square_p}
$$

$\lambda\omega_p$ provides higher-order universal quantification. This means that we can quantify over all propositions, e.g., $(\forall \alpha : *_p, \ldots)$, over all functions from propositions to propositions, e.g., $(\forall \alpha : *_p \rightarrow *_p, \ldots)$, etc. In $\lambda\omega_L$ more possibilities for quantification are needed. For universal quantification over all elements of a datatype the PTS-rule $(*, *_p)$ is needed:

$$
\frac{\Gamma \vdash \sigma \quad \Gamma, \tau : \sigma \vdash P : *_p}{\Gamma \vdash (\forall \sigma \quad P) : *_p}
$$

Using this rule, the proposition $\forall x : \text{nat}. (\leq x x)$ can be formed.
For universal quantification over all elements of a kind the PTS-rule \((\Box_s, \ast_p)\) is needed.

\[
(\Box_s, \ast_p) \quad \frac{\Gamma \vdash K \cdot \Box_s \quad \Gamma, \alpha \vdash P \vdash \ast_p}{\Gamma \vdash (\forall \alpha. K) \cdot P \vdash \ast_p}
\]

By taking \(K \equiv \ast_s\), this rule allows universal quantification over all datatypes. For example, this is used in the proposition \(\forall x: s, \forall y: \alpha. (\equiv_L \alpha x x)\), which expresses reflexivity of the relation \(\equiv_L\) for all datatypes \(\alpha\).

We have now discussed all the rules of \(\lambda\omega_L\).

3.14 DEFINITION. \(\lambda\omega_{L(4)}\) is the (D)PTS specified by

\[
\begin{align*}
S &= \{\ast_s, \Box_s, \ast_p, \Box_p\} \\
A &= \{(\ast_s \cdot \Box_s), (\ast_p \cdot \Box_p)\} \\
R &= \{ (\Box_s, \Box_s), (\ast_s, \ast_s), \\
&\quad (\Box_s, \Box_p), (\ast_s, \Box_p), (\Box_p, \Box_p), \\
&\quad (\ast_s, \ast_p), (\ast_p, \ast_p), (\ast_s, \ast_p) \}
\end{align*}
\]

The levels of terms we distinguish in \(\lambda\omega_L\) are those of \(\lambda\omega_s\) and \(\lambda\omega_p\); i.e., kinds, datatypes and datatype-constructors, programs, prop-kinds, propositions and prop-constructors, and proofs. They are defined as in definitions 3.2 and 3.9.

Observe that when \(R\) is regarded as a matrix, we only have rules below or on the diagonal from the top-left to the bottom-right. The system \(\lambda\omega_L\) consists of

- \(\lambda\omega_s\) for the programs and their datatypes: \(\{(\Box_s, \Box_s), (\Box_s, \ast_s), (\ast_s, \ast_s)\}\),

- \(\lambda\omega_p\) for the propositions and their proofs: \(\{(\Box_p, \Box_p), (\Box_p, \ast_p), (\ast_p, \ast_p)\}\),

- all possible dependencies of propositions and proofs on programs and types, provided by all rules of the form \((\ast_s, \ast_p)\) \(\{(\Box_s, \Box_p), (\Box_s, \ast_p), (\ast_s, \Box_p), (\ast_s, \ast_p)\}\)

\(\lambda\omega_L\) does not have all possible PTS-rules. It does not have the rules

- \((\ast_s, \Box_s)\).

As a result, there are no datatypes that depend on programs. An example of a datatype depending on a program is the type \((\text{less than } n)\) that consists of all natural numbers smaller than \(n\). Such datatypes are often simply called dependent types. They drastically change the nature of type system; it increases the expressive power to the extent that datatypes can express complete specifications. In chapter 1 we already discussed why we do not want such a very expressive type system for programs. In section 3.5 the main consequence of including the rule \((\ast_s, \Box_s)\) – the need for program extraction – is discussed.

If datatypes can depend on programs, then deciding equality of datatypes involves deciding equality of programs. A basic problem with this is that we want equality of datatypes to be decidable – otherwise type checking is undecidable –, whereas for programs we typically want an equality which is extensional and (hence) undecidable.
• (\ast_p, \Box_p) .

As a result, there are no propositions that depend on proofs. With such propositions properties of proofs can be expressed. We are only interested in proving properties of programs and not in proving properties of proofs. Hence there is no need for proof-dependent propositions.

• (\neg p, \neg p) As a result, there are no programs or datatypes that depend on propositions or proofs. The consequences of these dependencies are discussed in section 3.5.

Because there are no rules of the form (\neg p, \neg p), \lambda \omega L is a conservative extension of the programming language \lambda \omega, i.e. in \lambda \omega L we have the same kinds, datatype-constructors and programs as in \lambda \omega:

3.15 Theorem (Conservativity of \lambda \omega L over \lambda \omega).

Suppose \Gamma \vdash_{\lambda \omega L} a : A and a is a program, datatype-constructor or a kind (i.e. \Gamma \vdash_{\lambda \omega L} A \ast s, \Gamma \vdash_{\lambda \omega L} A : \Box_i, or A \equiv \Box_i )

Then \Gamma \vdash_{\lambda \omega} a : A

Proof. Induction on the derivation of \Gamma \vdash a : A \square

As for \lambda \omega, the different syntactic categories of \lambda \omega L can already be distinguished for the pseudoterms. It follows from the conservativity of \lambda \omega L over \lambda \omega that the kinds, datatype-constructors and programs of \lambda \omega L are elements of Kind, Cons and Prog as defined in definition 3.4. For the prop-kinds, prop-constructors and proofs we define:

3.16 Definition. The sets of pseudoterms Pkind, PCons and Proof are defined by

\begin{align*}
\text{Pkind} & := \ast_p | \text{Pkind} \rightarrow \text{Pkind} \\
\text{PCons} & := \text{Var}_p | \lambda \text{Var}_p \text{. Cons. PCons | PCons \ Prog} \\
& | \lambda \text{Var}_p \text{. Kind. PCons | PCons Cons} \\
& | \lambda \text{Var}_p \text{. Pkind. PCons | PCons PCons} \\
& | \text{PCons} \Rightarrow \text{PCons}
\end{align*}

\begin{align*}
\text{Proof} & := \text{Var}_p | \lambda \text{Var}_p \text{. PCons Proof | Proof Proof} \\
& | \lambda \text{Var}_p \text{. Cons. Proof | Proof Prog} \\
& | \lambda \text{Var}_p \text{. Kind Proof | Proof Cons} \\
& | \lambda \text{Var}_p \text{. Pkind. Proof | Proof PCons}
\end{align*}

where \Lambda \in \{ \lambda, \forall \} \square

Note that the different sets of pseudoterms for the different levels of \lambda \omega L can be defined in the following order: Kind, Cons, Prog, Pkind, PCons and Proof. Each of these sets only depends on the ones that precede it in this order. This is caused by the fact that the set of \lambda \omega L-rules \mathcal{R} in definition 3.14 does not include any rules above the diagonal. It will play an important role in chapter 6, where the semantics of the \lambda \omega L is treated, because it means that the interpretation of the different levels can be defined in the same order.

3.17 Lemma (Classification for \lambda \omega L).

1. If \Gamma \vdash K : \Box_i then K \in \text{Kind} .
2. If $\Gamma \vdash \sigma : K : \Box$, then $\sigma \in \text{Cons}$

3. If $\Gamma \vdash M : \sigma : *$, then $M \in \text{Prog}$

4. If $\Gamma \vdash P : \Box_p$, then $P \in \text{PKind}$

5. If $\Gamma \vdash P : \Box_p$, then $P \in \text{PCons}$

6. If $\Gamma \vdash p : P : *_p$, then $p \in \text{Proof}$

**Proof.** Parts 1 to 3 immediately follow theorem 3.15 and lemma 3.5. The proof of the other parts is then similar to that of lemma 3.5: by induction on the derivation of $\Gamma \vdash a : A$ we prove

\[
\Gamma \vdash a : A \Rightarrow \begin{cases} 
A \equiv \Box_p & \Rightarrow \ a \in \text{PKind} \\
\Gamma \vdash A : \Box_p & \Rightarrow \ a \in \text{PCons} \\
\Gamma \vdash A : *_p & \Rightarrow \ a \in \text{Proof}
\end{cases}
\]

\[\square\]

$\lambda \omega_L$ can be embedded in $\lambda C$, the Calculus of Constructions, simply by erasing the subscripts of $*$ and $\Box$:

**3.18 Lemma.** If $I \vdash a : A$ in $\lambda \omega_{I,(6)}$, then $|\Gamma| \vdash |a| : |A|$ in $\lambda C_{(6)}$, where $|z|$ is $z$ with $*$ substituted for $*_z$ and $*_p$ and $\Box$ substituted for $\Box_z$ and $\Box_p$

**Proof.** Induction on the derivation. In fact, it suffices to observe that erasing the subscripts in the specification of $\lambda \omega_{I,(6)}$ produces the specification of $\lambda C_{(6)}$.

What makes the system $\lambda \omega_L$ a lot simpler than $\lambda C$ is that there are fewer dependencies between the different levels. In $\lambda C$ different levels similar to those of $\lambda \omega_L$, can be distinguished, namely kinds, constructors and programs, but these are all mutually dependent. In other words, elements of one of these levels can occur as subexpressions of elements of any other.

By lemma 3.18 it follows immediately from the fact that $\lambda C_{(6)}$ is strongly normalising that $\lambda \omega_L$ and its extension with definitions $\lambda \omega_L,6$ are.

**3.19 Theorem** $\lambda \omega_{I,\xi}$ is $\text{SN}_{\xi}$.

**Basic propositions and prop-constructors.**

In $\lambda \omega_L$ there are three forms of existential quantification: over a prop-kind $P$, over a datatype $\sigma$ and over a kind $K$, i.e. of the form $(\exists X : P \vdash \sigma)$, $(\exists \sigma : P \vdash \sigma)$ and $(\exists K : P \vdash K \vdash \sigma)$, respectively. In definition 3.11 it was shown how in $\lambda \omega_\xi$ the first form of existential quantification can be defined in terms of $\forall$ and $\Rightarrow$. The other two can be defined similarly.

**3.20 Definition.** Existential quantification over a datatype $(\exists \sigma)$ and over a kind $K$ $(\exists K)$ are defined by

\[
\exists = \lambda \alpha. *_\alpha \vdash *_p \quad \forall R *_p (\forall \alpha (P \alpha \Rightarrow R) \Rightarrow R) \Rightarrow R
\]

\[
\exists = \lambda P *_p \vdash *_p \quad \forall R *_p (\forall \alpha (P \alpha \Rightarrow R) \Rightarrow R) \Rightarrow R
\]

\[
(\exists K \vdash *_p) \vdash *_p
\]

\[\square\]
In the first definition we have abstracted over all possible datatypes \( \alpha \). In the second definition we cannot abstract over all possible kinds \( K \), because this abstraction is not allowed in \( \lambda \omega_L \).

To use \( \lambda \omega_L \) as a programming logic, some basic predicates and their properties are needed. These can be declared in the context, together with the properties we require of them. Another option is to define them in \( \lambda \omega_L \). In [PPM90] it is shown how inductively defined propositions, predicates and relations can be represented in the Calculus of Constructions. The same technique can be used in \( \lambda \omega_L \). The most important relation — equality of programs — can be defined in \( \lambda \omega_L \) as follows:

\[ \Gamma_{LEIBNIZ} \]

3.21 Definition (Leibniz' Equality). \( \Gamma_{LEIBNIZ} \) is the context

\[ \begin{align*}
   \alpha & \overset{*}{=} \lambda x, y: \alpha \forall P, \alpha \rightarrow *_p, \ (P \ x) \Rightarrow (P \ y) \\
   \Pi \alpha \ & \overset{*}{=} \alpha \rightarrow *_p
\end{align*} \]

This context defines Leibniz' equality for programs. \( \approx_L \) will be written infix, with its first argument as a subscript instead of \( L \). So \( (\approx_L \sigma \ M \ N) \) is written as \( M \approx \sigma \ N \). From now on, we always assume that \( \Gamma_{LEIBNIZ} \) is part of the context, so that we can always use \( \approx_L \) to express equality of programs in \( \lambda \omega_L \).

Despite its asymmetric definition, the predicate \( \approx_L \) has all the required properties:

- It is reflexive, because all \( \beta \delta \)-convertible programs are Leibniz' equal, as shown below.

  Suppose \( \Gamma \vdash M : \sigma, \ \Gamma \vdash N : \sigma \) and \( \Gamma \vdash M \approx_{\beta \delta} N \). Then there is an inhabitant — i.e. a proof — of \( M \approx \sigma N \) in context \( \Gamma \). As an illustration of a formal proof in the logic, this proof and its type derivation are given below. We use the notation for natural deduction described in [Ned90], with "flags" denoting the scope of declarations.

  \[
  \begin{array}{c|l}
  1 & \frac{\quad}{P \ : \ \sigma \rightarrow *_p} \\
  2 & p : P \ M \\
  3 & p : P \ N & 2, M \approx_{\beta \delta} N, (\beta \delta \text{conv}) \\
  4 & (\lambda p : P \ M. \ p) : P \ M \Rightarrow P \ N & 3, (\Pi \text{intro}) \\
  5 & (\lambda P : \sigma \rightarrow *_p, \lambda p : P \ M. \ p) : (\forall P : \sigma \rightarrow *_p, P \ M \Rightarrow P \ N) & 4, (\Pi \text{intro}) \\
  6 & (\lambda P : \sigma \rightarrow *_p, \lambda p : P \ M. \ p) \cdot M =_\sigma N & 5, \text{def. } =_L, (\beta \delta \text{conv})
  \end{array}
  \]

- We can substitute equals for equals in propositions, because

  \[
  \forall \sigma : * \forall x, y : \sigma \ (x =_\sigma y) \Rightarrow \forall Q : \sigma \rightarrow *_p, \ (Q \ x) \Rightarrow (Q \ y)
  \]

  is provable in \( \lambda \omega_L \), as is shown below. Using this property it can be proved that \( =_L \) is symmetric and transitive.
42

CHAPTER 3. THE BASIC SYSTEM : $\lambda_\omega_L$

1
\[ \sigma : *_x, x, y : \sigma \]
2
\[ p : x =_y y \]
3
\[ Q : \sigma \rightarrow *_p \]
4
\[ q : Q x \]
5
\[ p : (\forall p. \sigma \rightarrow *_p (P x) \Rightarrow (P y)) \]
6
\[ p Q : (Q x) \Rightarrow (Q y) \]
7
\[ p Q q : (Q y) \]
8
\[ (\lambda \sigma : *, \lambda x, y : \sigma. \lambda p.x =_p y. \lambda Q . \sigma \rightarrow *_p. \lambda q. Q x. p Q q) \]
\[ \forall \sigma : *, \forall x, y \sigma. (x =_\sigma y) \Rightarrow \forall Q . \sigma \rightarrow *_p. (Q x) \Rightarrow (Q y) \]

In lemma 6.48 it is shown that $=_L$ is in fact interpreted as equality in the model.

As illustrated above, type derivations for proof terms correspond closely to conventional natural deduction proofs. Proof terms will usually be omitted in such derivations and in typing judgements. Instead of $\Gamma \vdash p : P$, we then write $\Gamma \vdash p : P$.

There are important differences between $\beta\delta$-conversion and Leibniz' equality. Whereas $\beta\delta$-conversion is defined for all (pseudo)terms, Leibniz' equality just for programs of the same type. The most important difference is that properties of Leibniz' equality can be assumed. For example, we can assume that two programs $M$ and $N$ of type $\sigma$ are Leibniz equal by introducing an assumption $p : M =_\sigma N$ in the context. As will be shown later in example 6.48, $=_L$ is interpreted as equality in the model. Consequently, any properties that hold for equality of programs in the model can safely be added as axioms. We will introduce axioms for Leibniz' equality that make it an extensional equality. As a consequence, Leibniz' equality for functions, e.g. $f =_L g$, will be undecidable. On the other hand, $\beta\delta$-conversion of all $\lambda_\omega_L$-terms is decidable, because $\beta\delta$-reduction is Church-Rosser and strongly normalising.

Consistency

As for $\lambda_\omega_P$, we are interested in the consistency of $\lambda_\omega_L$.

3.22 Theorem (Consistency of $\lambda_\omega_L$) : False is not provable in $\lambda_\omega_L$ in the context of $I_{\text{LOGIC}}$.

As for $\lambda_\omega_P$, there are two ways to prove consistency of $\lambda_\omega_L$; a syntactic proof using the fact that $\lambda_\omega_L$ is SN, or a semantic proof using the proof-irrelevance model of $\lambda_\omega_L$ given in chapter 6 (see lemma 6.49). And as for $\lambda_\omega_P$, the second proof can easily be extended to prove consistency of certain contexts and axioms.

For instance, we want to use classical logic, and we want to assume that equality of functions is extensional, i.e. that:

\[ f =_\sigma \rightarrow f' \iff (\forall x \sigma. f x =_\sigma f' x) \]
\[ q =_{\Pi \alpha : A} g \iff (\forall \alpha : A. g \alpha =_\sigma g' \alpha) \]

for all $f, f' : \sigma \rightarrow \tau$ and $q, g' \in (\Pi \alpha : A) \sigma$. The implications from left to right immediately follow from the definition of Leibniz' equality, but the implications from right to left have to be introduced as axioms.
3.3. **THE PROGRAMMING LOGIC $\lambda\omega_L$**

3.23 **Definition.** *AXIOM* is the set of axioms containing

- **classic**: $\forall P. P. (\neg\neg P) \Rightarrow P.$
- $AX_{\sigma\rightarrow\tau}: \forall f, g: \sigma \rightarrow \tau. (\forall x: \sigma. ((f x) =_\tau (g x))) \Rightarrow f =_{\sigma\rightarrow\tau} g.$
- $AX_{\Pi_0 K}. \sigma: \forall f, g: (\Pi_0 K. \sigma). (\forall \alpha: K. (f \alpha) =_\sigma (g \alpha)) \Rightarrow f =_{\Pi_0 K. \sigma} g$

for all $\sigma \rightarrow \tau, (\Pi_0 K. \sigma) \in \text{Cons}.$

We lack the necessary quantifications to express axioms in *AXIOM* in a finite number of $\lambda\omega_L$-axioms. We could quantify over all datatypes $\sigma, \tau : \kappa$ in $AX_{\sigma\rightarrow\tau}$, but we cannot quantify over all kinds $K : \Box \kappa$ in $AX_{\Pi_0 K. \sigma}$.

3.24 **Lemma (Consistency of *AXIOM* in $\lambda\omega_L$).**

*False is not provable in a context containing only axioms from *AXIOM*.***

**Proof.** See lemma 6.49.

So it is safe to use classical logic in $\lambda\omega_L$, and to assume that equality of programs is extensional.
3.4 Program and Proof Development

The system $\lambda\omega_L$ has been defined so that the problem of program construction can be stated as follows:

Given:
- a datatype $\sigma$, i.e. $\sigma : \ast$
- a predicate $P$ over that datatype, i.e. $P : \sigma \rightarrow \ast_p$

Find:
- a program $M$ of type $\sigma$, i.e. $M : \sigma$
- a proof $p_M$ that $M$ satisfies $P$, i.e. a proof $p_M : P M$.

A pair $(\sigma, P)$ consisting of a datatype and a predicate forms a specification. A pair $(M, p_M)$ consisting of a program $M : \sigma$ and proof $p_M : P M$ is a solution of this specification. We usually refer to the predicate $P$ or the proposition $P M$ as the specification.

The datatype $\sigma$ is only a partial specification. It gives some information about the program we want, for example that it should be a function from nat to nat, but not all of it. In the predicate $P$, however, all the required properties of the program can be expressed. It defines a subset of the datatype $\sigma$. This expressiveness comes at a price: in general it is not decidable whether a program $M$ satisfies a predicate $P$, whereas it is decidable whether it has type $\sigma$. To show that $M$ satisfies $P$, a proof $p_M$ has to be supplied.

We can think of the $\lambda\omega_L$-proofs and predicates as annotation of the $\lambda\omega_L$-programs and datatypes. Programs are annotated with proofs that they satisfy certain properties, datatypes are annotated with predicates.

<table>
<thead>
<tr>
<th>$\lambda\omega_L$ annotation</th>
<th>$\lambda\omega_L$ annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype $\sigma$ : $\ast_s$</td>
<td>predicate $P$ : $\ast_p$</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \rightarrow \ast_s$</td>
<td>$\Gamma \vdash P : \sigma \rightarrow \ast_p$</td>
</tr>
<tr>
<td>$\Gamma \vdash M : \sigma \rightarrow \ast_s$</td>
<td>$\Gamma \vdash p_M : P M \rightarrow \ast_p$</td>
</tr>
</tbody>
</table>

Not only the datatypes, but also the other datatype-constructors can be annotated. This will be discussed later, in subsection 3.4.3.

Despite the fact that a program and its correctness proofs are separate objects, we should not first have to find a program and afterwards prove its correctness. Ideally, programs and their correctness proofs should be constructed hand in hand. In the remainder of this section we look at a strategy for the compositional development of programs and proofs. Here the program is constructed, together with its correctness proof, by stepwise refinement. For this, pairs of derivable inference rules are needed, consisting of

(i) a $\lambda\omega_L$-derivation rule, that gives the type of a program in terms of the types of its component parts, and

(ii) an associated $\lambda\omega_L$-derivation rule, that gives a proof that this program satisfies a specification in terms of proofs that its component parts satisfy certain specifications.

So (i) is a type derivation rule for programs, and (ii) is a type derivation rule for proofs. Such pairs of derivation rules, consisting of a type derivation rule and an associated proof rule, will be called em coupled derivation rules. These coupled derivation rules can be used to develop a program together with its correctness proof. For every programming language construct there is usually a unique type inference rule. However, there may be more than one associated proof rule, for specifications of different forms.
A specification may have to be of a certain form in order to construct a program and its correctness proof together. For \( \lambda \omega \)-rules for forming a datatype there are corresponding rules in \( \lambda \omega L \) for forming specifications on that datatype, which produce specifications of the right form.

So we are interested in pairs of inference rules, that can be used to derive types for pairs \((M, p_M)\), consisting of a program and a correctness proof, or for pairs \((\sigma, P)\), consisting of a datatype and a predicate.

Although in \( \lambda \omega L \) program and proof construction both come down to finding an inhabitant of a certain type, there are important practical differences. We do care which particular program \( M \) we find, whereas we do not care (as much) which particular proof \( p_M \) we find. This means we do not have to be able to read proof terms in the same way we want to read programs. The proof term \( p_M \) is usually much longer than the program \( M \). Indeed, proof terms quickly become too long to be readable, in which case they will usually be omitted.

### 3.4.1 Coupled derivation rules for function types

First we look for \( \lambda \omega L \)-proof rules to accompany the \( \lambda \omega \)-type derivation rules for the introduction and elimination of function types. There are two programming language constructs involving function types, namely abstraction and application. In the examples below we consider how for programs of a certain form properties can be proved.

To make a clear distinction between judgements concerning programs and datatypes, and judgements concerning proofs and propositions, judgements \( \Gamma \vdash a : A \) where \( a \) and \( A \) are \( \lambda \omega \)-terms i.e. programs, datatype-constructors, or kinds - are written in boldface for the time being.

#### 3.25 Example (Abstraction).

Suppose we want a program \( f \) of type \( \sigma \rightarrow \tau \) satisfying a specification \((\forall x: \sigma. P x \Rightarrow Q x (fx))\), where \( \Gamma \vdash P: \sigma \rightarrow \mu \) and \( \Gamma \vdash Q: \sigma \rightarrow \tau \rightarrow \mu \).

So we want a program \( f \) and a proof \( p_f \) such that

\[
\Gamma \vdash f : \sigma \rightarrow \tau \quad \Gamma \vdash p_f : \forall x: \sigma. P x \Rightarrow Q x (fx)
\]

We can try a program of the form \((\lambda x: \sigma. M)\) for some \( M \). We then have to find a program \( M \) and a proof \( p_M \) such that

\[
(i) \quad \Gamma, x : \sigma \vdash M : \tau \quad (ii) \quad \Gamma, x : \sigma, p_x : P x \vdash p_M : Q x M
\]

From (i) it follows that \((\lambda x: \sigma. M)\) has the right type and from (ii) it follows that it satisfies the specification

\[
\Gamma \vdash (\lambda x : \sigma. M) : \sigma \rightarrow \tau \quad \Gamma \vdash (\lambda x : \sigma. \lambda p_x : P x \vdash p_M : Q x) \quad \forall x : \sigma. P x \Rightarrow Q x ((\lambda x \sigma. M) x)
\]

#### 3.26 Example (Application).

Suppose we want a program \( M \) of type \( \tau \) satisfying the specification \( Q' M \), where \( \Gamma \vdash \tau \rightarrow \mu \), and \( \Gamma \vdash Q' : \tau \rightarrow \mu \). So we want a program \( M \) and a proof \( p_M \) such that

\[
\Gamma \vdash M : \tau \quad \Gamma \vdash p_M : Q' M
\]
CHAPTER 3. THE BASIC SYSTEM : $\lambda \omega_L$

We can try a program of the form $fN$, with $N$ possibly occurring in the specification $Q'$
Then for some $\sigma : *_s$, $P \sigma \rightarrow *_p$ and $Q : \sigma \rightarrow \tau \rightarrow *_p$ such that $(Q N) \cong_\beta Q'$ we have to find a program $N$ with a proof $p_N$ such that

(i) $\Gamma \vdash N \ \sigma$  
(ii) $\Gamma \vdash p_N \ P N$

and a program $f$ with a proof $p_f$ such that

(iii) $\Gamma \vdash f : \sigma \rightarrow \tau$  
(iv) $\Gamma \vdash p_f : \forall x : \sigma. \ P x \Rightarrow Q x (fx)$

From (i) and (iii) it follows that $f N$ has the right type, and from (ii) and (iv) it follows that $f N$ satisfies the specification $Q'$:

$$\Gamma \vdash f N : \tau \quad \Gamma \vdash p_f N p_N \cdot Q N (f N) \cong_\beta Q' (f N)$$

In the previous example, the choice for a program of the form $(\lambda x : \sigma. \ M)$ was suggested by the type $\sigma \rightarrow \tau$ of the program. The choice for a program of the form $(f N)$ is not suggested by just the shape of its datatype (rather like to the choice of proving a proposition $\psi$ by proving $\phi \Rightarrow \psi$ and $\phi$ is not suggested by the shape of $\phi$).

In the examples above the predicates (specifications) on function types $\sigma \rightarrow \tau$ are of the form

$$\lambda f \ \sigma \rightarrow \tau \ \forall x : \sigma. \ P x \Rightarrow Q x (fx) \cdot (\sigma \rightarrow \tau) \rightarrow *_p$$

where $\Gamma \vdash P \ : \sigma \rightarrow *_p$ and $\Gamma \vdash Q : \sigma \rightarrow \tau \rightarrow *_p$. Here $P$ is the precondition, and $Q$ the post-condition. Not all predicates on a function type $\sigma \rightarrow \tau$ are of this form. However, all predicates that express some input-output relation (between input $x$ and output $f x$) are of this form. Note that for predicates of this form, different choices for $P$ and $Q$ are possible (e.g. $P' \equiv \lambda x : \sigma. \ True$ and $Q' \equiv \lambda x : \sigma. \ \lambda y : \tau. \ P x \Rightarrow Q x (f x)$). For the $\lambda \omega_s$-derivation rule for forming function types there is a corresponding derivable $\lambda \omega_L$-derivation rule for forming predicates of this form:

$$\frac{\Gamma \vdash \sigma : *_s}{\Gamma \vdash \tau : *_s} \quad \frac{\Gamma \vdash \sigma \rightarrow \tau}{\Gamma \vdash (\lambda f \ \sigma \rightarrow \tau. \ \forall x : \sigma. \ P x \Rightarrow Q x (fx)) \ : \ (\sigma \rightarrow \tau) \rightarrow *_p}$$

For $\Gamma \vdash \sigma : *_s, \tau : *_s, P : \sigma \rightarrow *_p, Q : \sigma \rightarrow \tau \rightarrow *_p$ the following coupled derivation rules for $\rightarrow$-introduction and elimination are derivable

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. \ M) : \sigma \rightarrow \tau} \quad \frac{\Gamma, x : \sigma, p_z : P x \vdash p_M : Q x M}{\Gamma \vdash (\lambda x : \sigma. \ \lambda p_z : P x. \ p_M) : (\forall x : \sigma. \ P x \Rightarrow Q x ((\lambda x : \sigma. \ M) x))}$$

$$\frac{\Gamma \vdash f : \sigma \rightarrow \tau}{\Gamma \vdash p_f : (\forall x : \sigma. \ P x \Rightarrow Q x (f x))} \quad \frac{\Gamma \vdash N : \sigma}{\Gamma \vdash p_f N p_N \cdot Q N (f N)}$$

3.4.2 Coupled derivation rules for polymorphic types

We now discuss proof rules to accompany the introduction and elimination rules for a polymorphic type ($\Pi_{\alpha, \mathcal{K}} : \tau$)
For predicates on polymorphic datatypes we could consider predicates that have a form similar to that of the predicates on function types. This would mean that on a datatype $(\Pi\alpha:K.\tau)$ we have predicates of the following form:

$$\lambda f:(\Pi\alpha:K.\tau)\;\forall\alpha K\;P\alpha \Rightarrow Q\alpha(f\alpha):(\Pi\alpha:K.\tau) \rightarrow *_p,$$

where $P:K \rightarrow *_p$ and $Q:(\Pi\alpha:K.\tau \rightarrow *_p)$. However, there are few - if any - interesting predicates $P$ on a kind $K$. So instead we consider specifications of the form

$$\lambda f:(\Pi\alpha:K.\tau).\;\forall\alpha K.\;Q(f\alpha):(\Pi\alpha:K.\tau) \rightarrow *_p,$$

where $\alpha:K \vdash Q:\tau \rightarrow *_p$. For such specifications we can find proof rules to accompany the introduction and elimination rules for a polymorphic type $(\Pi\alpha:K.\tau)$

Note that the dependency of $Q$ on $\alpha$ is left "implicit". It could be made explicit by considering predicates $Q$ with $\vdash Q:(\Pi\alpha:K.\tau \rightarrow *_p)$ instead of $\alpha:K \vdash Q:\tau \rightarrow *_p$. This would result in specifications of the form of the form $\lambda f:(\Pi\alpha:K.\tau).\;\forall\alpha K.\;Q'(f\alpha)$ Our only reason for not doing this is that the dependency of $\tau$ on $\alpha$ is also implicit.

For the $\lambda\omega$-derivation rule for forming polymorphic types there is a corresponding derivable $\lambda\omega_L$-derivation rule for forming specifications of the form (i):

\[
\frac{\Gamma, \alpha : K \vdash \tau : *_s}{\Gamma \vdash (\Pi \alpha : K.\tau) : \forall \alpha K. \tau \rightarrow *_p} \quad \frac{\Gamma, \alpha : K \vdash Q : \tau \rightarrow *_p}{\Gamma \vdash (\lambda f :(\Pi \alpha : K.\tau).\;\forall \alpha K.\;Q(f\alpha)) : (\Pi \alpha : K.\tau) \rightarrow *_p}
\]

For $\Gamma, \alpha : K \vdash \tau : *_s, Q : \tau \rightarrow *_p$ the following coupled derivation rules for $\Pi$-introduction and elimination are derivable:

\[
\frac{\Gamma, \alpha : K \vdash M : \tau}{\Gamma \vdash (\lambda \alpha : K.\tau)(\alpha : M) : (\Pi \alpha : K.\tau)} \quad \frac{\Gamma, \alpha : K \vdash p_M : Q M}{\Gamma \vdash (\lambda \alpha : K.\tau)(\alpha : p_M) : \forall \alpha K. Q M \Rightarrow \alpha \forall K. Q((\lambda \alpha : K.\tau)\alpha)}
\]

\[
\frac{\Gamma \vdash \rho : K}{\Gamma \vdash f : (\Pi \alpha : K.\tau)} \quad \frac{\Gamma \vdash f \rho : \tau[a := \rho]}{\Gamma \vdash p_f : \forall \alpha K. Q(f\alpha)} \quad \frac{\Gamma \vdash p_f : \forall \alpha K. Q(f\alpha)}{\Gamma \vdash p_f \rho : Q[a := \rho](f \rho)}
\]

The inference rules above can be specialised for specifications of a certain form, namely where $(\forall \alpha : K.\; Q(f\alpha))$ is of the form $(\forall \alpha : K.\; \forall \beta : P.\; \ldots)$, with $P$ some prop-kind that possibly depends on $\alpha$. This means the specification is of the form

$$\lambda f:(\Pi \alpha : K.\tau).\;\forall \alpha K.\;\forall \beta : P.\;\forall \beta : P.\;Q(f\alpha):(\Pi \alpha : K.\tau) \rightarrow *_p,$$

with $\alpha : K, P \vdash Q : \tau \rightarrow *_p$. Typically, if $K$ is $*_s$, i.e. $\forall \alpha K$ is a quantification over all datatypes, then $\forall \beta : P$ will be a quantification over all predicates of a certain type.

For the $\lambda\omega$-derivation rule for forming polymorphic types there is a corresponding $\lambda\omega_L$-derivation rule for forming predicates of the form (ii):

\[
\frac{\Gamma, \alpha : K \vdash \tau : *_s}{\Gamma \vdash (\Pi \alpha : K.\tau) : \forall \alpha K. \tau \rightarrow *_p} \quad \frac{\Gamma, \alpha : K, P : P \vdash Q : \tau \rightarrow *_p}{\Gamma \vdash (\lambda f :(\Pi \alpha : K.\tau).\;\forall \alpha K.\;\forall \beta : P.\;Q(f\alpha)) : (\Pi \alpha : K.\tau) \rightarrow *_p}
\]
For $\Gamma; \alpha : \mathcal{K}, P \vdash \tau : \ast_s, Q \quad \tau \rightarrow \ast_p$, the following coupled derivation rules for $\Pi$-introduction and elimination are derivable

\[
\begin{align*}
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash (\lambda \alpha : \mathcal{K}. M) : (\Pi \alpha : \mathcal{K}. \tau)} & \\
\frac{\Gamma \vdash \lambda \alpha : \mathcal{K}, P \vdash \tau \quad \Gamma \vdash P \vdash \Pi \cdot P \cdot \Pi \cdot Q \cdot M}{\Gamma \vdash (\lambda \alpha : \mathcal{K}. \lambda P \cdot P \cdot \Pi \cdot Q \cdot M) \cdot \forall \alpha : \mathcal{K} \cdot \forall P : \Pi. Q((\lambda \alpha : \mathcal{K}. M)\alpha) =_\beta \forall \alpha : \mathcal{K} \cdot \forall P : \Pi. Q} \\
\frac{\Gamma \vdash \rho : \mathcal{K}}{\Gamma \vdash f : (\Pi \alpha : \mathcal{K}. \tau)} & \\
\frac{\Gamma \vdash f \rho : \tau[\alpha := \rho]}{\Gamma \vdash f \rho : \forall \alpha : \mathcal{K}. \forall P : \Pi. Q(f \alpha)} & \\
\frac{\Gamma \vdash f \rho : \forall \alpha : \mathcal{K}. \forall P : \Pi. Q(f \alpha)}{\Gamma \vdash \rho : \Pi \vdash f \rho : \forall \alpha : \mathcal{K}. \forall P : \Pi. Q[\alpha := \rho]} & \\
\frac{\Gamma \vdash \rho : \Pi \vdash f \rho : \forall \alpha : \mathcal{K}. \forall P : \Pi. Q[\alpha := \rho]}{\Gamma \vdash \rho : \Pi \vdash f \rho : \forall \alpha : \mathcal{K}. \forall P : \Pi. Q[\alpha := \rho]} \\
\end{align*}
\]

3.27 Example The polymorphic identity

\[\text{Id} = (\lambda \alpha \ast_s. \lambda \tau \alpha. \tau) \quad (\Pi \alpha \ast_s. \alpha \rightarrow \alpha)\]
satisfies the specification

\[\forall \alpha \ast_s. \forall P \alpha \rightarrow \ast_p. \forall \tau \alpha. \forall x. \forall \alpha \ast_s. \forall x. \forall \alpha \ast_s. \Pi \rightarrow P (\text{Id} \alpha x)\]

Note that if we swap the second and third universal quantification, this becomes $\delta$-equal to \[\forall \alpha \ast_s. \forall x \alpha. x =_\alpha (\text{Id} \alpha x),\] with $=_\alpha$ as defined in definition 3.21. It follows from \[\alpha \ast_s, x \alpha \vdash x : \alpha\]
by the $\rightarrow$- and $\Pi$-introduction rules that $\text{Id}$ has type $(\Pi \alpha \ast_s. \alpha \rightarrow \alpha)$. It follows from \[\alpha : \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x \cdot P x \vdash p_x : P x\]
by the associated proof rules that $\text{Id}$ satisfies the specification mentioned above. These two derivations are:

\[
\begin{align*}
\frac{\alpha \ast_s, x \alpha \vdash x : \alpha}{\alpha \ast_s \vdash (\lambda x : \alpha. x) : \alpha \rightarrow \alpha} & \\
\frac{\alpha \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x : P x \vdash p_x : P x}{\alpha \ast_s \vdash P \alpha \rightarrow \ast_s, \tau \alpha, p_x \cdot P x \vdash p_x : P x \\
\frac{\alpha \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x : P x \vdash p_x : P x}{\alpha \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x : P x \vdash p_x : P x} & \\
\frac{\alpha \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x : P x \vdash p_x : P x}{\alpha \ast_s, P \alpha \rightarrow \ast_s, \tau \alpha, p_x : P x \vdash p_x : P x} \\
\end{align*}
\]

More and larger building blocks than just abstraction and application will be used for programs. Associated type derivation rules are needed to type programs constructed using these building blocks, and associated proof rules are needed to prove correctness of programs.

We will give a simple example of the development of a program with its correctness proof. But first two examples are given that introduce two basic datatypes, booleans and natural numbers, and coupled derivation rules for them.
3.4 PROGRAM AND PROOF DEVELOPMENT

3.28 EXAMPLE (Booleans). The following context introduces the booleans:

```
bool      : *s ,
true      : bool ,
false     : bool ,
if        : \alpha:*s, \alpha \to \alpha \to bool \to \alpha ,
pbool     : false \neq bool true ,
pif       : \forall \alpha:*s, \forall P.bool \to \alpha \to *p .
            \forall x \alpha . P true x \Rightarrow
            \forall y \alpha . P false y \Rightarrow
            \forall b:bool . P b (if b then x else y)
```

This context assumes that the two booleans are distinct and gives the specification of if . We write (if b then x else y) for (if \alpha xy b), so the type parameter \alpha is left implicit. false \neq bool true is short for \neg (false = bool true), with \neg and = bool as defined in definitions 3.11 and 3.21. Note that the specification of if is indeed of the form we discussed earlier.

Using pif we can define the following proof \piif:

```
\piif = \lambda h, \alpha, Q, x, p_x, y, p_y . pif (\lambda x:bool . \lambda a \alpha . b = bool true \Rightarrow (Q a)) x p_x y p_y
```

This gives the following coupled derivation rules for if:

```
\Gamma \vdash b : bool
\Gamma \vdash M : \alpha
\Gamma \vdash N : \alpha
---------------------
\Gamma \vdash if b then M else N : \alpha
\Gamma, p : (b = bool true) \vdash p M . Q M
\Gamma, p : (b = bool false) \vdash p N . Q N
\---------------------
\Gamma \vdash (\piif b bool Q M (\lambda p . p_M) N (\lambda p . p_N)) : Q (if b then M else N)
```

for \Gamma \vdash \sigma : *s and Q : \sigma \to *p .

Here the proof terms have already become too complicated to read. In the following examples they will be omitted.

3.29 EXAMPLE (Natural Numbers). The following context introduces the natural numbers and a primitive recursor with its specification

```
nat      : *s ,
0        : nat ,
S        : nat \to nat ,
primrec  \Pi \alpha:*s, \alpha \to (nat \to \alpha \to \alpha) \to nat \to \alpha ,
primrec  \forall \alpha:*s, \forall P.nat \to \alpha \to *p .
            \forall x \alpha . P 0 x \Rightarrow
            \forall f:nat \to \alpha \to \alpha . (\forall n:nat . y \alpha . P n y \Rightarrow P (Sn) (f n y)) \Rightarrow
            \forall n:nat . P n (primrec \alpha x f n)
```
Note that the specification of primrec is indeed of the form discussed earlier. This specification results in the following coupled derivation rules for primrec:

\[
\begin{align*}
\Gamma \vdash M : \sigma \\
\Gamma, n : \text{nat}, \text{rec} : \sigma \vdash N : \sigma \\
\Gamma \vdash \text{primrec} \sigma M (\lambda n \text{nat} : \lambda \text{rec} : \sigma N) : \text{nat} \rightarrow \sigma \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash p_M \cdot P \circ M \\
\Gamma, n : \text{nat}, \text{rec} : \sigma, p_{\text{rec}} : P \cdot n \text{rec} = p_N \cdot P (S n) N \\
\Gamma \vdash \ldots \ \ f = \text{primrec} \sigma M (\lambda n \text{nat} : \lambda \text{rec} : \sigma N) : \text{nat} \rightarrow \sigma \\
\end{align*}
\]

for \( \Gamma \vdash \sigma : \Sigma \) and \( P \cdot \text{nat} \rightarrow \sigma \rightarrow \Sigma \).

\[30\] **Example.** Using the booleans and natural numbers introduced in the previous example we now construct a program that tests if a natural number is odd or even. For this the predicates \( \text{Even} \) and \( \text{Odd} \) and their relevant properties are declared:

\[
\begin{align*}
\text{Even}, \text{Odd} : \text{nat} \rightarrow \Sigma_p, \\
\text{Even} \ 0 : \Sigma_p, \\
\forall n \text{nat} \ \text{Even} n \Rightarrow \text{Odd} (S n), \\
\forall n \text{nat} \ \text{Odd} n \Rightarrow \text{Even} (S n), \\
\end{align*}
\]

We define

\[
P = \lambda n \text{nat} : \lambda b \text{bool}. ((\text{Even} n \land b = \text{bool} \text{\ true}) \lor (\text{Odd} n \land b = \text{bool} \text{\ false}) \\
: \text{nat} \rightarrow \text{bool} \rightarrow \Sigma_p.
\]

It is easy to prove \( (P \ n \ \text{true} \iff \text{Even} \ n) \) and \( (P \ n \ \text{false} \iff \text{Odd} \ n) \). For the implication from left to right we need \( \text{false} \neq \text{bool} \ \text{true} \).

We are looking for a program \( \text{even} : \text{nat} \rightarrow \text{bool} \) that satisfies the specification \( \forall n \text{nat} \ P \ n (\text{even} n) \), so we are looking for a program \( \text{even} \) and a proof \( p_{\\text{even}} \) such that

\[
\Gamma \vdash \text{even} : \text{nat} \rightarrow \text{bool} \\
\Gamma \vdash p_{\\text{even}} : \forall n \text{nat} \ P \ n (\text{even} n)
\]

We choose to use the primitive recursor \( \text{primrec} \) for this program, i.e. \( \text{even} \) will be of the form \( (\text{primrec} \ \text{nat} \ M (\lambda n \text{nat} : \lambda \text{rec} : \sigma N)) \) for certain \( M \) and \( N \). By the coupled derivation rules for \( \text{primrec} \) this leaves us with the task of finding programs \( M \) and \( N \) with proofs \( p_N \) and \( p_M \) such that

\[
\Gamma \vdash M : \text{bool}, \ p_M : P \circ M \\
\forall n \text{nat}, b : \text{bool}, p_b : P \ n \ b \vdash N : \text{bool}, \ p_N : P (S n) N
\]

\( P \circ M \triangleq p_b (\text{Even} 0 \land M = \text{bool} \text{\ true}) \lor (\text{Odd} 0 \land M = \text{bool} \text{\ false}) \), so we choose \( M = \text{true} \). Then we have to prove \( P \ 0 \text{\ true} \), which follows from \( \text{Even} \ 0 \).

We choose \( N \equiv (\text{if} \ b \text{\ then} \ N_1 \text{\ else} \ N_2) \) for some \( N_1 \) and \( N_2 \). This amounts to distinguishing the cases \( b \text{\ is true} \) and \( b \text{\ is false} \), and by \( P \ n \ b \) this means distinguishing the cases that \( n \) is even and \( n \) is odd. We then have to find \( N_1 \) and \( N_2 \) with proofs \( p_{N_1} \) and \( p_{N_2} \) such that

\[
n \text{nat}, \ b : \text{bool}, \ p_b : P \ n \ b, \ p : b = \text{bool} \text{\ true} \vdash N_1 : \text{bool}, \ p_{N_1} : P (S n) N_1
\]
and
\[ n : \text{nat}, b : \text{bool}, p_b : P \text{ n b}, p : b = \text{bool} \quad \vdash \quad N_2 : \text{bool}, p_{N_2} : P(\text{n n}) N_2. \]

Finally, if \( P \text{ n b} \) and \( b = \text{bool} \text{ true} \), then \( \text{Even n} \), and hence \( \text{Odd (S n)} \). Similarly, if \( P \text{ n b} \) and \( b = \text{bool} \text{ false} \), then \( \text{Odd n} \), and hence \( \text{Even (S n)} \). \( \text{Odd (S n)} \) is equivalent with \( P(\text{n n}) \) false, \( \text{Even (S n)} \) with \( P(\text{n n}) \) true. So we take \( N_1 \equiv \text{false} \) and \( N_2 \equiv \text{true} \), and then proofs \( p_{N_1} \) and \( p_{N_2} \) can be found such that
\[
\begin{align*}
n : \text{nat}, \ b : \text{bool}, \ p_b : P \text{ n b}, \ p : b = \text{bool} \quad &\vdash \quad p_{N_1} : P(\text{n n}) \text{false} \\
n : \text{nat}, \ b : \text{bool}, \ p_b : P \text{ n b}, \ p : b = \text{bool} \quad &\vdash \quad p_{N_2} : P(\text{n n}) \text{true}
\end{align*}
\]

\[ \Box \]

### 3.4.3 Other datatype-constructors

So far, we have only considered the annotation of programs and datatypes. Programs are annotated with (correctness) proofs, datatypes with predicates. In \( \lambda \omega \), there are other datatype constructors besides the datatypes. We now consider how these can be annotated in \( \lambda \omega_L \).

These other datatype constructors are functions for constructing datatypes, such as \( \text{list} : *_* \rightarrow *, \) and \( \times : *_* \rightarrow *_* \rightarrow *_* \). They can be annotated with functions for constructing predicates. For instance, the datatype-constructor \( \text{list} : *_* \rightarrow *, \) which maps datatypes to datatypes, could be annotated with a polymorphic function \( \text{LIST} \), that maps predicates to predicates:
\[ \text{LIST} \cdot (\Pi \alpha : *_*, (\alpha \rightarrow *_*) \rightarrow (\text{list } \alpha) \rightarrow *_*) \]

So for any datatype \( \alpha \), \( \text{LIST } \alpha \) maps a predicate on \( \alpha \) to a predicate on \( \text{list } \alpha \). The obvious choice for \( \text{LIST} \) is of course the function that maps a predicate \( P \) \( \alpha \rightarrow *_p \) on \( \alpha \) to the predicate "\( P \) holds for all elements" on \( \text{list } \alpha \). This function \( \text{LIST} \) can be seen as a "lifted" version of \( \alpha \). Of course, not all interesting predicates on a datatype \( \text{list } \alpha \) are of the form \( \text{LIST } \alpha \ P \) for some \( P \), but some are. For example, we can expect that any polymorphic program \( f : (\Pi \alpha : *_*, (\alpha \rightarrow *_*) \rightarrow (\text{list } \alpha)) \) will satisfy
\[
\forall \alpha : *_*, \forall P : \alpha \rightarrow *_p \quad \forall l \quad (\text{LIST } \alpha \ P l) \Rightarrow (\text{LIST } \alpha \ P (f l)) \quad \tag{i}
\]

This means that the list that is the output of \( f \) can only contain elements that occur in the list that is its input.

In the remainder of this section, it will be shown that, for any closed datatype-constructor \( \text{list} : *_* \rightarrow *_* \) in \( \lambda \omega_L \), there exists a lifted version \( \text{LIST} \) in \( \lambda \omega_L \) such that all closed programs \( f : (\Pi \alpha : *_*, (\alpha \rightarrow *_*) \rightarrow (\text{list } \alpha)) \) satisfy property (i). To do this, lifted versions of all datatype constructors will be defined. In general, a datatype-constructor \( \sigma : K \) can be annotated by a lifted version, which is of type \( \text{type } \text{of } \text{lift_{K_1}}(\sigma) \).

### 3.31 Definition: For datatype-constructor \( \sigma \) of kind \( K \), with \( \Gamma \vdash \sigma : K \text{ \square}_a \), the \( \lambda \omega_L \)-propkind \( \text{type } \text{of } \text{lift_{K_1}}(\sigma) \), with \( \Gamma \vdash \text{type } \text{of } \text{lift_{K_1}}(\sigma) : \Box_p \), is defined as follows:
\[
\begin{align*}
\text{type } \text{of } \text{lift_{K_1}}(\sigma) & \equiv \sigma \rightarrow *_p \\
\text{type } \text{of } \text{lift_{K_1}}(\sigma) & \equiv (\Pi \alpha : K_1 : \text{type } \text{of } \text{lift_{K_1}}(\sigma) \rightarrow \text{type } \text{of } \text{lift_{K_2}}(\sigma) \alpha)
\end{align*}
\]

So \( \text{type } \text{of } \text{lift_{K_1}}(\text{list}) \equiv (\Pi \alpha : *_*, (\alpha \rightarrow *_p) \rightarrow (\text{list } \alpha) \rightarrow *_p \), which is indeed the type for \( \text{LIST} \) we gave earlier.

\[ \Box \]
3.32 Example. The annotation of a datatype-constructor \( \times \colon *_* \rightarrow *_* \rightarrow *_* \) would be of type

\[
\text{type-of-lift}_{*_* \rightarrow *_* \rightarrow *_*}(\times) \equiv \Pi \alpha \approx \Pi \beta \approx \Pi \gamma \approx (\alpha \times \beta) \rightarrow *_p ,
\]
where \( \times \) is written infix. So \( \times \) - a function that maps two datatypes to a datatype - can be annotated with a (polymorphic) function that maps two predicates to a predicate. The obvious candidate is the function which maps the predicates \( P \) on \( \alpha \) and \( Q \) on \( \beta \) to the predicate "\( P \) holds for the left component and \( Q \) for the right component" on \( \alpha \times \beta \). □

For a closed datatype-constructor \( \sigma \), it is possible to define a "lifted" version \( \text{lift}_\sigma \), a prop-constructor of type \( \text{type-of-lift}_{\text{K}(\sigma)}(\sigma) \), by induction on the structure of \( \sigma \).

3.33 Definition. For a datatype-constructor \( \sigma \) the prop-constructor \( \text{lift}_\sigma \) is defined by

\[
\begin{align*}
\text{lift}_\alpha & \equiv P_\alpha, \quad \text{if } \alpha \text{ is a variable} \\
\text{lift}_{p \rightarrow \sigma} & \equiv \lambda f \colon p \rightarrow \sigma. \forall x \colon p. \text{lift}_\sigma(x) \Rightarrow \text{lift}_\sigma(f(x)) \\
\text{lift}_{\Pi \alpha \approx \Pi \beta \approx \Pi \gamma \approx} & \equiv \lambda f \colon (\Pi \alpha \approx \Pi \beta \approx \Pi \gamma \approx \sigma). \forall \alpha, \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\alpha)}(\alpha) \Rightarrow \text{lift}_\gamma(f(\alpha)) \\
\text{lift}_{\lambda \alpha \approx \Pi \beta \approx} & \equiv \lambda g \colon (\lambda \alpha \approx \Pi \beta \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\gamma)}(\beta) \Rightarrow \text{lift}_\gamma(x) \\
\text{lift}_{\nu \alpha \approx \Pi \beta \approx} & \equiv \lambda g \colon (\nu \alpha \approx \Pi \beta \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\beta)}(\gamma) \Rightarrow \text{lift}_\gamma(x)
\end{align*}
\]

Here we assume that for every variable \( \alpha \in \text{Var}^p \), there is a variable \( P_\alpha \in \text{Var}^F \). □

3.34 Example. For the datatypes \( (\Pi \alpha \approx \ast_* \approx (\Pi \alpha \approx \ast_* \approx \alpha \rightarrow \alpha)) \) we get

\[
\begin{align*}
\text{lift}_{(\Pi \alpha \approx \ast_* \approx \Pi \alpha \approx \ast_* \approx \alpha \rightarrow \alpha)} & \equiv \lambda f \colon (\Pi \alpha \approx \ast_* \approx \Pi \alpha \approx \ast_* \approx \sigma). \forall P \colon \alpha \rightarrow \ast_* \rightarrow P(f(\alpha)) \\
\text{lift}_{(\Pi \alpha \approx \ast_* \approx \alpha \rightarrow \alpha)} & \equiv \lambda g \colon (\Pi \alpha \approx \ast_* \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\gamma)}(\beta) \Rightarrow \text{lift}_\gamma(g(x)) \\
\text{lift}_{(\nu \alpha \approx \Pi \beta \approx \ast_* \approx (\nu \alpha \approx \Pi \beta \approx \alpha \rightarrow \alpha))} & \equiv \lambda f \colon (\nu \alpha \approx \Pi \beta \approx \ast_* \approx (\nu \alpha \approx \Pi \beta \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\beta)}(\gamma) \Rightarrow \text{lift}_\gamma(f(x)) \\
\text{lift}_{(\nu \alpha \approx \Pi \beta \approx \alpha \rightarrow \alpha)} & \equiv \lambda g \colon (\nu \alpha \approx \Pi \beta \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\beta)}(\gamma) \Rightarrow \text{lift}_\gamma(x)
\end{align*}
\]

3.35 Example. Suppose \( \text{list} \colon *_* \rightarrow *_* \) as a closed datatype-constructor. Let \( \text{LIST} \equiv \text{lift}_{\text{list}} \). Then definition 3.33 results in the following predicate on \( (\Pi \alpha \approx \ast_* \approx \text{list} \rightarrow \text{list}(\alpha)) \)

\[
\begin{align*}
\text{lift}_{(\Pi \alpha \approx \ast_* \approx \nu \alpha \approx \Pi \beta \approx \ast_* \approx (\nu \alpha \approx \Pi \beta \approx \sigma \rightarrow \text{list}(\alpha)))} & \equiv \lambda f \colon (\Pi \alpha \approx \ast_* \approx \nu \alpha \approx \Pi \beta \approx \sigma). \forall P \colon \alpha \rightarrow \ast_* \rightarrow \text{list}(\alpha) \Rightarrow (\text{LIST} \circ P \circ f(\alpha)) \\
\text{lift}_{(\nu \alpha \approx \Pi \beta \approx \ast_* \approx (\nu \alpha \approx \Pi \beta \approx \sigma \rightarrow \text{list}(\alpha)))} & \equiv \lambda g \colon (\nu \alpha \approx \Pi \beta \approx \sigma). \forall x \colon \alpha. \forall \beta, \gamma \colon \text{type-of-lift}_{\text{K}(\beta)}(\gamma) \Rightarrow \text{lift}_\gamma(x)
\end{align*}
\]

For all closed datatype-constructors \( \sigma \) the prop-constructors \( \text{lift}_\sigma \) are of the right type.

3.36 Lemma. If \( \epsilon \vdash \sigma \vdash \square \), then \( \epsilon \vdash \text{lift}_\sigma : \text{type-of-lift}_{\text{K}(\sigma)}(\sigma) \).

Proof. A straightforward induction on the structure of \( \sigma \) proves that if \( \Gamma \vdash \lambda \omega, \sigma : \text{K} \vdash \square \), then \( \text{lift}(\Gamma) \vdash \lambda \omega \vdash \text{type-of-lift}_{\text{K}(\sigma)}(\sigma) \), where

\[
\begin{align*}
\text{lift}(\epsilon) & \equiv \epsilon \\
\text{lift}(\Gamma, \alpha, \sigma) & \equiv \text{lift}(\Gamma), \sigma, \alpha, \nu \beta, \text{lift}_\sigma \nu \beta \\
\text{lift}(\Gamma, \alpha, \text{K}) & \equiv \text{lift}(\Gamma), \alpha, \text{K}, \nu \beta, \text{type-of-lift}_{\text{K}(\sigma)}, \text{lift}_\sigma \nu \beta
\end{align*}
\]

Here we assume that for all variables \( \alpha \in \text{Var}^F \) and \( r \in \text{Var}^R \), there are variables \( P_\alpha \in \text{Var}^F \) and \( p_\beta \in \text{Var}^R \). □
This means that, for closed datatypes $\sigma$, $lift_{\sigma} : \text{type of } lift_{\sigma}(\sigma) \equiv \sigma \rightarrow *_{p}$, i.e. $lift_{\sigma}$ is a predicate on $\sigma$. We already saw examples of such predicates in examples 3.34 and 3.35. All closed programs of type $\sigma$ satisfy this predicate.

3.37 Theorem. If $\epsilon \vdash M : \sigma : *_{s}$, then there is a proof $p_{M}$ such that $\epsilon \vdash p_{M} : (lift_{\sigma} M)$.

Proof. The proof $p_{M}$ can be found by applying the $\lambda \omega_{L}$-derivation rules coupled to the $\lambda \omega_{s}$-derivation rules used in this derivation of $M : \sigma$. A straightforward induction on the structure of $M$ proves that if $\Gamma \vdash \lambda \omega_{s} \ M : \sigma : *_{s}$ then $lift(\Gamma) \vdash \lambda \omega_{L} lift_{M} : lift_{\sigma} M$, where

\[
\begin{align*}
lift_{x} & \triangleq p_{x} \text{ if } x \text{ is a program variable} \\
lift_{\alpha} & \triangleq P_{\alpha} \text{ if } \alpha \text{ is a datatype-constructor variable} \\
lift_{\lambda x. \rho M} & \triangleq \lambda x. \rho lift_{x} M \\
lift_{\lambda \alpha. K M} & \triangleq \lambda \alpha. K. lift_{\alpha} M \\
lift_{M N} & \triangleq lift_{M} N lift_{N} \\
lift_{M \rho} & \triangleq lift_{M} \rho lift_{\rho}
\end{align*}
\]

Here we assume that for all variables $\alpha \in \text{Var}^{D^{s}}$ and $x \in \text{Var}^{*s}$ there are variables $P_{\alpha} \in \text{Var}^{D^{s}}$, and $p_{x} \in \text{Var}^{*s}$.

3.38 Example. In example 3.34 we saw that

\[
\begin{align*}
lift\Pi\alpha. \alpha, \alpha & \equiv \lambda f. (\Pi \alpha : *_{s}. \alpha). \forall P : \alpha \rightarrow *_{p} . P (f \alpha) \\
lift\Pi\alpha. \alpha \rightarrow \alpha & \equiv \lambda g. (\Pi \alpha : *_{s}. \alpha \rightarrow \alpha) . \forall P : \alpha \rightarrow *_{p} . \forall x : \alpha . P x \Rightarrow P (g \alpha x)
\end{align*}
\]

Now by theorem 3.37, for all closed programs $f : (\Pi \alpha : *_{s}. \alpha)$ and $g : (\Pi \alpha : *_{s}. \alpha \rightarrow \alpha)$ there are proofs of $(\forall P : \alpha \rightarrow *_{p} . P (f \alpha))$ and $(\forall P : \alpha \rightarrow *_{p} . \forall x : \alpha . P x \Rightarrow P (g \alpha x))$.

This means that, for a datatype $\alpha$, $g \alpha$ does not just map programs of type $\alpha$ to programs of type $\alpha$, but it also maps elements of any subset of $\alpha$ to elements of that subset. This is also the case if this subset is a singleton subset, so this leaves only one possibility for $g \alpha$, namely the identity on $\alpha$.

Similarly, for a datatype $\alpha$, $f \alpha$ is not just a program of type $\alpha$, but also an element of every subset of $\alpha$, including the empty subset. Clearly, no such $f$ can exist.

What theorem 3.37 is basically saying is that any subset of a $\lambda \omega_{s}$-datatype could be allowed as a new datatype. Theorem 3.37 does not say anything about the relation between programs $f \rho$ and $f \sigma$ that are obtained by applying a single polymorphic program $f : (\Pi \alpha : R. \sigma)$ to different datatypes, like the notion of parametricity discussed in for instance [Rey83] [Wad89].
3.5 Comparison with Program Extraction in $\lambda C$

In [PM89b] Paulin-Mohring describes an approach to program construction that uses program extraction. This approach is quite different from the one we propose, as already discussed in the introduction. However, it involves closely related type systems, which is why we discuss it here in a bit more detail.

Program extraction is used in internal programming logics. Here the Curry-Howard-de Bruijn isomorphism is exploited by identifying the notions of program and proof and the notions of type and specification (to a certain extent). This idea dates back to Heyting's semantics of constructive proofs: a constructive proof $p$ of $\forall x \sigma. P x \rightarrow (\exists y . \tau. Q x y)$ contains an algorithm which, given a term $x$ and a proof $p_x$ of $P x$, returns a term $y$ and a proof $p_y$ of $Q x y$ (together, this witness $y$ and proof $p_y$ form a proof of $(\exists y . \tau. Q x y)$). The type of $p$ gives all the relevant information about this algorithm: it is its specification. Sufficiently expressive type systems, such as Martin-Löf's Type Theory and the Calculus of Constructions, can be used as programming logics in this way.

The main problem with this approach is that proofs contain redundant information. Typically, a large part of $p$ is devoted to the computation of $p_y$, and as programmers we are only interested in $y$. As a result, $p$ is inefficient and difficult to read.

One way to get rid of these irrelevant computations is to extract programs from proofs. In [PM89b] it is described how programs can be extracted from proofs in a variant of the Calculus of Constructions. The extracted programs are typable in $\lambda \omega$. For example, a $\lambda \omega$-program $f : \sigma \rightarrow \tau$ satisfying $\forall x . \sigma. P x \Rightarrow Q x (f x)$ is extracted from a constructive proof of $\forall x . \sigma. P x \Rightarrow (\exists y . \tau. Q x y)$. The proofs from which programs are extracted are constructed in the following variant of the Calculus of Constructions:

**3.39 Definition** $\lambda C_{pe}$ is the PTS specified by

$$S = \{*, \text{C}_1, *_{p}, \text{C}_p\}, \quad A = \{*, \text{D}_1, *_{p}; \text{D}_p\}, \quad R = S \times S$$

In [PM89b] the sort $*$ of $\lambda C_{pe}$ is called $\text{Spec}$, and $*_{p}$ is called $\text{Prop}$.

In this system, we can no longer think of all types $\sigma$ as datatypes, nor of all terms $M : \sigma$ as programs. For example, it is possible to form a program-dependent-type such as $(\leq n m) *_{p}$, where $\leq : \text{nat} \rightarrow \text{nat} \rightarrow *_{p}$, that consists of (constructive) proofs that $n$ is less than $m$. Inhabitants of $*_{p}$ are called informative propositions, and inhabitants of $*_{p}$ are called non-informative propositions.

The distinction between $*$ and $*_{p}$ is needed to control which parts of a proof are extracted to produce a program. The extraction procedure discards proofs of non-informative propositions, and keeps only the proofs of informative propositions. For instance, in order to extract a program of type $\sigma \rightarrow \tau$ from a proof of $\forall r : \sigma. P x \Rightarrow (\exists y . \tau. Q x y)$, the types $\sigma$ and $\tau$ have to be informative, i.e., $\sigma, \tau : *_{p}$, and the types $P x$ and $Q x y$ have to be non-informative, i.e., $P x, Q x y : *_{p}$.

The extraction procedure uses two mappings, $E$ and $R$. The function $E$ extracts programs and datatypes from $\lambda C_{pe}$-proofs and propositions, the function $R$ extracts the associated correctness proofs and predicates.
3.5. COMPARISON WITH PROGRAM EXTRACTION IN \(\lambda C\)

3.40 Theorem ([PM89b]). There are mappings \(\mathcal{E}\) and \(\mathcal{R}\) such that:

- if \(\Gamma \vdash \sigma : *_s\) in \(\lambda C_{ps}\), then
  \[\mathcal{E}(\Gamma) \vdash \mathcal{E}(S) : *_s\) in \(\lambda w_s\), and
  \[\mathcal{R}(\Gamma) \vdash \mathcal{R}(S) \quad \mathcal{E}(S) \rightarrow *_p\) in \(\lambda w_L\) extended with the rule \((*_p, \Box_p)\).

- if \(\Gamma \vdash s : S : *_s\) in \(\lambda C_{ps}\), then
  \[\mathcal{E}(\Gamma) \vdash \mathcal{E}(s) : \mathcal{E}(S)\) in \(\lambda w_s\), and
  \[\mathcal{R}(\Gamma) \vdash \lambda \omega_L \quad \mathcal{R}(s) : \mathcal{R}(S)\mathcal{E}(s)\) in \(\lambda w_L\) extended with the rule \((*_p, \Box_p)\).

In [PM89b] the sort \(*_s\) of \(\lambda w_s\) is called \textit{Data}. \(\square\)

So if \(S\) is an informative proposition, then \(\mathcal{E}(S)\) is a datatype and \(\mathcal{R}(S)\) is a predicate on that datatype. If \(s\) is a constructive proof of an informative proposition \(P\), then \(\mathcal{E}(s)\) is a program of type \(\mathcal{E}(S)\) and \(\mathcal{R}(s)\) is a proof that this program satisfies the predicate \(\mathcal{R}(S)\).

The example below shows how correctness proof and program are intertwined in a constructive proof.

3.41 Example. Suppose \(\Gamma \vdash \sigma : \sigma : *_s\), \(\Gamma \vdash P : \sigma \rightarrow *_p\) and \(\Gamma \vdash Q : \sigma \rightarrow \tau \rightarrow *_p\) in \(\lambda w_L\) (and hence in \(\lambda C_{ps}\)). Define the informative \(\lambda C_{ps}\)-proposition \(S\) as follows

\[S = \forall x : \sigma. P x \Rightarrow (\exists y : \tau. Q x y) : *_s\]

Then
\[
\begin{align*}
\mathcal{E}(S) &= \sigma \rightarrow \tau \\
\mathcal{R}(S) &= \lambda f : \sigma \rightarrow \tau. \forall x : \sigma. P x \Rightarrow Q x (f x)
\end{align*}
\]

Suppose that \(M\) is a \(\lambda w_s\)-term such that \(\Gamma, x : \sigma \vdash M : \tau\). Suppose that in \(\lambda C_{ps}\) we have the following inhabitant of \(S\)

\[m = \lambda x : \sigma. \lambda p : P x. \exists y : \tau. Q x y \quad \text{where } p = M \quad \text{is a proof of } (\exists y : \tau. Q x y) \quad \text{consisting of a witness } M \text{ and a proof}\]

\[p = Q x M \quad \text{Then } \mathcal{E}(S) = \lambda x : \sigma. M : \mathcal{E}(S) \quad \mathcal{R}(S) = \lambda x : \sigma. \lambda p : P x. p = M \quad \mathcal{R}(S)\mathcal{E}(s)\]

3.42 Theorem. Any program \(M : \sigma : *_s\) and correctness proof \(p_M : S M : *_p\) that can be constructed in \(\lambda w_L\) can also be obtained by program extraction from a proof of an informative proposition in \(\lambda C_{ps}\)

\[\square\]

Proof. The simplest way is to take as the \(\lambda C_{ps}\)-proposition \((\exists x : \sigma. S x) : *_s\), and as its (constructive) proof the pair consisting of the witness \(M\) and proof \(p_M\) that it is indeed a witness.

Note that this is not the only way, and not the best. There are many proofs of different informative propositions from which a certain program and correctness proof can be extracted. For example, a program \(f : \sigma \rightarrow \tau\) satisfying \(\forall x : \sigma. P x \Rightarrow Q x (f x)\) can be extracted from a proof of \(\forall x : \sigma. P x \Rightarrow (\exists y : \tau. Q x y)\) or \(\exists f : \sigma \rightarrow \tau. P f \Rightarrow Q x (f x)\). The whole idea behind using program extraction is that we prefer to prove the former. To prove the latter we have to provide a program, the witness, and a separate correctness proof, so in this case there is no real need for an extraction procedure to obtain these from the proof.
Not all programs and correctness proofs that are obtained by program extraction from a \( \lambda C_{ps} \)-proof can be constructed in \( \lambda \omega_L \). This is because \( \mathcal{R} \) produces predicates and correctness proofs that are typeable in \( \lambda \omega_L \) extended with \((*, \Box_p)\). This extra rule allows the formation of propositions depending on proofs, i.e. propositions that express properties of proofs. However, using this rule goes against the idea that the inhabitants of \( *_p \) are non-informative propositions, whose proofs do not have any computational meaning.
Chapter 4

Simple Extensions

Like all PTSs, the systems introduced in the previous chapter provide only very few primitives for term and type formation. In this chapter they are extended with more type constructors. In addition to the dependent product type constructor \( \Pi \), we introduce the type constructors \( \times \), \( + \) and \( \Sigma \), for the formation of (labelled) cartesian products, (labelled) disjoint sums, and weak dependent sums, respectively.

Especially in the programming language we want more primitives for term and type formation than those available in \( \lambda \omega \). We want more datatypes than just polymorphic types \((\Pi \alpha.K.\sigma)\) and function types \(\sigma \to \tau\), and more ways of constructing programs than just lambda abstraction and application. The extensions provide a richer syntax for the programming language. The labelled products and sums provide record and variant types. The dependent sums provide abstract datatypes.

In the logic the new type constructors provide the logical connectives conjunction, disjunction and existential quantification as primitives. In the logic there is not such a pressing need for more primitives for terms and types as in the programming language. Readability of proof terms is not as important as readability of programs. Also, we are not interested in the reduction behaviour of proofs.

The strength of the type systems is not really increased by the new type constructors. There are encodings of all the new types in the original systems, except for the labelled products and sums. There are several reasons for introducing \( + \), \( \times \) and \( \Sigma \) as primitive notions. Having them as primitives means that we can choose their interpretations in a model (which may be different from the interpretations of their encodings), which makes it easier to introduce specific axioms for them in the logic. Also, if \( + \), \( \times \) and \( \Sigma \) are primitives, they can be implemented more efficiently. Finally, there is no way to encode labels and labelled products and sums in \( \lambda \omega_L \).

The new primitives do not introduce new dependencies between the different levels, so that the hierarchy formed by these levels does not change.

Before we extend the particular type systems with \( + \), \( \times \) and \( \Sigma \), we first consider in section 4.1 how the new type constructors can be included in the PTS-framework. Then in sections 4.2 and 4.3 the programming language \( \lambda \omega \), and the programming logic \( \lambda \omega_L \) are extended with \( + \), \( \times \) and \( \Sigma \). The resulting system are called \( \lambda \omega^+_L \) and \( \lambda \omega^+_L \). Finally, in section 4.4 we consider the construction of programs with their correctness proofs.
4.1 Extending PTSs with $+$, $\times$ and $\Sigma$

In this section the PTS-framework is extended to include more types than just the dependent product types $(\Pi x. A. B)$, namely

- cartesian product types $A \times B$ and labelled cartesian products,
- disjoint sum types $A + B$ and labelled disjoint sums,
- dependent sum types $(\Sigma x. A. B)$ (also known as existential or weak sum types)

We do not intend to give the most general extension of PTSs with these additional type constructors, only one that is general enough for our purposes. In particular, for $+$- and $\Sigma$-types more powerful elimination rules can be given. The different type constructors are treated as uniformly as possible. We compare different notations for $+$- and $\Sigma$-types, and choose one which emphasizes the similarity between them. Later we may use one of the other notations if this is easier to read in a particular situation.

In the specification $(S, A, R)$ of a PTS the set of rules $R$ specifies which $\Pi$-types can be formed. For every new type constructor an additional set of rules will be required to specify which types can be formed with it. The set of rules $R$ controlling the formation of $\Pi$-types will be called $R_{\Pi}$ from now on.

4.1 Definition.

A specification of a (D)PTS with $+$, $\times$ and $\Sigma$ is a 6-tuple $(S, A, R^\Pi, R^\Sigma, R^\times, R^+)$ with

- $S$ is a set of symbols called the sorts
- $A \subseteq S \times S$,
- $R^\Pi, R^\Sigma \subseteq S \times S$,
- $R^\times, R^+ \subseteq S$.

If $S = (S, A, R^\Pi, R^\Sigma, R^\times, R^+)$ is a specification, then $\lambda S$ denotes the PTS with $+$, $\times$ and $\Sigma$ it specifies, and $\lambda S_{\lambda}$ denotes the DPTS with $+$, $\times$ and $\Sigma$ it specifies.

One simplification compared with the original definition of a PTS-specification (definition 2.1) is that $R^\Pi \subseteq S \times S$ instead of $R^\Pi \subseteq S \times S \times S$. For the system we are interested in no rules $(s_1, s_2, s_3)$ with $s_2 \neq s_3$ are needed to control the formation of $\Pi$-types (or $\Sigma$-types, for that matter). As a result, the definition of functional can be simplified:

4.2 Definition. A (system specified by $s$) specification $(S, A, R^\Pi, R^\Sigma, R^\times, R^+)$ is called functional if $A$ is a function, i.e. if

$$(s, s') \in A \land (s, s'') \in A \Rightarrow s' = s''.$$  

Extending a PTS or DPTS with $+$-, $\times$- and $\Sigma$-types comes down to

- extending the set of pseudotermwith new language constructs.
4.1. EXTENDING PTSs WITH $+$, $\times$ AND $\Sigma$

- adding new reduction rules, and
- adding new type inference rules.

The extension with $+$, $\times$ or $\Sigma$ can be defined simultaneously for PTSs and DPTSs. In the remainder of this section we give for each individual type constructor the new pseudoterm, reduction rules and inference rules. The (binary) cartesian products $A \times B$ and disjoint sums $A + B$ are special cases of the labelled ones. Because they are so much more familiar they are first treated separately.

Each new type constructor comes with its $\beta$- and $\eta$-reduction rules. As in the definition of PTSs in chapter 2, we will only have $\beta$-reduction in the type systems. The reason is that, as in chapter 2, the $\eta$-rules complicate matters because they destroy the Church-Rosser property for the pseudoterms. Also, for labelled products the $\eta$-rule will turn out to be a conditional rule that depends on the typing relation. The models that are discussed later will be extensional and respect $\eta$-equality.

4.1.1 Cartesian Products

Cartesian product types are of the form $A \times B$. The type $A \times B$ is the type of pairs consisting of a term of type $A$ and a term of type $B$. The associated term constructions are pairing and projection. The pair consisting of $a$ and $b$ is denoted by $(a, b)$. The first component of a pair $c$ is denoted by $c.1$, the second by $c.2$. The formation of product types is controlled by a set of rules $R^\times \subseteq S$. The type $A \times B$ can be formed only for types $A$ and $B$ that inhabit a sort $s \in R^\times$.

**4.3 DEFINITION (Extension of (D)PTS with Cartesian Products).**

The extension of a (D)PTS with cartesian products types is specified by a set of rules $R^\times \subseteq S$. The set of pseudoterms is extended as follows

$$ T ::= \ldots | T \times T | (T, T) | T \uparrow 1 | T \uparrow 2 . $$

The following reduction rule is added

$$ (a_1, a_2) : s \succ_\beta a_1 , $$

and the following type inference rules

$$(\times \text{form}) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A \times B : s} \quad \text{if } s \in R^\times$$

$$(\times \text{intro}) \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \times B : s}{\Gamma \vdash (a, b) : A \times B}$$

$$(\times \text{elim1}) \quad \frac{\Gamma \vdash c : A \times B}{\Gamma \vdash c.1 : A}$$

$$(\times \text{elim2}) \quad \frac{\Gamma \vdash c : A \times B}{\Gamma \vdash c.2 : B}$$

The $\eta$-rule for products is $(c.1, c.2) : s \succ_\eta c$. As is well known, the combination of $\beta$-reduction with this rule—known as surjective pairing—is not Church-Rosser for the untyped lambda calculus, and so it is also not Church-Rosser on the set of pseudoterms.
4.1.2 Disjoint Sums

Before we give the definition of (D)PTSs with disjoint sum types, we first discuss some of the possible alternatives for the syntax.

Disjoint sum types are of the form $A + B$. The formation of disjoint sums is controlled by a set of rules $R^+ \subseteq S$, in the same way the formation of product types is controlled by $R^\times$. The formation rule for $+\text{-}\text{types}$ is

\[
\frac{\Gamma \vdash A : a \quad \Gamma \vdash B : b}{\Gamma \vdash A + B : s} \quad \text{if } s \in R^+
\]

The disjoint sum or discriminated union of two type $A$ and $B$, $A + B$, is the union of two disjoint copies of $A$ and $B$. Informally, its inhabitants can be seen as pairs $(1, a)$ and pairs $(2, b)$ with $a$ of type $A$ and $b$ of type $B$. For sum types we have two introduction rules:

\[
\frac{I \vdash A + B : s \quad I \vdash a : A}{I \vdash \text{in}_{A+B} 1 a : A + B} \quad \frac{I \vdash A + B : s \quad I \vdash b : B}{I \vdash \text{in}_{A+B} 2 b : A + B}
\]

The subscript $A + B$ of $\text{in}$ is awkward but necessary for type inference. We will sometimes omit these subscripts if it is clear from the context what they should be. For instance, if we know that $f : A + B \rightarrow C$, then the subscript $A + B$ could be omitted in the term $f(\text{in}_{A+B} 1 a)$.

The parameter $a$ in $\text{in}_{A+B} 1 a$ could be treated as the argument in an ordinary application. Then the injection $\text{in}_{A+B} 1$ would be typable without its argument of type $A$, namely as

\[
\Gamma \vdash \text{in}_{A+B} 1 : A \rightarrow A + B
\]

However, this is only possible if the function type $A \rightarrow A + B$ can be formed, which requires $(s, s) \in R^\times$. One could go one step further, and treat $\text{in}_{A+B} 1$ as the selection of the first component of a pair. Then the injection into the sum type $A + B$ would be typed as

\[
\Gamma \vdash \text{in}_{A+B} : (A \rightarrow A + B) \times (B \rightarrow A + B)
\]

But this is only possible if the product type $(A \rightarrow A + B) \times (B \rightarrow A + B)$ can be formed, which not only requires $(s, s) \in R^\times$, but also $s \in R^\times$. In the systems we consider it will be possible to type the injection $\text{in}_{A+B}$ without its two arguments.

The elimination principle for a disjoint sum type involves a case distinction between terms constructed using the left injection and those constructed using the right injection. One possibility is to use a case-expression, with the inference rule

\[
(\begin{cases} \Gamma, x : A \vdash c_1 : C \\ \Gamma, y : B \vdash c_2 : C \end{cases} \quad I \vdash A + B \quad I \vdash d : \text{in} 1 x \Rightarrow c_1 \mid \text{in} 2 y \Rightarrow c_2 \quad \text{esac}) \Rightarrow C)
\]

and reduction rules

\[
(\begin{cases} (\text{case } (\text{in}_{A+B} 1 a) \text{ of in } 1 x \Rightarrow c_1 \mid \text{in} 2 y \Rightarrow c_2 \text{ esac}) \Rightarrow b \Rightarrow c_1[x := a] \\
(\text{case } (\text{in}_{A+B} 2 b) \text{ of in } 1 x \Rightarrow c_1 \mid \text{in} 2 y \Rightarrow c_2 \text{ esac}) \Rightarrow b \Rightarrow c_2[y := b] \\
(\text{case } c \text{ of in } 1 x \Rightarrow (\text{in}_{A+B} 1 c) \mid \text{in} 2 y \Rightarrow (\text{in}_{A+B} 2 y) \text{ esac}) \Rightarrow b \Rightarrow c
\end{cases})
\]

and reduction rules

\[
(\begin{cases} (\text{case } (\text{in}_{A+B} 1 a) \text{ of in } 1 x \Rightarrow c_1 \mid \text{in} 2 y \Rightarrow c_2 \text{ esac}) \Rightarrow b \Rightarrow c_1[x := a] \\
(\text{case } (\text{in}_{A+B} 2 b) \text{ of in } 1 x \Rightarrow c_1 \mid \text{in} 2 y \Rightarrow c_2 \text{ esac}) \Rightarrow b \Rightarrow c_2[y := b] \\
(\text{case } c \text{ of in } 1 x \Rightarrow (\text{in}_{A+B} 1 c) \mid \text{in} 2 y \Rightarrow (\text{in}_{A+B} 2 y) \text{ esac}) \Rightarrow b \Rightarrow c
\end{cases})
\]
4.1. EXTENDING PTSs WITH +, ∗ AND Σ

EXTENDING PTSs WITH +, ∗ AND Σ

case is a binding operator: in the term (case c of (in.1 x) → c_1 | (in.2 y) → c_2 esac) it binds
x in c_1 and y in c_2. If we use lambda-abstraction for these bindings, we obtain a less verbose
notation for the elimination principle for sum types:

\[ \Gamma \vdash f : A \rightarrow C \quad \Gamma \vdash g : B \rightarrow C \quad \Gamma \vdash A + B : s \]

\[ \Gamma \vdash f \uplus g : A + B \rightarrow C \]

\( f \uplus g \) is the function that maps a term \((in_{A+B}.1 \ a)\) to \(f \ a\), and a term \((in_{A+B}.1 \ b)\) to \(g \ b\). The
computation rules for \( f \uplus g \) are

\( f \uplus g \ (in_{A+B}.1 \ a) \quad \triangleright_\beta \quad f \ a \)

\( f \uplus g \ (in_{A+B}.2 \ b) \quad \triangleright_\beta \quad g \ b \)

The \( \eta \)-rule for \( f \uplus g \) is easiest to state in terms of \( in_{A+B}.1 \) and \( in_{A+B}.2 \):

\( (in_{A+B}.1 \ \uplus \ in_{A+B}.2 \ c) \quad \triangleright_\eta \quad c \)

The term (case \( d \) of \( in.1 \ x \mapsto c_1 \ | \ in.2 \ y \mapsto c_2 \ esac\)) can easily be expressed in terms of \( \uplus \),
namely as \((\lambda x : A. \ c_1) \uplus (\lambda y : B. \ c_2)\) \( d\). Depending on the situation, the notation using case or
\( \uplus \) may be preferable. Sometimes the case-construct is a little easier to read, for example in
the \( \eta \)-rule. On the other hand, the more compact notation using \( \uplus \) is useful to express some
equalities of functions on sum types, e.g.

\[ h_\circ (f \uplus g) =_\rho_1 \rho_2 \circ (h \circ f) \uplus (h \circ g) \]

where \( f : \rho_1 \mapsto \sigma, g : \rho_2 \mapsto \sigma \) and \( h : \sigma \mapsto \tau \).

There is subtle difference between the elimination rules (i) and (ii) given above, namely
in the restrictions they impose on the possible choices for \( C \). Rule (ii) is more restrictive
than rule (i). In (ii) it follows from the premisses that the types \( A \rightarrow C \) and \( B \rightarrow C \) can be
formed, which means that \( \Gamma \vdash C : s \) for some \((s, s') \in R^2\). In (i) on the other hand nothing
is required of \( C \). We will require that the domain and the range of \( f \uplus g \) have the same sort:

\[ \Gamma, x : A \vdash c_1 \quad \Gamma, y : B \vdash c_2 \quad \Gamma \vdash A + B : s \quad \Gamma \vdash C : s \]

\[ \Gamma \vdash f \uplus g : A + B \rightarrow C \]

There are two ways to generalise this elimination rule, but these generalisations will not be
included. One possibility is to allow the range of \( f \uplus g \) to be of another sort than its domain.
Another is to allow \( f \uplus g \) to have a dependent type. The most liberal elimination rule for
\( + \)-types, combining these two possibilities, is

\[ \Gamma, x : A \vdash c_1 : C[z := in_{A+B}.1 \ x] \]

\[ \Gamma, y : B \vdash c_2 : C[z := in_{A+B}.2 \ y] \]

\[ \Gamma \vdash d : A + B \quad \Gamma, z : A + B \vdash c : s \]

\[ \Gamma \vdash (case \ d \ of \ in.1 \ x \mapsto c_1 \ | \ in.2 \ y \mapsto c_2 \ esac) : C[z := d] \]

Later, in discussion \#38, we will discuss possible uses of these generalisations and our reasons
for excluding them.

Instead of the binary operator \( \uplus \) we will use a unary one that takes a pair as argument.
Then \( f \uplus g \) is in fact \( \uplus(f, g) \), and the elimination rule becomes:

\[ \Gamma \vdash h : (A \rightarrow C) \times (B \rightarrow C) \quad \Gamma \vdash A + B : s \quad \Gamma \vdash C : s \]

\[ \Gamma \vdash \uplus h : A + B \rightarrow C \]
This rule is only equivalent with the rule (iii) if the product \((A \rightarrow C) \times (B \rightarrow C)\) can be formed, which requires \(s \in R^\times\). In the systems we consider this will not be a problem, since we will always have \(R^\times = R^+\). The only reason for using the unary \(\nabla\) is that it is easy to generalise to n-ary sums and labelled sums. It is also more similar to the notation used for \(\Sigma\)-types.

The \(\beta\)-reduction rule becomes

\[
\nabla h \ ((m_{A+B} \; \alpha) \; \beta) \quad h \vDash b
\]

The term \(\nabla (f, g)\) has the expected reduction behaviour, except that now two steps are needed instead one:

\[
\nabla (f, g) \ (m_{A+B} \; 1 \; a) \quad \beta \quad (f, g) \; 1 \; a \quad \beta \quad f \; a
\]

\[
\nabla (f, g) \ (m_{A+B} \; 2 \; b) \quad \beta \quad (f, g) \; 2 \; b \quad \beta \quad g \; b
\]

The \(\eta\)-rule for \(\nabla\) is easiest to state in terms of \(m_{A+B}\),

\[
\nabla m_{A+B} \; c \quad \beta \quad c
\]

4.4 Definition (Extension of (D)PTS with Disjoint Sums)

The extension of a (D)PTS with disjoint sum types is specified by a set of rules \(R^+ \subseteq S\).

The set of pseudoterms is extended as follows

\[
T := \ T + T | m_{T+T} \ 1 \ T | m_{T+T} \ 2 \ T | \nabla T
\]

The following reduction rule is added

\[
\nabla h \ (m_{A+B} \; \alpha) \quad \beta \quad h \vDash a \quad , \text{ for } i \in \{1, 2\}
\]

and the following type inference rules

\[
\begin{array}{l}
\text{(+form)} \quad \Gamma \vdash A : s \quad \Gamma \vdash B : s \\
\quad \Gamma \vdash A + B : s
\end{array}
\]

if \(s \in R^\times\)

\[
\begin{array}{l}
\text{(+intro1)} \quad \Gamma \vdash A + B : s \\
\quad \Gamma \vdash m_{A+B} \; 1 \; a : A + B
\end{array}
\]

\[
\begin{array}{l}
\text{(+intro2)} \quad \Gamma \vdash A + B : s \\
\quad \Gamma \vdash m_{A+B} \; 2 \; b : A + B
\end{array}
\]

\[
\begin{array}{l}
\text{(+elim)} \quad \Gamma \vdash h : (A \rightarrow C) \times (B \rightarrow C) \\
\quad \Gamma \vdash A + B : s \\
\quad \Gamma \vdash C : s \\
\quad \Gamma \vdash \nabla h \ A + B \rightarrow C
\end{array}
\]

4.5 Convention Of the type constructors \(-\), \(+\) and \(\times\), the type constructor \(-\rightarrow\) has the lowest priority and \(\times\) the highest.

4.1.3 Dependent Sums

As for the disjoint sum types, before we give the definition of PTSs extended with dependent sum types, we first discuss some of the alternatives.

Dependent sum types are of the form \((\Sigma r : A \times B)\). The type constructor \(\Sigma\) can be seen as an infinitary version of +, the type \(B_1 + B_2\) is "\(\Sigma r \in \{1, 2\} \ B_r\)". Like inhabitants of \(B_1 + B_2\)
are pairs \((a, b)\) consisting of a index \(i \in \{1, 2\}\) and a term \(b : B_i\), inhabitants of \((\Sigma x : A. B)\) are essentially pairs \((a, b)\) consisting of a term \(a : A\) and a term \(b : B[x := a]\).

There are (at least) two kinds of \(\Sigma\)-types, known as weak and strong \(\Sigma\)-types, with different elimination rules. The \(\Sigma\)-types we use are the weak ones. The difference with the strong \(\Sigma\)-types are explained at the end of this section.

In the programming language \(\Sigma\)-types can be used as abstract datatypes, as explained in [MP84]. Examples of this are given in examples 4.25 and 4.27. As a logical connective, \(\Sigma\) is simply the existential quantifier \(\exists\). The rules for \(\Sigma\)-types look most familiar if \(\Sigma\) is read as \(\exists\); the introduction and elimination rule for \(\exists\) are then the usual introduction and elimination rule for an existential quantification.

The formation of dependent sums is controlled by a set \(R^\Sigma\), with \(R^\Sigma \subseteq S \times S\). The formation rule for \(\Sigma\)-types is:

\[
\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \frac{}{\Gamma \vdash (\Sigma x : A. B) : s_2} \quad \text{if } (s_1, s_2) \in R^\Sigma
\]

There will be no need to generalise this by having rules that are triples \((s_1, s_2, s_3)\) instead of pairs \((s_1, s_2)\) and replacing \(s_2\) in the conclusion by \(s_3\).

The introduction rule for \(\Sigma\)-types is:

\[
\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \quad \frac{}{\Gamma \vdash \text{in}_{\Sigma x : A} B \ a \ b : (\Sigma x : A. B)}
\]

If we read \(\exists\) for \(\Sigma\), then the rule above becomes the usual introduction rule for existential quantification: to prove \((\exists x : A. B)\) we have to provide a witness \(a : A\) with a proof of \(B[x := a]\). The term \((\text{in}_{\Sigma x : A} B \ a \ b)\) can be seen as a pair \((a, b)\). However, the elimination rule for \(\Sigma\)-types will not allow direct access to the two components \(a\) and \(b\) of an inhabitant \((\text{in}_{\Sigma x : A} B \ a \ b)\) of a \(\Sigma\)-type.

As for the \(\pi\)-types, the injection into a \(\Sigma\)-type could be typed without its two arguments,

\[
\text{in}_{\Sigma x : A} B : (\Pi x : A. B \rightarrow (\Sigma x : A. B))
\]

provided that this type can be formed. If \((\Sigma x : A. B)\) has been formed using \((s_1, s) \in R^\Sigma\), then we need \((s_1, s), (s, s) \in R^\Pi\) to form the type \((\Pi x : A. B \rightarrow (\Sigma x : A. B))\).

As for \(\pi\)-types, there are different ways to express the elimination principle for \(\Sigma\)-types. One way, similar to the case-expression for \(\pi\)-types, is a where-expression with the typing rule

\[
\Gamma, x : A, y : B \vdash c : C \quad \Gamma \vdash d : (\Sigma x : A. B) \quad \frac{}{\Gamma \vdash (c \text{ where } (\text{in } x \ y) = d) : C}
\]

and reduction rules

\[
c \text{ where } (\text{in } x \ y) = (\text{in}_{\Sigma x : A} B \ a \ b) \quad \triangleright_B \quad c[x := a][y := b]
\]

\[
c \text{ where } (\text{in } x \ y) = (\text{in}_{\Sigma x : A} B \ x \ y) \quad \triangleright_\eta \quad c
\]

where \(\triangleright\) is a binding operator; in the term \((c \text{ where } (\text{in } x \ y) = d)\) it binds \(x\) and \(y\) in \(c\). Using lambda-abstraction for these bindings produces an operator similar to the \(\nabla\) for \(\pi\)-types, with the following typing rule:

\[
\Gamma \vdash f : (\Pi x : A. B \rightarrow C) \quad \Gamma \vdash (\Sigma x : A. B) \quad s \quad \frac{}{\Gamma \vdash \nabla f \cdot (\Sigma x : A. B) \rightarrow C}
\]
Note that if we read $\exists$ for $\Sigma$, $\forall$ for $\Pi$ and $\Rightarrow$ for $\rightarrow$, this is the normal elimination rule for existential quantification: if $(\forall x : A. B \Rightarrow C)$ then $(\exists x : A. B) \Rightarrow C$. The premise $\Gamma \vdash C : s$ guarantees that $x \notin \text{FV}(C)$.

The $\beta$-reduction rule for $\nabla$ is

$$\nabla f \,(\text{in}_{\Sigma x : A \, B} \ a \, b) \quad \overset{\beta}{\Rightarrow} \quad f \ a \ b$$

The $\eta$-rule is easiest to express in terms of $\text{in}_{\Sigma x : A \, B}$:

$$\nabla \, \text{in}_{\Sigma x : A \, B} \, c \quad \overset{\eta}{\Rightarrow} \quad c .$$

The term $(c \text{ where } (\text{in} \, x \, y) = d)$ can be expressed in terms of the more primitive $\nabla$, namely as $\nabla(\lambda x : A. \, \lambda y : B. \ c) \ d$

As in the elimination rule for $+$-types, we have the possibility of controlling the possible sorts of $C$ in the elimination rules above. As for the $+$-types, we will only allow eliminations where the type of range $C$ is the same sort as the type of domain $(\Sigma x : A \, B)$.

4.6 Definition (Extension of (D)PTS with $\Sigma$-types)

The extension of a (D)PTS with $\Sigma$-types is specified by a set of rules $R^\Sigma \subseteq S \times S$. The set of pseudoterms is extended as follows

$$T := \quad | (\Sigma \text{Var} \, T) \quad | \text{in}_{\Sigma \text{Var} \, T} \quad | T \ T \ | \ \nabla \ T$$

The following reduction rule is added

$$\nabla f \,(\text{in}_{\Sigma x : A \, B} \ a \, b) \quad \overset{\beta}{\Rightarrow} \quad f \ a \ b ,$$

and the following type inference rules

$$(\Sigma \text{form}) \quad \frac{\Gamma \vdash A : s_1 \quad \text{in} \, \tau : A \, B \vdash B \ s_2}{\Gamma \vdash (\Sigma \tau : A. \ B) : s_2} \quad \text{if } (s_1, s_2) \in R^\Sigma$$

$$(\Sigma \text{intro}) \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a] \quad \Gamma \vdash (\Sigma \tau : A. \ B) : s}{\Gamma \vdash \text{in}_{\Sigma x : A \, B} \ a \, b : (\Sigma \tau : A. \ B)}$$

$$(\Sigma \text{elim}) \quad \frac{\Gamma \vdash f : (\Pi \tau : A. \ B \rightarrow C) \quad \Gamma \vdash (\Sigma \tau : A. \ B) : s \quad \Gamma \vdash C : s}{\Gamma \vdash \nabla f : (\Sigma \alpha : A \ B) \rightarrow C}$$

The elimination rule for $\Sigma$-types can be generalised in the same directions as the one for $+$-types. One possibility is to allow the range of $\nabla f$ to be of another sort than its domain. Another is to allow $\nabla f$ to have a dependent type. The most liberal elimination rule for $\Sigma$-types, combining these two possibilities, is

$$\Gamma, x : A, y : B \vdash c \quad \text{where } (\text{in} \, x \, y) = d \quad | \quad \Gamma \vdash (\Sigma \alpha : A \ B) \vdash C : s$$

With this elimination rule the $\Sigma$-types become so-called strong $\Sigma$-types, as used in Martin-Lof's type theories. The $\Sigma$-types in definition 4.6 - with the elimination rule $(\Sigma \text{elim})$ - are known as weak $\Sigma$-types. Inhabitants of a strong $\Sigma$-type $(\Sigma x : A \ B)$ are pairs $(a, b)$ with a $A$
and \( b : B[x := a] \) for which we do have direct access to the two components \( a \) and \( b \). Using the inference rule above it is possible to define functions

\[
\begin{align*}
\pi_1 &: (\Sigma x : A. B) \rightarrow A \\
\pi_2 &: (\Pi z : (\Sigma x : A. B) \ B[x := \pi_1(x)])
\end{align*}
\]

such that

\[
\begin{align*}
\pi_1(\text{in}_{\Sigma x : A} \ B \ a \ b) & \triangleright_{\beta} a \\
\pi_2(\text{in}_{\Sigma x : A} \ B \ a \ b) & \triangleright_{\beta} b
\end{align*}
\]

This means that the strong sums \((\Sigma x : A. B)\) with \( x \notin \text{FV}(B)\), is just a product type \( A \times B \). So strong sums can be seen as a generalisation of cartesian products.

4.7 REMARK. Even with the "weak" elimination rule rule \((\Sigma\text{elim})\) given in definition 4.6, it is possible to define the projections \( \pi_1 \) and \( \pi_2 \) for certain \((\Sigma x : A. B)\). A function \( \pi_1 \) satisfying (i) is definable if \((\Sigma x : A. B)\) is formed using a rule \( (s_1, s_2) \in \Sigma^2 \) with \( s_1 = s_2 \), namely as \( \nabla(\lambda x : A. \lambda y : B. \ x) \). A function \( \pi_2 \) satisfying (ii) is definable using \((\Sigma\text{elim})\) if \( x \notin \text{FV}(B) \), namely as \( \nabla(\lambda x : A. \lambda y : B. \ x) \). So, if the weak sum \((\Sigma x : A. B)\) is formed using a rule \( (s, s) \in \Sigma^2 \) and \( x \notin \text{FV}(B) \), then it is just a cartesian product type \( A \times B \).

4.1.4 Labelled Products

Labelled products are generalisations of the cartesian product types \( A \times B \) introduced earlier. They will be used as record types in the programming language. We have to introduce a collection of labels, and two very simple judgements involving labels:

4.8 DEFINITION.

1. \( \mathcal{L} \) is the set of all labels. We assume \( 0, 1, 2, \ldots \in \mathcal{L} \).

2. For judgements of the form \( \langle l_1, \ldots, l_n \rangle : \square_{\text{label}} \) and \( l : \langle l_1, \ldots, l_n \rangle \), where \( l, l_1, l_2, \ldots \in \mathcal{L} \), we have the following inference rules

\[
\begin{align*}
\frac{l_1, \ldots, l_n \in \mathcal{L} \text{ and all } l_i \text{ distinct}}{\langle l_1, \ldots, l_n \rangle : \square_{\text{label}}} \\
\frac{l \in \{ l_1, \ldots, l_n \}}{\langle l_1, \ldots, l_n \rangle : \square_{\text{label}}}
\end{align*}
\]

Because \( 0, 1, 2, \ldots \in \mathcal{L} \), we will be able to consider unlabelled products as special cases of labelled ones. Beware that the labels \( 0, 1, 2, \ldots \) and the natural numbers used as data in the programming language are not the same and belong to different syntactic categories.

Labelled product types are of the form \( \Pi (l_1 : A_1, \ldots, l_n : A_n) \). Their inhabitants are terms of the form \( (l_1 \mapsto a_1, \ldots, l_n \mapsto a_n) \). These terms are essentially mappings from labels to terms. The formation of labelled products is controlled by the same set of rules as the cartesian products. So the type \( \Pi (l_1 : A_1, \ldots, l_n : A_n) \) can be formed if all the \( A_i \) inhabit the same same
sort \( s \in R^X \). Although that labelled products are written using "\( \Pi \)", we will still refer to them as \( x \)-types, and reserve the name \( \Pi \)-types for the dependent products \( (\Pi \colon A \ B) \).

Clearly the order of the fields "\( l_i : A_i \)" and "\( l_i \mapsto a_i \)" in terms \( \Pi(l_1 : A_1, \ldots, l_n : A_n) \) and \( \langle l_1 \mapsto a_1, \ldots, l_n \mapsto a_n \rangle \) is irrelevant. Terms that are equal up to permutations of fields are identified, in the same way as terms that are equal up to renaming of bound variables are identified.

We will also allow the empty product type \( \Pi() \). This type is a type with only one inhabitant, namely the empty mapping (\( () \)) from labels to terms. This type requires some special attention. For every \( s \in R^X \) there is an empty product \( \Pi() \). So if there is more than one \( s \) in \( R^X \), these types \( \Pi() \) and their inhabitants (\( () \)) have to get a subscript \( s \) to distinguish them. Otherwise we lose the property that the type of a term is unique up to conversion. As far as the syntax is concerned, we do not gain anything by using the empty product type \( \Pi() \) as the unit type. Special type inference rules have to be given for the empty product because its introduction and formation rule do not have any premises. But as far as the semantics is concerned, the empty product can be treated as any other product.

4.9 Definition (Extension of (D)PTS with \( x \)-types)

The extension of a (D)PTS with labelled product types - or simply \( x \)-types for short - is specified by a set of rules \( R^X \subseteq S \):

The following reduction rule is added:

\[
(l_1 \mapsto a_1, \ldots, l_n \mapsto a_n) : A \quad \eta \implies a
\]

and the following type inference rules:

\[
\frac{}{(x\text{form})} \quad \frac{\Gamma \vdash l_i : A_i \quad s \text{ for } i = 0 \ldots n}{\Gamma \vdash \Pi(l_1 : A_1, \ldots, l_n : A_n) \quad s} \quad \text{if } s \in R^X \text{ and } n \neq 0
\]

\[
\frac{}{(x\text{intro})} \quad \frac{\Gamma \vdash A_i \quad \Gamma \vdash \Pi(l_1 : A_1, \ldots, l_n : A_n) \quad s}{\Gamma \vdash \langle l_1 \mapsto a_1, \ldots, l_n \mapsto a_n \rangle : \Pi(l_1 : A_1, \ldots, l_n : A_n)}
\]

\[
\frac{}{(x\text{elim})} \quad \frac{\Gamma \vdash \Pi(l_1 : A_1, \ldots, l_n : A_n) \quad \Gamma \vdash b : l_i : A_i}{\Gamma \vdash b.l_i : A_i}
\]

\[
\frac{\epsilon \vdash (\Pi()_s : s)}{(\Pi()\text{form})} \quad \text{if } s \in R^X
\]

\[
\frac{\epsilon \vdash (\Pi()_s : s)}{(\Pi()\text{intro})} \quad \text{if } s \in R^X
\]

The rules \( (x\text{form}) \), \( (x\text{intro}) \) and \( (x\text{elim}) \) are similar to the original PTS rules for formation, introduction and elimination of dependent products \( (\Pi \colon A \ B) \). Having \( s \in R^X \) can be seen as having \( (\text{label} : s) \in R^S \).

The \( \eta \)-reduction rule for labelled products is:

\[
(l_1 \mapsto a.l_1, \ldots, l_n \mapsto a.l_n) \eta \implies a \quad \text{if } a \Pi(l_1 : A_1, \ldots, l_n : A_n)
\]
Note that this rule depends on the type of \( a \). This is another reason to exclude \( \eta \)-reduction. For all the type systems in this thesis, the typing relation depends on the reduction relation (namely in the conversion rule \((\beta\text{conv})\)), but the reduction relation does not depend on the typing relation. Including the \( \eta \)-rule above would mean that the reduction relation in turn depends on the typing relation, so that the reduction relation can no longer be defined before the typing relation is defined.

For the empty product, the direction of the \( \eta \)-rule above does not make sense. It has to be reversed:

\[
a \triangleright \_ \quad \text{if } a : \Pi() \,.
\]

In fact, for the \( \eta \)-rules there is not a natural direction as there is for all the \( \beta \)-rules.

Because we assume all the natural numbers are labels, unlabelled products can be treated as a special case of labelled products.

4.10 Notation (Unlabelled \( n \)-ary products)

For terms \( A_1, \ldots, A_n \) and \( a_1, \ldots, a_n \) we write \( A_1 \times \cdots \times A_n \) for the \( n \)-ary product type \( \Pi(1; A_1, \ldots, n; A_n) \), and \( (a_1, \ldots, a_n) \) for the \( n \)-tuple \( (1 \mapsto a_1, \ldots, n \mapsto a_n) \).

Note that the reduction and inference rules for \( \Pi(1; A, 2; B) \) are indeed exactly the ones given for \( A \times B \) earlier. So the extension with \( \times \)-type given in definition 4.9 subsumes the one given in definition 4.3.

4.1.5 Labelled Sums

Labelled sum types, generalisations of the disjoint sum types \( A + B \) discussed earlier, are of the form \( \Sigma(l_1; A_1, \ldots, l_n; A_n) \). The inhabitants of \( \Sigma(l_1; A_1, \ldots, l_n; A_n) \) are essentially pairs \((l_i, a_i)\) with \( a_i \) of type \( A_i \). The formation of labelled sums is controlled by the same set of rules \( \mathcal{R}^+ \) as the disjoint sum types

\[
\frac{(l_1, \ldots, l_n) : \square_{\text{label}} \quad \Gamma \vdash A_1 : s \quad \ldots \quad \Gamma \vdash A_n : s}{\Gamma \vdash \Sigma(l_1; A_1, \ldots, l_n; A_n) : s} \quad \text{if } s \in \mathcal{R}^+ .
\]

The introduction rule for labelled sum types is

\[
\frac{}{\Gamma \vdash a_i : A_i \quad l_i : (l_1, \ldots, l_n) \quad \Gamma \vdash \Sigma(l_1; A_1, \ldots, l_n; A_n) : s} \quad \text{if } l_i \in \Sigma(l_1; A_1, \ldots, l_n; A_n).
\]

As for the disjoint sum, the subscript of \( \mathtt{in}_{\Sigma(l_1; A_1, \ldots, l_n; A_n)} l_i \) is necessary for type inference. A possible way of avoiding these subscripts would be to allow some form of subtyping. The type \( \Sigma(l_i; A_i) \) would then be a subtype of \( \Sigma(l_1; A_1, \ldots, l_n; A_n) \), so for \( a_i : A_i \), the term \((l_i, a_i)\) would have type \( \Sigma(l_i; A_i) \) and also the supertype \( \Sigma(l_1; A_1, \ldots, l_n; A_n) \). Subtyping will not be treated in this thesis.

The injection \( \mathtt{in}_{\Sigma(l_1; A_1, \ldots, l_n; A_n)} l_i, a \) could be typed without its argument \( a \), namely as

\[
\mathtt{in}_{\Sigma(l_1; A_1, \ldots, l_n; A_n)} l_i : A_i \rightarrow \Sigma(l_i; A_1, \ldots, l_n; A_n)
\]

or even without the projection \( l_i \), namely as

\[
\mathtt{in}_{\Sigma(l_1; A_1, \ldots, l_n; A_n)} : \Pi(l_1; A_1 \rightarrow \Sigma(l_1 A_1, \ldots, l_n; A_n), \ldots, l_n; A_n \rightarrow \Sigma(l_i; A_1, \ldots, l_n; A_n)).
\]
Of course this is only possible if these types can be formed.

As for the unlabelled binary sums, different ways to write the elimination principle exist. One possibility is a case-expression of the form \((\text{case } d \text{ of } l_1 x_1 \mapsto c_1 | \ldots | l_n x_n \mapsto c_n)\) with the type inference rule

\[
\Gamma \vdash d : \Sigma(l_1: A_1, \ldots, l_n: A_n) \quad \Gamma, x_i: A_i \vdash c_i : C \text{ for } i = 0 \ldots n
\]

\[
\Gamma \vdash (\text{case } d \text{ of } l_1 x_1 \mapsto c_1 | \ldots | l_n x_n \mapsto c_n) : C
\]

and the reduction rules

\[
(\text{case } \text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a) \text{ of } \text{in } l_1 x_1 \mapsto c_1 | \ldots | l_n x_n \mapsto c_n) \mapsto_B c_i [x_i := a_i]
\]

\[
(\text{case } c \text{ of } \text{in } l_1 x_1 \mapsto \text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a) \text{ of } \text{in } l_1 x_1 \mapsto \text{in} l_n x_n \mapsto \text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n)) \mapsto_B c
\]

We will use a generalisation of \(\nabla\), with the inference rule

\[
\Gamma \vdash f \cdot \Pi(l_1: A_1, \ldots, l_n: A_n \rightarrow C) \quad \Gamma \vdash \Sigma(l_1: A_1, \ldots, l_n: A_n) : s \quad \Gamma \vdash C : s
\]

\[
\Gamma \vdash \nabla f : \Sigma(l_1: A_1, \ldots, l_n: A_n) \rightarrow C
\]

So, as for the unlabelled binary disjoint sums, only eliminations where the type of the range \(C\) is the same as the type of domain \(\Sigma(l_1: A_1, \ldots, l_n: A_n)\) are allowed.

The \(\beta\)-reduction rules for \(\nabla\) is

\[
\nabla h \ (\text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a) \mapsto_B h \ l, a
\]

The \(\eta\)-rule is easiest to express in terms of \(\text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a)\)

\[
\nabla \text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a) : C \mapsto_\eta C
\]

To compare the inference rule for \(\nabla\) with the one for case, note that if \(\Gamma, x_i: A_i \vdash c_i : C\), then

\[
\Gamma \vdash (\lambda x_i: A_i, c_i) \ A_i \rightarrow C
\]

and then

\[
\Gamma \vdash (l_1 \mapsto (\lambda x_1: A_1, c_1), \ldots, l_n \mapsto (\lambda x_n: A_n, c_n)) : \Pi(l_1: A_1 \rightarrow C, \ldots, l_n: A_n \rightarrow C)
\]

The term \((\text{case } d \text{ of } \text{in } l_1 x_1 \mapsto c_1 | \ldots | l_n x_n \mapsto c_n)\) can be expressed in terms of \(\nabla\), namely as \(\nabla (l_1 \mapsto (\lambda x_1: A_1, c_1), \ldots, l_n \mapsto (\lambda x_n: A_n, c_n))\) \(a\). This term has the correct reduction behaviour:

\[
\nabla (l_1 \mapsto f_1, \ldots, l_n \mapsto f_n) \ (\text{in}(\Sigma(l_1: A_1, \ldots, l_n: A_n) \downarrow l, a) \mapsto_B (l_1 \mapsto f_1, \ldots, l_n \mapsto f_n) \ l, a
\]

Like the empty product type \(\Pi()\), the empty sum type \(\Sigma()\) requires some extra attention. The empty sum type \(\Sigma()\) is an empty type, for which there is no introduction rule to construct inhabitants. For every \(s \in \text{R}^t\) there is an empty sum \(\Sigma()\). Like \(\Pi()\), these get a subscript \(s\) to distinguish them. To preserve unicity of types more has to be done. Instantiating the elimination rule above for the empty sum produces

\[
\Gamma \vdash f \ \Pi() \quad \Gamma \vdash \Sigma() : s \quad \Gamma \vdash C : s
\]

\[
\Gamma \vdash \nabla f : \Sigma() \rightarrow C
\]

This means that for any \(f \ \Pi()\), the term \(\nabla f\) can have many types, because its range can be any type \(C : s\). To preserve uniqueness of types, the range of \(\nabla f\) is written as a subscript. So the elimination rule for empty sums becomes

\[
\Gamma \vdash f \ \Pi(s) \quad \Gamma \vdash \Sigma(s) : s \quad \Gamma \vdash C : s
\]

\[
\Gamma \vdash \nabla f : \Sigma(s) \rightarrow C
\]
4.1 DEFINITION (Extension of \((D)PTS\) with ++-types).
The extension of a \((D)PTS\) with labelled sum types – or simply ++-types for short – is specified by a set of rules \(R^+ \subseteq S\).
The set of pseudoterms is extended as follows:

\[
\mathcal{T} := \ldots | \Sigma(\mathcal{L}_1 : \ldots, \mathcal{L}_n) | \text{in}_{\Sigma(\mathcal{L}_1 : \ldots, \mathcal{L}_n)} \top | \top | \top | \top | \Sigma()_s .
\]

The following reduction rule is added:

\[
\nabla h \ (\text{in}_{\Sigma(l_1 A_1, \ldots, l_n A_n)} l_i a) \Downarrow \beta \ h \ l_i a ,
\]

and the following type inference rules:

\[
\begin{align*}
(+) \text{form} \quad & \frac{\Gamma \vdash A_i : s \text{ for } i = 0 \ldots n}{\Gamma \vdash \Sigma(l_1 A_1, \ldots, l_n A_n) : s} & \text{if } s \in R^+ \text{ and } n \neq 0 \\
(+) \text{intro} \quad & \frac{\Gamma \vdash a_i : A_i \quad \Gamma \vdash \Sigma(l_1 A_1, \ldots, l_n A_n) : s}{\Gamma \vdash \text{in}_{\Sigma(l_1 A_1, \ldots, l_n A_n)} l_i a_i : \Sigma(l_1 A_1, \ldots, l_n A_n)} \\
(+) \text{elim} \quad & \frac{\Gamma \vdash \nabla f : \Sigma(l_1 A_1, \ldots, l_n A_n) \rightarrow C}{\Gamma \vdash C : s} \quad \text{if } s \in R^+ \\
(\Sigma()) \text{form} \quad & \frac{\epsilon \vdash \Sigma()_s : s}{\Gamma \vdash \nabla \epsilon : \Sigma()_s \rightarrow C} \\
(\Sigma()) \text{elim} \quad & \frac{\Gamma \vdash \Sigma()_s \quad \Gamma \vdash C : s}{\Gamma \vdash \nabla \epsilon : \Sigma()_s \rightarrow C}
\end{align*}
\]

For labelled sums the notation using case seems more natural than the one using \(\nabla\). The notation using \(\nabla\) – or rather the infix \(\nabla\) – is useful for unlabelled sums:

4.12 NOTATION (Unlabelled n-ary sums)

For terms \(A_1, \ldots, A_n, f_1, \ldots, f_n\) we write \(A_1 + \ldots + A_n\) for the n-ary sum \(\Sigma(1 A_1, \ldots, n A_n)\), and \(f_1 \nabla \ldots \nabla f_n\) for \(\nabla (1 f_1, \ldots, n f_n)\).

Note that the reduction and inference rules for \(\Sigma(1 A, 2 B)\) are indeed exactly the ones given for \(A + B\) earlier. So the extension with ++-types given in definition 4.11 subsumes the one given in definition 4.4.

4.1.6 Properties of \((D)PTSs\) extended with \(+, \times\) and \(\Sigma\)

Extending a \(PTS\) or DPTS with \(+, \times\) and \(\Sigma\) preserves all the properties of arbitrary PTSs or DPTSs listed in chapter 2. For the reduction relation the property of Church-Rosser is preserved:

4.13 LEMMA. \(\beta\)-reduction is Church-Rosser.

\textbf{Proof.} This can be proved by the usual Tait-Martin-Lof method. Another way is to use the theory of Combinatory Reduction Systems (CRSs) and verify that \(\Downarrow \beta\) is an orthogonal CRS (see [KOR93]).
4.14 **Lemma** \( \beta \)-reduction is Church-Rosser.

**Proof.** This can be proved in exactly the same way as in lemma 2.26, i.e. using the fact that \( \varphi_\beta \) is \( \text{CR} \) and the fact that \( \delta \)-normal forms exist.

All other properties of PTSs and DPTSs in chapter 2 are proved by induction on type derivations or by induction on the structure of terms. Extending these induction proofs for the new inference rules and new term-constructions is completely straightforward. We just list the main properties

4.15 **Lemma** (Correctness of Types)  
For all (D)PTSs with +, \( \times \) and \( \Sigma \): if \( \Gamma \vdash a : A \) then \( \Gamma \vdash A \) \( s \) for some \( s \in S \).  

4.16 **Lemma** (SR\( _\beta \): Subject Reduction).  
For all PTSs with +, \( \times \) and \( \Sigma \): if \( \Gamma \vdash b : B \) and \( b \varphi_\beta b' \) then \( \Gamma \vdash b' : B \).

4.17 **Lemma** (UT\( _\beta \): Uniqueness of Types).  
For all functional PTSs with +, \( \times \) and \( \Sigma \): if \( \Gamma \vdash b : B \) and \( \Gamma \vdash b : B' \) then \( B \sim_\beta B' \).

4.18 **Lemma** (SR\( _{\beta, \delta} \): Subject Reduction).  
For all DPTSs with +, \( \times \) and \( \Sigma \): if \( \Gamma \vdash b : B \) and \( \Gamma \vdash b : B' \) then \( \Gamma \vdash b : B' \).

4.19 **Lemma** (UT\( _{\beta, \delta} \): Uniqueness of Types).  
For all functional DPTSs with +, \( \times \) and \( \Sigma \): if \( \Gamma \vdash b : B \) and \( \Gamma \vdash b : B' \) then \( B \sim_{\beta, \delta} B' \).

4.20 **Theorem** (Elimination of definitions)  
For all specifications \( S \): if \( \Gamma \vdash S a : A \) then \( \text{hnf}(\Gamma) \vdash_{\lambda S} \text{hnf}_S (a) : \text{hnf}_S(A) \).
4.2 Extension of the Programming Language

The programming language \( \lambda \omega \) is extended to provide with more datatypes as primitives.

4.21 Definition. \( \lambda \omega^+ \) (is) the (D)PTS with \( +, \times \) and \( \Sigma \) specified by

\[
\begin{align*}
S &= \{ \star, \Box \} \\
A &= \{ \star, \star, \star \} \\
R^\Pi &= \{ (\Box_1, \Box_2, (\Box_2, \star_3), (\Box_2, \star_4)) \} \\
R^\Sigma &= \{ (\star_3, \star_4) \} \\
R^\times &= R^+ \equiv \{ \star_3 \}
\end{align*}
\]

The new datatypes that can be formed using \( R^\times = \{ \star_3 \} \) are unlabelled product types of the form \( \sigma_1 \times \cdots \times \sigma_n \) inhabited by programs of the form \( (M_1, \ldots, M_n) \), and labelled product types - or record types - of the form \( \Pi(l_1 : \sigma_1, \ldots, l_n : \sigma_n) \), inhabited by programs of the form \( (l_1 \mapsto M_1, \ldots, l_n \mapsto M_n) \). The empty product \( \Pi() \) provides a datatype with just one inhabitant, for which we introduce a more suggestive name.

4.22 Definition (Unit Type). The datatype Unit and its inhabitant unit are defined by the following context \( \Gamma_{UNIT} \),

\[
\begin{align*}
\text{Unit} &= \Pi() : \star_3 \\
\text{unit} &= \langle \rangle_3 . \text{Unit}
\end{align*}
\]

The new datatypes that can be formed using \( R^+ = \{ \star_3 \} \) are unlabelled sum types of the form \( \sigma_1 + \cdots + \sigma_n \) and labelled sum types also known as variant types - of the form \( \Sigma(l_1 : \sigma_1, \ldots, l_n : \sigma_n) \). The empty sum gives us an empty datatype.

4.23 Definition (Empty Type). The datatype Empty is defined by the following context \( \Gamma_{EMPTY} \)

\[
\text{Empty} = \Sigma() : \star_3
\]

Using the empty product type Unit and +-types, a two-element type of booleans can be constructed:

4.24 Definition (Booleans) The datatype bool and its elements true and false are defined by the following context \( \Gamma_{BOOL} \)

\[
\begin{align*}
\text{bool} &= \text{Unit} + \text{Unit} : \star_3 \\
\text{true} &= \text{in}_{\text{Unit} + \text{Unit}} 1 \text{unit} : \text{bool} \\
\text{false} &= \text{in}_{\text{Unit} + \text{Unit}} 2 \text{unit} : \text{bool} \\
\text{if} &= \lambda \alpha. \star_3 . \lambda b: \text{bool} . \lambda x, y : \alpha . \nabla (\lambda u: \text{Unit} . x, \lambda u: \text{Unit} . y) \ b \\
&\quad \Pi \alpha. \star_3 . \text{bool} \rightarrow \alpha ightarrow \alpha \rightarrow \alpha
\end{align*}
\]

For programs \( b : \text{bool}, M_1 : \tau \) and \( M_2 : \tau \) we write \( (\text{if } b \text{ then } M_1 \text{ else } M_2) \) for \( (\if \tau M_1 M_2 b) \). Note that the type parameter \( \tau \) is omitted. It is not difficult to see that this missing parameter can be reconstructed (it is the type of \( M_1 \) and \( M_2 \)).
The term \((\text{if } b \text{ then } M_1 \text{ else } M_2)\) reduces as expected

\[
\Gamma_{\text{UNIT}}, \Gamma_{\text{BOOL}} \vdash \text{if true then } a_1 \text{ else } a_2 \qquad \Gamma_{\text{UNIT}}, \Gamma_{\text{BOOL}} \vdash \text{if false then } a_1 \text{ else } a_2
\]

We will always assume that \(\Gamma_{\text{BOOL}}, \Gamma_{\text{UNIT}}\) and \(\Gamma_{\text{EMPTY}}\) are part of the context.

The new datatypes that can be formed using \(R^\Sigma = \{(\alpha, \ast_\alpha)\}\) are of the form \((\Sigma \alpha \mathcal{K} \cdot \sigma)\). These datatypes are the types of programs of the form \((\text{inADT } \sigma \cdot N)\). Such programs can be understood as a pairs \((\tau, N)\) consisting of a datatype-constructor \(\tau\) and a program \(N\) of type \(\sigma[\alpha := \tau]\). \(\Sigma\)-types can be used as abstract datatypes. For a thorough explanation of this we refer to [MP84]. Two examples are given below.

4.25 Example. We consider a simple abstract datatype, for lists of booleans. An abstract datatype consists of a datatype and a collection of operations on it. These operations provide the only way to manipulate inhabitants of that datatype. For an abstract datatype of boolean lists we want a datatype \texttt{boollist} with the following operations:

\[
\begin{align*}
\text{empty} & : \texttt{boollist}, \\
\text{cons} & : \text{bool} \rightarrow \texttt{boollist} \rightarrow \texttt{boollist}, \\
\text{head} & : \texttt{boollist} \rightarrow \text{bool}, \\
\text{tail} & : \texttt{boollist} \rightarrow \texttt{boollist}
\end{align*}
\]

These 4 operations can be provided by a single program of the following record type

\[
\Pi\{ \text{empty} : \texttt{boollist}, \quad \text{cons} : \text{bool} \rightarrow \texttt{boollist} \rightarrow \texttt{boollist}, \\
\text{head} : \texttt{boollist} \rightarrow \text{bool}, \quad \text{tail} : \texttt{boollist} \rightarrow \texttt{boollist} \}
\]

This record type gives the signature of an abstract datatype of boolean lists. For an implementation of this abstract datatype we have to choose a particular datatype \texttt{boollist} and a program of the type above implementing the operations on boolean lists. In a program \(M\) that uses this abstract datatype we simply assume the existence of a datatype \texttt{boollist} and a program of the type above. For this \(\Sigma\)-types can be used. The datatype-constructor \(F\) as follows

\[
F = \lambda \texttt{boollist}. \ast_\ast \Pi\{ \text{empty} : \texttt{boollist}, \quad \text{cons} : \text{bool} \rightarrow \texttt{boollist} \rightarrow \texttt{boollist}, \\
\text{head} : \texttt{boollist} \rightarrow \text{bool}, \quad \text{tail} : \texttt{boollist} \rightarrow \texttt{boollist} \}
\]

This \(F\) gives the signature of our abstract datatype \(\text{ADT}\). The type

\[
\text{ADT} = (\Sigma \text{boollist } \ast_\ast, \ F \text{boollist}) \quad \ast_\ast
\]

can now serve as the type of implementations of the abstract datatype of boolean lists. For an implementation of this abstract datatype we have to choose a datatype \(\tau\) and a program of type \(F\tau\). The inhabitants of \(\tau\) then represent lists of booleans, and the program of type \(F\tau\) provides the implementations of the operations on these (representations of) lists. The introduction rule for the datatype \(\text{ADT}\) is

\[
\frac{
\Gamma \vdash \tau : \ast_\ast \quad \Gamma \vdash N : F\tau
}{\Gamma \vdash \text{inADT } \tau N : \text{ADT}}
\]
So, an inhabitant of $ADT$ – i.e., an implementation of the abstract datatype of boolean lists – can be built using a datatype $\tau$ and a program $N$ of type $F\tau$. Of course, there is no guarantee that this is indeed a correct implementation of lists, e.g., it may well be the case that $(N.\text{cons} \ b \ l) \not\in_{g_{\delta}} l$. Later, in examples 4.43 and 4.45, specifications for $ADT$ will be given.

To make a program using the abstract datatype of boolean lists, we simply assume the existence of a type $boolist : *_{\tau}$ and a program $x$ of type $(F\text{boolist})$ providing the operations on it. For example, suppose $\rho$ is a datatype and

$$\Gamma, \text{boolist} : *_{\tau}, x : F\text{boolist} \vdash M : \rho$$

Here the program $M$ can make use of the operations $x.\text{empty}, x.\text{cons}, x.\text{head}$ and $x.\text{tail}$ on the type $boolist$. Then by the elimination rule for the type $ADT$, the rule ($\Sigma$elim),

$$\Gamma \vdash \forall(\lambda\text{boolist} : *_{\tau}, \lambda x : F\text{boolist} \ M) : ADT \rightarrow \rho$$

The program $\forall(\lambda\text{boolist} : *_{\tau}, \lambda x : F\text{boolist} \ M)$ takes an implementation of the abstract datatype of boolean lists as an argument, for example the one defined above:

$$\forall(\lambda\text{boolist} : *_{\tau}, \lambda x : F\text{boolist} \ M) (\text{in}_{ADT} \tau \ N) : \rho$$

This program $\beta$-reduces to $M[\text{boolist} := \tau][x := N]$. The crucial point is that in the "main" program $M$ the variables $boolist$ and $x$ are visible, but not the bindings to $\tau$ and $N$. In other words, the interface of the abstract datatype is visible in $M$, but the implementation is not.

When $\Sigma$-types are used in this way, then implementations of abstract datatypes become first-class citizens in the programming language, i.e., they can be manipulated just like any other data or program. For instance, they can be passed as parameters, and the choice between different implementations can then be made at during the execution of a program.

\[4.26\text{ Notation.}\] We write $(\text{abstype } \alpha \ K \text{ with } x.\sigma \text{ is } \text{imp } \text{in } M)$ for $\forall(\lambda x : K. \lambda x.\sigma. \ M) \ \text{imp}$.

So the program given at the end of the example above is written as

$$(\text{abstype } \text{boolist } *_{\tau}, \text{with } \tau : F \text{boolist is } \text{(in}_{ADT} \tau \ N) \text{ in } M)$$

With this notation the interface of the abstract datatype and the implementation are no longer separated by the main program $M$.

$\lambda\omega^+_{\tau}$ does not only provide the abstract datatypes, but also abstract datatype-constructors

\[4.27\text{ Example.}\] An abstract datatype-constructor consists of a datatype-constructor and a collection of operations on the datatypes that can be built with it. As an example we consider an abstract datatype-constructor for building lists over any datatype. It comprises a datatype-constructor $\text{list} : *_{\tau} \rightarrow *_{\tau}$, and operations for manipulating inhabitants of the datatypes $(\text{list } \sigma)$, for all datatypes $\sigma$. i.e.

$\text{empty : list } \sigma, \text{cons } \sigma \rightarrow \text{list } \sigma \rightarrow \text{list } \sigma, \text{head : list } \sigma \rightarrow \sigma, \text{tail : list } \sigma \rightarrow \text{list } \sigma$. 

These operations can be provided by a single program of type

\[ \Pi \{ \text{empty} : \Pi \alpha \times \star, \; \text{cons} : \Pi \alpha \times \star, \; \text{head} : \Pi \alpha \times \star, \; \text{tail} : \Pi \alpha \times \star \} , \]

or, alternatively, by a program of type

\[ \Pi \alpha \times \star, \; \Pi \{ \text{empty} : \alpha \times \star, \; \text{cons} : \alpha \times \star, \; \text{head} : \alpha \times \star, \; \text{tail} : \alpha \times \star \} . \]

We choose the second option, and define a datatype-constructor \( F \) as follows

\[ F = \lambda \alpha. \Pi \alpha \times \star, \; \Pi \{ \text{empty} : \alpha \times \star, \; \text{cons} : \alpha \times \star, \; \text{head} : \alpha \times \star, \; \text{tail} : \alpha \times \star \} : (\alpha \times \star) \to \alpha \times \star . \]

The datatype

\[ ADT = (\Sigma \Pi \alpha \times \star, \; F \alpha \times \star) : \star \]

is then the type of implementations of an abstract datatype-constructor for making lists. Its inhabitants are of the form \((m_{ADT} r N)\), with \( r \times \star \to \star \) and \( N : F r \). If

\[ \Gamma \vdash \forall (\lambda \alpha \times \star, \; \lambda \alpha \times \star. \Pi \alpha \times \star) \cdot ADT \to \rho , \]

then in the program \( M \) the operations

\[ \lambda \sigma \cdot \text{empty} : \alpha \times \star \]
\[ \lambda \sigma \cdot \text{cons} : \alpha \times \star, \; \alpha \times \star \to \alpha \times \star \]
\[ \lambda \sigma \cdot \text{head} : \alpha \times \star \to \alpha \]
\[ \lambda \sigma \cdot \text{tail} : \alpha \times \star \to \alpha \times \star \]

can be used, for all datatypes \( \sigma \).

\[ \square \]

Properties of \( \lambda \omega_s^+ \) and \( \lambda \omega_s^{+ \downarrow} \)

As for the system without \( \times \), \( \times \) and \( \Sigma \), the context-free syntax of the pseudoterms can be refined, distinguishing the different levels:

4.28 Definition. For \( \lambda \omega_s^+ \), the sets of pseudoterms Cons and Prog defined in definition 3.2 are extended as follows:

\[ \text{Cons} = \mid \text{Cons} \times \cdot \times \text{Cons} \mid \Pi (L \; \text{Cons} \to \cdots, \; L \; \text{Cons}) \mid \Pi (\cdot) \star, \]
\[ \mid \text{Cons} \times, \times \text{Cons} \mid \Sigma (L \; \text{Cons} \to \cdots, \; L \; \text{Cons}) \mid \Sigma (\cdot) \star, \]
\[ \mid (\Sigma \text{Var}^O) \cdot \text{Kind} \; \text{Cons} \]

\[ \text{Prog} ::= \ldots \mid (\text{Prog}, \; \cdot \text{Prog}) \mid (L \to \text{Prog} \to \cdots, \; L \to \text{Prog}) \mid (\cdot) \star, \; \text{Prog} \; L \]
\[ \mid \text{in}_\text{Cons} \; L \; \text{Cons} \mid \forall \text{Prog} \mid \forall \text{Prog}_\text{Cons} \]
\[ \mid \text{in}_\text{Cons} \; \text{Cons} \; \text{Prog} \]

\[ \square \]
4.2. EXTENSION OF THE PROGRAMMING LANGUAGE

The set Kind containing the kinds is not affected by the new type-constructors. The dependencies between Kind, Cons and Prog do not change: Cons still only depends on Kind, and Prog depends on Cons and Kind. As for \( \lambda \omega \), the kinds, datatype-constructors and programs are elements of Kind, Cons and Prog, respectively:

4.29 Lemma (Classification for \( \lambda \omega^+ \))

1. If \( \Gamma \vdash K : \square_e \) then \( K \in \text{Kind} \).
2. If \( \Gamma \vdash \sigma : K : \square_e \) then \( \sigma \in \text{Cons} \).
3. If \( \Gamma \vdash M : \sigma : \ast_e \) then \( M \in \text{Prog} \).

Proof. In exactly the same way as lemma 3.5. \( \square \)

The usual encodings of +, \( \times \) and \( \Sigma \) in \( \lambda \omega \) can be used to show that the extension of \( \lambda \omega \) and \( \lambda \omega^+ \) with +, \( \times \) and \( \Sigma \) preserves the property of strong normalisation:

4.30 Corollary. The system \( \lambda \omega^+ \) is SN\(_{3} \).

Proof. This follows from the fact that \( \lambda \omega^+ \) is SN\(_{3} \), because there exists a mapping \( |-| \) which maps reduction sequences in \( \lambda \omega^+ \) to longer reduction sequences in \( \lambda \omega^+ \). This mapping translates +-, \( \times \)- and \( \Sigma \)-types to their usual encodings, which were given in definition 3.11 as encodings of \( \vee, \wedge \) and \( \exists \). N-ary products and sums can be treated as repeated binary ones, i.e. \( |\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n| = |\sigma_1 \times (\sigma_2 \times (\ldots \times \sigma_n))| \). Labelled products can be treated as unlabelled ones. We assume some ordering on the set of labels to fix a unique order of the fields, and then simply forget about the labels. \( \square \)
4.3 Extension of the Programming Logic

Extending $\lambda \omega_\Sigma$ with $+$, $\times$ and $\Sigma$ in the same way as $\lambda \omega_\Sigma$ produces the following system $\lambda \omega_\Sigma^+$:

4.31 Definition. $\lambda \omega_\Sigma^+_{\langle \delta \rangle}$ is the (D)PTS with $+$, $\times$ and $\Sigma$ specified by

$$
S = \{*_{\Sigma}, \square_{\Sigma}, *_{\Sigma}, \square_{\Sigma}\}
$$

$$
A = \{*_{\Sigma}, \square_{\Sigma}\}
$$

$$
R_H = \{(\square_{\Sigma}, *_{\Sigma}), (\square_{\Sigma}, *_{\Sigma}), (\square_{\Sigma}, *_{\Sigma})\}
$$

$$
R^\Sigma = \{\square_{\Sigma}, *_{\Sigma}\}
$$

In this system cartesian products are interpreted as conjunctions, disjoint sums as disjunctions, and dependent sums as existential quantifications. In $\lambda \omega_\Sigma$, we now also write $\wedge$ for $\times$, $\vee$ for $+$ and $\exists$ for $\Sigma$. The new propositions that can be formed are conjunctions $P_1 \wedge \ldots \wedge P_n$, and disjunctions $P_1 \vee \ldots \vee P_n$, and higher order existential quantifications ($\exists P_1 \cdots P_n Q$). There is no real need for the labelled conjunctions $\forall (l_1, P_1, \ldots, l_n, P_n)$ and labelled disjunctions $\exists (l_1, P_1, \ldots, l_n, P_n)$. In the programming language, labelled products and sums are needed for more readable programs. In the logic, on the other hand, we are not really concerned with the readability of proof terms. Indeed, for the logic there is no real need to have $\vee$, $\wedge$ and $\exists$ as primitives, because for the encodings given in definition 3.11 all the required properties can be proved.

Instead of defining the propositions $\text{True}$ and $\text{False}$ in terms of higher order quantification as in definition 3.11, they can now also be defined as the empty conjunction and disjunction:

4.32 Definition (True and False) The context defining the propositions $\text{True}$ and $\text{False}$ is

$$
\text{True} = \forall () *_{\Sigma} : *_{\Sigma},
$$

$$
\text{False} = \exists () *_{\Sigma} : *_{\Sigma}.
$$

Then for $\text{True}$ and $\text{False}$ we have the following inference rules:

$$
\frac{}{\text{True intro}} \quad \Gamma \vdash () *_{\Sigma} \quad \text{True}
$$

$$
\frac{}{\text{False elim}} \quad \Gamma \vdash p : \text{True} \quad \Gamma \vdash P : *_{\Sigma} \quad \Gamma \vdash \forall P P : \text{False} \rightarrow P
$$

The extension of $\lambda \omega_\Sigma$ with $+$, $\times$ and $\Sigma$, containing both $\lambda \omega_\Sigma^+$ and $\lambda \omega_\Sigma^+$, is the following system $\lambda \omega_\Sigma^+$.

4.33 Definition. $\lambda \omega_\Sigma^+_{\langle \delta \rangle}$ is the (D)PTS with $+$, $\times$ and $\Sigma$ specified by

$$
S = \{*_{\Sigma}, \square_{\Sigma}, *_{\Sigma}, \square_{\Sigma}\}
$$

$$
A = \{*_{\Sigma}, \square_{\Sigma}, *_{\Sigma}, \square_{\Sigma}\}
$$
\[ R^\Pi = \{ (\Box_s, \Box_s), \\
(\Box_s, *s), \ (s, *s), \\
(\Box_s, \Box_p), \ (*s, \Box_p), \ (\Box_p, \Box_p), \\
(\Box_s, *p), \ (*s, *p), \ (\Box_p, *p), \ (*p, *p) \} \]

\[ R^\Sigma = \{ (\Box_s, s), \ (s, *p), \ (\Box_p, *p) \} \]

\[ R^\times = R^+ = \{ *s, *p \} \]

The specification of \( \lambda \omega_+^\Pi \) consists of those of \( \lambda \omega_+^\Pi \) and \( \lambda \omega_+^\Sigma \), with four more rules for \( \Pi \)-types and two more rules for \( \Sigma \)-types. The rules in \( R^\Sigma \) are those of \( \lambda \omega_+^\Sigma \) and \( \lambda \omega_+^\Pi \), plus \((\Box_s, *p)\) and \((s, *p)\). These two rules enable existential quantification, over kinds \( K \) and over datatypes \( \sigma \), i.e. \((\exists \alpha. \sigma. \thinspace P)\) and \((\exists \alpha. K. \thinspace P)\). The rules added for \( \Pi \)-types but not for \( \Sigma \)-types are \((\Box_s, \Box_p)\) and \((s, \Box_p)\). There does not seem to be any use for \( \Sigma \)-types formed using these rules, nor any intuitive interpretation of them.

The context-free syntax given in definition 4.28 for \( \lambda \omega_+^\Pi \) can be extended to the whole of \( \lambda \omega_+^L \).

4.34 Definition. For \( \lambda \omega_+^L \), the sets of pseudoterms \( \text{PCons} \) and \( \text{Proof} \) defined in definition 3.2 are extended as follows:

\[ \text{PCons} ::= \ldots \ | \text{PCons} \land \ldots \text{PCons} \ | \forall (\ell. \text{PCons}, \ldots, \ell.\text{PCons}) \\
\ | \text{PCons} \lor \ldots \text{PCons} \ | \exists (\ell. \text{PCons}, \ldots, \ell.\text{PCons}) \\
\ | (\forall \alpha. \text{Kind} \thinspace \text{PCons}) \ | (\exists \alpha. \text{Cons} \thinspace \text{PCons}) \ | (\exists \alpha. \text{Pkind} \thinspace \text{PCons}) \\
\text{Proof} ::= \ldots \ | (\text{Proof}, \ldots, \text{Proof}) \ | (\ell \rightarrow \text{Proof}, \ldots, \ell \rightarrow \text{Proof}) \ | ()_{*p} \ | \text{Proof} \ell \\
\ | \text{inPCons} \ell \text{Proof} \ | \text{OptProof} \ |
\text{Proof} \text{PCons} \\
\ | \text{inPCons Cons Proof} \ | \text{inPCons Prog Proof} \ | \text{inPCons PCons Proof} \]

In exactly the same way lemma 3.17 was proved, it can be proved that:

4.35 Lemma (Classification for \( \lambda \omega_+^L \))

1. If \( \Gamma \vdash K : \Box_s \), then \( K \in \text{Kind} \)
2. If \( \Gamma \vdash \sigma : \Box_s \), then \( \sigma \in \text{Cons} \)
3. If \( \Gamma \vdash M : \sigma : *s \), then \( M \in \text{Prog} \)
4. If \( \Gamma \vdash P : \Box_p \), then \( P \in \text{Pkind} \)
5. If \( \Gamma \vdash P : \ell P : \Box_p \), then \( P \in \text{PCons} \)
6. If \( \Gamma \vdash p : P : *p \), then \( p \in \text{Proof} \).

\[ \square \]
The encodings of $+, \times$ and $\Sigma$ can be used to show that $\lambda\omega^+_L$ and $\lambda\omega^+_L$ are strongly normalising.

4.36 Corollary The system $\lambda\omega^+_L$ is $SN_{\theta(\delta)}$.

Proof. As corollary 4.30, now using $SN_{\theta(\delta)}$ of $\lambda\omega_L$ instead of $\lambda\omega$.

Because there are no rules of the form $(-p, -r)$ in $R^\Pi$ or $R^\Sigma$, $\lambda\omega^+_L$ is a conservative extension of $\lambda\omega^+_L$.

4.37 Theorem (Conservativity of $\lambda\omega^+_L$ over $\lambda\omega^+_L$).

Suppose $\Gamma \vdash a : A$ in $\lambda\omega^+_L$ and $a$ is a program, datatype-constructor or a kind (i.e. $a \in \text{Prog} \cup \text{Cons} \cup \text{Kind}$).

Then $\Gamma \vdash a : A$ in $\lambda\omega^+_L$.

Proof. Induction on the derivation of $\Gamma \vdash a : A$.

4.38 Discussion. We now discuss possible generalisations of the elimination rules for $+$-types, and our reasons for excluding them. Similar generalisation exists for the elimination rules for $\Sigma$-types. These generalised rules are instances of the rules for so-called inductive types given in [CP09].

Suppose $\Gamma \vdash \sigma + \tau : \star_s$. The elimination rule we have for this datatype is

$$
\Gamma \vdash f : \sigma \rightarrow \rho \quad \Gamma \vdash g : \tau \rightarrow \rho \quad \Gamma \vdash \rho : \star_s
$$

\[ \frac{\Gamma \vdash f \triangleright g \triangleright \sigma + \tau \rightarrow \rho}{\Gamma \vdash f \triangleright g \triangleright \sigma + \tau \rightarrow \rho} \]

- One possible generalisation is to allow the range of $f \triangleright g$ to be another type in the rule above to have

$$
\Gamma \vdash \rho : \star_s
$$

\[ \frac{\Gamma \vdash f \triangleright g \triangleright \sigma + \tau \rightarrow \star_s}{\Gamma \vdash P \triangleright Q : \sigma + \tau \rightarrow \star_s} \]

This inference rule allows a predicate on $\sigma + \tau$ to be defined using a case-distinction. Without rule (1), it is still possible to define a predicate on $\sigma + \tau$ in terms of predicates on $\sigma$ and $\tau$, for instance as

$$
R = \lambda z. \sigma + \tau \ (\exists y. \sigma \ z =_{\sigma + \tau} \text{in}_{\sigma + \tau} 1 \tau \land P \ y) \lor (\exists y. \tau \ z =_{\sigma + \tau} \text{in}_{\sigma + \tau} 2 \ y \land Q \ y)
$$

From an extensional point of view $P \triangleright Q$ and $R$ are the same. The only difference is in their reduction behaviour. The reduction rule for $\triangleright$ gives

$$(P \triangleright Q)(\text{in}_{\sigma + \tau} 1 \ x) \triangleright y \ P \ x$$

i.e. $(P \triangleright Q)(\text{in}_{\sigma + \tau} 1 \ x)$ and $P \ x$ are convertible. On the other hand, $R(\text{in}_{\sigma + \tau} 1 \ x)$ and $P \ x$ are not convertible. Instead, we have to use a proof $\rho$ such that

$$
\rho \ R(\text{in}_{\sigma + \tau} 1 \ x) \iff P \ x
$$
4.3. EXTENSION OF THE PROGRAMMING LOGIC

- Another way to generalise the elimination rule for +-types is to allow $f \forall g$ to have a $\Pi$-type, and not just an $\rightarrow$-type:

  \[(i) \quad \Gamma \vdash p_1 : \forall x. \sigma. P \left(\text{in}_\sigma 1 \ x\right) \quad \Gamma \vdash p_2 : \forall y. \tau. P \left(\text{in}_\tau 2 \ y\right) \quad \Gamma \vdash P \cdot \sigma + \tau \rightarrow *_p\]

  \[(ii) \quad \Gamma \vdash p_1 : \forall x. \sigma. P \left(\text{in}_\sigma 1 \ x\right) \quad \Gamma \vdash p_2 : \forall y. \sigma + \tau \rightarrow *_p \]

  This rule can be used to prove properties of inhabitants of $\sigma + \tau$, e.g.

  \[\forall x : \sigma + \tau \ (\exists x : \sigma. z = \text{in}_\sigma 1 \ x) \lor (\exists y : \tau. z = \text{in}_\tau 2 \ y)\]

  Instead of rule (ii), we can simply introduce an axiom

  \[AX_{\sigma + \tau} : \forall P : \sigma + \tau \rightarrow *_p \quad (\forall x : \sigma. P \left(\text{in}_\sigma 1 \ x\right)) \Rightarrow (\forall y : \tau. P \left(\text{in}_\tau 2 \ y\right)) \Rightarrow (\forall z : \sigma + \tau \ P \ z)\]

  Again, the only difference between having rule (ii) and having the axiom $AX_{\sigma + \tau}$ is in the reduction behaviour. The proof $(p_1 \forall p_2)(\text{in}_\sigma 1 \ x)$ of the proposition $P \ (\text{in}_\sigma 1 \ x)$ $\beta$-reduces to $(p_1 \ x)$, the proof $(AX_{\sigma + \tau} P p_1 p_2 (\text{in}_\sigma 1 \ x))$ of the same proposition does not. But, since we are not interested in the reduction behaviour of proofs, this difference does not matter.

  So the rules (i) and (ii) are not strictly necessary, and only rule (i) has any advantages (viz. a more powerful intensional equality of propositions). We exclude them for two reasons.

  First, if these rules are included, we can no longer use the fact that $\lambda C_\delta$ is $\beta$-strongly normalising to prove strong normalisation of the system. We do not want to have to pay attention to a strong normalisation proof here.

  A more fundamental reason is that in chapter 5 the programming language will be extended with a fixpoint operator. Types will be then be interpreted as cpos, and the datatype $\sigma + \tau$ will include an "undefined" element $\perp_{\sigma + \tau}$. Rule (ii) is then no longer sound. The problem with rule (i) is that it does not follow from the reduction rules for $\forall$ whether the predicate $P \forall Q$ is true or false in $\perp_{\sigma + \tau}$. To define a predicate on the cpo $\sigma + \tau$ in terms of predicates on $\sigma$ and $\tau$, we now also have to choose a value for the predicate in $\perp_{\sigma + \tau}$. (In the programming language this problem is solved by taking the function $f \forall g$ to be strict, i.e. the value of $f \forall g : \sigma + \tau \rightarrow \rho$ in $\perp_{\sigma + \tau}$ will be $\perp_{\rho}$.)

As in $\lambda \omega_L$, in $\lambda \omega_L^+$, we want to use the axioms in $AXIOM$ (definition 3.23) for classical logic and extensionality of functions in the programming language. To reason about the new datatypes some additional axioms are needed.

For $\times$-types we want axioms stating that pairing is surjective, i.e. that all inhabitants of $\Pi(l_1, \sigma_1 \ldots l_n, \sigma_n)$ are of the form $(l_1 \rightarrow N_1, \ldots, l_n \rightarrow N_n)$.

Similarly, for $+$- and $\Sigma$-types we want axioms stating that all their inhabitants can be constructed using (one of) the injection(s).

Finally, we introduce an axiom stating that true and false are not equal. Using this axiom we can prove, for any $+$-type, that $\text{in}_{\Sigma(l_1, \sigma_1 \ldots l_n, \sigma_n), l_1 N_1}$ and $\text{in}_{\Sigma(l_1, \sigma_1 \ldots l_n, \sigma_n), l_2 N_2}$ are not equal if $l_1 \neq l_2$. To do this, define a function $f : \Sigma(l_1, \sigma_1 \ldots l_n, \sigma_n) \rightarrow \text{bool}$ such that $f \ (\text{in}, l_i, N_i) \rightarrow \text{true}$ and $f \ (\text{in}, l_j, N_j) \rightarrow \text{false}$; then by the definition of Leibniz' equality it follows from false $\neq$ true that

\[\text{in}_{\Sigma(l_1, \sigma_1 \ldots l_n, \sigma_n), l_1 N_1} \neq \text{in}_{\Sigma(l_1, \sigma_1 \ldots l_n, \sigma_n), l_2 N_2}\]
4.39 Definition. $AXIOM^+$ is the set of axioms containing

- **classic**: $\forall P. P. (\neg\neg P) \Rightarrow P$,
- **pbool**: true $\neq$ bool false,
- $AX_{\sigma \rightarrow \tau}$: $\forall f, g : \sigma \rightarrow \tau. (\forall x: \sigma. (f \ x) =_\tau (g \ x)) \Rightarrow f =_\tau g$,
- $AX_{\Pi \alpha K. \sigma}$: $\forall f, g : (\Pi \alpha K. \sigma). (\forall \alpha K. (f \alpha) =_\sigma (g \alpha)) \Rightarrow f =_\sigma K. \sigma g$,
- $AX_{\Sigma \alpha K. \sigma}$: $\forall P. \Pi(l_1: \sigma_1, \ldots, l_n: \sigma_n) \Rightarrow P$,

\[
\forall x_1: \sigma_1, \ldots, x_n: \sigma_n. P(l_1 \rightarrow x_1, \ldots, l_n \rightarrow x_n)
\]

\[
\Rightarrow (\forall x. \Pi(l_1: \sigma_1, \ldots, l_n: \sigma_n). P \ x)
\]

\[
AX_{\Sigma \alpha K. \sigma}
\]

\[
\Rightarrow (\forall x. \Sigma(l_1: \sigma_1, \ldots, l_n: \sigma_n). P \ x)
\]

\[
AX_{\Sigma \alpha K. \sigma}
\]

\[
\Rightarrow (\forall z. (\Sigma \alpha. K. \sigma). P) \ x)
\]

for all $\sigma \rightarrow \tau. (\Pi \alpha K. \sigma), \Pi(l_1: \sigma_1, \ldots, l_n: \sigma_n), \Sigma(l_1: \sigma_1, \ldots, l_n: \sigma_n), (\Sigma \alpha. K. \sigma) \in \text{Cons}$

Using axioms in $AXIOM^+$ we can for example prove the propositions $(\forall x: \text{Empty}. \text{False})$, $(\forall x: \text{Unit}. x =_\text{Unit} \text{unit})$, and $(\forall x: \text{bool}. x =_\text{bool} \text{true} \lor x =_\text{bool} \text{false})$

The axioms in $AXIOM^+$ can be safely used in $\lambda \omega_L^+$.

4.40 Lemma (Consistency of $AXIOM^+$ in $\lambda \omega_L^+$). False is not provable in a context containing only axioms from $AXIOM^+$

**Proof.** See lemma 7.27.

### 4.4 Program and Proof Development

In chapter 3 we gave examples of coupled derivation rules in $\lambda \omega_1$, pairs of derivable rules consisting of a type inference rule and a corresponding proof rule for a certain program construction. $\lambda \omega_1^+$ offers many new ways for forming programs in addition to the lambda abstraction and application available in $\lambda \omega_2$. For these we can give proof rules to accompany the type inference rules. Some examples are give below

4.41 Lemma (Coupled derivation rule for if - then - else ). The following rules are derivable using axioms from $AXIOM^+$

- $\Gamma \vdash b : \text{bool}$
- $\Gamma \vdash \rho \ : \ast_p$
- $\Gamma \vdash M : \rho$
- $\Gamma \vdash N : \rho$
- $\Gamma \vdash \text{if } b \text{ then } M \text{ else } N : \rho$
- $\Gamma \vdash R : \rho \rightarrow \ast_p$
- $\Gamma, p : (b =_\text{bool} \text{true}) \vdash R M$
- $\Gamma, p : (b =_\text{bool} \text{false}) \vdash R N$
- $\Gamma \vdash \text{if } b \text{ then } M \text{ else } N$
(When we present such coupled derivation rules as above, it is left implicit that the premisses of the type inference rule are also premisses of the proof rule. Repeating all these premisses would be tedious. In this particular case, just the premiss $\Gamma \vdash \rho : \sigma$ is needed for the proof rule to be derivable.)

**Proof.** Showing that the type inference rule (on the left) is derivable is trivial (as will be the case in the examples that follow). To show that the proof rule (on the right) is derivable, two axioms from $AXIOM^+$ are needed. Recall that $bool \triangleq \text{Unit} + \text{Unit}$ and $\text{Unit} \triangleq \Pi(\tau)$. From $AX_{\Pi(\tau)}$ and $AX_{\text{Unit}+\text{Unit}}$ it follows that $\forall b \in bool. b =_{bool} \text{true} \lor b =_{bool} \text{false}$ (i).

Then

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premisses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\rho : \sigma, M, N : \rho, b : \text{bool}, R : \rho \rightarrow \ast \rho$, $b =<em>{\text{bool}} \text{true} \Rightarrow RM$, $b =</em>{\text{bool}} \text{false} \Rightarrow RN$</td>
</tr>
<tr>
<td>2</td>
<td>$b =_{\text{bool}} \text{true}$</td>
</tr>
<tr>
<td>3</td>
<td>$RM$</td>
</tr>
<tr>
<td>4</td>
<td>$R \text{(if true then } M \text{ else } N)$</td>
</tr>
<tr>
<td>5</td>
<td>$R \text{(if } b \text{ then } M \text{ else } N)$</td>
</tr>
<tr>
<td>6</td>
<td>$b =_{\text{bool}} \text{true} \Rightarrow R \text{(if } b \text{ then } M \text{ else } N)$</td>
</tr>
<tr>
<td>7</td>
<td>$b =_{\text{bool}} \text{false} \Rightarrow R \text{(if } b \text{ then } M \text{ else } N)$</td>
</tr>
<tr>
<td>8</td>
<td>$R \text{(if } b =<em>{\text{bool}} \text{false} \lor b =</em>{\text{bool}} \text{true}) \Rightarrow R \text{(if } b \text{ then } M \text{ else } N)$</td>
</tr>
<tr>
<td>9</td>
<td>$\forall b \in bool. R \text{(if } b \text{ then } M \text{ else } N)$</td>
</tr>
</tbody>
</table>

Similar coupled derivation rules can be given for all the $+\text{-types}$:

### 4.4.2 Lemma (Coupled derivation rule for case)

The following rules are derivable using axioms from $AXIOM^+$:

\[
\begin{align*}
\Gamma &\vdash \Sigma(l_1, \sigma_1, \ldots, l_n, \sigma_n) : \ast \sigma \\
\Gamma &\vdash \rho : \ast \sigma \\
\Gamma &\vdash \tau : \sigma_1 \triangleright M_1 : \rho \text{ for } i = 0 \ldots n \\
\Gamma &\vdash (\text{case } N \text{ of } \text{in } l_1 x_1 \mapsto M_1 \mid \ldots | \text{in } l_n x_n \mapsto M_n) \vdash \rho
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash R : \rho \rightarrow \ast \rho \\
\Gamma, p : (N =_{\Sigma(l_1, \sigma_1, \ldots, l_n, \sigma_n)} \text{in } l_i x_i) &\vdash R M_i \text{ for } i = 0 \ldots n \\
\Gamma &\vdash \vdash (\text{case } N \text{ of } \text{in } l_1 x_1 \mapsto M_1 \mid \ldots | \text{in } l_n x_n \mapsto M_n) \\
\end{align*}
\]

Subscripts of $\text{in}$ have been omitted here to keep things readable; it is clear from the context what they should be.

**Proof.** Analogous to lemma 4.41, now using $AX_{\Sigma(l_1, \sigma_1, \ldots, l_n, \sigma_n)}$ to distinguish the cases $N =_{\Sigma(l_1, \sigma_1, \ldots, l_n, \sigma_n)} \text{in } l_i N_i$, where $i = 1 \ldots n$. ☐

The specification of an abstract datatype $(\Sigma \sigma : \ast \sigma \sigma)$ involves a polymorphic predicate $P : (\Pi \ast \sigma \sigma : \sigma \rightarrow \ast \sigma)$. As an example we look at the abstract datatype of boolean lists introduced in example 4.25.
4.43 EXAMPLE. Recall that in example 4.25 the datatype-constructors ADT and \( F \) were defined as:

\[
\begin{align*}
F & = \lambda \text{boolist} : * \ 	ext{II( empty : boolist,} \\
& \quad \text{cons : bool \rightarrow boolist \rightarrow boolist,} \\
& \quad \text{head : boolist \rightarrow bool,} \\
& \quad \text{tail : boolist \rightarrow boolist }) \cdot * \rightarrow * \\
\text{ADT} & = (\Sigma \text{boolist} : * \ F \ \text{boolist}) \ * 
\end{align*}
\]

To specify the abstract datatype \( \text{ADT} \) of boolean lists we can use the following polymorphic predicate \( P \),

\[
P = \lambda \text{boolist} : * \ \lambda r : \text{boolist} \\
x \ \text{tail} \ x \ \text{empty} = \text{boolist in 2 unit} \land x. \text{head} \ x. \ \text{empty} = \text{bool in 2 unit} \\
\land (\forall b \ \text{bool}, \ b \ \text{boolist} \ x \ \text{tail}(x. \ \text{cons} \ b l) = \text{boolist in 1 1} \land x. \ \text{head}(x. \ \text{cons} \ b l) = \text{bool in 1 b}) \\
\ni \ \text{boolist} : * \ F \ \text{boolist} \rightarrow * \rho
\]

Again, subscripts of \( i \) have been omitted here to keep things readable.

An implementation \((\text{in}_{\text{ADT}} \tau \ N) \) \( \text{ADT} \) of the abstract datatype is correct if we can give a proof of \((P \rightarrow \tau)\). For a program \( \forall (\lambda \text{boolist} : * \ \lambda r : F \ \text{boolist} \ M) : \text{ADT} \rightarrow \rho \) we can then prove correctness of \( M \) under the assumption that \((P \ \text{boolist} \ \tau)\). \( \square \)

For the abstype-construction defined in 4.26 we can give the following proof rule:

4.44 LEMMA (Coupled derivation rule for abstype). The following rules are derivable using axioms from \( \text{AXIOM}^+ \):

\[
\begin{align*}
\Gamma \vdash (\Sigma \alpha : \sigma \ * \sigma) : * \\
\Gamma \vdash \rho \ * * \\
\Gamma \vdash \tau \ * * \\
\Gamma \vdash N : \sigma | \alpha = \tau | \\
\Gamma, \alpha : \star, \tau : \sigma \vdash M : \rho \\
\Gamma \vdash (\text{abstype} \ \alpha : \star \ with \ \tau : \sigma \ is \ (\text{in}_{\Sigma \alpha : \sigma} \ * \tau \ N) \ in \ M) \ \rho \\
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash P : (\ll \rho, * \sigma \rightarrow * \rho) \\
\Gamma \vdash R : \rho \rightarrow * * \\
\Gamma \vdash P : \tau \ N \\
\Gamma, \alpha : *, x. \sigma, y. \tau \vdash P \ \alpha x \vdash R \ M \\
\Gamma \vdash R : (\text{abstype} \ \alpha : * \ with \ x : \sigma \ is \ (\text{in}_{\Sigma \alpha : \sigma} \ * \tau \ N) \ in \ M)
\end{align*}
\]

PROOF. To prove that the proof rule is derivable, we have to show that

\[
(\forall x : * \ , \sigma \ P \ \alpha x \Rightarrow \ R M) \Rightarrow R : (\text{abstype} \ \alpha : * \ with \ x : \sigma \ is \ (\text{in}_{\Sigma \alpha : \sigma} \ * \tau \ N) \ in \ M)
\]

can be derived. This immediately follows from the fact that

\[
(\text{abstype} \ \alpha : * \ with \ x : \sigma \ is \ (\text{in}_{\Sigma \alpha : \sigma} \ * \tau \ N) \ in \ M) \Rightarrow M[\alpha := \tau] \ M \Rightarrow \tau \ N)
\]

\( \square \)
The derivation rules above can be generalised for \((\Sigma \alpha \rightarrow K \sigma)\) with \(K\) an arbitrary kind, simply by taking \(P : (\Pi \alpha : K. \sigma \rightarrow *_p)\). The coupled derivation rules for the abstype-construct can also be specialised, to take into account a representation invariant for the abstract datatype. As an example, we first reconsider the specification for \(ADT\):

4.45 Example. The specification given for \(ADT\) in example 4.43 can be improved by requiring that there is a representation invariant \(I : bool list \rightarrow *_p\) which holds for \(empty\) and is preserved by \(cons\) and \(tail\):

\[
P = \lambda bool list : *_s. \lambda I : bool list \rightarrow *_s. \lambda x : F bool list.
\]

\[
I x. empty \land x.tail x.empty = bool list in.2 unit \land x.head x.empty = bool in.2 unit
\land (\forall \langle b : bool, l : bool list \rangle)
\]

\[
(I l) \Rightarrow (x.tail(x.cons b l) = bool list in.1 l \land x.head(x.cons b l) = bool in.1 b
\land I (x.tail l) \land I (x.cons b l))
\]

\[
:\Pi bool list : *_s (\langle l : *_p \rangle) \rightarrow (F bool list \rightarrow *_p)
\]

Compared with the specification given in example 4.43, the conditions "\(x.tail(x.cons b l) = l\)" and "\(x.head(x.cons b l) = b\)" now only have to hold for lists \(l\) that satisfy the representation invariant \(I\). Now an implementation \((in_{ADT} \tau N)\) \(ADT\) is correct if we can give an invariant \(I \cdot \tau \rightarrow *_p\) for which there is a proof of \((P \cdot I N)\).

Recall that in section 3.4.3 we discussed the annotation for datatype-constructors. The prop-constructor \(P\) annotates the datatype-constructors \(F : *_s \rightarrow *_s\). Note that the type of \(P\) is indeed \(type_of . lift *_s . \tau (F)\) (as defined in definition 3.31).

For the abstype-construction another proof rule can now be given:

4.46 Lemma (Coupled derivation rule for abstype).

The following rules are derivable using axioms from AXIOM:

\[
\Gamma \vdash (\Sigma \alpha:*_s \sigma) : *_s
\]

\[
\Gamma \vdash \rho : *_s
\]

\[
\Gamma \vdash \tau : *_s
\]

\[
\Gamma \vdash N : \sigma[\alpha := \tau]
\]

\[
\Gamma, \alpha:*_s, x: \sigma \vdash M : \rho
\]

\[
\Gamma \vdash P : (\Pi \alpha:*_s, (\alpha \rightarrow *_s) \rightarrow (\sigma \rightarrow *_p))
\]

\[
\Gamma \vdash R : \rho \rightarrow *_p
\]

\[
\Gamma \vdash I : \tau \rightarrow *_p
\]

\[
\Gamma \vdash \tau \cdot I N
\]

\[
\Gamma, \alpha:*_s, I \alpha \rightarrow *_p, \sigma, p \cdot \tau \vdash \tau \cdot P \cdot I N
\]

\[
\Gamma \vdash \tau \cdot R (\alpha:*_s, x: \sigma, p \cdot \tau \vdash \tau \cdot R M)
\]

Proof. Similar to lemma 4.44.

Again, these derivation rules can be generalised to \((\Sigma \alpha \rightarrow K \sigma)\) with \(K\) an arbitrary kind. Then we have to take \(P : (\Pi \alpha \rightarrow K \cdot type_of . lift K (\alpha) \rightarrow (\sigma \rightarrow *_p))\) and \(I \cdot type_of . lift K (\alpha)\).
The coupled derivation rules are all *compositional* rules, i.e., they deduce the type and the correctness of a program from the type and correctness of its component parts. To prove correctness of programs we may also want to use *transformation* rules, i.e., using equalities between programs to replace (sub)programs by equal ones. Using axioms from $AXIOM^+$ many equalities can be derived in $\lambda\omega_L$.

4.47 Definition (composition).
Composition for programs is defined by the following context $\Gamma_{composition}$

\[
\begin{align*}
\mathcal{C} & = \lambda\alpha,\beta,\gamma.\lambda g.\beta \rightarrow \gamma.\lambda f.\alpha \rightarrow \beta.\lambda x.\alpha.\ g\ (f\ x) \\
\mathcal{P} & = \Pi\alpha,\beta,\gamma.\ (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)
\end{align*}
\]

will be written *infix*, without its three type parameters. So we write $gof$ for $(\mathcal{C} \circ \mathcal{P} \circ g \circ f)$. These missing type parameters can easily be reconstructed (by looking at the types of $f$ and $g$).

It is not difficult to prove the transformation rule for distributing $\circ$ over $\triangleright$:

4.48 Lemma. The following proposition is provable using axioms from $AXIOM^+$

\[
\forall \rho_1, \rho_2, \tau, \sigma : \ast, f : \rho_1 \rightarrow \sigma, g : \rho_2 \rightarrow \sigma, h : \sigma \rightarrow \tau \quad ho(f \triangleright g) = \rho_1 + \rho_2 \rightarrow \tau \quad (ho f)\triangleright (ho g)
\]

Proof.

\[
\begin{align*}
1 & \quad p_1, p_2, \tau, \sigma : \ast, f : p_1 \rightarrow \sigma, g : p_2 \rightarrow \sigma, h : \sigma \rightarrow \tau \\
2 & \quad x : p_1 \\
3 & \quad (ho(f \triangleright g))(in_{p_1 + p_2} x) = \tau \quad ((ho f)\triangleright (ho g))(in_{p_1 + p_2} x) \quad \text{both sides $\beta\eta$-reduce} \\
4 & \quad \forall x : p_1. \ (ho(f \triangleright g))(in_{p_1 + p_2} 1 x) = \tau \quad ((ho f)\triangleright (ho g))(in_{p_1 + p_2} 1 x) \\
5 & \quad \forall y : p_2. \ (ho(f \triangleright g))(in_{p_1 + p_2} 2 y) = \tau \quad ((ho f)\triangleright (ho g))(in_{p_1 + p_2} 2 y) \quad \text{similarly} \\
6 & \quad \forall z : p_1 + p_2. \ (ho(f \triangleright g)) z = \tau \quad ((ho f)\triangleright (ho g)) z \\
7 & \quad ho(f \triangleright g) = \rho_1 + \rho_2 \rightarrow \tau \quad (ho f)\triangleright (ho g) \\
\end{align*}
\]

Many such transformation rules can be proved. For instance, for the distribution of composition over $(if \ then \ else)$

4.49 Lemma. The following propositions are provable using axioms from $AXIOM^+$

1. $\forall c : \mathbf{bool}, \sigma, \tau : \ast. \forall f : (c \rightarrow \sigma) + \rho \rightarrow \tau. \forall h : \rho \rightarrow \sigma. \quad (if\ c\ then\ f\ else\ g)\circ h = \rho_1 + \rho_2 \rightarrow \tau \quad \text{if } h\ \text{then } f\circ h\ \text{else } g\circ h$

2. $\forall c : \mathbf{bool}, \sigma, \tau : \ast. \forall h : \sigma \rightarrow \tau. \forall f : (c \rightarrow \sigma) + \rho \rightarrow \tau. \quad ho(\text{if } c\ \text{then } f\ else\ g) = \rho_1 + \rho_2 \rightarrow \tau \quad \text{if } h\ \text{then } h\circ f\ \text{else } h\circ g$

By the definition of $\mathbf{bool}$ and $(if\ then\ else)$, the second part of this lemma is an instantiation of the previous lemma.
Chapter 5

Recursion

Recursive datatypes and recursive programs are essential in any typed functional programming language. In this chapter the programming language is extended with recursion at both the level of datatypes and the level of programs. The new programming language $\lambda\omega^p_L$ is the extension of $\lambda\omega^p_L$ with two new primitives, namely a type constructor $\mu$ to form recursive datatypes and a fixpoint combinator $Y$ to form recursive programs.

The extension with unrestricted recursion drastically changes the nature of the programming language. It now offers the full power of the untyped lambda calculus. The price for this unrestricted recursion is that the programming language now includes non-terminating programs: programs of type $\sigma \rightarrow \tau$ are now partial functions from $\sigma$ to $\tau$. This affects the semantics of the programming language and (hence) the way we reason about programs in the programming logic.

In chapter 8 the semantics of the programming language with recursion is discussed. Datatypes can no longer interpreted be as sets. Instead, they will be interpreted as cpos. This means that domain theory can be used as the basis for reasoning about the new programming language. This is the conventional way of reasoning about non-termination and partial objects; the best known example of a programming logic based on domain theory is LCF [GMW79].

The programming logic $\lambda\omega^p_T$ will be extended to a system $\lambda\omega^p_L$. No new logical primitives are needed in $\lambda\omega^p_L$. Some domain-theoretic notions, such as the ordering $\subseteq$ on programs, and their properties are needed, but these can all be expressed using the primitives of $\lambda\omega^p_T$.

It will be shown that all the domain theory that is needed can be formalized inside the system $\lambda\omega^p_L$. The higher-order nature of the logic turns out to be useful for this. In particular, it enables us to define the notion of admissibility (or chain-completeness) for predicates in the system. Consequently, the logic is more powerful than LCF, because there this notion has to live in the meta-theory.

The structure of this chapter is similar to the previous ones. In section 5.1 the programming language $\lambda\omega^p_L$ is defined. In section 5.2 the associated programming logic $\lambda\omega^p_L$ is defined, and a formalisation of domain theory in this system is given. Finally, in section 5.3 we give some examples of ways to prove correctness of programs in $\lambda\omega^p_L$. 

85
CHAPTER 5. RECURSION

5.1 Extension of the Programming Language

In this section we define the program language \( \lambda \omega^\mu \), which extends the system \( \lambda \omega^+ \) defined in the previous chapter with recursive datatypes and recursive programs.

For recursive datatypes, a new type constructor \( \mu \) is introduced, with which recursive types of the form \( (\mu \alpha \ast \sigma) \) can be formed. A recursive type \( (\mu \alpha \ast \sigma) \) is the solution of the recursive type equation

\[
\alpha \cong \sigma,
\]

where \( \alpha \in \text{Var}^{Q} \) is a datatype-variable. In other words, the datatype \( (\mu \alpha \ast \sigma) \) is isomorphic with its unfolding \( \sigma[\alpha := (\mu \alpha \ast \sigma)] \). This isomorphism is given by two operations \( \text{unfold}_{\mu \alpha \ast \sigma} \) and \( \text{fold}_{\mu \alpha \ast \sigma} \), which map inhabitants of a recursive type to its unfolding and back. These operations come with the following reduction rules:

\[
\begin{align*}
\text{unfold}_{\mu \alpha \ast \sigma}(\text{fold}_{\mu \alpha \ast \sigma} M) & \triangleright_{\mu} M \\
\text{fold}_{\mu \alpha \ast \sigma}(\text{unfold}_{\mu \alpha \ast \sigma} M) & \triangleright_{\mu} M
\end{align*}
\]

It follows from these reduction rules that \( \text{fold}_{\mu \alpha \ast \sigma} \) and \( \text{unfold}_{\mu \alpha \ast \sigma} \) do indeed give an isomorphism.

An alternative to operations \( \text{fold}_{\mu \alpha \ast \sigma} \) and \( \text{unfold}_{\mu \alpha \ast \sigma} \) is to identify a recursive type and its unfolding. Later, in discussion 5.10, it is explained why we have chosen not to do this.

Recursion at program level is provided by a fixpoint-operator \( Y \),

\[
Y \ (\Pi \alpha \ast \sigma \ (\alpha \rightarrow \alpha) \rightarrow \alpha)
\]

with the reduction rule

\[
Y \sigma f \triangleright_{\mu} f (Y \sigma f)
\]

In fact, there is no real need to add a fixpoint-operator as a primitive. Because we have solutions of all recursive type equations - including for instance a solution \( (\mu \alpha \ast \alpha) \) of \( \alpha \cong \alpha \rightarrow \alpha \) - we have the full power of the untyped lambda calculus, and fixpoint-operators can be defined. An example of this is given later, in example 5.2. The reason for including a fixpoint-operator \( Y \) as a primitive is that plays such an important role. Also, a direct implementation of \( Y \) will be more efficient than any definable fixpoint-operator.

The programming language \( \lambda \omega^\mu \) and its extension with definitions \( \lambda \omega^\mu_D \) are now defined as follows:

5.1 Definition \( (\lambda \omega^\mu_D(\delta)) \)

The programming language \( \lambda \omega^\mu(\delta) \) is the extension of \( \lambda \omega^+(\delta) \) with the pseudoterms, reduction rules and type inference given below. The set of pseudoterms is extended as follows

\[
T ::= (\mu \text{Var}^{\bar{\delta}} \ast \ T) \mid \text{fold}_{\mu \alpha \ast \sigma} \ T \mid \text{unfold}_{\mu \alpha \ast \sigma} \ T \mid Y
\]

The following reduction rules are added

\[
\begin{align*}
\text{unfold}_{\mu \alpha \ast \sigma}(\text{fold}_{\mu \alpha \ast \sigma} M) & \triangleright_{\mu} M \\
\text{fold}_{\mu \alpha \ast \sigma}(\text{unfold}_{\mu \alpha \ast \sigma} M) & \triangleright_{\mu} M \\
Y \sigma f & \triangleright_{\mu} f (Y \sigma f)
\end{align*}
\]
5.1. EXTENSION OF THE PROGRAMMING LANGUAGE

and the following type inference rules

\[(\mu\text{form})\quad \frac{\Gamma, \alpha : A \vdash \sigma : B}{\Gamma \vdash (\mu\alpha : A \cdot \sigma) : B}\]

\[(\mu\text{intro})\quad \frac{\Gamma \vdash M : (\mu\alpha : A \cdot \sigma)}{\Gamma \vdash \text{fold}_{\mu\alpha} \cdot \sigma M : (\mu\alpha : A \cdot \sigma)}\]

\[(\mu\text{elim})\quad \frac{\Gamma \vdash M : (\mu\alpha : A \cdot \sigma)}{\Gamma \vdash \text{unfold}_{\mu\alpha} \cdot \sigma M : \sigma[\alpha := (\mu\alpha : A \cdot \sigma)]}\]

\[(\Y\text{intro})\quad \epsilon \vdash \Y : (\Pi\alpha : A \cdot (\alpha \rightarrow \alpha) \rightarrow \alpha)\]

Unlike the reduction rules for +, × and Σ introduced in the previous chapter, the new reduction rules are not included in the notion of β-reduction, but instead we have a separate notion of ρ-reduction. As a result, β- and ρ-reduction are kept strongly normalising. At this point there is no reason for having separate reduction relations ρβ and ρρ. But for the programming logic \(\Lambda T^\omega\) introduced in section 5.2 it will be important that β(δ)-reduction is kept strongly normalising (or rather that β(δ)-conversion is kept decidable). This will be explained in more detail in the next section in discussion 5.10. For the same reason we do not identify recursive types with their unfoldings. Then β-reduction would also not be strongly normalising and β-conversion would be undecidable.

The two reduction rules for \(\text{fold}_{\mu\alpha} \cdot \sigma\) and \(\text{unfold}_{\mu\alpha} \cdot \sigma\) can be seen as a β- and an η-reduction rule. It is the convention that the reduction of a term formed by an introduction construction applied to an elimination construction is called β-reduction, and that the reduction of a term formed by an elimination construction applied to an introduction construction is called η-reduction. For example, in the β-reduces \((\lambda x : A \cdot b) a \rightarrow (a, b)\) and \((a, b) \cdot 1\) elimination constructions - application and selection - are applied to introduction constructions - abstraction and pairing. In the η-reduces \((\lambda x : A \cdot b x) a \rightarrow (a, b)\) and \((a, b) \cdot 1\) introduction constructions are applied to elimination constructions.

By this convention \(\text{unfold}_{\mu\alpha} \cdot \sigma\) (\(\text{fold}_{\mu\alpha} \cdot \sigma \cdot M\) \(\rho\mu\), \(M\) is the β-reduction rule, and \(\text{fold}_{\mu\alpha} \cdot \sigma\) (\(\text{unfold}_{\mu\alpha} \cdot \sigma \cdot M\) \(\rho\mu\), \(M\) the η-reduction rule for \(\mu\)-types. Of the two reduction rules for \(\text{fold}_{\mu\alpha} \cdot \sigma\) and \(\text{unfold}_{\mu\alpha} \cdot \sigma\) it is this "β"-reduction rule that appears to be the crucial one. For instance, in example 5.2 below it is this reduction rule that is needed to produce an infinite reduction sequence. We do not know if there are infinite reduction sequences in which the "η"-rule \(\text{fold}_{\mu\alpha} \cdot \sigma\) (\(\text{unfold}_{\mu\alpha} \cdot \sigma \cdot M\) \(\rho\mu\), \(M\) is used, but the "β"-rule \(\text{unfold}_{\mu\alpha} \cdot \sigma\) (\(\text{fold}_{\mu\alpha} \cdot \sigma \cdot M\) \(\rho\mu\), \(M\) not used

5.2 EXAMPLE An example of a fixpoint-operator for programs that is now typable is

\[\Y_{\text{Turing}} = \lambda \alpha : \ast \cdot \tau = (\mu \beta : \ast \cdot \beta \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha) : \ast\]

in \(w = (\lambda \tau : \ast \cdot \lambda x : (\alpha \rightarrow \alpha) \cdot f ((\text{unfold}_{\tau} x) \cdot f(x)))\)

\[\tau \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha\]

in \(w = (\text{fold}_{\tau} w)\)

\[\Y_{\text{Turing}} : (\Pi\alpha : \ast \cdot (\alpha \rightarrow \alpha) \rightarrow \alpha)\]

This is a typed version of Turing's fixpoint combinator for the untyped lambda calculus, which we can get simply by erasing all the type information and the operations fold_ and unfold_.
It is easy to verify that $\lambda_{\text{Y}_{\text{Uring}}}f$ is indeed a fixpoint-operator:

\[
\begin{align*}
\lambda_{\text{Y}_{\text{Uring}}} f &\downarrow_{\delta} \Rightarrow \tau = \text{true} \quad \text{in } w = \text{true} \quad \text{in } (\text{fold } w) f \\
\lambda_{\text{Y}_{\text{Uring}}} f &\downarrow_{\in} \Rightarrow \tau = \text{true} \quad \text{in } w = \text{true} \quad \text{in } (\lambda y. f f ((\text{unfold } y) y f)) (\text{fold } w) f \\
\lambda_{\text{Y}_{\text{Uring}}} f &\downarrow_{\mu} \Rightarrow \tau = \text{true} \quad \text{in } w = \text{true} \quad \text{in } f ((\text{unfold } (\text{fold } w)) (\text{fold } w) f) \\
\lambda_{\text{Y}_{\text{Uring}}} f &\downarrow_{\nu} \Rightarrow \tau = \text{true} \quad \text{in } w = \text{true} \quad \text{in } f (w (\text{fold } w) f)
\end{align*}
\]

\[= f (\lambda_{\text{Y}_{\text{Uring}}} f)
\]

\(\square\)

In contrast to the programming languages introduced earlier, all datatypes are now inhabited, as is shown below. In $\lambda_{\text{nat}}^+$ and $\lambda_{\text{nat}}$ there are no programs inhabiting the datatype $(\Pi \alpha \cdot \alpha$, $\alpha)$, but in $\lambda_{\text{nat}}^+$ there are, as the following definition demonstrates.

5.3 Definition. The context $\Gamma_{\downarrow}$ is

\[
\downarrow = \lambda \alpha. \lambda (\alpha \cdot \alpha (\lambda x. x)) : (\Pi \alpha \cdot \alpha, \alpha)
\]

Consequently, any datatype $\sigma$ has at least one inhabitant, namely $\downarrow \sigma$. Given the reduction rule for $\lambda_{\text{Y}_{\text{Uring}}}$, it is easy to see that this program does not have a normal form. The interpretation of $\downarrow \sigma$ in the cpo-model given in chapter 8 will of course be the bottom element of the cpo that is the interpretation of $\sigma$

We will always assume that $\Gamma_{\downarrow}$ is part of the context. The type parameter of $\downarrow$ will be written as a subscript, i.e. we write $\downarrow_{\sigma}$ for $\downarrow \sigma$.

With the type constructor $\mu$ we can at last make "real" datatypes, such as a datatype for the natural numbers:

5.4 Example. The context $\Gamma_{\text{nat}}$ defining the datatype of natural numbers is

\[
\begin{align*}
\text{nat} &\equiv (\mu \alpha \cdot \alpha. \text{Unit} + \alpha) : \alpha \\
0 &\equiv \text{fold}_{\text{nat}} (\text{in}_{\text{nat} + \text{nat}} \cdot \text{Unit}) : \text{nat} \\
S &\equiv \lambda n. \lambda m. \text{fold}_{\text{nat}} (\text{in}_{\text{nat} + \text{nat}} \cdot 2 n) : \text{nat} \to \text{nat} \\
P &\equiv \lambda n. \lambda m. \lambda l. \text{fold}_{\text{nat}} (\text{in}_{\text{nat} + \text{nat}} \cdot \text{Unit} \cdot 0, \lambda m. \lambda n. \lambda l. (\text{unfold}_{\text{nat}} n) : \text{nat} \to \text{nat}
\end{align*}
\]

Of course, the price for having all recursive types and a fixpoint combinator is that this datatype $\text{nat}$ not only has inhabitants $S^n 0$ with $n \in \mathbb{N}$, but also an inhabitant $\downarrow_{\text{nat}}$

Properties of $\lambda_{\text{nat}}^+$ and $\lambda_{\text{nat}}$

The type inference rule ($\mu_{\text{form}}$) gives new datatypes, the other new type inference rules give new programs. The sets of pseudoterms Cons and Prog have to be extended accordingly:

5.5 Definition. The sets of pseudoterms Cons and Prog defined in definition 4.28 for $\lambda_{\text{nat}}^+$ are extended as follows

\[
\begin{align*}
\text{Cons} &\equiv \ldots | (\mu \text{Var} \nu \cdot \nu) \text{ Cons} \\
\text{Prog} &\equiv \ldots | \text{fold}_{\mu_{\text{nat}}} \cdot \sigma \text{ Prog} | \text{unfold}_{\mu_{\text{nat}}} \cdot \sigma \text{ Prog} | \text{Y}
\end{align*}
\]

\(\square\)
5.1. EXTENSION OF THE PROGRAMMING LANGUAGE

Then:

5.6 LEMMA (Classification for $\lambda \omega^\mu$).
1. If $\Gamma \vdash R : \mathcal{C}$, then $R \in \text{Kind}$.
2. If $\Gamma \vdash \sigma : \mathcal{C}$, then $\sigma \in \text{Cons}$.
3. If $\Gamma \vdash M : \sigma : \ast$, then $M \in \text{Prog}$.

PROOF Exactly the same way as lemmas 3.5 and 4.29.

With the exception of strong normalisation, the systems $\lambda \omega_\mu$ and $\lambda \omega^\mu$ inherit all the properties of $\lambda \omega^\Sigma$ and $\lambda \omega_\delta^\Sigma$. And even though $\beta_\mu$- and $\beta_\mu \delta$-reduction are not strongly normalising, strong normalisation for $\beta$- and $\beta \delta$-reduction is preserved.

Only programs can $\mu$-reduce, the kinds, datatypes and other datatype-constructors cannot. Because of this, types are not just unique up $\beta_\mu (\delta)$-conversion, but they are even unique up to $\beta(\delta)$-conversion.

5.7 THEOREM (Properties of $\lambda \omega^\mu$).

- CR$_{\beta\mu}$ : $\beta_\mu$-reduction is Church-Rosser
- SR$_{\beta\mu \delta}$ : if $\Gamma \vdash b : B$ and $b \triangleright_\beta \triangleright_\mu \triangleright_\delta b'$ then $\Gamma \vdash b' : B$.
- UT$_\beta$ : if $\Gamma \vdash b : B$ and $\Gamma \vdash b : B'$ then $B \simeq_\beta B'$.
- SN$_\beta$ : if $\Gamma \vdash b : B$ then $b$ and $B$ are $\beta$-strongly normalising.

PROOF As for PTSs with $+$, $\times$ and $\Sigma$ (Lemma 4.13), Church-Rosser can be proved using the theory of Combinatory Reduction Systems (CRSs). The relation $\triangleright_\beta \triangleright_\mu$ is not an orthogonal CRS, because it has so-called overlapping redices of the form

$$\text{fold}_{\mu \omega^\mu} \sigma \left( \underline{\text{unfold}_{\mu \omega^\mu} \sigma} \left( \text{fold}_{\mu \omega^\mu} \sigma M \right) \right).$$

This term contains two overlapping redices, indicated by the over- and underlining. However, contracting either redex result in the same term, namely $(\text{fold}_{\mu \omega^\mu} \sigma M)$, so $\triangleright_\beta \triangleright_\mu$ is a weakly orthogonal CRS. Confluence for weakly orthogonal CRSs is proved in [Raa92].

SN$_\beta$ for $\lambda \omega^\mu$ follows from SN$_\beta$ for $\lambda \omega^\Sigma$. All the terms that are typable in $\lambda \omega^\mu$ are - with a slightly different syntax for the $\mu$-types and with extra parameters for fold$_{\mu \omega^\mu} \sigma$ and unfold$_{\mu \omega^\mu} \sigma$ - also typable in $\lambda \omega^\Sigma$ in the following context.

\[
\begin{align*}
\mu & \quad (\ast_\mu \rightarrow \ast_\mu) \rightarrow \ast_\mu \\
\text{fold} & \quad \Pi F : \ast_\mu \rightarrow \ast_\mu \cdot F(\mu F) \rightarrow \mu F \\
\text{unfold} & \quad \Pi F : \ast_\mu \rightarrow \ast_\mu \cdot \mu F \rightarrow F(\mu F) \\
Y & \quad (\Pi \alpha : \ast_\mu \cdot (\alpha \rightarrow \alpha) \rightarrow \alpha)
\end{align*}
\]

It then follows from SN$_\beta$ for $\lambda \omega^\Sigma$ that all these terms are $\beta$-strongly normalising.

The properties SR and UT can be proved in the standard way, i.e. by the same inductions used in chapter 2 for PTSs (and in chapter 4 for PTSs with $+$, $\times$ and $\Sigma$).
5.8 **Theorem** (Properties of \( \lambda \omega^*_\delta \)).

- **CR\(_{\delta \mu \delta} \)**: \( \delta \mu \delta \)-reduction is Church-Rosser.
- **SR\(_{\delta \mu} \)**: if \( \Gamma \vdash b : B \) and \( b \gg \beta \mu \delta \delta' \) then \( \Gamma \vdash \delta' : B \).
- **UT\(_{\delta \delta} \)**: if \( \Gamma \vdash b : B \) and \( \Gamma \vdash b \cdot B' \) then \( B \simeq_{\mu \delta} B' \).
- **SN\(_{\delta \delta} \)**: if \( \Gamma \vdash b : B \) then \( b \) and \( B \) are \( \beta \delta \)-strongly normalising in \( \Gamma \).
- **Elimination of definitions**: if \( \Gamma \vdash a \cdot A \in \lambda \omega^*_\delta \) then \( \delta \text{nf}(\Gamma) \vdash \delta \text{nf}(a) \cdot \delta \text{nf}(A) \) in \( \lambda \omega^*_\delta \).

**Proof.** SN\(_{\delta \delta} \) can be proved in the same way as SN\(_{\delta} \) for \( \lambda \omega^*_\delta \) in the previous theorem. All the other properties can be proved in exactly the same way as in chapter 2 for DPTSs and in chapter 4 for DPTSs with \(+\), \( \times \) and \( \Sigma \). \( \square \)
5.2 Extension of the Programming Logic

In this section the programming logic \( \lambda \omega^\mu_L \) is defined, in which we can reason about the programming language \( \lambda \omega^\mu_L \).

As we mentioned earlier, domain theory is used as the basis for reasoning about \( \lambda \omega^\mu_L \). However, no notions or axioms from domain theory such as the cpo ordering \( \subseteq \) and its properties have to be included as primitives in the system \( \lambda \omega^\mu_L \). Instead, all the domain theory that is needed will be formalised in the system, i.e. we will give a context declaring and defining all the domain-theoretic notions and axioms that are needed. No new logical primitives other than the ones already offered by \( \lambda \omega^\mu_L \) are needed for this, so as the system \( \lambda \omega^\mu_L \) we can simply take the system \( \lambda \omega^\mu_L \) extended to \( \lambda \omega^\mu_L \).

5.9 Definition (\( \lambda \omega^\mu_{L(\delta)} \))
The programming logic \( \lambda \omega^\mu_{L(\delta)} \) is the extension of the system \( \lambda \omega^\mu_L \) with the pseudoterms, reduction and type inference rules given in definition 5.1.

5.10 Discussion (\( \beta(\delta) \)-conversion vs \( \mu \)-conversion of types).
In \( \lambda \omega^\mu_L \) there is an inference rule to \( \beta \)-convert types, namely the rule

\[
\frac{\Gamma \vdash b : B \quad \Gamma \vdash B' \colon s \quad B \equiv_\beta B'}{\Gamma \vdash b : B'}\]

which allows a type to be replaced by a \( \beta \)-convertible one. In \( \lambda \omega^\mu_{L(\delta)} \) we have the more powerful conversion rule (\( \beta\delta\text{conv} \)), which allows the \( \beta\delta \)-conversion of types. With this rule, the following inference rule is derivable.

\[
\frac{\Gamma \vdash b : B \quad \Gamma \vdash B' \colon s \quad \Gamma \vdash B \equiv_\delta B'}{\Gamma \vdash b : B'}\]

It is easy to see that, because of these rules, type checking involves testing convertibility of types. For example, to check if \( r : A \vdash x : A' \) is derivable in \( \lambda \omega^\mu_{L(\delta)} \) we have to check among other things - if \( A \) and \( A' \) are \( \beta(\delta) \)-convertible. Because \( \beta(\delta) \)-reduction is Church-Rosser and strongly normalising, \( \beta(\delta) \)-conversion is decidable.

The type inference rules of \( \lambda \omega^\mu_L \) do not include a \( \mu \)-conversion rule

\[
\frac{\Gamma \vdash b : B \quad \Gamma \vdash B' \colon s \quad B \equiv_\mu B'}{\Gamma \vdash b : B'}\]

which allows a type to be replaced by a \( \mu \)-convertible one. In the programming language \( \lambda \omega^\mu_L \) there is no use for this rule, because in \( \lambda \omega^\mu_L \) there are no types that can be \( \mu \)-reduced (only programs can be \( \mu \)-reduced). But in \( \lambda \omega^\mu_L \) there are types that \( \mu \)-reduce, because programs can occur as subexpressions of propositions. For example, if \( P : \sigma \rightarrow sp \) then \( P (\forall \sigma f) \) is a proposition that \( \mu \)-reduces to \( P (f (\forall \sigma f)) \). The rule (\( \mu\text{conv} \)) could be used to deduce \( \Gamma \vdash p : P (\forall \sigma f) \) from \( \Gamma \vdash p : P (f (\forall \sigma f)) \) and vice versa, and hence it could be used to prove \( (\forall \sigma f) =_\sigma f (\forall \sigma f) \).

The reason for excluding the rule (\( \mu\text{conv} \)) is that it would make type inference undecidable, because \( \mu \)-conversion is undecidable. To be precise, it would make type inference for proof terms undecidable, i.e. it would no longer be decidable if \( \Gamma \vdash p : P \) for proof terms \( p \in \text{Proof} \) and propositions \( P \in \text{PCons} \).
For the same reason, we do not identify a recursive type and its unfolding. This would make the operations fold$_{\mu\alpha\sigma}$ and unfold$_{\mu\alpha\sigma}$ redundant, but it would also make type inference undecidable, because $\beta$-conversion is then no longer decidable.

The rules ($\beta$conv) and ($\delta$conv) can be used to prove that all $\beta\delta$-convertible programs are Leibniz equal (see page 41, where the notion of Leibniz equality of programs is introduced). Because we do not have the rule ($\mu$conv), we cannot prove Leibniz' equality of $\mu$-convertible programs in the same way. The properties fold$_{\mu\alpha\sigma}$, $\sigma$ (unfold$_{\mu\alpha\sigma}$, $\sigma$ $M$) = $\mu\alpha\sigma$, $\sigma$ $M$, unfold$_{\mu\alpha\sigma}$, $\sigma$ (fold$_{\mu\alpha\sigma}$, $\sigma$ $N$) = $\sigma$[$\alpha$ = (fold$_{\mu\alpha\sigma}$, $\sigma$) $N$], and $(Y\sigma f)\sigma$ = $f(Y\sigma f)$ will have to be introduced as axioms (see definition 5.17).

So, unlike $\beta$- and $\delta$-conversions, any $\mu$-conversions that are done in a proof leave a trace in the proof term that represents that proof. Proof terms for the properties mentioned above effectively record all the $\mu$-conversions that are done in a proof.

Properties of $\lambda_\omega_L^\mu$ and $\lambda_\omega_L^{\mu\delta}$

In the system $\lambda_\omega_L^\mu$ we have the same inference rules for prop-kinds, prop-constructors and proofs as in $\lambda_\omega_L^\mu$. This means that the context-free syntax for these levels remains unchanged, i.e. for $\lambda_\omega_L^\mu$ the sets Pkind, PCons and Proof can be defined by the same context-free grammar given in definition 4.34 for $\lambda_\omega_L^\mu$.

The systems $\lambda_\omega_L^\mu$ and $\lambda_\omega_L^{\mu\delta}$ inherit all the properties of $\lambda_\omega_L^\mu$ and $\lambda_\omega_L^{\mu\delta}$. The Church-Rosser properties for $\beta\mu$ and $\beta\mu\delta$ which do not depend on the typing relation have already been proved for $\lambda_\omega_L^\mu$ and $\lambda_\omega_L^{\mu\delta}$ in the previous section. Strong normalisation of $\beta(\delta)$-reduction for $\lambda_\omega_L^{\mu\delta(\delta)}$ can be proved in the same way as for $\lambda_\omega_L^{\mu(\delta)}$. The remaining properties can all be proved by the same inductions used to prove these properties for the previous systems.

5.11 Theorem (Properties of $\lambda_\omega_L^\mu$).

- $\text{CR}_{\beta\mu} \cdot \beta\mu$-reduction is Church-Rosser.
- $\text{SR}_{\beta\mu} \cdot \text{if } \Gamma \vdash b : B \text{ and } b \equiv_{\beta\mu} b' \text{ then } \Gamma \vdash b' : B$.
- $\text{UT}_{\delta} \cdot \text{if } \Gamma \vdash b : B \text{ and } \Gamma \vdash b : B' \text{ then } B \equiv_{\beta\delta} B'$.
- $\text{SN}_{\beta\delta} \cdot \text{if } \Gamma \vdash b : B \text{ then } b \text{ and } B \text{ are } \beta\delta\text{-strongly normalising}$.
- Conservation of $\lambda_\omega_L^\mu$ over $\lambda_\omega_L^{\mu\delta}$.
  - if $\Gamma \vdash a : A$ in $\lambda_\omega_L^\mu$ and $a \in \text{Prog} \cup \text{Cons} \cup \text{Kind}$ then $\Gamma^{\text{data}} \vdash a : A$ in $\lambda_\omega_L^{\mu\delta}$.

5.12 Theorem (Properties of $\lambda_\omega_L^{\mu\delta}$).

- $\text{CR}_{\beta\mu\delta} \cdot \beta\mu\delta$-reduction is Church-Rosser.
- $\text{SR}_{\beta\mu\delta} \cdot \text{if } \Gamma \vdash b : B \text{ and } b \equiv_{\beta\mu\delta} b' \text{ then } \Gamma \vdash b' : B$.
- $\text{UT}_{\beta\delta} \cdot \text{if } \Gamma \vdash b : B \text{ and } \Gamma \vdash b : B' \text{ then } B \equiv_{\beta\delta} B'$.
- $\text{SN}_{\beta\delta} \cdot \text{if } \Gamma \vdash b : B \text{ then } b \text{ and } B \text{ are } \beta\delta\text{-strongly normalising}$.
- Elimination of definitions:
  - if $\Gamma \vdash a : A$ in $\lambda_\omega_L^{\mu\delta}$ then $\delta\text{nf}(\Gamma) \vdash \delta\text{nf}(a) : \delta\text{nf}(A)$ in $\lambda_\omega_L^{\mu\delta}$.
Formalisation of Domain Theory in $\lambda\omega^L$

To reason about recursive programs in $\lambda\omega^L$, a context $\Gamma_{CPO}$ will be defined, which declares the cpo ordering $\sqsubseteq$ and all the properties of $\sqsubseteq$ and of $Y$. In this context Scott's rule for computational induction - also known as fixpoint induction - will be provable. The model given in chapter 8 gives the justification for the axioms in $\Gamma_{CPO}$.

First we list the notions from the theory of cpos that have to be formalised.

5.13 Definition (chain, cpo, monotonic, continuous).
Let $A$ be a set, partially ordered by $\sqsubseteq_A$, i.e. $\sqsubseteq_A$ is reflexive, transitive and antisymmetric.

1. a subset $C$ of $A$ is called a chain if it is not empty and if for all $x, y \in C$, $x \sqsubseteq_A y$ or $y \sqsubseteq_A x$.

2. $A$ is called a cpo if every chain $C$ in $A$ has a least upper bound - or limit - $\bigcup C$, and it has a least element $\bot_A$ (which is the least upper bound of the empty set in $A$).

Suppose the sets $A$ and $B$, partially ordered by $\sqsubseteq_A$ and $\sqsubseteq_B$, respectively, are cpos.

3. a function $f$ from $A$ to $B$ is called monotonic if for all $x, y \in A$, if $x \sqsubseteq_A y$ then $f(x) \sqsubseteq_B f(y)$.

4. a function $f$ from $A$ to $B$ is called continuous if $f(\bigcup C) = \bigcup \{f(x) \mid x \in C\}$ for every chain $C$ in $A$.

If $A$ be a cpo and $f$ a continuous function from $A$ to $A$, then $f$ has a least fixpoint $Y(f)$. This means that $Y(f) = f(Y(f))$, and if $f(x) = x$ then $Y(f) \sqsubseteq_A x$. This least fixpoint of $Y(f)$ is the least upper bound of $\{\bot_A, f(\bot_A), f^2(\bot_A), \ldots\}$. The least fixpoint of $f$ is also the least prefixpoint of $f$, so if $f(x) \sqsubseteq_A x$ then $Y(f) \sqsubseteq_A x$.

Scott’s fixpoint induction rule - given in lemma 5.15 below - provides a way to reason about least fixpoints. It requires the following definition.

5.14 Definition A predicate $P$ on a set $A$ is called admissible if for every chain $C$ in $A$ for which all elements satisfy $P$ the limit of $C$ also satisfies $P$.

5.15 Lemma (Fixpoint Induction). Let $A$ be a cpo, $f$ a continuous function from $A$ to $A$, and $P$ an admissible predicate on $A$. If

- $P(\bot_A)$
- for all $x \in A$, if $P(x)$ then $P(f(x))$

then $P(Y(f))$.

A $\lambda\omega^L$-context $\Gamma_{CPO}$ will be defined, which declares an ordering $\sqsubseteq$ on datatypes, and axioms for $\sqsubseteq$ and $Y$, from which it follows that all datatypes are cpos, that all programs of some type $\sigma \rightarrow \tau$ are continuous (and hence monotonic), and that $Y$ produces least fixed points. Before this context is defined we first discuss the ingredients one by one.
The most important notion needed is the cpo-ordering \( \sqsubseteq \) on programs. Like Leibniz' equality of programs, introduced in definition 3.21, this is a polymorphic predicate of type (\( \Pi x : \sigma \), \( \alpha \to \alpha \to \ast_p \)), i.e.

\[
\sqsubseteq : \Pi x : \sigma . \alpha \to \alpha \to \ast_p .
\]

For \( \sqsubseteq \) the same notational conventions are used as for \( =_\sigma \), i.e. \( \sqsubseteq \) is written infix with its first argument as a subscript. So for instance we write \( (M \sqsubseteq_\sigma N) \) for \( (\sqsubseteq \sigma \ M \sqsubseteq N) \).

It is easy to give axioms stating that \( \sqsubseteq \) is reflexive, transitive and antisymmetric, and that all functions are monotonic:

\[
\begin{align*}
\text{refl} & : \forall \alpha : \ast, \forall x : \alpha \ . \ x \sqsubseteq \alpha x \\
\text{trans} & : \forall \alpha : \ast, \forall x, y, z : \alpha \ . \ x \sqsubseteq \alpha y \Rightarrow y \sqsubseteq \alpha z \Rightarrow x \sqsubseteq \alpha z \\
\text{antisym} & : \forall \alpha : \ast, \forall x, y : \alpha \ . \ x \sqsubseteq \alpha y \Rightarrow y \sqsubseteq \alpha x \Rightarrow x = \alpha y \\
\text{monotonicity} & : \forall \alpha, \beta : \ast, \forall f : \alpha \to \beta \ . \ \forall x, y : \alpha \ . \ x \sqsubseteq \alpha y \Rightarrow f x \sqsubseteq \beta f y
\end{align*}
\]

We also need some properties of the fixpoint-operator \( Y \) provided by the programming language, namely that \( (Y \sigma f) \) is the least fixed-point of \( f \) for all \( \sigma \to \sigma \). The axioms below state that \( (Y \sigma f) \) is a fixpoint of \( f \) and that it is smaller than any other prefix-point:

\[
\begin{align*}
\text{fix} & : \forall \alpha : \ast, \forall f : \alpha \to \alpha \ . \ Y \alpha f = _\alpha f (Y \alpha f) \\
\text{leastfix} & : \forall \alpha : \ast, \forall f : \alpha \to \alpha \ . \ \forall \alpha : \ast \ . \ f \sqsubseteq \alpha \Rightarrow Y \alpha f \sqsubseteq \alpha \tau
\end{align*}
\]

It immediately follows from the axiom \( \text{leastfix} \) that \( (\forall \alpha \sigma, \forall \tau : \alpha \sqsubseteq \alpha x) \), with \( \bot \) as defined in definition 5.3. So every datatype \( \alpha \) has a least element, namely the program \( \bot_\alpha \).

We still need the property that all chains have least upper bounds, and that all functions are continuous. Expressing these properties in \( \lambda \omega^\alpha \) requires a bit more work.

First of all, the notion of (least) upper bound of a subset \( \{ M_1, \ldots, M_n \} \) of a datatype \( \sigma \) is needed. Predicates on datatypes will be used to characterise such subsets. The following terms \( \text{UB} \) and \( \text{LUB} \) express the notions of upper bound and least upper bound of a predicate on a datatype, respectively:

\[
\begin{align*}
\text{UB} & = \lambda \alpha : \ast, \lambda P : \sigma \to \ast_p . \ \lambda x : \alpha \ . \ \forall y : \alpha \ . \ P y \Rightarrow y \sqsubseteq \alpha x \\
\text{LUB} & = \lambda \alpha : \ast, \lambda P : \sigma \to \ast_p . \ \lambda x : \alpha \ . \ \left( \forall y : \alpha \ . \ UB \alpha P y \right) \land \left( \forall y : \alpha \ . \ UB \alpha P y \Rightarrow x \sqsubseteq \alpha y \right)
\end{align*}
\]

If \( P : \sigma \to \ast_p \), then the proposition \( UB \sigma P x \) states that \( x \) is an upper bound of the set characterised by \( P \) and \( (LUB \sigma P x) \) states that \( x \) is a least upper bound of or limit of this set.

The following term defines what is means for a predicate on a datatype to be a chain:

\[
\begin{align*}
\text{chain} & = \lambda \alpha : \ast, \lambda C : \sigma \to \ast_p \ . \ \lambda x : \alpha \ . \ \left( \exists x, y : \alpha . \ C x \land C y \Rightarrow x \sqsubseteq \alpha y \lor y \sqsubseteq \alpha x \right) \\
& = \Pi \alpha : \ast, \left( \ast_p \to \ast_p \right)
\end{align*}
\]

If \( C \) is predicate on a datatype \( \alpha \), then \( (\text{chain } \alpha C) \cdot \ast_p \) is the proposition "\( C \) is a chain."

Using predicates to characterise sets, and using \( LUB \) and \( \text{chain} \) defined above, we then get the following axiom for continuity of all functions:

\[
\begin{align*}
\text{continuity} & : \forall \sigma, \tau : \ast_p, \forall g : \sigma \to \tau . \ \forall C : \sigma \to \ast_p . \ \forall \text{ lub } \sigma . \ \forall \text{ lub } P \ . \\
& \ C = \lambda y : \tau . \ \exists x \sigma . \ g x = \ast y \land C x \Rightarrow \tau \to \ast_p \ \text{in} \\
& \ (\text{chain } \sigma C) \Rightarrow (LUB \sigma C \text{ lub}) \Rightarrow (LUB \tau C' \text{ lub'}) \Rightarrow g \text{ lub } \ast, \text{ lub'}
\end{align*}
\]
5.2. EXTENSION OF THE PROGRAMMING LOGIC

Here $C': x \to \ast_p$ is the image of $C$ under $g$.

We can now give the last property of $\subseteq$, namely that all chains have a least upper bound:

\[
\text{complete} : \forall \alpha \ast_s, \forall C \alpha \to \ast_s, \text{chain } \alpha C \Rightarrow (\exists \text{lub } \alpha. \text{lub } \alpha C \text{lub})
\]

Recall that from leastfix, it followed that all datatypes have a least element, so, from leastfix and complete together it follows that all datatypes are cpos.

Note that we cannot hope to define - or even to declare - a function

\[
\bigcup : (\Pi \alpha \ast_s. (\alpha \to \ast_p) \to \alpha)
\]

that maps predicates to their limits, because this type cannot be formed in $\lambda \omega_{L,L}$. A term $(\bigcup \sigma P)$ would be an inhabitant of the datatype $\sigma$ depending on a predicate $P$. So it is a term in the programming language depending on a term in the logic. This is exactly one of the dependencies we have chosen to exclude in $\lambda \omega_L$ and its extensions.

A set $P$ is called admissible - or chain-complete - if any chain in $P$ has a limit in $P$. A predicate $P : \alpha \to \ast_p$ is called admissible if the set it characterises is, i.e. if for any chain $C$ for which all elements satisfy $P$ the limit also satisfies $P$. So we define

\[
\text{admissible} = \lambda \alpha \ast_s. \lambda P \alpha \to \ast_p \\
\forall C \alpha \to \ast_p \forall \text{lub } \alpha. \\
(\text{chain } \alpha C) \Rightarrow (\text{lub } \alpha C \text{lub}) \Rightarrow (\forall x. C x \Rightarrow P x) \Rightarrow (P \text{lub})
\]

The fact the notion of admissibility can be expressed and that admissibility of programs can be proved in the system means that the logic we have is stronger than LCF ([Pau87]). There the condition of the admissibility of $P$ in fixpoint induction is strengthened to a syntactic check on the shape of $P$. Even though the notion of admissibility is definable, such a syntactic check can still be useful, because proving admissibility of a predicate can be quite complicated.

All declarations and definitions in $\Gamma_{CPO}$ have now been discussed, with the exception of the (self-explanatory) definition of strict

5.16 DEFINITION. $\Gamma_{CPO}$ is the $\lambda \omega_{L,L}$-context

\[
\begin{align*}
\subseteq & : \Pi \alpha \ast_s. \alpha \to \alpha \to \ast_p \\
\text{refl} & : \forall \alpha \ast_s, \forall x. x \subseteq_\alpha x \\
\text{trans} & : \forall \alpha \ast_s, \forall x, y, z : \alpha. x \subseteq_\alpha y \Rightarrow y \subseteq_\alpha z \Rightarrow x \subseteq_\alpha z \\
\text{antisym} & : \forall \alpha \ast_s. \forall x, y : \alpha. x \subseteq_\alpha y \Rightarrow y \subseteq_\alpha x \Rightarrow x =_\alpha y \\
\text{monotonicity} & : \forall \alpha, \beta : \ast_s. \forall f. \alpha \to \beta. \forall x, y : \alpha. x \subseteq_\alpha y \Rightarrow f x \subseteq_\beta f y \\
\text{fix} & : \forall \alpha \ast_s. \forall f. \alpha \to \alpha. \forall \alpha f =_\alpha f (Y \alpha f) \\
\text{leastfix} & : \forall \alpha \ast_s. \forall f. \alpha \to \alpha. \forall x : \alpha. f x \subseteq_\alpha x \Rightarrow Y \alpha f \subseteq_\alpha x \\
\text{UB} & : \lambda \alpha \ast_s. \lambda P \alpha \to \ast_p. \lambda x : \alpha. \forall y. P y \Rightarrow y \subseteq_\alpha x \\
& : \Pi \alpha \ast_s. (\alpha \to \ast_p) \to \alpha \to \ast_p
\end{align*}
\]
CHAPTER 5. RECURSION

\[ LUB = \lambda \alpha \cdot \lambda x. (UB \alpha P x) \wedge (\forall \alpha. UB \alpha P y \Rightarrow \tau \subseteq \alpha y) \]
\[ : \Pi \alpha \cdot (\alpha \rightarrow \ast_p) \rightarrow \alpha \rightarrow \ast_p \]
\[ chain = \lambda \alpha : \ast_p. \lambda C : \alpha \ast_p \ast_s. \]
\[ (\exists x : \alpha. C x) \wedge (\forall \alpha, \exists y. C x \wedge C y \Rightarrow x \subseteq \alpha y \vee y \subseteq \alpha x) \]
\[ : \Pi \alpha : \ast_p. (\alpha \rightarrow \ast_p) \rightarrow \ast_p \]
\[ continuity = \forall \sigma, \tau : \ast_p. \forall g : \tau \rightarrow \tau. \forall C : \sigma \rightarrow \ast_p. \forall \text{lub} : \sigma. \forall \text{lub}' : \tau. \]
\[ C' = (\lambda y : \tau. \exists x : \sigma. g x = \tau, y \wedge C x) : \tau \rightarrow \ast_p \in \text{in} \]
\[ (chain \sigma C) \Rightarrow (LUB \sigma C \text{lub}) \Rightarrow (LUB \tau C' \text{lub'}) \Rightarrow g \text{lub} = \tau, \text{lub}' \]
\[ complete = \forall \alpha : \ast_p. \forall C : \alpha \rightarrow \ast_s. \text{chain} \alpha C \Rightarrow (\exists \text{lub} : LUB \alpha C \text{lub}) \]
\[ admissible = \lambda \alpha : \ast_p. \lambda P : \alpha \rightarrow \ast_p. \]
\[ \forall C : \alpha \rightarrow \ast_p. \forall \text{lub} : \alpha \]
\[ (chain \alpha C) \Rightarrow (LUB \alpha C \text{lub}) \Rightarrow (LUB \tau C \text{lub}) \Rightarrow (P \text{lub}) \]
\[ : \Pi \alpha : \ast_p. (\alpha \rightarrow \ast_p) \rightarrow \ast_p \]
\[ strict = \lambda \sigma, \tau : \ast_p. \lambda f : \sigma \rightarrow \tau. \forall f. \tau \Rightarrow f \Downarrow \tau \Rightarrow \Downarrow \tau \]
\[ : \Pi \sigma, \tau : \ast, (\sigma \rightarrow \tau) \rightarrow \ast_p \]

\[ \square \]

More axioms are needed than just the ones in \( \Gamma_{\propto} \). We want to use axioms similar to the ones in \( A\text{XION}^+ \) (definition 4.39), e.g., axioms for classical logic, axioms for extensionality of \( \rightarrow \) and \( \Pi \)-types, and axioms for structural induction on \( +, \times, \text{and } \Sigma \)-types. Below a set of axioms \( A\text{XION}^\mu \) is defined that can be safely used in \( \lambda \omega_{\mu}^\text{\(\delta\)} \). That these axioms are sound is shown in chapter 8, using the cpo-model.

Compared with the axioms in \( A\text{XION}^+ \) there are a few differences. First of all, we want axioms relating the cpo-ordering on the different datatypes. For the \( \rightarrow, \Pi \)-, and \( \times \)-datatypes we want to be able to prove

\[ f \subseteq_\sigma g \iff (\forall x : \sigma. f x \subseteq_\tau g x) \quad (i) \]
\[ f \subseteq_\Pi \text{K} g \iff (\forall \alpha : \text{K}. f \alpha \subseteq_\sigma g \alpha) \quad (ii) \]
\[ M \subseteq_{\Pi l_1, l_2, \ldots} N \iff N l_1 \subseteq_{\sigma_1} M l_1 \land \ldots \land N l_n \subseteq_{\sigma_n} M l_n \quad (iii) \]

All the implications from left to right can be proved using monotonicity. For example, if \( f \subseteq_{\Pi \text{K}} g \alpha \), then by monotonicity

\[ (\lambda x : (\Pi \alpha \cdot (\Pi \text{K} \cdot \sigma). \tau \alpha) f) \subseteq_\sigma (\lambda x : (\Pi \alpha \cdot (\Pi \text{K} \cdot \sigma). \tau \alpha) g) \]

and hence \( f \alpha \subseteq_\sigma g \alpha \).

For (i) and (ii), the implications from right to left are included as axioms in \( A\text{XION}^\mu \). They can replace the axioms for extensionality of \( \rightarrow \) and \( \Pi \)-types that were included in \( A\text{XION} \) and \( A\text{XION}^+, \) because these follow from (i) and (ii) by the antisymmetry of \( \subseteq \).

For (iii), the implication from right to left does not have to be included as an axiom, because it follows from monotonicity of the function \( \text{tuple } \sigma_1 \rightarrow \rightarrow \sigma_n \rightarrow \Pi(l_1, \sigma_1 \ldots l_n \sigma_n) \) with \( \text{tuple } N_1 \{ N_a \triangleright \beta \ (l_1 \mapsto N_1, \ldots, l_n \mapsto N_n) \)
5.2. EXTENSION OF THE PROGRAMMING LOGIC

No properties of the ordering $\sqsubseteq$ on $+$-types and $\Sigma$-types have to be included as axioms. For instance, we can already prove that

$$N \sqsubseteq_{\sigma_1} N' \iff \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)}$$

The implication from left to right follows from the fact that $(\lambda x: \Sigma_1. \Sigma_1(\sigma_1, l_\sigma_1)) \cdot l_1 x$ is monotonic. The implication from right to left can be proved by defining a program $f$ of type $\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n) \rightarrow \sigma_i$ such that $f((\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n))(l_1 x)) \gg_{g_2} x$, and using the fact that this function $f$ is monotonic.

We can also prove, for any $l_i \neq l_j$, that

$$\Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)}$$

To do this, define functions $f, g : \Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n) \rightarrow \text{bool}$ such that $f((l_i x) \gg g false, g((l_j x) \gg g false, and $f((l_j x) \gg g true$. Then by monotonicity of $f$ and $g$ and antisymmetry of $\sqsubseteq$ it follows that

$$\Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)} \cdot \Sigma_{\ell_1(\sigma_1, l_\sigma_1)}$$

The axioms $AX_{\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)}$ and $AX_{\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)}$ in $AXIOM^+$ for structural induction on $+$- and $\Sigma$-types have to change. A type $\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)$ not only contains elements of the form $\Sigma(l_1, \sigma_1 \ldots l_n \cdot \sigma_n)$, but it now also contains a bottom element $\bot_{\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)}$. For example, for the datatypes $(\Sigma(1, 2, 3, 4))$, i.e. $\sigma_1 + \sigma_2$, we now get

$$AX_{\sigma_1 + \sigma_2} : \forall P \cdot \sigma_1 + \sigma_2 \rightarrow \ast_p \cdot (\forall x_1 : \sigma_1. P((\Sigma(\sigma_1 + \sigma_2))(l_1 x_1))) \Rightarrow (\forall x_2 : \sigma_2. P((\Sigma(\sigma_1 + \sigma_2))(l_2 x_2))) \Rightarrow P \bot_{\sigma_1 + \sigma_2} \Rightarrow (\forall x : \sigma_1 + \sigma_2. P x)$$

Datatypes $(\Sigma(\alpha, K, \sigma \ldots l_n \cdot \sigma_n))$ also contain bottom elements, so for these the axioms $AX_{\Sigma(\alpha, K, \sigma \ldots l_n \cdot \sigma_n)}$ are changed in the same way.

For $+$- or $\Sigma$-types we also have to include axioms that state that all functions constructed using the elimination construction $\triangledown$ are strict.

Finally, for the $\mu$-types we have to include axioms $AX_{\mu \sigma}$ that state that fold$\mu \sigma$ and unfold$\mu \sigma$ are isomorphisms.

5.17 DEFINITION $AXIOM^\mu$ is the set of axioms containing

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>classic</td>
<td>$\forall P : \ast, (\neg P) = P$</td>
</tr>
<tr>
<td>$\text{prop}$</td>
<td>true $\neq \text{false}$</td>
</tr>
<tr>
<td>$AX_{\tau}$</td>
<td>$\forall f : \tau \rightarrow \tau$ (\forall \tau : \tau \rightarrow \tau g \cdot \tau) $\Rightarrow f \sqsubseteq \tau g$</td>
</tr>
<tr>
<td>$AX_{\Pi(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)}$</td>
<td>$\forall P : \Pi(l_1, \sigma_1 \ldots l_n \cdot \sigma_n) \rightarrow \ast_p$</td>
</tr>
<tr>
<td>$AX_{\Sigma(\ell_1, \sigma_1 \ldots l_n \cdot \sigma_n)}$</td>
<td>$\forall P : \Sigma(l_1, \sigma_1 \ldots l_n \cdot \sigma_n) \rightarrow \ast_p$</td>
</tr>
</tbody>
</table>
98  

CHAPTER 5. RECURSION  

\[ \Rightarrow (\forall x_n. \sigma_n. P(\Sigma(l_1 \sigma_1 \ldots l_n \sigma_n), l_n x_n)) \]
\[ \Rightarrow P \perp \Sigma(l_1 \sigma_1 \ldots l_n \sigma_n) \]
\[ \Rightarrow (\forall z: \Sigma(l_1 \sigma_1 \ldots l_n \sigma_n). P z) . \]

**AX\SigmaK \sigma** : \( \forall P. (\Sigma \alpha \cdot K \cdot \sigma) \rightarrow ^* P .\)
\( (\forall \alpha : \cdot K. \forall x. \sigma P(\Sigma \alpha \cdot K \cdot \sigma \cdot x)) \)
\[ \Rightarrow P \perp \Sigma \alpha \cdot K \cdot \sigma \]
\[ \Rightarrow (\forall z: (\Sigma \alpha \cdot K \cdot \sigma). P z) . \)

**AX\muax, \sigma** : \( (\forall z (\mu \alpha^* \cdot \sigma). \text{fold}_{\mu \alpha^*} \cdot \sigma (\text{unfold}_{\mu \alpha^*} \cdot \sigma z) = \mu \alpha^* \cdot \sigma z) \land \)
\( (\forall z. \sigma[\alpha := (\mu \alpha^* \cdot \sigma)]. \text{unfold}_{\mu \alpha^*} \cdot \sigma (\text{fold}_{\mu \alpha^*} \cdot \sigma z) = \sigma[\alpha = (\mu \alpha^* \cdot \sigma)] z) \)

**STRICT \Sigma(l_1 \sigma_1 \ldots l_n \sigma_n)** : \( \forall \rho^* \cdot \forall f. (l_1 \sigma_1 \rightarrow \rho, \ldots, l_n \sigma_n \rightarrow \rho). \nabla f \perp \Sigma(l_1 \sigma_1 \ldots l_n \sigma_n) = \rho \perp \rho \)

**STRICT \SigmaK \sigma** : \( \forall \rho^* \cdot \forall f. (\Sigma \alpha \cdot K \cdot \sigma \rightarrow \rho). \nabla f \perp \Sigma \alpha \cdot K \cdot \sigma = \rho \perp \rho \)

for all \( \sigma \rightarrow \tau, (\Sigma \alpha \cdot K \cdot \sigma), \Pi(l_1 \sigma_1 \ldots l_n \sigma_n), \Sigma(l_1 \sigma_1 \ldots l_n \sigma_n), (\Sigma \alpha \cdot K \cdot \sigma), (\mu \alpha^* \cdot \sigma) \in \text{Cons} . \)

\( \square \)

Of course, we want to know if the axioms in \( \Gamma_{CPO} \) and \( AXIOM^\mu \) are consistent. This can be proved using the model given in chapter 8.

5.18 **LEMMA** (Consistency of \( \Gamma_{CPO} \) and \( AXIOM^\mu \) in \( \lambda \omega_L \)).
False is not provable in the context \( \Gamma_{CPO} \) extended with axioms from \( AXIOM^\mu \).

**PROOF** See lemma 8.33.  \( \square \)
5.3. PROGRAM AND PROOF DEVELOPMENT

5.3 Program and Proof Development

As in the previous programming logics, in $\lambda \omega^k$ we can prove propositions that can be used to deduce a property of a program from certain properties of its subprograms, and we can prove propositions that state equalities between programs, which can be used as transformation rules.

First we consider the problem of proving properties of recursive programs, i.e. programs constructed with the fixpoint combinator $Y$. The following lemma is crucial for this:

5.19 Lemma (Fixpoint Induction).

In the context $\Gamma_{CPO}$ the following proposition is provable:

$$\forall \sigma. \forall f : \sigma \rightarrow \sigma. \forall p : \sigma \rightarrow *p. \text{admissible } \sigma P \Rightarrow \sigma \triangleleft_{\sigma} \Rightarrow (\forall x : \sigma. P x \Rightarrow P (f x)) \Rightarrow P (Y \sigma f)$$

Proof. Let $\Gamma \vdash \sigma : *p$, $\Gamma \vdash P : \sigma \rightarrow *p$, and $\Gamma \vdash f : \sigma \rightarrow \sigma$, and suppose that $\Gamma \vdash _{\cdot} (\text{admissible } \sigma P), \Gamma \vdash _{\cdot} P \triangleleft_{\sigma}$ and $\Gamma \vdash _{\cdot} (\forall x : \sigma. P x \Rightarrow P (f x))$. To prove: $P (Y \sigma f)$.

The plan of the proof is as follows: If $P \triangleleft_{\sigma}$ and $(\forall x : \sigma. P x \Rightarrow P (f x))$, then all elements in the chain

$$\sigma \subseteq \sigma \triangleleft_{\sigma} \subseteq \sigma \triangleleft_{\sigma} \subseteq \sigma \triangleleft_{\sigma} \subseteq \ldots$$

satisfy $P$. Since $P$ is admissible it then follows that the least upper bound of this chain also satisfies $P$. So to prove $P (Y \sigma f)$, we have to show that $\{ \sigma \triangleleft_{\sigma}, \sigma \triangleleft_{\sigma}, \sigma \triangleleft_{\sigma}, \ldots \}$ is indeed a chain and that $(Y \sigma f)$ is its least upper bound.

The predicate on $\sigma$ that characterises the chain $\{ \sigma \triangleleft_{\sigma}, \sigma \triangleleft_{\sigma}, \sigma \triangleleft_{\sigma}, \ldots \}$ is defined as follows:

$$C = \lambda x : \sigma. (\forall Q : \sigma. Q \sigma \Rightarrow (\forall y : \sigma. Q y \Rightarrow Q (f y))) \Rightarrow Q x$$

$$: \sigma \rightarrow *p$$

It is easy to verify that $C : \sigma \rightarrow *p$ is true in $\sigma$ and closed under $f$, and for any predicate $Q : \sigma \rightarrow *p$ that is also true in $\sigma$ and closed under $f$, we have $(\forall x : \sigma. C x \Rightarrow Q x)$. So $C : \sigma \rightarrow *p$ is the smallest predicate that is true in $\sigma$ and closed under $f$.

We have to prove that $C$ is a chain and that $(Y \sigma f)$ is the least upper bound of $C$. Proving of (chain $\sigma C$) is not difficult: define

$$Q = \lambda x : \sigma. (\forall z : \sigma. C z \Rightarrow (z \subseteq_{\sigma} x \vee x \subseteq_{\sigma} z))$$

$$: \sigma \rightarrow *p$$

and prove that this predicate is true in $\sigma$ and closed under $f$. A fairly detailed proof of $(LUB \sigma C (Y \sigma f))$ is given in figure 5.1 on page 100. It is essentially a conventional proof of the fact that the least fixpoint of a continuous function $f$ is the least upper bound of the chain $\sigma \subseteq f \sigma \subseteq f^2 \sigma \subseteq f^3 \sigma \subseteq \ldots$. The type parameter of $\subseteq$ is omitted in figure 5.1.

This lemma gives the following proof rule to accompany the type inference rule for $Y$:

5.20 Corollary (Coupled derivation rule for $Y$). In the context $\Gamma_{CPO}$ the following rules are derivable:

$$\frac{\Gamma \vdash \sigma \rightarrow *p}{\Gamma \vdash P : \sigma \rightarrow *p}$$

$$\frac{\Gamma \vdash \cdot \text{ admissible } \sigma P}{\Gamma \vdash \cdot \sigma \triangleleft_{\sigma}}$$

$$\frac{\Gamma \vdash f : \sigma \rightarrow \sigma}{\Gamma \vdash \cdot \sigma \triangleleft_{\sigma} \Rightarrow \Gamma \vdash _{\cdot} P \triangleleft_{\sigma} \Rightarrow \Gamma \vdash _{\cdot} (\forall x : \sigma. P x \Rightarrow P (f x))}$$

$$\frac{\Gamma \vdash \cdot \cdot P (Y \sigma f)}{\Gamma \vdash \cdot \cdot}$$


<table>
<thead>
<tr>
<th>No.</th>
<th>Statement</th>
<th>Proof References</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{chain } \sigma \ C )</td>
<td>easy</td>
</tr>
<tr>
<td>2</td>
<td>( \exists \text{lub}: \sigma. (\text{LUB } \sigma \ C \ \text{lub}) )</td>
<td>1, complete</td>
</tr>
<tr>
<td>3</td>
<td>( \text{lub} : \sigma, (\text{LUB } \sigma \ C \ \text{lub}) )</td>
<td>1, complete</td>
</tr>
<tr>
<td>4</td>
<td>( C' = (\lambda y. (\exists x. (C x \land f x = \sigma y)) : \sigma \rightarrow \ast_p )</td>
<td>easy</td>
</tr>
<tr>
<td>5</td>
<td>( \exists \text{lub}' : \sigma. (\text{LUB } \sigma \ C' \ \text{lub}') )</td>
<td>5, complete</td>
</tr>
<tr>
<td>6</td>
<td>( \text{lub}' = \sigma \ f \ \text{lub} )</td>
<td>3, 7, continuity, def ( C' )</td>
</tr>
<tr>
<td>7</td>
<td>( \text{LUB } \sigma \ C' (f \ \text{lub}) )</td>
<td>7, 8</td>
</tr>
<tr>
<td>8</td>
<td>( \forall x : \sigma. C' x \Rightarrow C x )</td>
<td>def ( C, C' )</td>
</tr>
<tr>
<td>9</td>
<td>( U B \ \sigma \ C' \ \text{lub} )</td>
<td>3, 11, def ( UB, LUB )</td>
</tr>
<tr>
<td>10</td>
<td>( f \ \text{lub} \subseteq \text{lub} )</td>
<td>10, 12, def ( LUB )</td>
</tr>
<tr>
<td>11</td>
<td>( Y \ \sigma f \subseteq \text{lub} )</td>
<td>13, least fix</td>
</tr>
<tr>
<td>12</td>
<td>( \perp_{\sigma} \subseteq Y \ \sigma f )</td>
<td>easy</td>
</tr>
<tr>
<td>13</td>
<td>( f x \subseteq f (Y \ \sigma f) )</td>
<td>16, monotonicity</td>
</tr>
<tr>
<td>14</td>
<td>( f x \subseteq Y \ \sigma f )</td>
<td>17, fix</td>
</tr>
<tr>
<td>15</td>
<td>( \forall x : \sigma. \tau \subseteq Y \ \sigma f \Rightarrow f \ \tau \subseteq Y \ \sigma f )</td>
<td>18, ( \forall \text{intro}, \Rightarrow \text{intro} )</td>
</tr>
<tr>
<td>16</td>
<td>( \forall x : \sigma. C \ \tau \Rightarrow f \ \tau \subseteq Y \ \sigma f )</td>
<td>15, 19, def ( C )</td>
</tr>
<tr>
<td>17</td>
<td>( U B \ \sigma \ C (Y \ \sigma f) )</td>
<td>20, def ( UB )</td>
</tr>
<tr>
<td>18</td>
<td>( \text{lub} \subseteq Y \ \sigma f )</td>
<td>21, 3, def ( LUB )</td>
</tr>
<tr>
<td>19</td>
<td>( \text{lub} = \sigma Y \ \sigma f )</td>
<td>14, 22, antisym</td>
</tr>
<tr>
<td>20</td>
<td>( LUB \ \sigma \ C (Y \ \sigma f) )</td>
<td>3, 23</td>
</tr>
<tr>
<td>21</td>
<td>( LUB \ \sigma \ C (Y \ \sigma f) )</td>
<td>2, 24, ( \forall \text{intro}, \exists \text{elim} )</td>
</tr>
</tbody>
</table>

Figure 5.1: proof of lemma 5.19
Lemma 5.19 and its corollary are of course useless if we cannot prove admissibility of predicates. Large classes of predicates can be shown to be admissible, see for instance [ZNV72] [Fau87]. To prove that a predicate is admissible, it is useful to prove admissibility for all predicates of a certain form, as in lemma 5.21 below, and to prove that a certain construction preserve admissibility, as in lemma 5.22 below. These two lemmas indicate a well-known class of admissible predicates.

5.21 Lemma. In context $\Gamma_{CP0}$ the following proposition is provable.

$$\forall \alpha, \beta \vdash_{s}, \forall g, h. \alpha \rightarrow \beta. \text{admissible } \alpha (\lambda x: \alpha. g x \subseteq_{\beta} h x)$$

Proof. Suppose $\alpha, \beta : \ast_{s}$ and $g, h : \alpha \rightarrow \beta$. Define $P = (\lambda x: \alpha. g x \subseteq_{\beta} h x) : \alpha \rightarrow \ast_{p}$. To prove admissible $\alpha P$ we have to prove

$$\forall C : \alpha \rightarrow \ast_{p}, \text{lub } \alpha. \text{ (chain } \alpha C) \Rightarrow (\text{lub } \alpha) C \Rightarrow (\forall x: \alpha. C x \Rightarrow P x) \Rightarrow (P \text{lub})$$

Suppose $C : \alpha \rightarrow \ast_{p}, \text{lub } \alpha$, (chain $\alpha C$), ($\text{lub } \alpha C \text{lub}$), and $(\forall x: \alpha. C x \Rightarrow P x)$. So $C$ is a chain with least upper bound $\text{lub}$ and all elements in $C$ satisfy $P$.

To prove: $(P \text{lub})$, i.e $g \text{lub} \subseteq h \text{lub}$.

Define predicates $C_{g}$ and $C_{h}$ as follows:

$$C_{g} = (\lambda y : \tau. \exists x : \sigma. P x \land y =_{\tau} g x) \cdot \tau \rightarrow \ast_{p}$$

$$C_{h} = (\lambda y : \tau. \exists x : \sigma. P x \land y =_{\tau} h x) \cdot \tau \rightarrow \ast_{p}$$

By continuity $(g \text{lub})$ is the limit of $C_{g}$, and $(h \text{lub})$ is the limit of $C_{h}$, i.e. $(\text{lub } \tau C_{g} (g \text{lub}))$ and $(\text{lub } \tau C_{h} (h \text{lub}))$. It follows from $(\forall x: \alpha. C x \Rightarrow P x)$ - which is $\delta$-convertible with $(\forall x: \alpha. C x \Rightarrow g x \subseteq_{\beta} h x)$ - than for every element of $C_{g}$ there is a larger element of $C_{h}$, i.e.

$$\forall y : \tau. C_{g} y \Rightarrow (\exists z : \tau. C_{h} z \land y \subseteq z)$$

Then $(h \text{lub})$ - the least upper bound of $C_{h}$ - is also an upper bound of $C_{g}$. But $(g \text{lub})$ is the least upper bound of $C_{g}$ and so $g \text{lub} \subseteq h \text{lub}$.

It is well-known that conjunction "preserves" admissibility.

5.22 Lemma. In context $\Gamma_{CP0}$ the following proposition is provable:

$$\forall \alpha \vdash_{s}, \forall P, Q. \alpha \rightarrow \ast_{s},$$

admissible $\alpha P \Rightarrow$ admissible $\alpha Q \Rightarrow$ admissible $\alpha (\lambda x: \alpha. P x \land Q x)$

Proof. Easy. 

Using the last two lemmas admissibility for a large class of predicates can be proved. For example, by the antisymmetry of $\subseteq$ it immediately follows from these lemmas that all predicates of the form $(\lambda x: \sigma. g x \Rightarrow h x) : \sigma \rightarrow \ast_{p}$ are admissible.

Fixpoint induction can not only be used to prove properties of individual recursive programs, but can also be used to prove many transformation rules for recursive programs. An example is given in the next lemma.
1. \( \sigma : \ast_0 \)
2. \( f, g : \sigma \rightarrow \sigma \)
3. \( f \perp = g \perp, f; g = g; f \)
4. admissible \( (\lambda x : \sigma \ x \in Y g) \)
5. \( \perp \subseteq Y g \)
6. \( x : \sigma, x \subseteq Y g \)
7. \( f x \subseteq f (Y g) \)
8. admissible \( (\lambda y : \sigma \ f y \subseteq Y g) \)
9. \( g \perp \subseteq Y g \)
10. \( f \perp \subseteq Y g \)
11. \( \bigvee y : \sigma, f y \subseteq Y g \)
12. \( g(f y) \subseteq g(Y g) \)
13. \( f(g y) \subseteq g(Y g) \)
14. \( f(g y) \subseteq Y g \)
15. \( \forall y : \sigma, f y \subseteq Y g \Rightarrow f(y y) \subseteq Y g \)
16. \( f(Y g) \subseteq Y g \)
17. \( f(r) \subseteq Y g \)
18. \( \forall x : \sigma, r \subseteq Y g \Rightarrow f(r) \subseteq Y g \)
19. \( Y f \subseteq Y g \)
20. \( \forall f, g : \sigma \rightarrow \sigma \ f \perp = g \perp \Rightarrow f; g = g; f \Rightarrow Y f \subseteq Y g \)
21. \( f, g : \sigma \rightarrow \sigma, f \perp = g \perp, f; g = g; f \Rightarrow Y f = Y g \)
22. \( Y f = Y g \)
23. \( \forall f, g : \sigma \rightarrow \sigma \ f \perp = g \perp \Rightarrow f; g = g; f \Rightarrow Y f = Y g \)

Figure 5.2 proof of lemma 5.23
5.3 PROGRAM AND PROOF DEVELOPMENT

5.23 LEMMA. In the context $\Gamma_{CPO}$ the following proposition is provable:

$$\forall \sigma, \varphi, g, \sigma \rightarrow f, f \perp g \Rightarrow \ Y \sigma f = \sigma \ Y g$$

PROOF. This proof, given in figure 5.2 on page 102, is still small enough to be given in full in natural deduction format. The type parameters of $\perp, Y, =, \in$ and admissible – which are nearly always $\sigma$ – are omitted in this proof.

In lemma 4.48 we gave an example of a transformation rule in $\lambda \omega L^\tau$, namely for the distribution of composition over $\nabla$. Because $+$-types now include bottom elements, in $\lambda \omega L^\mu$ this rule requires an extra condition:

5.24 LEMMA In the context $\Gamma_{CPO}$ the following proposition is provable using axioms from $AXIOM^\mu$:

$$\forall \rho_1, \rho_2, \tau, \sigma, x, \rho : \rho_1 \rightarrow \sigma, g : \rho_2 \rightarrow \sigma, h : \sigma \rightarrow \tau$$

$$\left(\text{strict } \sigma \rightarrow \tau\right) \Rightarrow h (f \nabla g) = \rho_1 + \rho_2 \rightarrow (h \sigma f) \nabla (h \sigma g)$$

PROOF

<table>
<thead>
<tr>
<th>1</th>
<th>$\rho_1, \rho_2, \tau, \sigma : \ast, f : \rho_1 \rightarrow \sigma, g : \rho_2 \rightarrow \sigma, h : \sigma \rightarrow \tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\text{strict } \sigma \rightarrow \tau$</td>
</tr>
<tr>
<td>3</td>
<td>$f_1$</td>
</tr>
<tr>
<td>4</td>
<td>$\left(\text{strict } \sigma \rightarrow \tau\right)$</td>
</tr>
<tr>
<td>5</td>
<td>$\forall x : \rho_1 \left(\left(\text{strict } \sigma \rightarrow \tau\right) \Rightarrow \left(h \sigma f \nabla g\right)(\in_1 + \rho_2, 1) = \tau \left(\left(h \sigma f \nabla g\right)(\in_1 + \rho_2, 1\right)</td>
</tr>
</tbody>
</table><p>ight)$ |
| 6 | $\forall y : \rho_2, \left(h \sigma f \nabla g\right)(\in_1 + \rho_2, 2) = \tau \left(\left(h \sigma f \nabla g\right)(\in_1 + \rho_2, 2)\right)$ |
| 7 | $\left(\left\langle\left(\text{strict } \sigma \rightarrow \tau\right)\right\rangle\right) \Rightarrow \left(h \sigma f \nabla g\right) \Rightarrow \left(h \sigma f \nabla g\right)$ |
| 8 | $\left(h \sigma f \nabla g\right) \nabla \rho_1 + \rho_2 = \tau \Rightarrow \tau$ |
| 9 | $\left(h \sigma f \nabla g\right) \nabla \rho_1 + \rho_2 = \tau \nabla \tau$ |
| 10 | $\left(h \sigma f \nabla g\right) \nabla \rho_1 + \rho_2 = \tau \left(\left(h \sigma f \nabla g\right) \nabla \rho_1 + \rho_2\right)$ |
| 11 | $\forall z : \rho_1 + \rho_2, \left(h \sigma f \nabla g\right) z = \tau \left(\left(h \sigma f \nabla g\right) z\right)$ |
| 12 | $h \sigma f \nabla g = \rho_1 + \rho_2 \rightarrow (h \sigma f) \nabla (h \sigma g)$ |</p>

In lemma 4.49 we gave two distribution rules for composition over (if then else) in $\lambda \omega L^\tau$. In $\lambda \omega L^\mu$ the second of these requires an extra condition.

5.25 LEMMA. In the context $\Gamma_{CPO}$ the following proposition is provable using axioms from $AXIOM^\mu$:

1. $\forall h : \text{bool}, \forall \sigma, \tau : \ast, \forall f, g : \sigma \rightarrow \tau, \forall \rho : \rho \rightarrow \sigma$

$$\left(h \text{ then } f \text{ else } g\right) h = \rho \rightarrow \tau, \text{ if } h \text{ then } f h \text{ else } g h$$
2. \( \forall b \in \text{bool. } \forall \rho, \sigma, \tau_1 \in \rho . \forall h \in \tau \times \rho . \forall f, g : \rho \to \sigma. \)
\[
\text{strict } \rho \times \tau \to \rho \Rightarrow h \circ (\text{if } b \text{ then } f \text{ else } g) = \rho \rightarrow \text{ if } b \text{ then } h \circ f \text{ else } h \circ g
\]

By the definition of bool and (if then else), the second part of this lemma is an instantiation of the previous lemma.

It is awkward to have to use the cpo-ordering \( \leq \) every time we want to prove termination of a recursive program. Some standard recursion patterns will be used often in recursive programs, such as primitive recursion, or recursion on the natural numbers or lists with all recursive calls on smaller natural numbers or shorter lists. For these common recursion patterns we would like to prove termination once and for all. Also, we would want to derive induction rules tailored to those recursion patterns, which can be used from then on to prove properties of programs that use one of these recursion patterns.
Part II

Semantics
Chapter 6

The Basic System

This chapter treats the semantics of the basic system $\lambda \omega_L$ introduced in chapter 3. The semantics gives a sound interpretation of the system. It provides a denotational semantics for the programming language and a truth-table semantics for the logic, which shows that the logic is free from inconsistencies. The model provides the justification (and possibly also the inspiration) for extensions of the basic system. This can be extensions of the programming language, e.g., new datatypes, datatype-constructors and associated primitive operations, as will be considered in subsequent chapters. It can also be extensions of the logic, typically axioms expressing properties of programs or datatypes. Some examples of such axioms have already been given in chapter 3.

The structure of this chapter closely follows that of chapter 3. We begin by considering the two subsystems of $\lambda \omega_L$: $\lambda \omega_s$ and $\lambda \omega_p$. Although syntactically these two systems are identical — they are copies of $\lambda \omega$ — we want different interpretations for them. This is because in $\lambda \omega_s$ our main interest concerns the interpretation of the terms (programs), whereas in $\lambda \omega_p$, our main interest concerns the interpretation of the types (propositions). Also, the models will have to accommodate extensions of the systems, and $\lambda \omega_s$ and $\lambda \omega_p$ will be extended in different directions.

Still, the global structure of the models for $\lambda \omega_s$ and $\lambda \omega_p$ is the same. In section 6.1 we describe the general structure of a model for $\lambda \omega$, the higher order lambda calculus. This is a straightforward extension of the environment models defined in [BMM90] for the second order lambda calculus.

In section 6.2 we give a model for $\lambda \omega_s$ by showing that the PER-model, which is the best known model for the second and higher order lambda calculus, is indeed an instantiation of this general model definition.

In section 6.3 it is shown how the general model definition can be instantiated to produce a model for $\lambda \omega_p$. The model for $\lambda \omega_p$ is much simpler than the one for $\lambda \omega_s$. It uses the classical interpretation of propositions as truth values. It identifies all proof terms, which is why it is called a proof-irrelevance model. This simple model suffices to prove consistency of the logic.

In section 6.4 we consider the semantics of the whole of $\lambda \omega_L$. It is shown how the proof-irrelevance for $\lambda \omega_p$ can be combined with any model for $\lambda \omega_s$ to produce a model for $\lambda \omega_L$. In $\lambda \omega_L$ there are more predicates, propositions and proofs than in $\lambda \omega_p$, but the truth-value interpretation of $\lambda \omega_p$ can easily be extended to include the additional constructs of $\lambda \omega_L$. This model proves consistency of the logic, and consistency of the axioms in $AXIOM$. 

107
In defining the semantics of \( \lambda \omega \) and its subsystems, the distinction between the different levels plays an important role. In the definition of a PTS there is just one collection of (pseudo)terms. But for the particular PTSs we consider a more refined context-free syntax has been introduced in definitions 3.4 and 3.16, where the different sets of pseudoterms Kind, Cons, Prog, Pkind, PCons and Proof are defined. The interpretation of terms in the different models will be based on this context-free syntax.

As far as the semantics is concerned, only PTSs, and not DPTSs, have to be considered. By theorem 2.36 any model for a PTS also models the associated DPTS. As the interpretation of a DPTS-term \( A \) in context \( \Gamma \) we can take the interpretation of its \( \delta \)-normal form, the PTS-term \( \nulft \delta(A) \). This means the interpretation of DPTS-terms is given in two steps: first take the \( \delta \)-normal form, and then take the interpretation of the resulting PTS-term. Defining the interpretation of DPTS-terms directly would only obscure the model definitions without any added value, because it would simply amount to composing the function \( \nulft \delta(\_\_\_) \) and the interpretation function for PTS-terms.

### 6.1 General Model Definition for \( \lambda \omega \)

In this section we give the general definition of the notion of environment model for the system \( \lambda \omega \). This describes the overall structure of an environment model for \( \lambda \omega \), and because \( \lambda \omega \) and \( \lambda \omega_p \) are copies of \( \lambda \omega \) also of environment models for \( \lambda \omega \) and \( \lambda \omega_p \).

In [BMM90] the notion of environment model for the second order lambda calculus is defined. It is based on the notion of environment model for the untyped lambda calculus, as given in [Mey82]. The general model definition for \( \lambda \omega \) - the higher order lambda calculus - is a simple extension of the one for the second order lambda calculus.

The aim of such a general model definition is to separate the general characteristics of a \( \lambda \omega \)-model from those that are particular to specific models. Some properties can be proven with respect to this general model definition, which saves us from having to prove them for each \( \lambda \omega \)-model individually. In particular, soundness of type assignment, i.e.

\[
\Gamma \vdash a : A \Rightarrow [a] \in [A],
\]

and soundness of conversion, i.e.

\[
a \simeq_B b \Rightarrow [a] = [b],
\]

will be proven for the general model definition.

Because the general model definition specifies exactly what constitutes a model for \( \lambda \omega \), it can also help the design of a specific model. This will be the case in chapter 8, where a model is constructed by solving the domain equations that are part of the general model definition.

The general model definition is explained using the copy \( \lambda \omega \), of \( \lambda \omega \). Recall that in \( \lambda \omega \), three levels were distinguished: kinds, datatype-constructors and programs. The two top layers of \( \lambda \omega \), the datatype-constructors and their kinds, do not depend on the bottom layer, the programs. This means the interpretation of the kinds and datatype-constructors can be considered separately, before we turn to the interpretation of the programs.
6.1. GENERAL MODEL DEFINITION FOR $\lambda\omega$

The semantics of the kinds and datatype-constructors

The datatype-constructors with their kinds form a simply typed lambda calculus with a single type constant $*$, and term constants $\rightarrow, \ast, \ast$ and $\Pi$ : $(K \rightarrow \ast) \rightarrow \ast$ for all kinds $K$. So a model for the datatype-constructors and their kinds is a model for the simply typed lambda calculus. Before we give a formal definition of what constitutes a model for the kinds and datatype-constructors, we discuss the different ingredients.

The kinds are generated by

$$ K := * | K \rightarrow K,$$

so all kinds are closed expressions. For the interpretation of kinds we simply have a set $\text{Kind}_K$ for every kind $K$.

The datatype-constructors of type $K$ are interpreted as elements of the set $\text{Kind}_K$. So if $\Gamma \vdash \sigma : K$, then

$$[\Gamma \vdash \sigma : K] \eta \in \text{Kind}_K;$$

$[\Gamma \vdash \sigma : K] \eta$ is the meaning of the datatype-constructor $\sigma$ in the environment $\eta$. This environment $\eta$ is a function which gives the meaning of the free variables of $\sigma$.

By lemma 3.5 the datatype-constructors are of the form

$$\sigma := \alpha | (\lambda \alpha : K. \sigma) | \sigma \sigma | \sigma \rightarrow \sigma | (\Pi \alpha : K. \sigma)$$

where $\alpha \in \text{Var}^{\omega}$ and $K$ ranges over kinds. To define the meaning of abstractions and applications in datatype-constructors (i.e. $(\lambda \alpha : K. \sigma)$ and $\sigma_1 \sigma_2$), the sets $\text{Kind}_K$ have to solve the domain equations

$$\text{Kind}_{K_1 \rightarrow K_2} \cong [\text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2}]$$

where the square brackets denote some subset of the function space. The associated isomorphisms are the element-to-function mappings, well-known from models of the type-free lambda calculus.

In fact, $\text{Kind}_{K_1} \rightarrow K$ and $\text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2}$ can be taken equal and not just isomorphic. We maintain the isomorphisms here to emphasize the similarity with the definition of the semantics of programs that will be given later.

Solving the domain equations above takes care of the interpretation of abstraction and application in datatype-constructors. This leaves only the datatypes of the form $\sigma \rightarrow \tau$ and $(\Pi \alpha : K. \sigma)$ to be dealt with.

The meaning of $\sigma \rightarrow \tau$ ($\in \text{Kind}_{\ast}$) is defined in terms of the meanings of $\sigma$ and $\tau$ (both $\in \text{Kind}_{\ast}$). For this a function \( \leftrightarrow \) is needed, with

$$ \leftrightarrow \in \text{Kind}_{\ast} \rightarrow \text{Kind}_{\ast} \rightarrow \text{Kind}_{\ast}.$$

The meaning of $(\Pi \alpha : K. \sigma)$ ($\in \text{Kind}_{\ast}$) is defined in terms of the function that maps all possible interpretations of $\alpha$ ($\in \text{Kind}_{\ast}$) to the resulting meanings of $\sigma$ ($\in \text{Kind}_{\ast}$). So

$$[\Gamma \vdash (\Pi \alpha : K. \sigma) \rightarrow \ast, \eta] \eta$$

is defined in terms of the function which maps $\alpha \in \text{Kind}_K$ to

$$[\Gamma, \alpha : K \vdash \sigma : \ast, \eta[\alpha := a] \in \text{Kind}_{\ast}.$$ For this a function \( \Pi \) is needed, with

$$\Pi \in \prod_{\Gamma \vdash \ast} (\text{Kind}_K \rightarrow \text{Kind}_{\ast}) \rightarrow \text{Kind}_{\ast}.$$

So \( \leftrightarrow \) provides the interpretation of $\rightarrow$, and \( \Pi \) provides the interpretation of $\Pi$. We have now introduced everything that is needed for a model for the $\lambda\omega$-datatype-constructors.
6.1 Definition. An environment frame for the datatype-constructors of \( \lambda \omega \) is a 4-tuple 
\( (\text{Kind}, \Phi_{\text{Kind}}, \Box, \Pi) \), where

- \( \text{Kind} = \{ \text{Kind}_{\mathcal{K}} \mid \mathcal{K} \text{ is a kind} \} \) is a family of sets, indexed by kinds.
- \( \Phi_{\text{Kind}} = \{ \Phi_{\mathcal{K}_1, \mathcal{K}_2} \mid \mathcal{K}_1 \rightarrow \mathcal{K}_2 \text{ is a kind} \} \) is a family of bijections with

\[
\Phi_{\mathcal{K}_1, \mathcal{K}_2} \in \text{Kind}_{\mathcal{K}_1 \rightarrow \mathcal{K}_2} \rightarrow [\text{Kind}_{\mathcal{K}_1} \rightarrow \text{Kind}_{\mathcal{K}_2}]
\]

where the square brackets denote some subset of the function space.

- \( \Box \in \text{Kind}_{\bullet} \rightarrow \text{Kind}_{\bullet} \rightarrow \text{Kind}_{\bullet} \).
- \( \Pi \in \prod_{\mathcal{K} \in \mathcal{K}_0} (\text{Kind}_{\mathcal{K}} \rightarrow \text{Kind}_{\bullet}) \rightarrow \text{Kind}_{\bullet} \).

The meaning of a datatype-constructor \( \sigma \) depends on an environment giving the meaning of its free variables. The meanings assigned to the free variables of \( \sigma \) by this environment have to match the kinds assigned to them in the context. The free variables of a datatype-constructor are all datatype-constructors variables, i.e., if \( \Gamma \vdash \sigma : \mathcal{K} \), then the free variables of \( \sigma \) are declared in \( \Gamma^{\mathcal{K}_0} \).

6.2 Definition. Let \( \Gamma \) be a \( \lambda \omega_2 \)-context. An environment \( \eta \) satisfies \( \Gamma^{\mathcal{K}_0} \) written \( \eta \models \Gamma^{\mathcal{K}_0} \) if \( \eta(\alpha) \in \text{Kind}_{\mathcal{K}} \) for all \( \alpha : \mathcal{K} \) in \( \Gamma^{\mathcal{K}_0} \).

The meaning of datatype-constructors is defined by induction on its type derivation rather than by induction on its structure. This is done because the definition involves (type) information which cannot be found in the datatype-constructors, but which can be found in their type derivations. For example, the meaning of \( \sigma \tau \) depends on \( \Phi_{\mathcal{K}_1, \mathcal{K}_2} \), where \( \mathcal{K}_1 \rightarrow \mathcal{K}_2 \) is the type of \( \sigma \). The meaning of programs will later also be defined by induction on type derivations.

A type derivation for a term follows the structure of that term fairly closely, so there is not much difference between the two forms of induction. However, there is a difference. It is caused by the inference rules (weaken) and (\( \beta \text{conv} \)), which can be applied at many points in a derivation.

By lemma 2.15 we can restrict the places where the rule (weaken) is used, namely only to derive \( \Gamma \vdash \sigma : \mathcal{K} \) where \( \sigma \) is a variable. In these cases, the interpretation of \( \sigma \) is immediately given by the environment.

This leaves (\( \beta \text{conv} \)) as the only inference rule whose use is not directed by the syntax. So the only way to produce different type derivations for the same term is by different applications of the rule (\( \beta \text{conv} \)). Whenever we define the meaning of terms by induction on their type derivation, the first thing to do is of course prove that different type derivation do not result in different meanings of the same term.

6.3 Definition (Semantics of \( \lambda \omega_2 \)-datatype-constructors). For datatype-constructors \( \sigma \) we define \( \models \Gamma \vdash \sigma : \mathcal{K} \) \( \eta \) for all \( \Gamma \vdash \sigma : \mathcal{K} \) and \( \eta \models \Gamma^{\mathcal{K}_0} \) by
induction on the derivation of $\Gamma \vdash \sigma : \mathcal{K}$, as follows
\[
\begin{align*}
\left[\Gamma, \alpha : \mathcal{K}, \Gamma' \vdash \alpha : \mathcal{K}\right] \eta &= \eta(\alpha) \\
\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta &= \left[\Gamma \vdash \sigma : \mathcal{K}'\right] \eta \quad \text{if } \Gamma \vdash \sigma : \mathcal{K} \text{ follows from } \Gamma \vdash \sigma : \mathcal{K}' \text{ by } (\beta\text{conv}). \\
\left[\Gamma \vdash \sigma : \mathcal{K}_2\right] \eta &= \Phi_{K_1,K_2}^{-1} \left[\Gamma \vdash \sigma : \mathcal{K}_1 \rightarrow \mathcal{K}_2\right] \eta \quad (\forall a \in \text{Kind}_{K_1}, \left[\Gamma, \alpha : \mathcal{K}_1 \vdash \sigma : \mathcal{K}_2\right] \eta[\alpha := a]) \\
\left[\Gamma \vdash \sigma : \tau : \ast,\ast\right] \eta &= \left[\Gamma \vdash \sigma : \ast,\ast\right] \eta[\longrightarrow] \left[\Gamma \vdash \tau : \ast,\ast\right] \eta \\
\left[\Gamma \vdash (\Pi\alpha : \mathcal{K} \cdot \sigma) : \ast,\ast\right] \eta &= \prod_{\mathcal{K}} (\exists a \in \text{Kind}_{K_1}, \left[\Gamma, \alpha : \mathcal{K} \vdash \sigma : \ast,\ast\right] \eta[\alpha := a])
\end{align*}
\]

The rule $(\beta\text{conv})$ is not really needed in type derivations for datatype-constructors. The type of a datatype-constructor is a kind and because all kinds are in normal form, this type is unique, not just unique up to $\delta$-conversion. This means that $\mathcal{K} \equiv \mathcal{K}'$ in all applications of $(\beta\text{conv})$. An immediate consequence is that $\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta$ does not depend on which particular type derivation we consider. \qed

The subscript $\mathcal{K}$ of $\prod_{\mathcal{K}}$ will usually be omitted.

By lemma 3.5 the clauses in the definition above cover all possible type derivations for datatype-constructors. Still, $\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta$ is not well-defined for all $\Gamma \vdash \sigma : \mathcal{K}$ and $\eta \models \Gamma^\omega$, for every environment frame. The problem is that
\[
\lambda a \in \text{Kind}_{K_1}, \left[\Gamma, \alpha : \mathcal{K}_1 \vdash \sigma : \mathcal{K}_2\right] \eta[\alpha := a]
\]
is possibly not in the subset $[\text{Kind}_{K_1} \longrightarrow \text{Kind}_{K_2}]$ of $\text{Kind}_{K_1} \longrightarrow \text{Kind}_{K_2}$, i.e. the range of the $\Phi_{K_1,K_2}$.

6.4 Definition. An environment model for the datatype-constructors of $\lambda\omega$ is an environment frame for the datatype-constructors for which $\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta$ is defined for all $\Gamma \vdash \sigma : \mathcal{K}$ and $\eta \models \Gamma^\omega$. \qed

In the actual models that we consider this will never be a problem, as the range $\text{Kind}_{K_1} \rightarrow \mathcal{K}_2$ of $\Phi_{K_1,K_2}$ will always be the whole function space $\text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2}$.

For any environment model for the datatype-constructors we have the following properties:

6.5 Theorem (Soundness of type assignment for datatype-constructors).
If $\Gamma \vdash \sigma : \mathcal{K}$ then $\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta \in \text{Kind}_{\mathcal{K}}$ for all $\eta \models \Gamma^\omega$.

Proof. Induction on the derivation of $\Gamma \vdash \sigma : \mathcal{K}$. \qed

6.6 Lemma (Soundness of $\beta$-reduction for datatype-constructors).
Suppose $\sigma \triangleright \sigma'$, $\Gamma \vdash \sigma : \mathcal{K}$, $\Delta$, and $\Gamma \vdash \sigma' : \mathcal{K} : \Delta$.

Then $\left[\Gamma \vdash \sigma : \mathcal{K}\right] \eta = \left[\Gamma \vdash \sigma' : \mathcal{K}\right] \eta$ for all $\eta \models \Gamma^\omega$.

Proof. Induction on the structure of $\sigma$. The interesting case is the case that $\sigma$ is a redex. Here we need the substitution lemma given below. \qed
6.7 LEMMA (Substitution in datatype-constructors).
Suppose \( \Gamma, y : B, \Gamma' \vdash_\sigma \eta \), \( \mathcal{K} \) and \( \Gamma \vdash b : B \). Then
\[
[\Gamma, \Gamma'[y := b] \vdash \sigma[y := b] : \eta] = [\Gamma, y : B, \Gamma' \vdash \sigma \mathcal{K}] \eta[y := \{ \Gamma \vdash b \} \eta],
\]
for all \( \eta \models (\Gamma, \Gamma'[y := b]) \), and \( \eta[y = \{ \Gamma \vdash b \} \eta] = (\Gamma, y : B, \Gamma') \eta \).

PROOF. Induction on the structure of \( \sigma \). \( \square \)

6.8 THEOREM (Soundness of \( \beta \)-conversion for datatype-constructors).
Suppose \( \sigma \approx_\beta \sigma', \Gamma \vdash \sigma \mathcal{K} : \mathcal{O} \), and \( \Gamma \vdash \sigma' \mathcal{K} : \mathcal{O} \).
Then \( [\Gamma \vdash \sigma : \mathcal{K}] \eta = [\Gamma \vdash \sigma' : \mathcal{K}] \eta \) for all \( \eta \models \Gamma \).

PROOF. This follows from soundness of \( \beta \)-reduction, \( \text{SR} \beta \) and \( \text{CR} \beta \). By \( \text{CR} \beta \) \( \sigma \) and \( \sigma' \) have a common reduct \( \sigma'' \). By \( \text{SR} \beta \) all terms on the reduction path from \( \sigma \) or \( \sigma' \) to \( \sigma'' \) have type \( \mathcal{K} \). By soundness of \( \beta \)-reduction, all these terms have the same meaning. \( \square \)

The semantics of programs

The definition of the semantics of programs is similar to the definition of the semantics of datatype-constructors. Instead of having a family of sets \( \text{Kind} \) indexed by kinds, we now have a family of sets \( \text{Dom} \) indexed by \( \text{Kind} \), i.e. indexed by the interpretations of datatypes. The meaning of a program of type \( \sigma \) is an element of the set that \( \text{Dom} \) associated to the meaning of \( \sigma \). So if \( \Gamma \vdash M : \sigma \), then
\[
[\Gamma \vdash M : \sigma] \eta \in \text{Dom}_{[\Gamma \vdash \sigma] \eta}.
\]

By lemma 3.5 programs are of the form
\[
M := x \mid (\lambda x. \sigma. M) \mid MM \mid (\lambda x. \mathcal{K}. M) \mid M \tau,
\]
where \( \sigma \) ranges over datatype-constructors, \( \mathcal{K} \) over kinds, \( \tau \in \text{Var}^* \) and \( \alpha \in \text{Var}^\mathcal{O} \). Defining the semantics of programs requires mappings similar to the element-to-function mappings \( \Phi_{\mathcal{K}_1, \mathcal{K}_2} \) needed to define the semantics of datatype-constructors. However, because there are two kinds of abstraction in programs, over a datatype and over a kind, things are slightly more complicated.

Suppose \( I \vdash M : \sigma \rightarrow \tau \). Then for all \( I \vdash N : \sigma \) we have \( I \vdash MN : \tau \), so we should be able to define the meaning of \( MN \) \((\in \text{Dom}_{[I \vdash \sigma] \eta} \) in terms of the meanings of \( M \) \((\in \text{Dom}_{[I \vdash \sigma] \eta} \) and \( N \) \((\in \text{Dom}_{[I \vdash \tau. \mathcal{K}] \eta} \). This means the meaning of \( M \) has to be considered as a mapping from \( \text{Dom}_{[I \vdash \sigma] \eta} \) to \( \text{Dom}_{[I \vdash \tau. \mathcal{K}] \eta} \). This requires
\[
\text{Dom}_{[I \vdash \sigma] \eta} \cong [\text{Dom}_{[I \vdash \sigma. \mathcal{K}] \eta} \rightarrow \text{Dom}_{[I \vdash \tau. \mathcal{K}] \eta}], \tag{i}
\]
where the square brackets denote some subset of the function space. The associated isomorphism
\[
\Phi_{[I \vdash \sigma \rightarrow \tau. \mathcal{K}] \eta} \in \text{Dom}_{[I \vdash \sigma. \mathcal{K}] \eta} \rightarrow [\text{Dom}_{[I \vdash \sigma. \mathcal{K}] \eta} \rightarrow \text{Dom}_{[I \vdash \tau. \mathcal{K}] \eta}]
\]
is the element-to-function mapping needed to define the meaning of term abstraction and application.
Suppose $\Gamma \vdash M : (\Pi \alpha : \mathcal{K} \cdot \sigma)$. Then $\Gamma \vdash M \tau \cdot \sigma[\alpha := \tau]$ for all $\Gamma \vdash \tau : \mathcal{K}$. So for all $\tau$ we have to define the meaning of $M \tau (\in \text{Dom}_{\Gamma \vdash (\Pi \alpha : \mathcal{K} \cdot \sigma) \eta})$ in terms of the meanings of $M (\in \text{Dom}_{\Gamma \vdash (\Pi \alpha : \mathcal{K} \cdot \sigma) \eta})$ and $\tau (\in \text{Kind}_{\mathcal{K}})$. This requires

$$\text{Dom}_{\Gamma \vdash (\Pi \alpha : \mathcal{K} \cdot \sigma) \eta} \cong \left[ \prod_{\alpha \in \text{Kind}_{\mathcal{K}}} \text{Dom}_{\Gamma \vdash \alpha : \mathcal{K} \cdot \sigma \eta}[\alpha = \alpha] \right],$$

where the square brackets denote some subset of the generalised Cartesian product. The associated isomorphism

$$\Phi_{\Gamma \vdash (\Pi \alpha : \mathcal{K} \cdot \sigma) \eta} \in \text{Dom}_{\Gamma \vdash (\Pi \alpha : \mathcal{K} \cdot \sigma) \eta} \longrightarrow \prod_{\alpha \in \text{Kind}_{\mathcal{K}}} \text{Dom}_{\Gamma \vdash \alpha \cdot \mathcal{K} \cdot \sigma \eta}[\alpha = \alpha]$$

is used to define the meaning of kind abstraction and application.

So for a model for $\lambda \omega$, we need a family of sets $\text{Dom}$ such that

$$\text{Dom}_a \overset{\square}{\square}_b \cong \left[ \text{Dom}_a \longrightarrow \text{Dom}_b \right],$$

$$\text{Dom} \overset{\square}{\square}_F \cong \left[ \prod_{a \in \text{Kind}_{\mathcal{K}}} \text{Dom}_F(a) \right],$$

for all $a, b \in \text{Kind}_{\mathcal{K}}$, and all $F \in \text{Kind}_{\mathcal{K} \rightarrow \mathcal{K}}$.

6.9 Definition. An environment frame for $\lambda \omega$ is a 4-tuple $(\text{KIND}, \text{Dom}, \Phi^-, \Phi^\Pi)$, where

- $\text{KIND} = (\text{Kind}, \Phi_{\text{Kind}})$ is an environment model for the datatype-constructors.
- $\text{Dom} = \{\text{Dom}_a | a \in \text{Kind}_{\mathcal{K}}\}$ is a family of sets, indexed by $\text{Kind}_{\mathcal{K}}$.
- $\Phi^- = \{\Phi^-_{a,b} | a, b \in \text{Kind}_{\mathcal{K}}\}$ and $\Phi^\Pi = \{\Phi^\Pi_F : \square, F \in \text{Kind}_{\mathcal{K} \rightarrow \text{Kind}_{\mathcal{K}}}\}$ are families of bijections with

$$\Phi^-_{a,b} \in \text{Dom}_a \overset{\square}{\square}_{b} \longrightarrow \left[ \text{Dom}_a \longrightarrow \text{Dom}_b \right],$$

$$\Phi^\Pi_F \in \text{Dom} \overset{\square}{\square}_F \longrightarrow \left[ \prod_{a \in \text{Kind}_{\mathcal{K}}} \text{Dom}_F(a) \right],$$

where the square brackets denote some subset of the function space and the product, respectively.

The meaning of a program depends on an environment that gives the meaning of its free variables. The meanings assigned to the free variables have to match the types assigned to them in the context.

6.10 Definition Let $\Gamma$ be a $\lambda \omega$-context. An environment $\eta$ satisfies $\Gamma^\square$ - written $\eta \models \Gamma^\square$ - if

- $\eta \models \Gamma^\square$, i.e. $\eta(\alpha) \in \text{Kind}_{\mathcal{K}}$ for all $\alpha : \mathcal{K}$ in $\Gamma^\square$, and
- $\eta(\tau) \in \text{Dom}_{\Gamma \vdash (\tau \cdot \sigma) \eta}$ for all $\tau \cdot \sigma$ in $\Gamma^\square$. 

\[\square\]
6.11 Definition (Semantics of programs).

For programs $M$ we define $[\Gamma \vdash M : \sigma ] \eta$ for all $\Gamma \vdash M : \sigma$ and $\eta \models \Gamma^{\omega \ast \omega}$, by induction on the derivation of $\Gamma \vdash M : \sigma$, as follows:

\[
\begin{align*}
[ \Gamma, x : \sigma, \Gamma' \vdash x : \sigma ] \eta &\xlongleftarrow{\eta} \eta(x) \\
[ \Gamma \vdash M : \sigma ] \eta &\xlongleftarrow{\eta} [ \Gamma \vdash M : \sigma' ] \eta \\
\text{if } \Gamma \vdash M : \sigma \text{ follows from } \Gamma \vdash M : \sigma' \text{ by } (\delta \text{conv}).
\end{align*}
\]

\[
\begin{align*}
[ \Gamma \vdash MN : \tau ] \eta &\xlongleftarrow{\eta} (\Phi_1 [ [ \Gamma \vdash M : \sigma \rightarrow \tau ] \eta \vert [ \Gamma \vdash N : \sigma ] \eta \\
[ \Gamma \vdash (\lambda x. M) : \sigma \rightarrow \tau ] \eta &\xlongleftarrow{\eta} \Phi_2^{-1}(\lambda x \in \text{Dom}_{[I \vdash \tau \ast \ast]}. \ [ \Gamma, x : \sigma \vdash M : \tau ] \eta \vert \alpha = \text{id}_\tau)
\end{align*}
\]

where $\Phi_1 = \Phi_1[I \vdash \tau, \ast \ast] \vert [I \vdash \tau, \ast \ast] \eta$ and $\Phi_2 = \Phi_2^{-1}(\lambda x \in \text{Kind}_K. \ [ \Gamma, \alpha \vdash \tau ] \eta \vert \alpha = \text{id}\).

By lemma 3.5 the clauses in this definition cover all possible type derivations for programs. However, not for every environment frame is $[ \Gamma \vdash M : \sigma ] \eta$ well-defined for all $\Gamma \vdash M : \sigma$ and $\eta \models \Gamma^{\omega \ast \omega}$. In order for the meaning of an abstraction $(\lambda x. M)$ or $(\lambda \alpha. K \ M)$ to be defined the ranges of the $\Phi_1$ and $\Phi_2$ above have to be large enough.

6.12 Definition. A $\lambda \omega$-environment model is a $\lambda \omega$-environment frame for which $[ \Gamma \vdash \sigma : K ] \eta$ is defined for all $\Gamma \vdash \sigma : K$ and $\eta \models \Gamma^{\omega \ast \omega}$. \hfill $\square$

Of course we want to show that the meaning of a program does not depend on the particular type derivation we consider. Unlike for datatype-constructors, for a program there can be many type derivations, in which the rule $(\delta \text{conv})$ is used in different places, and different (but $\beta$-convertible) types are derived for subexpressions of the program. To show that this does not influence the interpretation of the program, we need the fact that these $\beta$-convertible types have the same interpretation. This has already been proved in theorem 6.8, soundness of $\beta$-conversion for the datatype-constructors.

6.13 Lemma. Let $\delta r_1$ and $\delta r_2$ be derivations of $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \sigma'$, respectively. Then

\[
[ \Gamma \vdash M : \sigma ] \eta \xlongleftarrow{\eta} [ \Gamma \vdash M : \sigma' ] \eta,
\]

where the meanings are defined using $\delta r_1$ and $\delta r_2$ respectively.

Proof. Induction on the structure of $M$ using $UT_B$ and the soundness of $\beta$-conversion for datatype-constructors (theorem 6.8). \hfill $\square$

6.14 Theorem (Soundness of type assignment for programs)

If $\Gamma \vdash M : \sigma$ then $[ \Gamma \vdash M : \sigma ] \eta \in \text{Dom}_{[I \vdash \tau, \ast \ast]}$ for all $\eta \models \Gamma^{\omega \ast \omega}$.

Proof. Induction on the derivation of $\Gamma \vdash M : \sigma$, using soundness of type assignment for datatype-constructors (theorem 6.5). \hfill $\square$
6.15 Lemma (Soundness of β-reduction for programs).

Suppose $M \Rightarrow M'$, $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M' : \sigma'$.

Then $[\Gamma \vdash M : \sigma] \eta = [\Gamma \vdash M' : \sigma'] \eta$ for all $\eta \models \Gamma \Rightarrow \ast$.

Proof: Induction on the structure of $M$.

Datatype-constructors can be subterms of programs, so the reduction in $M$ can be a reduction in a datatype-constructor in $M$. Hence we need soundness of β-reduction for the datatype-constructors (lemma 6.6).

The interesting case is the case that $M$ is a redex, where we use the substitution lemma for programs given below.

6.16 Lemma (Substitution in programs).

Suppose $\Gamma, y : B, \Gamma' \vdash M : \sigma$ and $\Gamma \vdash b : B$. Then

$$[\Gamma, y := b] M \vdash \eta = [\Gamma, y : B, \Gamma' \vdash M : \sigma] \eta[y := \eta] = [\Gamma \vdash b : B] \eta$$

for all $\eta \models \Gamma \vdash b : B$ and $\eta[y := \eta] \models \Gamma \vdash b : B$.

Proof: Induction on the structure of $M$, using the substitution lemma for the datatype-constructors (lemma 6.7). We treat just one case, the case that $M$ is an abstraction, i.e. $M \equiv (\lambda x. \rho \ N)$. Its type is then a function type, i.e. $\sigma \equiv \rho \rightarrow \tau$.

First some abbreviations: $\overline{A}$ is $A[y := b]$ for $\lambda \omega$-terms $A$, and

$\overline{\Delta} \equiv \Gamma, \Gamma'[y := b]$

$\overline{\Delta} \equiv \Gamma, \Gamma[y : B, \Gamma' \vdash M : \sigma] \eta[y := \eta]$

$\overline{\eta} \equiv \eta[y := \eta] = [\Gamma \vdash b : B] \eta$

So $\overline{\Delta} \vdash b \ B$ and $\overline{\Delta} \vdash (\lambda x. \rho \ N) \vdash \rho \rightarrow \tau$.

Suppose $\eta \models \Gamma, \Gamma'[y := b]$ and $\eta[y := \eta] \models \Gamma \vdash b : B$. Then $\eta \models \overline{\Delta}$, and $\overline{\eta} \models \overline{\Delta}$.

To prove

$$[\overline{\Delta} \vdash (\lambda x. \rho \ N) \vdash \rho \rightarrow \tau] \overline{\eta} = [\overline{\Delta} \vdash (\lambda x. \rho \ N) \vdash \rho \rightarrow \tau] \eta$$

(i)

By the definition of $\left[ \ldots \right.$:

$$\left[ \overline{\Delta} \vdash (\lambda x. \rho \ N) \vdash \rho \rightarrow \tau \right] \overline{\eta} = \Phi_{\overline{\Delta} \vdash \rho \rightarrow \tau} \eta$$

$$\left[ \overline{\Delta} \vdash (\lambda x. \rho \ N) \vdash \rho \rightarrow \tau \right] \eta = \Phi_{\overline{\Delta} \vdash \rho \rightarrow \tau} \eta$$

By the substitution lemma for datatype-constructors

$$[\overline{\Delta} \vdash \rho \rightarrow \tau] \eta = [\overline{\Delta} \vdash \rho \rightarrow \tau \ast \overline{\eta}] \eta$$

and by the IH

$$[\overline{\Delta}, x : \rho \vdash N : \tau] \overline{\eta}[x := \xi] = [\overline{\Delta}, x : \rho \vdash N : \tau] \eta[x := \xi]$$

and so (i). Note that the IH only applies if $\eta[\tau = \xi] \models (\overline{\Delta}, x : \rho)$ and $\overline{\eta}[x := \xi] \models (\overline{\Delta}, x : \rho)$. This follows from $\eta \models \overline{\Delta}, \overline{\eta} \models \overline{\Delta}$ and

$$\xi \in \text{Dom}_\mathcal{F}(\overline{\Delta}, \rho, \ast).$$

\[\square\]
6.17 THEOREM (Soundness of $\beta$-conversion for programs).

Suppose $M \approx_{\beta} M'$, $\Gamma \vdash M : \sigma$, and $\Gamma \vdash M' : \sigma'$.

Then $\{ \Gamma \vdash M : \sigma \} \eta = \{ \Gamma \vdash M' : \sigma' \} \eta$ for all $\eta \models \Gamma \equiv_{\ast \ast}$.

PROOF. This follows from soundness of $\beta$-reduction, $SR_\eta$ and $CR_\eta$. 

In fact, the environment models for $\lambda\omega$, respect not only $\beta$- but also $\eta$-equality. This follows immediately from the fact that the $\Phi_{R_1, R_2}$, $\Phi_{\Delta}$, and $\Phi_{\Pi}$ are all bijections.
6.2 PER-Model for \( \lambda \omega_s \)

The best-known models for the second and higher-order lambda calculus are the so-called \( \text{PER-models} \), first given in [Gir72]. In these models datatypes are interpreted as partial equivalence relations over some partial combinatory algebra, and programs as equivalence classes. In this section it is shown that the \( \text{PER-model} \) given in [Gir72], called \( HEO^n \), can easily be fitted in the general model definition for \( \lambda \omega_s \).

6.18 Definition. 1. \( (\mathbb{N}, \_\_\_) \) is Kleene’s applicative structure. This means we have some enumeration of the partial recursive functions, and \( n \_m \) is the result of applying the \( n^{th} \) partial recursive function to \( m \).

2. \( \text{PER} \) is the set of all partial equivalence relations on \( \mathbb{N} \).

3. If \( R \in \text{PER} \) and \( n \in \mathbb{N} \), then \( [n]_R \) is the equivalence class of \( R \) that contains \( n \), i.e. \( [n]_R \equiv \{ m \in \mathbb{N} \mid (n, m) \in R \} \).

For the model for the datatype-constructors sets \( \text{Kind}_K \) are needed, and functions \( \rightarrow \) and \( \Pi \):

6.19 Definition

1. For all kinds \( K \) the set \( \text{Kind}_K \) is defined as follows:

\[
\begin{align*}
\text{Kind}_s & \equiv \text{PER} \\
\text{Kind}_{K_1 \rightarrow K_2} & \equiv \text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2}
\end{align*}
\]

Here \( \text{Kind}_{K_1 \rightarrow K_2} \) is the set of all functions from \( \text{Kind}_{K_1} \) to \( \text{Kind}_{K_2} \).

2. The functions \( \rightarrow \in \text{PER} \rightarrow \text{PER} \rightarrow \text{PER} \) and \( \Pi \in \Pi_{K \leq K} (\text{Kind}_K \rightarrow \text{PER}) \rightarrow \text{PER} \) are defined by

\[
\begin{align*}
S \rightarrow T & \equiv \{ (n, n') \in \mathbb{N}^2 \mid (n, m, n') \in T \} \\
\Pi F & \equiv \underset{a \in \text{Kind}_K}{\bigcap} F(a)
\end{align*}
\]

6.20 Lemma. \( \text{KIND}_{HEO} = ( \text{Kind}, \Phi_{\text{Kind}}, \rightarrow, \Pi ) \), where all the bijections in \( \Phi_{\text{Kind}} \) are identities, is an environment model for the datatype-constructors.

Proof. Trivial, because all \( \Phi_{K_1, K_2} \) are identities.

To complete the model for \( \lambda \omega_s \) we need a family of sets \( \text{Dom} \) and associated families of bijections \( \Phi_{\rightarrow} \) and \( \Phi_{\Pi} \).

6.21 Definition. Let \( R, S, T \in \text{PER} \) and \( F \in \text{Kind}_K \rightarrow \text{PER} \) for some kind \( K \). Then

1. \( \text{Dom}_R \equiv \{ [n]_R \mid (n, n) \in R \} \).
2. $\Phi_{S,T} \in \text{Dom}_S \rightarrow \text{Dom}_T$ is defined by

$$\Phi_{S,T} [m]_S = [m]_T$$

3. $\Phi^\Pi \in \text{Dom}_{\Pi} \rightarrow \prod_{a \in \text{Kind}_K} \text{Dom}_{F(a)}$ is defined by

$$\Phi^\Pi [m]_\Pi = [m]_{F(a)}$$

The definitions of $\Phi_{S,T}$ and $\Phi^\Pi$ are given in terms of representatives of equivalence classes. Checking that they are well-defined as functions on equivalence classes is straightforward. For $\Phi_{S,T}$ we have to verify that $(m, n, n') \in T$ if $(m, n') \in S$ (i.e. that

$$[m, n]_T = [m', n']_T$$

and $[n]_S = [n']_S$.) For $\Phi^\Pi$ we have to verify that $(m, m') \in F(a)$ if $(m, m') \in \Pi F$. These properties immediately follow from the definitions of $\rightarrow$ and $\Pi$.

6.22 Lemma. $(\text{KIND}_{K E O}, \text{Dom}, \Phi^-, \Phi^\Pi)$ is an environment frame for $\lambda\omega$.

Proof. We only have to show that

$$\Phi_{S,T} \in \text{Dom}_S \rightarrow \text{Dom}_T$$

$$\Phi^\Pi \in \text{Dom}_{\Pi} \rightarrow \prod_{a \in \text{Kind}_K} \text{Dom}_{F(a)}$$

are bijections, for certain subsets $\text{Dom}_S \rightarrow \text{Dom}_T$ of $\text{Dom}_S \rightarrow \text{Dom}_T$, and certain subsets $\prod_{a \in \text{Kind}_K} \text{Dom}_{F(a)}$ of $\prod_{a \in \text{Kind}_K} \text{Dom}_{F(a)}$. As these subsets we take the ranges of $\Phi_{S,T}$ and $\Phi^\Pi$.

6.23 Lemma. $(\text{KIND}_{K E O}, \text{Dom}, \Phi^-, \Phi^\Pi)$ is an environment model for $\lambda\omega$.

Proof. We have to prove that $[\Gamma \vdash M : \sigma]_{\eta}$ is defined for all $\Gamma \vdash M : \sigma$ and $\eta \models \Gamma^\bullet$. To prove this we prove by induction on the structure of $M$ that if $\Gamma \vdash M : \sigma$ with $\Gamma^* \equiv \chi_1 : \rho_1, \ldots, \chi_m : \rho_m$, then there is a partial recursive function $f \in \mathbb{N}^m \rightarrow \mathbb{N}$ such that for all $\eta \models \Gamma^\bullet$, $\eta(x_i) = [f(n_1, \ldots, n_m)]_{\Gamma^* \vdash \chi_i : \rho_i}$. In other words, given representatives $n_i$ of the equivalence classes $\eta(x_i)$, $f$ returns a representative of the equivalence class $[\Gamma \vdash M : \sigma]_{\eta}$.

The crucial case in this induction is the case that $M$ is an abstraction, for which we need Kleene's s-m-n theorem (see e.g. [Rog67]).
6.3 Proof-irrelevance Model for $\lambda \omega_p$

In this section the general model definition given in 6.1 is instantiated to produce a model for the logic $\lambda \omega_p$. The model for $\lambda \omega_p$ can be much simpler than that for $\lambda \omega_s$, because our main concern is the interpretation of the types (propositions), and not the interpretation of the terms (proofs). Propositions are interpreted as truth values in this model, and all proof terms are identified, which is why the model is called a proof-irrelevance model. This simple model is all that is needed to prove consistency of $\lambda \omega_p$, and to check the consistency of any additional axioms.

To stress that this model is meant as a model for $\lambda \omega_p$ and not for $\lambda \omega_s$, we use $\text{PKind}$, $\text{PDom}$, $\bigvee$ and $\bigwedge$ instead of $\text{Kind}$, $\text{Dom}$, $\prod$ and $\Pi$, respectively. Indeed, the model is useless as a model for the programming language $\lambda \omega_s$, because it would identify all programs. As we shall see, some of the technical machinery needed for an environment model can be avoided for this particular model, because it is so simple.

A proposition is interpreted as either the empty set 0 (representing "false") or as a singleton set 1 (representing "true").

6.24 Definition For all propkinds $\mathcal{P}$ the set $\text{PKind}_{\mathcal{P}}$ is defined as

$$\text{PKind}_{\mathcal{P}} = \{0, 1\}$$

Here $\text{PKind}_{\mathcal{P}_1} \rightarrow \text{PKind}_{\mathcal{P}_2}$ is the set of all functions from $\text{PKind}_{\mathcal{P}_1}$ to $\text{PKind}_{\mathcal{P}_2}$.

For the interpretation of propositions that are formed using implication or universal quantification we define:

6.25 Definition. The functions $\bigvee \in \text{PKind}_{\mathcal{P}} \rightarrow \text{PKind}_{\mathcal{P}}$ and $\bigwedge \in \prod_{\mathcal{P}} \text{PKind}_{\mathcal{P}} \rightarrow \text{PKind}_{\mathcal{P}}$ are defined as follows

$$\bigvee P Q \equiv \begin{cases} 1 & \text{if } P = 0 \text{ or } Q = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\bigwedge F \equiv \bigcap_{X \in \text{PKind}_{\mathcal{P}}} F(X)$$

So $\bigwedge F = 1$ iff $F(X) = 1$ for all $X \in \text{PKind}_{\mathcal{P}}$.

6.26 Lemma. $\text{PKIND} = \langle \text{PKind}, \Phi_{\text{PKind}}, \bigvee, \bigwedge \rangle$, where all the bijections in $\Phi_{\text{PKind}}$ are identities, is an environment model for the $\lambda \omega_p$ prop-constructors.

Proof. Trivial, because all $\Phi_{\mathcal{P}_1, \mathcal{P}_2}$ are identities.

6.27 Example. For the propositional constants and logical connectives defined in definition 3.11 we get

$$[\text{False} : *_p] = 0$$

$$[\text{True} : *_p] = 1$$

$$[\neg *_p] = \lambda X \in \{0, 1\} \begin{cases} 1 & \text{if } X = 0 \\ 0 & \text{if } X = 1 \end{cases}$$

$$[\land *_p : *_p] = \lambda X, Y \in \{0, 1\}. X \cap Y$$

$$[\lor *_p : *_p] = \lambda X, Y \in \{0, 1\}. X \cup Y$$

$$[\exists *_p (\mathcal{P} \rightarrow *_p) : *_p] = \lambda F \in \text{PKind}_{\mathcal{P}} \rightarrow \{0, 1\} \cup_{X \in \text{PKind}_{\mathcal{P}}} F(X)$$
As the family of sets $PDom$ that maps each element $P$ of $PKnd_{\mathcal{P}}$ to a set $PDom_{\mathcal{P}}$ we can simply take the identity:

6.28 Definition. $PDom_1 \cong \mathcal{P}$ and $PDom_0 \cong \emptyset$. 

Then the domain equations

$$PDom_P \xrightarrow{\pi} Q \cong PDom_{\mathcal{P}} \rightarrow PDom_Q$$

$$PDom \bigvee_{\mathcal{P}} F \cong \prod_{X \in PKnd_{\mathcal{P}}} PDom_{\mathcal{P}}(X)$$

become

$$P \xrightarrow{\pi} Q \cong P \rightarrow Q$$

$$\bigvee_{\mathcal{P}} F \cong \prod_{X \in PKnd_{\mathcal{P}}} F(X)$$

By the definition of $\xrightarrow{\pi}$ and $\bigvee$, either both sides are the empty set or both sides are singleton sets. So the domain equations for $PDom$ are solved. Moreover, all the associated isomorphisms are unique, since the isomorphisms are either between two one-element sets or between two empty sets. If we let $\Phi^P_Q$ and $\Phi^P$ be the isomorphisms solving the domain equations above, then

6.29 Lemma. $(PKIND, PDom, \Phi^p, \Phi^v)$ is an environment model for $\lambda\omega_{\mathcal{P}}$.

Proof. Easy, because $P \xrightarrow{\pi} Q$ and $\bigvee_{\mathcal{P}} F$ are isomorphic with the whole function space $P \rightarrow Q$ and the whole product $\prod_{X \in PKnd_{\mathcal{P}}} F(X)$, respectively. 

If $\Gamma \vdash \mathcal{P} : \mathcal{P} \cdot \mathcal{P}$ and $\eta \models \Gamma \vdash_{\mathcal{P}} \mathcal{P}$ then by soundness of type assignment

$$\{ \Gamma \vdash \mathcal{P} \} \eta \in \{ \Gamma \vdash \mathcal{P} \cdot \mathcal{P} \} \eta \in \{0, 1\}$$

But this is only possible if $\{ \Gamma \vdash \mathcal{P} \cdot \mathcal{P} \} \eta \models \eta = 1$, and then $\{ \Gamma \vdash \mathcal{P} \cdot \mathcal{P} \} \eta \models \eta \in 1$. This means that all proofs are identified in this model - they are all interpreted as the element of 1. Instead of defining the interpretation of a proof by induction on its type derivation it could simply be defined as the element of 1.

It also follows immediately from soundness of type assignment that:

6.30 Corollary. $\mathcal{P}$ is not provable in context $\Gamma$ if $\{ \Gamma \vdash \mathcal{P} \cdot \mathcal{P} \} \eta \models \eta = 0$ for some $\eta \models \Gamma \vdash_{\mathcal{P}} \mathcal{P}$. 

This can be used to prove consistency (of contexts) in $\lambda\omega_{\mathcal{P}}$, for example

6.31 Corollary (Consistency)

1. False is not provable in the empty context.

2. False is not provable in the context containing just the assumption

$$\text{classic} \cdot \forall \mathcal{P} : \mathcal{P} (\rightarrow \mathcal{P} \models \mathcal{P}) \Rightarrow \mathcal{P}$$
6.3. PROOF-IRRELEVANCE MODEL FOR $\lambda \omega_\mathcal{P}$

PROOF. 1. follows immediately from the fact that $[\text{False} : *_p] = 0$. For 2. we also have to show that there is an environment $\eta$ that satisfies the context classic : $\forall P : *_p. (\neg \neg P) \Rightarrow P$. The only possible choice for $\eta$ is the environment that maps classic to the element of 1. That this environment satisfies the context follows from $[\forall P : *_p. (\neg \neg P) \Rightarrow P : *_p] = 1$. 

This proof-irrelevance model can be seen as a degenerated PER-model, namely the PER-model where partial equivalence relations over 1 instead of $\mathbb{N}$ are used, and the only possible function in $1^2 \rightarrow 1$ is used instead of the Kleene application $\in \mathbb{N}^2 \rightarrow \mathbb{N}$. 
6.4 Model for \( \lambda \omega_L \)

We now turn to the semantics of the whole system \( \lambda \omega_L \). Recall that \( \lambda \omega_L \) consists of three parts:

- the programming language \( \lambda \omega_s \),
- the purely logical part \( \lambda \omega_p \),
- four more PTS-rules that allow the formation of proofs and propositions that depend on programs and datatypes.

This means that a model for \( \lambda \omega_L \) will contain a submodel for \( \lambda \omega_s \) and a submodel for \( \lambda \omega_p \). We will in fact produce a model for \( \lambda \omega_L \) by combining a model for \( \lambda \omega_s \) and a model for \( \lambda \omega_p \).

To be precise, an arbitrary model for \( \lambda \omega_s \) and the proof-irrelevance model for \( \lambda \omega_p \) will be combined. \( \lambda \omega_L \) has the same programs and datatypes as \( \lambda \omega_s \), so any model for \( \lambda \omega_s \) can be used to interpret the programs and datatypes of \( \lambda \omega_L \). This means we do not have to fix a particular interpretation for \( \lambda \omega_s \) yet. A model for \( \lambda \omega_p \) can be given with respect to an arbitrary model for the \( \lambda \omega_s \)-part.

On the other hand, \( \lambda \omega_L \) does have more proofs and propositions than \( \lambda \omega_p \). We fix a particular interpretation for the \( \lambda \omega_p \)-part, namely the proof-irrelevance model defined in the previous section. This proof-irrelevance model can easily be extended to interpret these additional constructs in the logic-part of \( \lambda \omega_L \).

Recall that in \( \lambda \omega_L \) six syntactic categories were distinguished, namely

1. kinds, which are elements of \( \text{Kind} \),
2. datatype-constructors, which are elements of \( \text{Cons} \),
3. programs, which are elements of \( \text{Prog} \),
4. propkinds, which are elements of \( \text{Pkind} \),
5. prop-constructors, which are elements of \( \text{PCons} \),
6. proofs, which are elements of \( \text{Proof} \).

Each of these depends only on (some of) the ones that precede it in the order given above. This layered structure is a result of the fact that some \((s_1, s_2)\) combinations are excluded as PTS-rules in the definition of \( \lambda \omega_L \). Here the layered structure pays off. It can be exploited by giving the semantics of each of the levels separately.

Let

\[
M = \langle \text{KIND}, \text{Dom}, \Phi \rangle
\]

be any environment model for \( \lambda \omega_s \), with

\[
\text{KIND} = \langle \text{Kind}, \Phi, \Phi^{\downarrow}, \Phi^{\downarrow} \rangle
\]

In the rest of this section a \( \lambda \omega_L \)-model will be defined in which this model \( M \) is the \( \lambda \omega_s \)-submodel.
6.4. MODEL FOR $\lambda \omega_L$

The semantics of datatype-constructors and programs

$\lambda \omega_L$ has the same kinds, datatype-constructors and programs as $\lambda \omega$, their interpretation is simply given by $\mathcal{M}$:

6.32 DEFINITION. 1. $\eta \models \Gamma \vdash \sigma$ and $\eta \models \Gamma \vdash \sigma \rightarrow \tau$ are defined for $\lambda \omega_L$-contexts $\Gamma$ as for $\lambda \omega$-contexts (definitions 6.2 and 6.10).

2. $[ \Gamma \vdash \sigma : \mathcal{K} ] \eta$ is defined for all $\Gamma \vdash \sigma : \mathcal{K}$ and $\eta \models \Gamma \vdash \sigma$ as in definition 6.3.

3. $[ \Gamma \vdash M : \sigma ] \eta$ is defined for all $\Gamma \vdash M \sigma$ as in definition 6.11.

Because $\mathcal{M}$ is an environment model of $\lambda \omega$, this interpretation has all the properties we have for environment models for $\lambda \omega$, e.g. soundness of type assignment and conversion.

The semantics of prop-kinds

By lemma 3.17 the prop-kinds are of the form

$\mathcal{P} ::= *_{\sigma} \mid \mathcal{P} \rightarrow \mathcal{P} \mid \sigma \rightarrow \mathcal{P} \mid (\Pi_{\alpha : \mathcal{K}}. \mathcal{P})$,

where $\mathcal{K}$ ranges over kinds, $\sigma$ over datatype-constructors, and $\alpha \in \text{Var}^{\mathcal{L}}$. So prop-kinds can contain free variables. For example, in $\alpha : *_{\sigma} \vdash \alpha \rightarrow *_{\sigma} : \square_p$, the propkind $\alpha \rightarrow *_{\sigma}$ contains a free variable $\alpha$. This means that the interpretation of a prop-kind depends on an environment that gives the interpretation of its free variables.

6.33 DEFINITION (Semantics of prop-kinds)

For $\lambda \omega_L$-propkinds $\mathcal{P}$ we define $[ \Gamma \vdash \mathcal{P} : \square_p ] \eta$ for all $\Gamma \vdash \mathcal{P} \square_p$ and $\eta \models \Gamma \vdash \sigma$, by induction on the derivation of $\Gamma \vdash \mathcal{P} : \square_p$, as follows

$[ \Gamma \vdash *_{\sigma} \square_p ] \eta \equiv \{0, 1\}$

$[ \Gamma \vdash \mathcal{P}_1 \rightarrow \mathcal{P}_2 \square_p ] \eta \equiv [ \Gamma \vdash \mathcal{P}_1 : \square_p ] \eta \rightarrow [ \Gamma \vdash \mathcal{P}_2 : \square_p ] \eta$

$[ \Gamma \vdash \sigma \rightarrow \mathcal{P} \square_p ] \eta \equiv \text{Dom}[\Gamma \vdash \sigma \rightarrow \mathcal{P}] \eta \rightarrow [ \Gamma \vdash \mathcal{P} : \square_p ] \eta$

$[ \Gamma \vdash (\Pi_{\alpha : \mathcal{K}}. \mathcal{P}) \square_p ] \eta \equiv \prod_{\alpha \in \text{Var}^{\mathcal{L}}} [ \Gamma, \alpha : \mathcal{K} \vdash \mathcal{P} : \square_p ] \eta[\alpha := a]$

Here in the right hand sides, $A \rightarrow B$ is the set of all functions from $A$ to $B$, and $\prod_{x \in A} B(x)$ is the set of all functions that map an $a \in A$ to an element of $B(a)$. By lemma 3.17 the clauses above define $[ \Gamma \vdash \mathcal{P} : \square_p ] \eta$ for all propkinds $\mathcal{P}$.

6.34 LEMMA (Soundness of $\beta$-reduction for prop-kinds).

Suppose $\mathcal{P} \triangleright_{\beta} \mathcal{P}'$, $\Gamma \vdash \mathcal{P} : \square_p$ and $\Gamma \vdash \mathcal{P}' : \square_p$.

Then $[ \Gamma \vdash \mathcal{P} : \square_p ] \eta = [ \Gamma \vdash \mathcal{P}' : \square_p ] \eta$ for all $\eta \models \Gamma \vdash \sigma$.

PROOF. Induction on the structure of $\mathcal{P}$, using soundness of $\beta$-reduction for the datatype-constructors (lemma 6.6)
The substitution lemma for propkinds is not needed to prove soundness of β-reduction for propkinds, because all reductions in a propkind occur in subexpressions that are datatype-constructors. We will need the substitution lemma later.

6.35 Lemma (Substitution in propkinds).
Suppose \( \Gamma, \beta : B, \Gamma' \vdash \Box \) and \( \Gamma \vdash b : B \). Then
\[
[\Gamma, \beta'[y := b] : B, \Gamma' \vdash \Box \eta(y) = [\Gamma, \beta : B, \Gamma' \vdash \Box \eta] \quad \text{for all } \eta \models (\Gamma, \beta'[y := b])^{\Box},
\]
and \( \eta[y := [\Gamma \vdash t : B] \eta] = (\Gamma, \beta : B, \Gamma')^{\Box} \).

Proof. Induction on the structure of \( \Box \), using the substitution lemma for the datatype-constructors (lemma 6.7).

6.36 Theorem (Soundness of β-conversion for propkinds).
Suppose \( \Pi \equiv \Box \), \( \Gamma \vdash \Box \) and \( \Gamma \vdash \Box' : \Box_p \).
Then \( [\Gamma \vdash \Box \vdash \Box_p] \eta = [\Gamma \vdash \Box' \vdash \Box_p] \eta \) for all \( \eta \models \Gamma^{\Box} \).

Proof. This follows SR, CR, and from soundness of β-reduction for propkinds (lemma 6.34).

The semantics of prop-constructors

By lemma 3.17 the prop-constructors are of the form
\[
P \equiv x | (\lambda y : A. P) | Pa | (\forall y : A. P) | P \Rightarrow P,
\]
where \( A \) ranges over the kinds, datatype-constructors and prop-kinds, \( a \) over the datatype-constructors, programs and prop-constructors, and \( x \in \Var^{\Box} \).

The interpretation of a prop-constructor \( P : \Box \) will be an element of the interpretation of \( \Box \), i.e., if \( \Gamma \vdash P : \Box \vdash \Box_p \) then \( [\Gamma \vdash P : \Box] \eta \in [\Gamma \vdash \Box : \Box_p] \eta \). So for example predicates on a datatype \( \sigma \) are interpreted as characteristic functions of subsets of \( \Dom_{\Gamma \vdash \Box : \Box_p} \eta \).

6.37 Definition. Let \( \Gamma \) be a \( \lambda \omega \)-context. An environment \( \eta \) satisfies \( \Gamma^{\Box} \times \Box_p \) written \( \eta \models \Gamma^{\Box} \times \Box_p \) if \( \eta \models \Gamma^{\Box} \times \Box_p \) and \( \eta(P) \in [\Gamma \vdash \Box : \Box_p] \eta \) for all \( P \in \Box \) in \( \Gamma^{\Box} \).

Given the definition of \( [\Gamma \vdash \Box : \Box_p] \eta \), the interpretation of application and abstraction in prop-constructors is straightforward. For the interpretation of products in propositions we use intersections, as in the model for \( \lambda \omega \).

6.38 Definition (Semantics of prop-constructors).
For \( \lambda \omega \)-prop-constructors \( P \) we define \( [\Gamma \vdash P : \Box] \eta \) for all \( \Gamma \vdash P : \Box \) and \( \eta \models \Gamma^{\Box} \times \Box_p \), by induction on the derivation of \( \Gamma \vdash P : \Box \), as follows
\[
[\Gamma, \Delta \vdash \Box, \Gamma' \vdash T] \eta = \eta(\Delta)
\]
\[
[\Gamma \vdash P : \Box] \eta = [\Gamma \vdash P : \Box] \eta
\]
if \( \Gamma \vdash P : \Box \) follows from \( \Gamma \vdash \Box \) by \( (\beta \text{conv}) \)
\[
[\Gamma \vdash Pa : \Box[x = a]] \eta = [\Gamma \vdash P : (\Pi x : A. \Box)] \eta \Gamma \vdash a : A \eta
\]
\[
[\Gamma \vdash (\lambda x : A. P) : (\Pi x : A. \Box)] \eta = \forall \xi \in A \Gamma \vdash x : A \vdash P : \Box [\eta[x = \xi]]
\]
\[
[\Gamma \vdash (\forall y : A. P) : *] \eta = \forall \xi \eta \in A \Gamma \vdash x : A \vdash P : \Box \eta[x := \xi]
\]
6.4 \textit{MODEL FOR } \lambda \omega_L \\

\begin{align*}
\text{where } A &\equiv \begin{cases} 
\mathbf{Kind}_A & \text{if } \Gamma \vdash A : \Box_s \\
\text{Dom}(\mathcal{F}_A) & \text{if } \Gamma \vdash A : \ast_s \\
\{ \Gamma \vdash A : O_p \} \eta & \text{if } \Gamma \vdash A : O_p \\
\{ \Gamma \vdash A : *_p \} \eta & \text{if } \Gamma \vdash A : *_p
\end{cases}
\end{align*}

Here we take \( \cap_{x \in \mathbb{N}} B(x) = 1 \). By lemma 3.17 the clauses above define \( \llbracket \Gamma \vdash P : \mathcal{P} \rrbracket \eta \) for all propconstructors \( P \). Note that in the last clause \( A \) may be a proposition, in which case \( (\forall x : A. \ P) \) is an implication \( A \Rightarrow P \). \( \square \)

6.39 \textsc{Lemma}. Let \( \text{der}_1 \) and \( \text{der}_2 \) be derivations of \( \Gamma \vdash P : \mathcal{P} \) and \( \Gamma \vdash P : \mathcal{P}' \), respectively. Then

\( \llbracket \Gamma \vdash P : \mathcal{P} \rrbracket \eta = \llbracket \Gamma \vdash P : \mathcal{P}' \rrbracket \eta \),

where the meanings are defined using \( \text{der}_1 \) and \( \text{der}_2 \) respectively.

\textsc{Proof}. Induction on the structure of \( P \).

6.40 \textsc{Theorem} (Soundness of type assignment for prop-constructors).

If \( \Gamma \vdash P : \mathcal{P} : \Box_p \), then \( \llbracket \Gamma \vdash P : \mathcal{P} \rrbracket \eta \in \llbracket \Gamma \vdash P : \Box_p \rrbracket \eta \) for all \( \eta \models \Gamma \Box_s \ast_s \).

\textsc{Proof}. Induction on the derivation of \( \Gamma \vdash P : \mathcal{P} \), using soundness of type assignment for datatypeconstructors and programs (theorems 6.5 and 6.14). \( \square \)

6.41 \textsc{Lemma} (Soundness of \( \beta \)-reduction for prop-constructors).

Suppose \( P \triangleright_\beta P' \), \( \Gamma \vdash P : \mathcal{P} \) and \( \Gamma \vdash P' : \mathcal{P}' \).

Then \( \llbracket \Gamma \vdash P : \mathcal{P} \rrbracket \eta = \llbracket \Gamma \vdash P' : \mathcal{P}' \rrbracket \eta \) for all \( \eta \models \Gamma \Box_s \ast_s \).

\textsc{Proof}. Induction on the structure of \( P \). Datatype-constructors, programs and propkinds can be subterms of prop-constructors, so soundness of \( \beta \)-reduction for these expressions is needed (lemmas 6.6, 6.15 and 6.34).

As usual, for the case that \( M \) is the redex we use the substitution lemma given below. \( \square \)

6.42 \textsc{Lemma} (Substitution in prop-constructors).

Suppose \( \Gamma, y : B, \Gamma' \vdash P : \mathcal{P} \) and \( \Gamma \vdash t : B \). Then

\( \llbracket \Gamma, \Gamma'[y := b] \vdash P[y := b] : \mathcal{P} \rrbracket \eta = \llbracket \Gamma, y : B, \Gamma' \vdash P : \mathcal{P} \rrbracket \eta[y := \llbracket \Gamma \vdash t : B \rrbracket \eta] \),

for all \( \eta \models (\Gamma, \Gamma'[y := b]) \Box_s \ast_s \) and \( \eta[y := \llbracket \Gamma \vdash t : B \rrbracket] \models (\Gamma, y : B, \Gamma') \Box_s \ast_s \).

\textsc{Proof}. Induction on the structure of \( P \), using the substitution lemmas for the datatypeconstructors, programs and propkinds (lemmas 6.7, 6.16 and 6.35). \( \square \)

6.43 \textsc{Theorem} (Soundness of \( \beta \)-conversion for prop-constructors).

Suppose \( P \equiv_\beta P' \), \( \Gamma \vdash P : \mathcal{P} \) and \( \Gamma \vdash P' : \mathcal{P} \).

Then \( \llbracket \Gamma \vdash P : \mathcal{P} \rrbracket \eta = \llbracket \Gamma \vdash P' : \mathcal{P} \rrbracket \eta \) for all \( \eta \models \Gamma \Box_s \ast_s \).

\textsc{Proof}. This follows from soundness of \( \beta \)-reduction for prop-constructors, SR\( \beta \) and CR\( \beta \). \( \square \)
The semantics of proofs

6.44 Definition. Let \( \Gamma \) be a \( \lambda \omega \)-context. An environment \( \eta \) satisfies \( \Gamma \) — written \( \eta \models \Gamma \) — if \( \eta \models \Gamma^\alpha \times \sigma \times \tau \) and \( \eta(x) \in [\Gamma \vdash P : \ast_p] \eta \) for all \( x : P \) in \( \Gamma^\sigma \).

So \( \eta \models \Gamma \) iff

- \( \eta(x) \in \text{Kind}_K \) for all \( x : K \) in \( \Gamma^\alpha \), and
- \( \eta(x) \in \text{Dom}[I \rightarrow \sigma \times \tau] \) for all \( x : \sigma \) in \( \Gamma^\alpha \), and
- \( \eta(x) \in [\Gamma \vdash P : \Omega_p] \eta \) for all \( x : P \) in \( \Gamma^\alpha \), and
- \( \eta(x) \in [\Gamma \vdash P : \ast_p] \eta \) for all \( x : P \) in \( \Gamma^\alpha \).

Because \( [\Gamma \vdash P : \ast_p] \eta \in \{0, 1\}, \) \( \eta(x) \in [\Gamma \vdash P : \ast_p] \eta \) is only possible if \( [\Gamma \vdash P : \ast_p] \eta = 1 \).

6.45 Definition (Semantics of proofs). For \( \lambda \omega \)-proofs \( p \) we define \( [\Gamma \vdash p : P] \eta \) for all \( \eta \models \Gamma \) as the element of \( \Gamma \).

6.46 Theorem (Soundness of type assignment for proofs). If \( \Gamma \vdash p : P \) then \( [\Gamma \vdash p : P] \eta \in [\Gamma \vdash P : \ast_p] \eta \) for all \( \eta \models \Gamma \) (i.e. \( [\Gamma \vdash P : \ast_p] \eta = 1 \)).

Proof. We prove that, for proofs \( p, \)

\[
\text{if } \Gamma \vdash p : P \text{ then } [\Gamma \vdash P : \ast_p] \eta = 1 \text{ for all } \eta \models \Gamma,
\]

by induction on the derivation of \( \Gamma \vdash p : P \), using soundness of type assignment for datatypes, constructors, programs and propconstructors (theorems 6.5, 6.14 and 6.40).

As for the proof-irrelevance model for \( \lambda \omega_p \), it follows immediately from soundness of type assignment that:

6.47 Corollary. \( P \) is not provable in context \( \Gamma \) if \( [\Gamma \vdash P : \ast_p] \eta = 0 \) for some \( \eta \models \Gamma \).

This can be used to prove consistency (of contexts) in \( \lambda \omega_i \), as for instance in corollary 6.49.

6.48 Example. As an example of the interpretation of a \( \lambda \omega \)-propconstructor, we consider Leibniz' equality, which was defined in definition 3.21 by

\[
=_{L} = \lambda \alpha : \star, \lambda \beta : \star. \lambda x, y : \alpha. \forall P : \alpha \rightarrow \ast_p. (P \ x) \Rightarrow (P \ y)
\]

\( \prod \alpha : \ast_p. \alpha = \alpha \rightarrow \ast_p \).

This will turn out to be interpreted as equality in the \( \lambda \omega_i \)-model.

Suppose \( M \) and \( N \) are programs with \( \Gamma \vdash M, N \) and \( \eta \models \Gamma \). Then

\[
[\Gamma \vdash M =_{L} N : \ast_p] \eta
= [\Gamma \vdash \forall P : \sigma \rightarrow \ast_p. (P \ M) \Rightarrow (P \ N) : \ast_p \] \eta
= \bigcap_{\xi \in \text{Dom}[I \rightarrow \sigma \times \ast_p]} [\Gamma, \Gamma \vdash P : \sigma \rightarrow \ast_p \vdash (P \ M) \Rightarrow (P \ N) : \ast_p \] \eta \mid P = \xi
\]

The above expression is valid for \( \eta \models \Gamma \).
For $\xi \in \text{Dom}([\Gamma \vdash M =_\sigma N : *_p]) \rightarrow \{0, 1\}$ it follows by the definition of $[\ ]$ that

$$\begin{align*}
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi] \\
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi]
\end{align*}$$

and that

$$\begin{align*}
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi] \\
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi]
\end{align*}$$

and that

$$\begin{align*}
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi] \\
\Gamma, P : \sigma \vdash *_p \Gamma (P M) &\Rightarrow (P N) : *_p \eta[P := \xi]
\end{align*}$$

Now, let $\eta$ be an arbitrary function $\eta : \{0, 1\} \rightarrow \text{Dom}(\Gamma \vdash \sigma)$.

So for $\Gamma \vdash M =_\sigma N : *_p \eta$ we get:

$$\begin{align*}
\Gamma &\vdash \forall \sigma \Gamma \vdash *_p (P M) \Rightarrow (P N) : *_p \eta \\
\eta &\in \text{Dom}([\Gamma \vdash M =_\sigma N : *_p]) \rightarrow \{0, 1\}
\end{align*}$$

This can be simplified further if we distinguish two possibilities for $\Gamma \vdash M =_\sigma N : *_p \eta$ and $\Gamma \vdash N : \sigma \eta$:

(i) Suppose $\Gamma \vdash M =_\sigma N : *_p \eta$.

Then $\eta(\Gamma \vdash M =_\sigma N : *_p) = 1$ since $0 \rightarrow 0 = 1$ and $1 \rightarrow 1 = 1$, this means that $\eta(\Gamma \vdash (M : \sigma \eta) \Rightarrow (N : \sigma \eta)) = 1$ for all $\eta \in \text{Dom}([\Gamma \vdash M =_\sigma N : *_p]) \rightarrow \{0, 1\}$.

(ii) Suppose $\Gamma \vdash M =_\sigma N : *_p \eta$.

Then there is a function $\xi \in \text{Dom}([\Gamma \vdash M =_\sigma N : *_p]) \rightarrow \{0, 1\}$ such that $\eta(\Gamma \vdash (M : \sigma \eta) \Rightarrow (N : \sigma \eta)) = 0$ and $\eta(\xi(\Gamma \vdash (M : \sigma \eta) \Rightarrow (N : \sigma \eta)) = 0$ for this function $\xi$.

So

$$\begin{align*}
\Gamma &\vdash \forall \sigma \Gamma \vdash *_p (P M) \Rightarrow (P N) : *_p \eta \\
\eta &\in \text{Dom}([\Gamma \vdash M =_\sigma N : *_p]) \rightarrow \{0, 1\}
\end{align*}$$

This means that $\Gamma \vdash N =_p M : *_p \eta = 1 \iff (\Gamma \vdash N : \sigma \eta = \Gamma \vdash M : \sigma \eta$. The interpretation of Leibniz' equal of programs is 1 iff they interpretations in the $\lambda\omega_\sigma$-model are equal - and that

$$\begin{align*}
([ =_\lambda \Pi \alpha \rightarrow \alpha \rightarrow *_p] = \lambda \alpha \in \text{Km}_\omega \alpha \lambda X, Y \in \text{Dom}_\alpha \left\{ \begin{array}{ll}
1, & \text{if } X = Y \\
0, & \text{if } X \neq Y
\end{array} \right.
\end{align*}$$
So Leibniz' equality, as defined in definition 3.21, is interpreted as equality in the $\lambda\omega_2$-submodel, irrespective of the $\lambda\omega_2$-model we choose. This means that axioms for Leibniz' equality of programs can be safely added on the basis of equalities that hold in the $\lambda\omega_2$-model. For example, because equality of programs in any $\lambda\omega_2$-environment model is extensional, it can be assumed that Leibniz' equality is extensional. This is stated in the corollary below.

6.49 COROLLARY (Consistency of AXIOM – classical logic and extensionality in $\lambda\omega_2$). False is not provable in a context containing only axioms from AXIOM.

PROOF. $[\text{False} : \ast_n] = 0$, so by corollary 6.47 we only have to show that there is an environment $\eta$ that satisfies this context. The only possible value that an environment can assign to the variables classic, EXT and EXT' is the element of 1. That this environment satisfies the context above follows from the fact that all the propositions assumed in this context are interpreted as 1. \qed
Chapter 7

Simple Extensions

This chapter treats the semantics of the systems introduced in chapter 4, i.e. of the
programming language $\lambda\omega^+_s$ and of the associated programming logic $\lambda\omega^+_s$. It builds on the
model definitions given in the previous chapter for the programming language $\lambda\omega_s$ and the
programming logic $\lambda\omega_L$. These models will be extended to provide interpretations for the
$\rightarrow$, $\times$- and $\Sigma$-types and their inhabitants.

As in the previous chapter, only the systems without the definition mechanism have
to be considered. Any model for a PTSs with $\rightarrow$, $\times$ and $\Sigma$ also provides a model for the
corresponding DPTSs with $\rightarrow$, $\times$ and $\Sigma$, because, by theorem 4.20 – elimination of definitions
for DPTSs with $\rightarrow$, $\times$, and $\Sigma$ – a term in a system with definitions can be interpreted by first
eliminating all definitions, and then interpreting the resulting term in the system without
definitions.

The structure of this chapter closely follows that of chapter 6.

In section 7.1 we give a general model definition for $\lambda\omega^+_s$, which is an extension of the one
for $\lambda\omega_s$ given in the section 6.1. It is not difficult to incorporate the datatype-constructors
$\rightarrow$ and $\times$ in the general model definition for $\lambda\omega_s$. The datatype-constructor $\Sigma$ turns out to
be slightly more complicated

In section 7.2 it is shown that the PER-model for $\lambda\omega^+_s$, which is an extension of the
PER-model for $\lambda\omega_s$ given in section 6.2 – is an instantiation of this general model definition.

In section 7.3 it is shown how the proof-irrelevance model for $\lambda\omega_L$ given in section 6.4
can be extended to provide a model for $\lambda\omega^+_L$. This model proves consistency of $\lambda\omega^+_L$ and
consistency of the axioms in $AXIOM^+$.

7.1 General Model Definition for $\lambda\omega^+_s$

The general model definition for $\lambda\omega_s$ given in section 6.1 will be extended to provide a general
model definition for $\lambda\omega^+_s$. As in $\lambda\omega_s$, in $\lambda\omega^+_s$ we can first consider the semantics of kinds, then
the semantics of the datatype-constructors, and then finally the semantics of the programs.

To obtain a $\lambda\omega^+_s$-model from a given $\lambda\omega_s$-model, it has to be extended to provide interpreta-
tions for the new datatypes and for the new programs. In the general model definition
for $\lambda\omega_s$ the functions $\rightarrow$ and $\Pi$ give the meaning of $\rightarrow$- and $\Pi$-datatypes. We now
also need functions $\rightarrow$, $\times$ and $\Sigma$ to give the meaning of $\rightarrow$, $\times$- and $\Sigma$-datatypes. In
the general model definition for $\lambda\omega_s$, the domain equations for $\rightarrow$- and $\Pi$-datatypes provide
the interpretation for the associated program constructions. We now have to include domain equations for the \( + \), \( \times \) and \( \Sigma \)-datatypes.

**The semantics of the kinds and datatype-constructors**

As far as the kinds are concerned, there is no difference between \( \lambda \omega \) and \( \lambda \omega \)'. So, as for the \( \lambda \omega \)-model, a family of sets \( \text{Kind} = \{ \text{Kind}_K \mid K \text{ is a kind} \} \) is needed

The new datatype-constructors are given by the definition of Cons in definition 4.28 (We will not bother with the subscript \( * \) of the empty product \( \Pi(\_\_ \_ \_ ) \) and the empty sum \( \Sigma(\_\_ \_ \_ \_ ) \) here.) Functions \( \begin{array}{c} + \\ \times \\ \Sigma \end{array} \) are needed to give the meaning of the new datatype-constructors. The function \( \Sigma \) will have the same type as \( \begin{array}{c} \Pi \\ \Sigma \end{array} \), i.e

\[
\Sigma \in \prod_{K \in \text{Kind}} (\text{Kind}_K \rightarrow \text{Kind}_* ) \rightarrow \text{Kind}_*
\]

Using \( \Sigma \), the meaning of a datatype \( (\Sigma: \mathbb{K}' \rightarrow \sigma) \) in \( \text{Kind}_* \) is defined in terms of the function in \( \text{Kind}_K \rightarrow \text{Kind}_* \) that maps all possible interpretations of \( \sigma \) to the resulting meaning of \( \sigma \).

The meanings of datatypes \( \Pi(l_1 \sigma_1, \ldots, l_n \sigma_n) \) and \( \Sigma(l_1 \sigma_1, \ldots, l_n \sigma_n) \) in \( \text{Kind}_* \) are defined in terms of the function in \( \{l_1, \ldots, l_n\} \rightarrow \text{Kind}_* \) that maps a label \( l \) to the meaning of \( \sigma \).

For this functions \( \begin{array}{c} \times \\ + \end{array} \) are used, with

\[
\begin{array}{c} \times \\ + \end{array} \in \prod_{L \in \text{Label}} (L \rightarrow \text{Kind}_* ) \rightarrow \text{Kind}_*
\]

This results in the following definition for an environment frame for the constructors:

**7.1 DEFINITION.** An environment frame for the datatype-constructors of \( \lambda \omega \) is a 7-tuple \( (\text{Kind}, \Phi_{\text{Kind}}, \begin{array}{c} \longrightarrow \\ \Pi \\ \Sigma \\ \times \\ + \end{array}, \), where

- \( \text{Kind} = \{ \text{Kind}_K \mid K \text{ is a kind} \} \) is a family of sets, indexed by kinds
- \( \Phi_{\text{Kind}} = \{ \phi_{K_1, K_2} \mid K_1 \rightarrow K_2 \text{ is a kind} \} \) is a family of bijections with \( \phi_{K_1, K_2} \in \text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2} \), where the square brackets denote some subset of the function space.
- \( \begin{array}{c} \longrightarrow \\ \Pi \\ \Sigma \\ \times \\ + \end{array} \in \prod_{\text{Label}} (\text{Label} \rightarrow \text{Kind}_* ) \rightarrow \text{Kind}_*
\)

We will not distinguish the term \( (l_1, \ldots, l_n) : \text{Label} \) and the set \( \{l_1, \ldots, l_n\} \subseteq \mathbb{L} \). Beware that \( \vdash L : \text{Label} \) is not equivalent with \( L \subseteq \mathbb{L} \), because if \( L \subseteq \mathbb{L} \) is an infinite set, then we do not have \( \vdash L : \text{Label} \). The fact that labelled cartesian products and disjoint sums are finite products and sums may be needed for certain model constructions.

Defining the interpretation of the new datatype-constructors using \( \begin{array}{c} \longrightarrow \\ \times \\ \Sigma \end{array} \) is straightforward:
7.1 GENERAL MODEL DEFINITION FOR $\lambda\omega_5^+$

7.2 DEFINITION (semantics of $\lambda\omega_5^+$-datatype-constructors).
For datatype-constructors $\sigma$ we define $[\Gamma \vdash \sigma : \mathcal{K}]\eta$ for all $\Gamma \vdash \sigma : \mathcal{K}$ and $\eta \models \Gamma^\sigma$ by induction on the derivation of $\Gamma \vdash \sigma : \mathcal{K}$ as in definition 6.3, extended with the following clauses:

- $\Gamma \vdash \Pi(l_1 : \sigma_1, \ldots, l_n : \sigma_n) : \ast_\sigma \eta \equiv \boxed{x}(l_1, \ldots, l_n) \in \{l_1, \ldots, l_n\}$. $\lbrack \Gamma \vdash \sigma_1 : \ast_\sigma \eta \rbrack_\Pi(l_1 : \sigma_1, \ldots, l_n : \sigma_n)$
- $\Gamma \vdash \Sigma(l_1 : \sigma_1, \ldots, l_n : \sigma_n) : \ast_\sigma \eta \equiv \boxed{+}(l_1, \ldots, l_n) \in \{l_1, \ldots, l_n\}$. $\lbrack \Gamma \vdash \sigma_1 : \ast_\sigma \eta \rbrack_\Sigma(l_1 : \sigma_1, \ldots, l_n : \sigma_n)$
- $\Gamma \vdash (\Sigma\alpha)\mathcal{K} : \sigma_\ast \eta \equiv \sum{\mathcal{K}}(\alpha \in \text{Kind}_{\mathcal{K}} \lbrack \Gamma, \alpha : \mathcal{K} \vdash \sigma : \ast_\sigma \eta[\alpha := a] \rbrack)$

For $\boxed{x}$ and $\boxed{+}$ the same notational conventions are used as for their syntactical counterparts. For instance, we write $S_1 \boxed{\times} S_2$ for $\boxed{\prod}(l_1 \in \{1, 2\}, S_i)$. The indices $L$ and $\mathcal{K}$ of $\boxed{\times}$, $\boxed{+}$, $\prod\mathcal{K}$ and $\sum\mathcal{K}$ will usually be omitted.

As in chapter 6, the meaning of some datatype-constructor may be undefined for a given frame, for exactly the same reason, namely that the range of the $\Phi_{K_1 \rightarrow K_2}$ may not be large enough. Analogous to definition 6.4 we define:

7.3 DEFINITION. An environment model for the datatype-constructors of $\lambda\omega_5^+$ is an environment frame for the datatype-constructors of $\lambda\omega_5^*$ for which $[\Gamma \vdash \sigma : \mathcal{K}] \eta$ is defined for all $\Gamma \vdash \sigma : \mathcal{K}$ and $\eta \models \Gamma^\sigma$.

The properties proved in chapter 6 for arbitrary environment models for the datatype-constructors of $\lambda\omega_5$ also hold for arbitrary environment models for the datatype-constructors of $\lambda\omega_5^+$:

7.4 LEMMA (Soundness of type assignment for datatype-constructors).
If $\Gamma \vdash \sigma : \mathcal{K}$ then $[\Gamma \vdash \sigma : \mathcal{K}] \eta \in \text{Kind}_{\mathcal{K}}$ for all $\eta \models \Gamma^\sigma$.

7.5 LEMMA (Soundness of $\beta$-conversion for datatype-constructors).
Suppose $\Gamma \vdash \sigma : \mathcal{K}$, $\Gamma \vdash \sigma' : \mathcal{K}$, and $\sigma \simeq_\beta \sigma'$.
Then $[\Gamma \vdash \sigma : \mathcal{K}] \eta = [\Gamma \vdash \sigma' : \mathcal{K}] \eta$ for all $\eta \models \Gamma^\sigma$.

These lemmas can be proved in exactly the same way as in chapter 6.

The semantics of programs

Upgrading the general model definition from $\lambda\omega_5$ to $\lambda\omega_5^+$ is more work for the programs than it is for the datatype-constructors. The general model definition for $\lambda\omega_5$ includes domain equations for $\rightarrow$- and $\Pi$-types:

$$\text{Dom}[\Gamma \vdash \sigma : \ast_\rightarrow] \eta \equiv \left[ \text{Dom}[\Gamma \vdash \sigma : \ast] \eta \rightarrow \text{Dom}[\Gamma \vdash \sigma : \ast] \eta \right]$$

$$\text{Dom}[\Gamma \vdash (\Pi\alpha)\mathcal{K} : \ast_\Pi] \eta \equiv \left[ \prod_{\alpha \in \text{Kind}_{\mathcal{K}}} \text{Dom}[\Gamma, \alpha : \mathcal{K} \vdash \sigma : \ast_\sigma \eta[\alpha := a]] \right]$$

with the square brackets denoting some subset of the function space and the generalised Cartesian product. Now for $\lambda\omega_5^+$ domain equations have to be added for the $+$- and $\ast$-
\( \Sigma \)-types, in such a way that the new program constructions for inhabitants of these types can be interpreted using the associated isomorphisms.

For \( + \) and \( \times \)-types, the following equations have to be satisfied:

\[
\text{Dom}(\text{A} \rightarrow \text{B})_\eta = \coprod_{x \in \text{Dom}(\text{A})_\eta} \text{Dom}(\text{B})_\eta
\]

\[
\text{Dom}(\Sigma A x_0 \ldots x_n)_\eta = \coprod_{t \in \{1, \ldots, n\}} \text{Dom}(\Sigma A x_0 \ldots x_n)_\eta
\]

The function mapping \( l_i \) to \( \sigma_i \) is left implicit. In the right-hand sides \( \prod \) is the generalised cartesian product, and \( \Sigma \) is the set-theoretic discriminated union, i.e

\[
\sum_{l_i \in \{1, \ldots, n\}} \text{Dom}(\Sigma \sigma_i x_0 \ldots x_n)_\eta = \coprod_{l_i \in \{1, \ldots, n\}} \{(l_i, x) \mid x \in \text{Dom}(\Sigma \sigma_i x_0 \ldots x_n)_\eta\}
\]

Note that the whole cartesian product and discriminated union are used in the domain equations for product and sum types. This in contrast to the domain equations for function type and polymorphic types, where we are content with subsets of the function space and generalised product. This is because some functions in \( \text{Dom}_x \rightarrow \text{Dom}_z \) or \( \prod_{t \in \text{Kind}_x} \text{Dom}_t \) may not be \( \lambda \)-definable, whereas all tuples in \( \prod_{t \in \{1, \ldots, n\}} \text{Dom}_t \) and all injections into \( \sum_{l_i \in \{1, \ldots, n\}} \text{Dom}_i \) are \( \lambda \)-definable (or rather, \( (,) \)-and \( -\)-indefinable).

Given the interpretation of datatypes in definition 7.2, the domain equations for \( + \) and \( \times \)-types are

\[
\text{Dom}(L F) \equiv \prod_{i \in L} \text{Dom}(i) \quad \text{and} \quad \text{Dom}(L F) \equiv \sum_{i \in L} \text{Dom}(i)
\]

for all \( L \subseteq \text{L} \) and \( F : L \rightarrow \text{Kind}_x \). For the binary products and sums we get

\[
\text{Dom}(\text{A} \times \text{B})_\eta = \text{Dom}(\text{A})_\eta \times \text{Dom}(\text{B})_\eta
\]

\[
\text{Dom}(\text{A} + \text{B})_\eta = \text{Dom}(\text{A})_\eta + \text{Dom}(\text{B})_\eta
\]

where in the right hand sides \( \times \) and \( + \) are the set-theoretic cartesian product and discriminated union, i.e

\[
\text{Dom}(\text{A} \times \text{B})_\eta + \text{Dom}(\text{A} \times \text{B})_\eta = \{(1, x) \mid x \in \text{Dom}(\text{A})_\eta\} \cup \{(2, x) \mid x \in \text{Dom}(\text{B})_\eta\}
\]

For the empty product \( \Pi() \) and the empty sum \( \Sigma() \) we get that \( \text{Dom}(\Pi \sigma \ldots) \) is isomorphic to a singleton set and \( \text{Dom}(\Sigma \sigma \ldots) \) is isomorphic to the empty set.

For the \( \Sigma \)-types the situation is more complicated. An obvious choice for the domain equation for a \( \Sigma \)-type is one similar to the one for a \( + \)-type:

\[
\text{Dom}(\Sigma \sigma \ldots)_\eta = \prod_{x \in \text{Kind}_x} \text{Dom}(\Sigma \sigma \ldots)_\eta[0 = a]
\]

The associated isomorphism can indeed be used to interpret the program constructions for \( \Sigma \)-types. With the isomorphism from left to right, the \( \text{Dom}(\Sigma \sigma \ldots)_\eta \) can be defined in terms of the meanings of \( \tau \) and \( N \). With the isomorphism from left to right, the meaning of \( \nabla f \) (\( \Sigma \sigma : \text{K} \rightarrow \rho \)) can be defined in terms of the meaning of \( f : \Pi \sigma : \text{K} \rightarrow \rho \). However,
the domain equation above requires more than is strictly necessary. Only an interpretation for a weak \( \Sigma \)-type is needed, and (i) provides the interpretation of the strong \( \Sigma \)-type (discussed in section 4.1.3). This is because (i) inhabitants of a \( \Sigma \)-type are interpreted as pairs consisting of a datatype-constructor and a program, and the associated isomorphism can be used to define projections \( \pi_1 \) and \( \pi_2 \) such that

\[
\begin{align*}
\pi_1([ \Gamma \vdash (\eta_{\Sigma[K]} \sigma \tau N) : (\Sigma:\alpha[K] \sigma)] \eta) &= \left[ \Gamma \vdash \tau : \ast_{\Sigma} \right] \eta \\
\pi_2([ \Gamma \vdash (\eta_{\Sigma[K]} \sigma \tau N) : (\Sigma:\alpha[K] \sigma)] \eta) &= \left[ \Gamma \vdash N : \tau[\alpha := \tau] \right] \eta
\end{align*}
\]

Indeed, in PER-model equation (i) cannot be solved, as can be shown by considering the cardinality of the two sides. Take \( K \equiv \ast \). Then the left hand side of (i) is a set of equivalence classes in a partial equivalence relation on \( \mathbb{N} \), and the right hand side is a set of pairs consisting of a partial equivalence relation on \( \mathbb{N} \) and an equivalence class.

Later, in chapter 8, a model will be given that does satisfy (i) (but in which the \( \text{Dom_a} \) are cpos instead of sets).

We will give another domain equation for \( \Sigma \)-types. This domain equation does not require that \( \text{Dom}_{[\Sigma,F]} \) is isomorphic to \( \sum \text{Dom}_{[\Sigma,F]} \), but only requires that it is isomorphic to \( \sum \text{Dom}_{[\Sigma,F]} \) modulo some equivalence relation \( \cong \). The following example explains the idea behind this.

7.6 EXAMPLE. Suppose \( (\eta_{\Sigma[K]} \sigma \tau N) \) and \( (\eta_{\Sigma[K]} \sigma \tau' N') \) are of type \( (\Sigma: \alpha[K] \sigma) \), and suppose for all functions \( f : (\Pi\sigma[K] \sigma \rightarrow \rho) \) and all environments \( \eta \) that satisfy the context

\[
[ \Gamma \vdash f \tau N : \rho ] \eta = [ \Gamma \vdash f \tau' N' : \rho ] \eta .
\]

Then the model does not have to distinguish \( (\eta_{\Sigma[K]} \sigma \tau N) \) and \( (\eta_{\Sigma[K]} \sigma \tau' N') \), because there is no function \( \forall f : (\Sigma: \alpha[K] \sigma) \rightarrow \rho \) that can distinguish them. We will define a relation \( \cong \) on \( \sum \text{Dom}_{[\Sigma,F]} \) so that the pairs \( ([\tau], [\tau]) \) and \( ([\tau'], [\tau']) \) are in this relation.

To define \( \cong \) the following notation is used.

7.7 DEFINITION \( (\text{App}(\_,\_)) \)

For \( f \in \text{Dom}_{[\Pi,K]} \) and \( a \in \text{Kind}_K \) we write \( \text{App}(f,a) \) for \( \phi_{\Pi}^F(f)(a) \).

For \( f \in \text{Dom}_{a \boxplus b} \) and \( \xi \in \text{Dom}_a \) we write \( \text{App}(f,\xi) \) for \( \phi_\alpha^{-b}(f)(\xi) \).

So for example \( [\Gamma \vdash f \tau N : \rho] \eta = \text{App}(\text{App}([\Gamma], [\tau]), [N]) \), for \([\_]\) as defined in definition 6 32

7.8 DEFINITION \( (\cong) \)

Let \( F \in \text{Kind}_K \rightarrow \text{Kind}_* \). The relation \( \cong \) on \( \sum \in \text{Kind}_K \) \( \text{Dom}_{[\Sigma,F]} \) is then defined as follows:

\[
\begin{align*}
(a, \chi) \cong (a', \chi') \iff \\
\text{App}(\text{App}(f,a), \chi) = \text{App}(\text{App}(f,a'), \chi')
\end{align*}
\]

for all \( b \in \text{Kind}_* \) and all \( f \in \text{Dom}_{[\Pi,K] \rightarrow \text{Kind}_K} \) \( (\lambda a \in \text{Kind}_K) F(a) \boxplus b \).
So for the datatypes $\tau, \tau', \sigma$, and the programs $N : \sigma[\alpha := \tau]$ and $N' : \sigma[\alpha := \tau']$ in example 7.6 we have $([\tau], [N]) \approx ([\tau'], [N'])$

As the domain equation for $\Sigma$-types we now take

$$\text{Dom} \left[ \Sigma \right]_{K F} \equiv \left( \sum_{a \in \text{Kind}_K} \text{Dom}_F(a) \right) / \approx .$$

The associated isomorphism can be used to interpret the program-constructions for $\Sigma$-types. With the isomorphism $\Phi^{-1}$ from right to left the meaning of $(\sigma : \Sigma) \sigma$ can be defined in terms of the meanings of $\tau$ and $N$, namely as $\Phi^{-1}$ applied to the equivalence class containing $([\tau], [N])$. With the isomorphism $\Phi$ from left to right the meaning of $\bigtriangledown \phi : (\Sigma:K, \sigma) \rightarrow \rho$ can be defined in terms of the meaning of $\phi : (\Pi:K, \sigma \rightarrow \rho)$, namely as the function which maps an element of $\xi \in \text{Dom} \left[ \Sigma \right]_{K F}$ to the result of applying $[\phi]$ to $a$ and $\chi$, where $(a, \chi)$ is any element in the equivalence class $\Phi(\xi)$. The definition of $\approx$ guarantees that it does not matter which $(a, \chi) \in \Phi(\xi)$ we choose.

With the domain equations for $+$, $\times$ and $\Sigma$ that have been given, the $\lambda \omega^+_t$-environment frames are now defined as follows

7.9 Definition ($\lambda \omega^+_t$ environment frame).
An environment frame for $\lambda \omega^+_t$ is a tuple $(KIND, \text{Dom}, \Phi^-, \Phi^\Pi, \Phi^\Sigma, \Phi^+)$, where

- $KIND = (\text{Kind}, \Phi_{\text{Kind}}, \rightarrow, [\Pi], [\Sigma], [\times], [+])$ is an environment model for the $\lambda \omega^+_t$-constructors.
- $\text{Dom} = \{ \text{Dom}_a | a \in \text{Kind}_* \}$ is a family of sets
- $\Phi^-$ and $\Phi^\Pi$ are as in definition 6.9, $\Phi^\Sigma = \{ \Phi^\Sigma_F \mid K, \square, F \in \text{Kind}_K \rightarrow \text{Kind}_* \}$, $\Phi^\times = \{ \Phi^\times_F \mid L, \square, F \in L \rightarrow \text{Kind}_* \}$, and $\Phi^+ = \{ \Phi^+_F \mid L, \square, F \in L \rightarrow \text{Kind}_* \}$ are families of bijections with

  \[
  \begin{align*}
  \Phi^\Sigma_F & \subseteq \text{Dom} [\Sigma] \rightarrow \left( \sum_{a \in \text{Kind}_K} \text{Dom}_F(a) \right) / \approx \ \\
  \Phi^\times_F & \subseteq \text{Dom} [\times] \rightarrow \prod_{\xi \in [\Sigma]} \text{Dom}_F(\xi) \ \\
  \Phi^+_F & \subseteq \text{Dom} [+] \rightarrow \sum_{\xi \in [\Sigma]} \text{Dom}_F(\xi)
  \end{align*}
  \]

We now define the interpretation of the new program constructions in $\lambda \omega^+_t$ with respect to an arbitrary environment frame. This simply amounts to inserting the right isomorphisms in the right places, but some of the results look very disappointingly. In example 7.11 the interpretations of case and abstype are given, which are a lot easier to read than the interpretation of $\bigtriangledown$ for $+\cdot$- and $\Sigma$-datatypes.

7.10 Definition (semantics of $\lambda \omega^+_t$-programs)
For a $\lambda \omega^+_t$-program $M$ with $\Gamma \vdash M : \sigma$, $[\Gamma \vdash M : \sigma]$ is defined for $\eta \equiv \Gamma^{\downarrow \cdot \top}$ by induction on the derivation of $\Gamma \vdash M : \sigma$, as in definition 6.32, extended with the clauses given below.
For $x$-types.

Let $\sigma = \Pi(l_1: \sigma_1, \ldots, l_n: \sigma_n)$ and $F = (\lambda l_i \in \{l_1, \ldots, l_n\}: \Gamma \vdash \sigma_i : \star) \eta$. Then we define

$$[\Gamma \vdash \langle l_1 \rightarrow M_1, \ldots, l_n \rightarrow M_n \rangle : \sigma \eta] = \Phi_1^{-1}(l_i, [\Gamma \vdash M_i : \sigma_i] \eta)$$

$$[\Gamma \vdash \Pi(l_1: \sigma_1, \ldots, l_n: \sigma_n) : \sigma \eta] = (\Phi_1 \mid \Gamma \vdash \sigma \star) \eta) l_i$$

where $\Phi_1^x \in \text{Dom}_{\Gamma} F \rightarrow \prod_i \text{Dom}_{\Gamma(i)} F$.

For $\ast$-types.

Let $\sigma = \Sigma(l_1: \sigma_1, \ldots, l_n: \sigma_n), F = (\lambda l_i \in \{l_1, \ldots, l_n\}: \Gamma \vdash \sigma_i : \star) \eta, R = [\Gamma \vdash \rho : \star] \eta,$ and $G = (\lambda l_i \in \{l_1, \ldots, l_n\}: F(l) \rightarrow R)$. So $[\Sigma] F = [\Gamma \vdash \sigma \star] \eta,$ and $[\prod] F = [\Gamma \vdash \Pi(l_1: \sigma_1 \rightarrow \rho_1, \ldots, l_n: \sigma_n \rightarrow \rho) : \sigma \star] \eta$.

Then we define

$$[\Gamma \vdash \Pi \in \sigma, l_i : M_i : \sigma \eta] = \Phi_1^{-1}(l_i, [\Gamma \vdash M_i : \sigma_i] \eta)$$

$$[\Gamma \vdash \bigvee f : \sigma \rightarrow \rho \eta] = \Phi_2^{-1}(\lambda \xi \in \text{Dom}_{\Gamma} E, F \rightarrow \prod_i \text{Dom}_{\Gamma(i)} F, \Phi_4((\Phi_3 \mid \Gamma \vdash f : \Pi(l_1: \sigma_1 \rightarrow \rho_1, \ldots, l_n: \sigma_n \rightarrow \rho) : \sigma \star) \eta) l_i \chi)$$

where $\Phi_1^\ast \in \text{Dom}_{\Gamma} F \rightarrow \sum_i \text{Dom}_{\Gamma(i)} F$

$$(l, \chi) = \Phi_1(\lambda \xi \in \text{Dom}_{\Gamma} E, F \rightarrow \prod_i \text{Dom}_{\Gamma(i)} F, \Phi_4((\Phi_3 \mid \Gamma \vdash f : \Pi(l_1: \sigma_1 \rightarrow \rho_1, \ldots, l_n: \sigma_n \rightarrow \rho) : \sigma \star) \eta) l_i \chi)$$

For $\Sigma$-types.

Let $F = (\lambda a \in \text{Kind}_K : [\Gamma, \alpha : K, \sigma : \star] \eta, a) G = (\lambda a \in \text{Kind}_K : F(a) \rightarrow R)$. So $[\Sigma] F = [\Gamma \vdash \sigma \star] \eta, \text{and} \prod G = [\Gamma \vdash \Pi(\lambda \alpha : K, \sigma : \star) : \sigma \star] \eta$. Then we define

$$[\Gamma \vdash \Pi a : \sigma \rightarrow N : (\Sigma \alpha : K, \sigma) \eta] = \Phi_1^{-1}(\lambda \xi \in \text{Dom}_{\Gamma} E, F \rightarrow \prod_i \text{Dom}_{\Gamma(i)} F, \Phi_4((\Phi_3 \mid \Gamma \vdash f : \Pi(\lambda \alpha : K, \sigma : \star) \eta) a) \chi)$$

where $\Phi_1^\Sigma \in \text{Dom}_{\Gamma} F \rightarrow (\Sigma a \text{Dom}_{\Gamma(a)})/\approx$

$$(a, \chi) = \Phi_1^\Sigma(\lambda \xi \in \text{Dom}_{\Gamma} E, F \rightarrow \prod_i \text{Dom}_{\Gamma(i)} F, \Phi_4((\Phi_3 \mid \Gamma \vdash f : \Pi(\lambda \alpha : K, \sigma : \star) \eta) a) \chi)$$

The definition of $\approx$ guarantees that it does matter which $(a, \chi) \in \check{\Phi}_1(\xi)$ is chosen. $\square$
CHAPTER 7. SIMPLE EXTENSIONS

7.11 EXAMPLE The interpretations of the case- and abstype-constructions are easier to read then the interpretation of \( \triangledown \) given above. Definition 7.10 gives the following interpretation for case:

\[
\begin{align*}
\Gamma \vdash (\text{case } N \text{ of in } 1 \; x \rightarrow M_1 \mid \ldots \mid \text{in } 2 \; y \rightarrow M_n \; \text{esac}) \cdot \tau \eta \\
= \Gamma \vdash \triangledown(l_1 \rightarrow (\lambda x \cdot \sigma_1, M_1), \ldots, l_n \rightarrow (\lambda x \cdot \sigma_n, M_n)) \; N : \tau \eta \\
= \Gamma, x : \sigma_i \vdash M_i \cdot \tau \eta[x = \lambda] \quad \text{by definition of case}
\end{align*}
\]

where \((l_i, \chi) = \Phi^\triangledown_F(\Gamma \vdash N : \Sigma(l_1, \sigma_1, \ldots, l_n, \sigma_n) \eta) \in \sum \text{Dom}_F(D), \) with

\( F = (\mathcal{A}_x \in \{l_1, \ldots, l_n\}; \eta) \)

As the interpretation of abstype we get

\[
\begin{align*}
\Gamma \vdash (\text{abstype } \alpha \in \mathcal{K} \text{ with } x : \sigma \text{ is } N \text{ in } M) \cdot \rho \eta \\
= \Gamma \vdash \triangledown(\lambda x \cdot \mathcal{K}_\alpha \times \tau \sigma \cdot M) \cdot N : \rho \eta \\
= \Gamma, x : \sigma \vdash M \cdot \rho \eta[x = \alpha][x = \chi] \quad \text{by definition of abstype}
\end{align*}
\]

where \((\alpha, \chi) \) is any element of the \( \equiv \)-equivalence class \( \Phi_F^\triangledown(\Gamma \vdash N : (\Sigma \alpha \mathcal{K}_\alpha \sigma)) \eta), \) with

\( F = (\mathcal{A}_x \in \text{Kind}_\mathcal{K}; \eta) \)

Analogous to definition 6.12 we define:

7.12 DEFINITION A \( \lambda \omega^+ \)-environment model is a \( \lambda \omega^+ \)-environment frame which \( \Gamma \vdash M : \sigma \eta \) is defined for all \( \Gamma \vdash M : \sigma \) and \( \eta = \Gamma^\triangledown \).

If a \( \lambda \omega^+ \)-frame is not a \( \lambda \omega^+ \)-model - i.e. the meaning of some program is not defined for the frame - then the reason is that the range of one of the bijections \( \Phi_{\alpha, b}^\triangledown \) or \( \Phi_{\rho}^\triangledown \) is too small. The reason cannot be that the range of a bijection \( \Phi_{\alpha}^\triangledown \) or \( \Phi_{\rho}^\triangledown \) is too small. This is because the ranges of these bijections are prescribed by the definition of an environment frame, whereas all subsets of \( \text{Dom}_{\alpha} \rightarrow \text{Dom}_{\rho} \) and \( \prod_{\alpha} \text{Dom}_{\alpha, \eta} \) are allowed as the range of \( \Phi_{\alpha, b}^\triangledown \) and \( \Phi_{\rho}^\triangledown \).

The properties proved in chapter 6 for arbitrary environment models of \( \lambda \omega \) also hold for arbitrary environment models of \( \lambda \omega^+ \), and they can be proved in exactly the same way as in chapter 6.

7.13 LEMMA. Let \( \text{der}_1 \) and \( \text{der}_2 \) be derivations of \( \Gamma \vdash M : \sigma \) and \( \Gamma \vdash M : \sigma' \), respectively. Then

\[
\Gamma \vdash \text{der}_1 [\Gamma \vdash M : \sigma] \eta = \Gamma \vdash \text{der}_2 [\Gamma \vdash M : \sigma'] \eta,
\]

where the meanings are defined using \( \text{der}_1 \) and \( \text{der}_2 \) respectively.

\( \square \)

7.14 LEMMA (Soundness of type assignment for programs)
If \( \Gamma \vdash M : \sigma \) then \( \Gamma \vdash M : \sigma \eta \in \text{Dom}_\eta^{\triangledown} \) for all \( \eta = \Gamma^\triangledown \).

\( \square \)

7.15 LEMMA (Soundness of \( \beta \)-conversion for programs).
Suppose \( M \equiv \beta M' \), \( \Gamma \vdash M : \sigma \), and \( \Gamma \vdash M' : \sigma' \).
Then \( \Gamma \vdash M : \sigma \eta = \Gamma \vdash M' : \sigma' \eta \) for all \( \eta = \Gamma^\triangledown \).

\( \square \)
7.2 PER-Model for $\lambda \omega^+$

In section 6.2 it was shown that the PER-model for $\lambda \omega$ is an instantiation of the general model definition for $\lambda \omega$. We will now give a PER-model for $\lambda \omega^+$ that extends the one given in section 6.2 and show that is an instantiation of the general model definition for $\lambda \omega^+$.

Most accounts of PER-models in the literature only deal with --> and II-types. In [Gir72] interpretations for +-, x- and $\Sigma$-types in the PER-model can be found. The interpretation of x-types given below is different from the one given there. This is because we have n-ary product types and not just binary ones.

For the sake of simplicity we assume that all labels are of the form $l_n$ for some $n \in \mathbb{N}$. This gives us an easy way to associate a unique natural number with every label.

The interpretations $\langle \rightarrow \rangle$ and $\Pi$ of --> and II are defined as in definition 6.19. To interpret the + types we need a pairing function and projections.

7.16 Definition. The function $\langle \_ \_ \rangle : \mathbb{N}^2 \rightarrow \mathbb{N}$ is a recursive surjective pairing function and $\pi_1(\_ \_), \pi_2(\_ \_) : \mathbb{N} \rightarrow \mathbb{N}$ are the associated projection functions.

The datatype-constructors +, x and $\Sigma$ are now interpreted as the following functions:

7.17 Definition.

1. The functions $\times, + \in \Pi_{L \subseteq \text{labels}} (L \rightarrow \text{PER}) \rightarrow \text{PER}$ are defined by

$$\times \langle \_ \_ \rangle \equiv \{(n, n') \in \mathbb{N}^2 \mid (n, n', j) \in F(l_j) \text{ for } l_j \in L\}$$

$$\langle + \_ \rangle \equiv \{(n, n') \in \mathbb{N}^2 \mid (n, n', j) \in F(l_j) \text{ for } l_j \in L\}$$

for all $F : L \rightarrow \text{PER}$.

2. The function $\sum \in \Pi_{K \subseteq \text{context}} (K \rightarrow \text{PER}) \rightarrow \text{PER}$ is defined by

$$\sum \langle \_ \_ \rangle \equiv \{(n, n') \in \mathbb{N}^2 \mid (n, n'), (n', n') \in \bigcup_{a \in K \text{context}} F(a) \text{, and } (f, n, f', n') \in R \text{ for all } R \in \text{PER and } (f, f') \in \Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle \}$$

for all $F : K \rightarrow \text{PER}$.

In the definition of $\sum$ we can recognize the definition of $\approx$. If $(n, n) \in F(a)$, $(n', n') \in F(a')$, and $(f, f') \in \Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle$, and we define

$$[f] \equiv \langle f \rangle_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle} \in \text{Dom}_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle}$$

$$\langle f \_ \rangle \equiv \langle f \rangle_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle} \in \text{Dom}_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle}$$

$$\langle [f] \_ \rangle \equiv \langle [f] \rangle_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle} \in \text{Dom}_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle}$$

$$\langle [n] \_ \rangle \equiv \langle [n] \rangle_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle} \in \text{Dom}_{\Pi \langle \lambda a \in K \text{context} : F(a) \rightarrow \text{PER} \rangle}$$

then $(f \cdot n, f' \cdot n') \in R$ is equivalent with $\text{App}(\text{App}([f], a), [n]) = \text{App}(\text{App}([f'], a'), [n'])$. 


7.18 Lemma. $\text{KIND}_{HFO} = \langle \text{Kind } \Phi_{\text{KIND}}(\rightarrow), \prod, \Sigma, \times, + \rangle$ is an environment model for the $\lambda\omega^+_s$ datatype-constructors.

The sets $\text{Dom}_K$ and the bijections $\Phi^-_K, \Phi^+_K, \Phi^\pi_K$ are defined as in definition 6.21. To complete the model bijections $\Phi^L_K, \Phi^H_K$ and $\Phi^\pi_K$ are needed:

7.19 Definition. For $G \in L \rightarrow \text{PER}$, and $F \in \text{Kind}_K \rightarrow \text{PER}$, the functions

$$
\Phi^L_K \in \text{Dom}_K \rightarrow \prod_{i \in L} \text{Dom}_K(i),
$$

$$
\Phi^H_K \in \text{Dom}_K \rightarrow \Sigma_{i \in \text{Dom}_K(i)},
$$

$$
\Phi^\pi_K \in \text{Dom}_K \rightarrow (\sum_{a \in \text{Kind}_K} \text{Dom}_K(a))/\simeq
$$

are defined by

$$
\Phi^L_K(n) \cong \Phi(n)_L \cong \lambda j \in L. [n \cdot j]_{C(i)},
$$

$$
\Phi^H_K((j, n)) \cong \lambda j \in L. [j \cdot \lambda n \in \text{Dom}_K(i) \rightarrow [a, [n]_{F(a)}]_\simeq,
$$

where $a$ is an arbitrary element of $\text{Kind}_K$.

Checking that these $\Phi^L_K, \Phi^H_K$ and $\Phi^\pi_K$ are well-defined functions on equivalence classes is straightforward. For $\Phi^L_K$ - for which this is the most complicated - this amounts to proving

7.20 Lemma. Let $F \in \text{Kind}_K \rightarrow \text{PER}$. $(n, n') \in \Sigma F$ (so $[n]_{\Sigma F} = [n']_{\Sigma F}$) and let $a, a' \in \text{Kind}_K$. Then

$$
\langle a, [n]_{F(a)} \rangle \simeq \langle a', [n']_{F(a')} \rangle
$$

(and hence $\langle a, [n]_{F(a)} \rangle_\simeq = \langle a', [n']_{F(a')} \rangle_\simeq$).

Proof. To prove $(a, [n]_{F(a)}) \approx (a', [n']_{F(a')})$ we have to prove that

$$
\text{App}(\text{App}(f, a), n) = \text{App}(\text{App}(f, a'), n'),
$$

where $R \in \text{PER}$ and $f \in \text{Dom}_K(\prod_{a \in \text{Kind}_K} \text{Dom}_K(a) \rightarrow R)$. This $f$ is an equivalence class in the partial equivalence relation $\prod_{a \in \text{Kind}_K} \text{Dom}_K(a) \rightarrow R$.

Let $m \in f$, i.e.

$$
f = [m]_{\prod_{a \in \text{Kind}_K} \text{Dom}_K(a) \rightarrow R}.
$$

Then by the definitions of $\text{App}(.-, \cdot)$ (definition 7.7) and of $\Phi^L$ and $\Phi^H$ (definition 6.21) (i) is equivalent with

$$
\text{App}(\text{App}(f, a), n) \simeq \text{App}(\text{App}(f, a'), n').
$$

and by the definition of $\Sigma F$ this follows from $(n, n') \in \Sigma F$.

To prove that we have an environment frame for $\lambda\omega^+_s$ we have to show:

7.21 Lemma. The functions $\Phi^+_K, \Phi^H_K$ and $\Phi^\pi_K$ are bijections.

Proof. Easy.

Finally

7.22 Lemma. $\langle \text{KIND}_{HFO}, \text{Dom}, \Phi^-, \Phi^H, \Phi^{-}, \Phi^\pi, \Phi^\pi, \Phi^+ \rangle$ is an environment model for $\lambda\omega^+_s$

Proof. Proving that $\sum_{\Gamma \vdash M : \sigma} \eta$ is defined for all $\Gamma \vdash M$ and $\eta \vdash \Gamma^\pi \cdot \cdot$ is done exactly as in lemma 6.23.
7.3 The Programming Logic

In the previous chapter proof-irrelevance models were given for \( \lambda \omega_p \) and \( \lambda \omega_L \). In section 6.3 the general model definition for \( \lambda \omega \) was instantiated to produce a proof-irrelevance model for \( \lambda \omega_p \). In section 6.4 a proof-irrelevance model for the "logical" part of \( \lambda \omega_L \) was defined given an arbitrary model for the programming language \( \lambda \omega_p \). Extending these models to deal with the additional primitives of \( \lambda \omega_p^+ \) and \( \lambda \omega_L^+ \) is very easy, and will therefore not be treated in much detail.

7.3.1 Proof-irrelevance model for \( \lambda \omega_p^+ \)

The proof-irrelevance model for \( \lambda \omega_p \) defined in section 6.3 can be extended to a model for \( \lambda \omega_p^+ \). The new prop-constructors \( \bigwedge \), \( \bigvee \) and \( \exists \) are given their usual truth-table interpretations:

**Definition.** The functions \( \bigwedge \), \( \bigvee \) \( L \rightarrow \{0,1\} \rightarrow \{0,1\} \) and \( \exists \) \( K \rightarrow \{0,1\} \rightarrow \{0,1\} \) are defined by

\[
\bigwedge L F \equiv \bigcap_{l \in L} F(l)
\]
\[
\bigvee L F \equiv \bigcup_{l \in L} F(l)
\]
\[
\exists \ K F \equiv \bigcup_\nu P \nu F(X)
\]

where we take \( \bigcap_{l \in \emptyset} F(l) = 1 \).

However, the proof-irrelevance model for \( \lambda \omega_p^+ \) cannot meet the general model definition given in section 7.1. The problem is that this general model definition is already too specific. In particular, it requires that \( \text{Dom}_a \bigvee b \) is isomorphic with the set-theoretic disjoint union of \( \text{Dom}_a \) and \( \text{Dom}_b \). So, if \( \text{Dom}_a \) and \( \text{Dom}_b \) are singletons, \( \text{Dom}_a \bigvee b \) has to be a two-element set. This is clearly not the case in the proof-irrelevance model, because there \( \text{Dom}_a \bigvee b \) is the singleton set 1 if \( \text{Dom}_a \) and \( \text{Dom}_b \) are both singletons.

7.3.2 Proof-irrelevance model for \( \lambda \omega_L^+ \)

Let

\[ M = (\text{Kind}, \text{Dom}, \Phi^-, \Phi^+, \Phi^\text{at}, \Phi^\text{int}, \Phi^\text{dim}) \]

be any environment model for \( \lambda \omega_L^+ \). A model for \( \lambda \omega_L^+ \) can be defined in which \( M \) is the submodel for \( \lambda \omega_L^+ \). The only difference with the construction in section 6.4 are the interpretations for the new prop-constructors, which are of course interpreted as in the \( \lambda \omega_L^+ \)-model above.

The interpretation of the kinds, datatype-constructors and programs is given by \( M \), because by lemma 4.37 \( \lambda \omega_L^+ \) has the same kinds, datatype-constructors and programs as \( \lambda \omega_L^+ \).

The interpretation of the propkinds is defined as for \( \lambda \omega_L \) (definition 6.33).

The new prop-constructors are given by the definition of \( \text{PCons} \) in definition 4.34. The definition given in section 6.4 for the interpretation of prop-constructors has to be extended to deal with these new ones.
7.24 Definition (semantics of the new $\lambda\omega_2^+$-propconstructors)
For $\lambda\omega_L$-propconstructors $P$ we define $\llbracket \Gamma \vdash P : \mathcal{P} \rrbracket_\eta$ for all $\Gamma \vdash P : \mathcal{P}$ and $\eta \models \Gamma \sqcap \ast$ by induction on the derivation of $\Gamma \vdash P : \mathcal{P}$, as in definition 6.38, extended with the following clauses:

\[
\llbracket \Gamma \vdash \forall (t_1, P_1, \ldots, t_n, P_n) : *_p \rrbracket_\eta \equiv \bigcap_{i \in [1, n]} \llbracket \Gamma \vdash P_i : *_p \rrbracket_\eta
\]
\[
\llbracket \Gamma \vdash \exists (t_1, P_1, \ldots, t_n, P_n) : *_p \rrbracket_\eta \equiv \bigcup_{i \in [1, n]} \llbracket \Gamma \vdash P_i : *_p \rrbracket_\eta
\]
\[
\llbracket \Gamma \vdash (\exists x. A, P) : *_p \rrbracket_\eta \equiv \bigcup_{x \in \mathcal{A}} \llbracket \Gamma, \tau. A \vdash P : *_p \rrbracket_\eta[x := \xi]
\]

where $\bigcap_{i \in \emptyset} P_i = 1$, and $\mathcal{A} = \{ \text{Kind}_A, \text{Dom}_{[r \vdash A \ast]} \eta \}$.

Finally, as in $\lambda\omega_L$, all proof terms are interpreted as the element of 1. The properties proved in section 6.4 for the proof-irrelevance model for $\lambda\omega_L$ also hold for this proof-irrelevance model for $\lambda\omega_2^+$, and they can be proved in exactly the same way.

7.25 Theorem (Soundness)
1. Let $\text{der}_1$ and $\text{der}_2$ be derivations of $\Gamma \vdash a : A$ and $\Gamma \vdash a : A'$, respectively. Then
\[
\llbracket \Gamma \vdash a : A \rrbracket_\eta = \llbracket \Gamma \vdash a : A' \rrbracket_\eta,
\]
where the meanings are defined using $\text{der}_1$ and $\text{der}_2$ respectively.

2. If $\Gamma \vdash a : A$, then for all $\eta \models \Gamma$
\[
\llbracket \Gamma \vdash a : A \rrbracket_\eta \in \begin{cases}
\text{Kind}_A & \text{if } \Gamma \vdash A : \Box_p \\
\text{Dom}_{[r \vdash A \ast]} \eta & \text{if } \Gamma \vdash A : *_p \\
\llbracket \Gamma \vdash A : \Box_p \rrbracket_\eta & \text{if } \Gamma \vdash A : \Box_p \\
\llbracket \Gamma \vdash A : *_p \rrbracket_\eta & \text{if } \Gamma \vdash A : *_p
\end{cases}
\]

3. If $a \approx_\eta a', \Gamma \vdash a : A$, and $\Gamma \vdash a : A'$, then $\llbracket \Gamma \vdash a : A \rrbracket_\eta = \llbracket \Gamma \vdash a' : A \rrbracket_\eta$ for all $\eta \models \Gamma$.

Like the proof-irrelevance models given before, this one can be used to prove consistency.

7.26 Corollary (Consistency of $\lambda\omega_2^+$). $\lambda\omega_2^+$ is consistent (False is not provable).

7.27 Lemma (Consistency of $AXIOM^+$ in $\lambda\omega_2^+$)
False is not provable in a context containing only axioms from $AXIOM^+$.

Proof. As in the proof of lemma 6.49, we have to supply a model $\mathcal{M}$ for the programming language - the PER-model will do - and check that all axioms from $AXIOM^+$ are then interpreted as 1 ("true") in the resulting $\lambda\omega_2^+$-model.

In fact, any model $\mathcal{M}$ that meets the general model definition for $\lambda\omega_2^+$ can be used to prove consistency of all axioms in $AXIOM^+$. The domain equations ensure that the properties of $\ast$, $\times$- and $\Sigma$-datatypes given by axioms in $AXIOM^+$ are true.
Chapter 8

Recursion

In this chapter we treat the semantics of the systems introduced in chapter 5: the programming language $\lambda\omega^\mu$ and the associated programming logic $\lambda\omega^\mu$. This chapter is organised in exactly the same way as chapter 7. First, in section 8.1, a general model definition for $\lambda\omega^\mu$ is given. Then, in section 8.2, a model is constructed that is an instantiation of this general model definition. And finally, in section 8.3, this model for the programming language is combined with a proof-irrelevance model for the logic, and the resulting model is used to prove soundness of $\lambda\omega^\mu$ and soundness of all the axioms introduced in chapter 5.

The main difference between the programming language $\lambda\omega^\mu$ and the programming languages $\lambda\omega$, and $\lambda\omega^+$ is that in $\lambda\omega^\mu$ we have unrestricted recursion at both program and datatype level. Consequently, in the general model definition for $\lambda\omega^\mu$ datatypes can no longer be interpreted as sets. Instead types will be interpreted as cpos, the most commonly used semantic domains in denotational semantics. Recursive programs and recursive datatypes can then be interpreted as usual: recursive programs as least least fixpoints in cpos, and recursive datatypes as solutions of recursive domain equations.

The general model definition gives a collection of domain equations any model has to satisfy. In section 8.2 a model is constructed by solving these coupled domain equations. Because datatypes are interpreted as cpos, the standard technique can be used to do this. Earlier versions of this model construction – for different systems – are given in [EH91] and [PHE93]. Compared to the PER-models given in sections 6.2 and 7.2 this cpo-model is more in line with conventional programming language semantics, and it is compatible with conventional denotational semantics.

Given any model for $\lambda\omega^\mu$ that meets the general model definition, a proof-irrelevance model for $\lambda\omega^\mu$ can be defined. Because $\lambda\omega^\mu$ provides the same logical primitives as $\lambda\omega^+$, this is done in exactly the same way as for $\lambda\omega^+$ in section 7.3. Since datatypes are interpreted as cpos, it is easy to verify the soundness of the axioms introduced in chapter 5 for reasoning about recursive programs. This proves the consistency of $\lambda\omega^\mu$, of the context $\Gamma_{CPO}$, and of the axioms in $AXIOM^\mu$.

As in the previous chapters, only the semantics of the systems without the definition mechanism is treated. By theorems 5.8.5 and 5.12.5 – elimination of definitions for $\lambda\omega^\mu$ and $\lambda\omega^+$ – any model for $\lambda\omega^\mu$ or $\lambda\omega^+$ can also be used as an interpretation of $\lambda\omega^\mu$ or $\lambda\omega^+$.
8.1 General Model Definition $\lambda \omega_8^\mu$

In this section the general model definition for $\lambda \omega_8^+\mu$ given in section 7.1 is adapted to provide a general model definition for $\lambda \omega_8^\mu$. The main difference is that programs are now interpreted as elements of cpos rather than sets. So instead of a set, we now associate a cpo $\text{Dom}_{\{f \mapsto \sigma\}}$ with every datatype $\sigma$.

From now on, by a cpo we always mean an $\omega$-cpo.

8.1 DEFINITION. An $\omega$-cpo is a set with a partial order $\subseteq$, a least element $\bot$, and limits of all chains of the form $c_0 \subseteq c_1 \subseteq c_2 \subseteq \ldots$.

For countable sets this definition is equivalent with the one given in definition 5.13. For the general model definition it does not really matter what kind of domains are used.

New domain equations have to be included for the $\mu$-types in order to interpret the associated program construction, i.e. the functions fold and unfold.

A final difference with respect to the general model definition for $\lambda \omega_8^+$ is that the domain equations for the $\Sigma$-types now have to be changed, because the sum construction on cpos differs from the one on sets. Apart from elements constructed using the injections $\text{in}_{\mathcal{C}(\mu, A): \lambda_{\alpha, A}}$ and $\text{in}_{\mathcal{C}(\mathcal{A}, B)}$, the $\Sigma$-types now also contain bottom elements.

The semantics of the kinds and datatype-constructors

As far as the semantics of kinds and datatype-constructors is concerned, there is not much difference between $\lambda \omega_8^+$ and $\lambda \omega_8^\mu$. We only have to extend the definition of an environment frame for the datatype-constructors of $\lambda \omega_8^+$ - definition 7.1 - with a function $\mu$,

$$\mu \in \text{Kind}_* \rightarrow \text{Kind}_* \rightarrow \text{Kind}_*,$$

which gives the meaning of the new datatype-constructor $\mu$.

8.2 DEFINITION. An environment frame for the datatype-constructors of $\lambda \omega_8^\mu$ is a 8-tuple $(\text{Kind}, \Phi_{\text{Kind}}, \rightarrow, \Pi, \Sigma, \times, +, \mu)$, where

- $\text{Kind} = \{\text{Kind}_k \mid k \text{ is a kind}\}$ is a family of sets, indexed by kinds.
- $\Phi_{\text{Kind}} = \{\Phi_{K_1, K_2} \mid K_1 \rightarrow K_2 \text{ is a kind}\}$ is a family of bijections with $\Phi_{K_1, K_2} \in \text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2}$, where the square brackets denote a function space.
- $\rightarrow \in \text{Kind}_* \rightarrow \text{Kind}_* \rightarrow \text{Kind}_*$.
- $\Pi, \Sigma \in \prod_{K \in \text{Kind}_k} (\text{Kind}_k \rightarrow \text{Kind}_*) \rightarrow \text{Kind}_*$.
- $\times, + \in \prod_{L \in \text{Kind}_k} (L \rightarrow \text{Kind}_*) \rightarrow \text{Kind}_*$.
- $\mu \in (\text{Kind}_* \rightarrow \text{Kind}_*) \rightarrow \text{Kind}_*$.

The function $\mu$ is used to define the semantics of the recursive datatypes.
8.1. GENERAL MODEL DEFINITION \(\lambda\omega^*_n\)

8.3 DEFINITION (semantics of \(\lambda\omega^*_n\)-datatypewriter-constructors).
For datatypewriter-constructors \(\sigma\) we define \([\Gamma \vdash \sigma : \mathcal{K}] \eta\) for all \(\Gamma \vdash \sigma : \mathcal{K}\) and \(\eta \models \Gamma^\sigma\) by induction on the derivation of \(\Gamma \vdash \sigma : \mathcal{K}\) as in definition 7.2, extended with the following clause:

\[
[\Gamma \vdash (\mu \alpha ::, \sigma) : \ast_\ast] \eta = \bigmu(a \in \text{Kind}_{\ast_\ast}) [\Gamma, \alpha : \ast, \Gamma \vdash \sigma : \ast] \eta[\alpha := \sigma] .
\]

Analogous to definitions 6.4 and 7.3 we define:

8.4 DEFINITION An environment model for the datatypewriter-constructors of \(\lambda\omega^*_n\) is an environment frame for the constructors of \(\lambda\omega^*_n\) for which \([\Gamma \vdash \sigma : \mathcal{K}] \eta\) is defined for all \(\Gamma \vdash \sigma : \mathcal{K}\) and \(\eta \models \Gamma^\sigma\).

For any environment model for the datatypewriter-constructors of \(\lambda\omega^*_n\) we can prove the same properties as for environment models for the datatypewriter-constructors of \(\lambda\omega_n\) or \(\lambda\omega^*_n\):

8.5 LEMMA (Soundness of type assignment for datatypewriter-constructors).
If \(\Gamma \vdash \sigma : \mathcal{K}\) then \([\Gamma \vdash \sigma : \mathcal{K}] \eta \in \text{Kind}_{\mathcal{K}}\) for all \(\eta \models \Gamma^\sigma\).

8.6 LEMMA (Soundness of \(\beta\)-conversion for datatypewriter-constructors).
Suppose \(\Gamma \vdash \sigma : \mathcal{K}\), \(\Gamma \vdash \sigma' : \mathcal{K}\), and \(\sigma \approx \sigma'\).
Then \([\Gamma \vdash \sigma : \mathcal{K}] \eta = [\Gamma \vdash \sigma' : \mathcal{K}] \eta\) for all \(\eta \models \Gamma^\sigma\).

These lemmas can be proved in exactly the same way as in chapters 6 and 7.

The semantics of \(\lambda\omega^*_n\)-terms

As mentioned before, instead of \(\sigma\), we now associate a cpo \(\text{Dom}_{\{\Gamma \vdash \sigma : \ast\}}\) with every datatypewriter-constructors \(\sigma\). So for every \(a \in \text{Kind}_{\ast_\ast}\), there is a cpo \(\text{Dom}_a\), and programs of type \(\sigma\) will be interpreted as elements of the cpo \(\text{Dom}_{\{\Gamma \vdash \sigma : \ast\}}\). For a datatypewriter-constructors \(\sigma\), the program \(Y \sigma (\sigma \rightarrow \sigma) \rightarrow \sigma\) is as a program interpreted as the least fixpoint operator on the cpo \(\text{Dom}_{\{\Gamma \vdash \sigma : \ast\}}\).

Recursive types should be isomorphic to their unfoldings. So it is obvious what the domain equation for a recursive type \(\langle \mu \alpha ::, \sigma \rangle\) should be, namely

\[
\text{Dom}_{\{\Gamma \vdash (\mu \alpha ::, \sigma) : \ast_\ast\}} \eta \cong \text{Dom}_{\{\Gamma \vdash [\sigma[\alpha \mapsto (\mu \alpha ::, \sigma)] \ast_\ast\}} \eta .
\]

The associated isomorphism then gives the interpretation of the operations \(\text{fold}_{\mu \alpha ::, \sigma}\) and \(\text{unfold}_{\mu \alpha ::, \sigma}\). Given the definition of \([\Gamma \vdash (\mu \alpha ::, \sigma) : \ast_\ast] \eta\) in definition 8.3, the new domain equations for the recursive types can also be given as

\[
\text{Dom}_{\{\mu \}} \cong \text{Dom}_{\{\mu \}} .
\]

Because cpos are used instead of sets, we have to reconsider the domain equations for \(\Rightarrow\), \(\Pi\), \(\ast\), \(\times\), and \(\Sigma\)-types given in the previous chapters. These were

- \(\text{Dom}_{\ast} \cong [\text{Dom}_{a} \longrightarrow \text{Dom}_{b}]\)
- \(\text{Dom}_{\Pi} \cong [\Pi_{\alpha \in \text{Kind}_{\ast}} \text{Dom}_{F(a)}]\)
- \(\text{Dom}_{\times} \cong \Pi_{\alpha \in \mathcal{L}} \text{Dom}_{F(\alpha)}\)
- \(\text{Dom}_{\Sigma} \cong \sum_{\alpha \in \text{Kind}_{\ast}} \text{Dom}_{F(\alpha)}\)

These domain equations reflect the properties of cpos.

\[
\text{Dom}_{\text{T}} \cong \text{Dom}_{\{\mu \}} .
\]
where in the right-hand sides $\prod$, $\sum$ denote the set-theoretic function space, cartesian product and disjoint sum, respectively, and $|A|$ denotes some subset of $A$.

We now have to turn the right-hand sides into epos. For (i), (ii) and (iii) this is not a problem, because the natural ordering on these sets produces a cpo. If we look at the underlying sets, then only the domain equations (iv) and (v) will change.

8.7 Definition (FS). Let $B$ and $C$ be epos, and let $(D_i | i \in I)$ be a family of epos indexed by some set $I$. Then

1. the cpo $\text{FS}(B, C)$ is the set of the continuous functions from $B$ to $C$ with the ordering pointwise.

2. the cpo $\text{GP}(D_i | i \in I)$ is the set $\Pi_{i \in I} D_i$ with the ordering coordinate-wise, i.e.
   \[
   (d_i | i \in I) \leq (d'_i | i \in I) \iff \forall_{i \in I} d_i \leq d'_i .
   \]

3. the cpo $\text{GS}(D_i | i \in I)$ is the set $\{ \bot \} \cup \sum_{i \in I} D_i$, partially ordered as follows
   \[
   c \leq c' \iff (c = \bot) \lor (c = (i, d) \wedge c' = (i, d') \wedge d \subseteq d') .
   \]

The new domain equations are

\[
\begin{align*}
\text{Dom}_a \prod b & \cong \text{FS}(\text{Dom}_a, \text{Dom}_b) \quad (i) \\
\text{Dom} \prod I & \cong \text{GP}(\text{Dom}_I | a \in \text{Kind}_I) \quad (ii) \\
\text{Dom} \sum I & \cong \text{GP}(\text{Dom}_I | i \in I) \quad (iii) \\
\text{Dom} \subseteq I & \cong \text{GS}(\text{Dom}_I | a \in \text{Kind}_I) \quad (iv) \\
\text{Dom} \mu I & \cong \text{GS}(\text{Dom}_I | i \in I) \quad (v) \\
\text{Dom} \nu I & \cong \text{Dom}_I \nu I \quad (vi)
\end{align*}
\]

where the square brackets denote a sub-cpo instead of a subset of $\text{GP}(\text{Dom}_I | a \in \text{Kind}_I)$.

Below the differences between the old and the new domain equations for $\rightarrow$, $\Pi$, $+$, $\times$, and $\Sigma$-types are discussed.

- (i)
  The old domain equations required $\text{Dom}_a \prod b$ to be isomorphic to some subset of the function space $\text{Dom}_a \rightarrow \text{Dom}_b$. In the new domain equation it is fixed what this subset should be, namely the collection of continuous functions from $\text{Dom}_a$ to $\text{Dom}_b$, because this is the natural, if not the only possible, choice. We cannot take the strict continuous functions, because by the $\beta$-reduction rule $\lambda x : A \ b \ a \equiv b[i \rightarrow a]$ functions are not always strict. For instance, if $x \notin \text{FV}(b)$ then $(\lambda x : A \ b) \downarrow = b$ and so $(\lambda x : A \ b)$ is not strict if $b \neq \bot.$
8.1. GENERAL MODEL DEFINITION $\lambda^\omega_S$

- (ii),(iii)
  For the $\Pi$- and $\times$-types there is no real change. As far as the underlying sets are concerned, the new domain equations are the same as the old ones.
  Unlike for the $\rightarrow$-types, for the $\Pi$-types there is no obvious choice for the sub-cpo in the right-hand side of the domain equation. In the model we construct in section 8.2 we will simply take the whole cpo $\text{gp}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_S)$.
  Equation (iii) requires that the domain for Unit, the empty product type $\Pi()$, is isomorphic with a one-point cpo. This means that the program unit of type Unit is interpreted as the only element of this cpo, i.e. its bottom element. An alternative is to require that the domain for Unit is isomorphic with a two-point cpo, and take the interpretation of unit different from $\bot_{\text{Unit}}$.

- (iv),(v)
  For the $\Sigma$- and $+$-types there has been a real change. For example, the old domain equation for the type $\sigma + r$ was
  \[ \text{Dom}_{[r \rightarrow \sigma + r \ast r]} \equiv \{ (1, x) \mid x \in \text{Dom}_{[r \rightarrow \sigma \ast r]} \} \cup \{ (2, x) \mid x \in \text{Dom}_{[r \rightarrow r \ast r]} \} \]
  The problem is that the set on the right hand side cannot be partially ordered to produce a cpo. At least, not in a such way that all the programs that can be constructed using in and $\triangleright$ are all continuous.
  There are two constructions on cpos that are possible interpretations of $\sigma + r$: the lazy sum – also known as the disjoint or separated sum $+$, and the strict or coalesced sum. Forgetting the cpo-ordering for the moment, and treating cpos as sets, the lazy sum would result in the following domain equation.
  \[ \text{Dom}_{[r \rightarrow \sigma + r \ast r]} = \{ \bot \} \cup \{ (1, x) \mid x \in \text{Dom}_{[r \rightarrow \sigma \ast r]} \} \cup \{ (2, x) \mid x \in \text{Dom}_{[r \rightarrow r \ast r]} \} \]
  i.e a new bottom element is provided. The strict sum would result in the following domain equation:
  \[ \text{Dom}_{[r \rightarrow \sigma + r \ast r]} = \{ \bot \} \cup \{ (1, x) \mid x \in \text{Dom}_{[r \rightarrow \sigma \ast r]} \} \cup \{ (2, x) \mid x \in \text{Dom}_{[r \rightarrow r \ast r]} \} \cup \{ (3, x) \mid x \in \text{Dom}_{[r \rightarrow \sigma \ast r]} \} \]
  Here the bottom elements of $\text{Dom}_{[r \rightarrow \sigma \ast r]}$ and $\text{Dom}_{[r \rightarrow r \ast r]}$ are identified.
  The reduction rules for $+$-types rule out the second possibility. If we interpret $\sigma + r$ as a strict sum, then the value of the function $f \triangleright g : \sigma + r \rightarrow \rho$ in $(\text{in}_{\sigma + r} 1 \perp_{\sigma})$ and $(\text{in}_{\sigma + r} 2 \perp_r)$. But this is in conflict with the reduction behaviour of $f \triangleright g$:
  \[
  (f \triangleright g)(\text{in}_{\sigma + r} 1 \perp_{\sigma}) \triangleright g \perp_{\rho},
  \]
  and $f \perp_{\sigma}$ and $g \perp_{r}$ may well be distinct.
  Similarly, the $\beta$-reduction rules make it impossible to interpret $\Pi$- and $\times$-types as smashed products.
  Note that $\text{GS}(\cdot)$ is a one-point cpo. So the datatype $\text{Empty} \equiv \Sigma()$ is no longer interpreted as an empty type, but as a one-element type.
The domain equation

\[ \text{Dom} \; \Sigma \; F \; \cong \; \text{GS}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_{\mathcal{K}}) \tag{i} \]

requires more than is necessary for the interpretation of the associated program constructions, as we already discussed in chapter 7 for the same domain equation with GS replaced by \( \Sigma \). As for \( \lambda \omega^+_E \), it would be sufficient if the following domain equation is satisfied

\[ \text{Dom} \; \Sigma \; F \; \cong \; \left( \text{GS}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_{\mathcal{K}}) \right) / \approx, \tag{ii} \]

with \( \approx \) as defined in definition 7.8. But for the particular cpo model that is given in section 8.2 the domain equations (i) and (ii) are identical. In this model all equivalence classes in \( \approx \) are singletons, which is caused by the fact that \( \text{Dom} \; \Sigma \; F \) is isomorphic to the whole cpo \( \text{GP}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_{\mathcal{K}}) \). Because of this, we will not go to the extra trouble of using (ii) instead of (i).

8.8 Definition (\( \lambda \omega^+_E \) cpo environment frame).

A cpo environment frame for \( \lambda \omega^+_E \) is a tuple \( (\text{KIND}, \text{Dom}, \Phi^-, \Phi^+, \Phi_\Sigma, \Phi', \Phi^\mu) \), where

- \( \text{KIND} = (\text{Kind}, \Phi_{\text{Kind}}, \rightarrow, \Sigma, \Xi, +, \mu) \) is an environment model for the \( \lambda \omega^+_E \)-datatype-constructors.
- \( \text{Dom} = \{ \text{Dom}_a \mid a \in \text{Kind}_+ \} \) is a family of epos
- \( \Phi^- = \{ \Phi^-_{a,b} \mid a, b \in \text{Kind}_+ \} \), \( \Phi^+ = \{ \Phi^+_{a} \mid F \in \text{Kind}_R \rightarrow \text{Kind}_+ \} \), \( \Phi^\Sigma = \{ \Phi^\Sigma_{a} \mid F \in \text{Kind}_R \rightarrow \text{Kind}_+ \} \), \( \Phi = \{ \Phi_\Sigma_{a} \mid F \in \text{Kind}_R \rightarrow \text{Kind}_+ \} \), \( \Phi^\mu = \{ \Phi^\mu_{a} \mid F \in \text{Kind}_+ \rightarrow \text{Kind}_+ \} \) are families of bijections with

\[
\begin{align*}
\Phi^-_{a,b} & \in \text{Dom}_a \rightarrow \text{Dom}_b, \\
\Phi^+_{a} & \in \text{Dom}_a, \\
\Phi_\Sigma_{a} & \in \text{Dom}_a, \\
\Phi^\mu_{a} & \in \text{Dom}_a.
\end{align*}
\]

We now define the interpretation of the \( \lambda \omega^+_E \)-programs in an arbitrary environment frame. Only the differences with respect to definition 7.10, which defines the semantics of \( \lambda \omega^+_E \)-programs, are given. The differences are the interpretation of \( \nabla \) where we now have to take the bottom elements into account, and the interpretation of the new program constructions (unfold and \( Y \)). In example 8.10 the interpretations of case and abstype are given, which are a lot easier to read than the interpretation of \( \nabla \) for + and \( \Sigma \)-types.

8.9 Definition (semantics of \( \lambda \omega^+_E \)-programs).

For programs \( M \) we define \( \Gamma \vdash M \sigma \eta \) for all \( \Gamma \vdash M \sigma \) and \( \eta = \Gamma^\mu \), as in definition 7.10—i.e., by induction on the derivation of \( \Gamma \vdash M \sigma \) with the following extra clauses—
For μ-types:
Let \( F = (Aa \in \text{Kind}_a) \) \([\Gamma, \alpha : \star, \sigma : \star] \eta[\alpha := a] \) \( \in \text{Kind}_a \). Then
\[
\begin{align*}
\Phi &= \Phi^\mu_F \in \text{Dom}_F \rightarrow \text{Dom}_F = \\
\Phi^1 &= \Phi^{\mu}_F \in \text{Dom}_F \rightarrow \text{Dom}_F
\end{align*}
\]
For \( \Sigma \)-types.
Let \( F = (Aa \in \text{Kind}_a) \) \([\Gamma, \alpha : \star, \sigma : \star] \eta[\alpha := a] \) \( \in \text{Kind}_a \). Then
\[
\begin{align*}
\Phi &= \Phi^\Sigma_F \in \text{Dom}_F \rightarrow \text{Dom}_F = \\
\Phi^1 &= \Phi^{\Sigma}_F \in \text{Dom}_F \rightarrow \text{Dom}_F
\end{align*}
\]
where $\Phi_1 = \Phi \circ [\Sigma \circ F] \in \text{Dom}(\Sigma \circ F) \rightarrow \{\bot\} \cup \sum_a \text{Dom}_{F(a)}$

$\Phi_2 = \Phi \circ \left(\Sigma \circ F \circ \Pi \circ R \rightarrow \left[\text{Dom}(\Sigma \circ F) \rightarrow \text{Dom}_R\right]\right)$

$\Phi_3 = \Phi \circ \left(\Pi \circ G \rightarrow \Pi \circ \text{Dom}_{F(a)} \circ \Sigma \circ F \circ \Pi \circ R\right)$

$\Phi_4 = \Phi \circ \left(\text{Dom}_{F(a)} \circ \Sigma \circ F \circ \Pi \circ R \rightarrow \left[\text{Dom}(\Sigma \circ F) \rightarrow \text{Dom}_R\right]\right)$

8.10 Example. As in chapter 7, the interpretation of the case- and abtype-construction is easier to read then the interpretation of $\varnothing$. Definition 8.9 results in the following interpretation for the case- and abtype-construction:

\[
\begin{align*}
\lbrack \Gamma \vdash \text{(case } N \text{ of } m.n.x \rightarrow M_1 \mid \ldots \mid m.n.y \rightarrow M_n \text{ esac)} : \tau \rbrack \eta \\
= \begin{cases}
\lbrack \Gamma \vdash \text{case } x :: \tau \rightarrow M_1 \mid \ldots \mid M_n : \text{esac} : \tau \rbrack \eta \\
= \lbrack \Gamma \vdash \text{case } x :: \tau \rightarrow M_1 \mid \ldots \mid M_n : \text{esac} : \tau \rbrack \eta \\
= \begin{cases}
\lbrack \Gamma, \text{case } x :: \tau \rightarrow M_1 \mid \ldots \mid M_n : \text{esac} : \tau \rbrack \eta \rbrack \eta = \begin{cases}
\text{if } \Phi \lbrack \Gamma \vdash N : \Sigma (l_1, \sigma_1), \ldots, l_n, \sigma_n) \rbrack \eta = \begin{cases}
\text{if } \Phi \lbrack \Gamma \vdash N : \Sigma (l_1, \sigma_1), \ldots, l_n, \sigma_n) \rbrack \eta = \bot
\end{cases}
\end{cases}
\end{cases}
\end{align*}
\]

where $\Phi = \Phi \circ \left(\Sigma \circ F \circ \Pi \circ R \rightarrow \left[\text{Dom}(\Sigma \circ F) \rightarrow \text{Dom}_R\right]\right)$

8.11 Definition. A $\lambda\omega^\ast$-environment model is a $\lambda\omega^\ast$-environment frame for which $\lbrack \Gamma \vdash M : \sigma \rbrack \eta$ is defined for all $\Gamma \vdash M : \sigma$ and $\eta \models \Gamma^\bullet_{\sigma^\ast}$.

The properties proved in chapters 6 and 7 for arbitrary environment models of $\lambda\omega_\ast$ and $\lambda\omega_\ast^\ast$ also hold for arbitrary environment models of $\lambda\omega^\ast_\ast$, and they can be proved in exactly the same way.

8.12 Lemma. Let $\text{der}_1$ and $\text{der}_2$ be derivations of $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \sigma'$, respectively. Then

\[
\lbrack \Gamma \vdash M : \sigma \rbrack \eta = \lbrack \Gamma \vdash M : \sigma' \rbrack \eta,
\]

where the meanings are defined using $\text{der}_1$ and $\text{der}_2$ respectively.

8.13 Lemma (Soundness of type assignment for programs). If $\Gamma \vdash M : \sigma$ then $\lbrack \Gamma \vdash M : \sigma \rbrack \eta \in \text{Dom}_{\Gamma^\circ_{\sigma^\ast}} \eta$ for all $\eta \models \Gamma^\bullet_{\sigma^\ast}$.

8.14 Lemma (Soundness of $\beta^\ast$-conversion for programs). Suppose $M \approx_{\beta^\ast} M'$, $\Gamma \vdash M : \sigma$, and $\Gamma \vdash M' : \sigma'$.

Then $\lbrack \Gamma \vdash M : \sigma \rbrack \eta = \lbrack \Gamma \vdash M' : \sigma' \rbrack \eta$ for all $\eta \models \Gamma^\bullet_{\sigma^\ast}$.
8.2 CPO-MODEL FOR $\lambda \omega^\mu_s$  

In this section we construct a cpo-model that is an $\lambda \omega^\mu_s$-environment model as defined in the previous section. The general model definition gives a system of coupled domain equations that have to be satisfied:

\[
\begin{align*}
\text{Dom}_a & \cong \text{FS}(\text{Dom}_a, \text{Dom}_b) & (i) \\
\text{Dom} \left[ \prod \right] F & \cong \left[ \text{GP}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_c) \right] & (ii) \\
\text{Dom} \left[ \exists \right] F & \cong \left[ \text{GS}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_c) \right] & (iii) \\
\text{Dom} \left[ \times \right] F & \cong \text{GP}(\text{Dom}_{F(l)} \mid l \in L) & (iv) \\
\text{Dom} \left[ + \right] F & \cong \left[ \text{GS}(\text{Dom}_{F(l)} \mid l \in L) \right] & (v) \\
\text{Dom} \left[ \square \right] F & \cong \text{Dom}_{F(\left[ \square \right] F)} & (vi)
\end{align*}
\]

These equations are mutually recursive, because of the equations (ii), (iii) and (vi). It is obvious that (vi) is recursive. For (ii) and (iii) this is less obvious, but the right-hand sides of these equations refer to all Dom$_{F(a)}$, which includes many Dom$_{F(a)}$ with $\prod F$ or $\Sigma F$ a subexpression of $F(a)$ (for instance Dom$_{F(\left[ \prod \right] F)}$ or Dom$_{F(\left[ \Sigma \right] F)}$).

To construct a $\lambda \omega^\mu_s$-model we will construct a family of cpos Dom = $\{\text{Dom}_a \mid a \in \text{Kind}_c\}$ that solves this system of coupled domain equations. For this the standard technique for solving domain equations, as described in for instance [SP82], is used. The necessary tools for this technique, which involve some category theory, are introduced in subsection 8.2.1. In subsection 8.2.2 the results given in subsection 8.2.1 are applied to construct a solution of the domain equations above.

The domain equations above leave open the choice of a particular sub-cpo of the cpo $\text{GP}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_c)$. Here we choose the whole cpo. So the model that is constructed will satisfy

\[
\text{Dom} \left[ \prod \right] F \cong \text{GP}(\text{Dom}_{F(a)} \mid a \in \text{Kind}_c)
\]

### 8.2.1 Category-theoretic solution of recursive domain equations

We assume the reader is familiar with the basic notions of category theory. For a category $C$, we write $\text{Obj}(C)$ for the collection of objects in $C$. For $c, c' \in \text{Obj}(C)$, we write $\text{Hom}_C(c, c')$ for the collection of morphisms from $c$ to $c'$ in $C$.

#### Fixed-points of Functors

Solutions of a recursive domain equations $\text{Dom} \cong F(\text{Dom})$ can be regarded as fixpoints of a functor $F$ on some category.

8.15 Definition. A fixpoint of a functor $F$ on a category $C$, is a pair $(\text{Dom}, \phi)$, where $\text{Dom}$ is a $C$-object and $\phi$ an isomorphism between $\text{Dom}$ and $F(\text{Dom})$. The initial fixpoint theorem given below gives a class of functors for which (initial) fixpoints exists, namely so-called $\omega$-continuous functors on $\omega$-categories.
8.16 Definition (\(\omega\)-chain, \(\omega\)-continuous, \(\omega\)-colimit, \(\omega\)-category)

1. An \(\omega\)-chain is a diagram of the form \(c_0 \xrightarrow{\phi_0} c_1 \xrightarrow{\phi_1} c_2 \).

2. A functor is called \(\omega\)-continuous if it preserves \(\omega\)-colimits, i.e., colimits of \(\omega\)-chains.

3. An \(\omega\)-category is a category with an initial object which has all \(\omega\)-colimits, i.e., in which every \(\omega\)-chain has a colimit.

8.17 Theorem (Initial fixpoint theorem).
Every \(\omega\)-continuous functor on an \(\omega\)-category has an initial fixed-point.

This theorem is similar to the least fixpoint theorem for \(\omega\)-cpos, which states that every continuous function on a \(\omega\)-cpos has a least fixed point. In fact, the fixpoint theorem for cpos is a particular case of the initial fixed point theorem for \(\omega\)-categories. In denotational semantics, this general result is applied to construct solutions of a recursive domain equation \(\text{Dom} \cong F(\text{Dom})\).

CPO and functors on CPO

We now introduce the particular categories and functors that will be used. The following category plays a central role.

8.18 Definition. CPO is the category with \(\omega\)-cpos, i.e., partially ordered sets with an initial object and limits of all chains \(c_0 \subseteq c_1 \subseteq c_2 \subseteq \cdots\) as objects, and continuous functions as morphisms.

Because of the interdependence of the domain equations they all have to be solved simultaneously. Therefore we will construct a solution in a *product category*.

8.19 Definition (product category).
Let \(I\) be an index set and \(C\) a category. The *product category* \(\prod_I C\) is then defined as follows:

- a \(\prod_I C\)-object is a family \((c_i | i \in I)\), where each \(c_i\) is a \(C\)-object,
- a \(\prod_I C\)-morphism from \((c_i | i \in I)\) to \((c'_i | i \in I)\) is a family \((\phi_i | i \in I)\), where each \(\phi_i\) is a \(C\)-morphism from \(c_i\) to \(c'_i\).

A product category \(\prod_I C\) inherits many of the properties of \(C\). In particular:

8.20 Lemma ([HS73] corollary 25.7)
If \(C\) has all \(\omega\)-colimits, i.e., \(C\) has colimits of all \(\omega\)-chains, then so does \(\prod_I C\).

So the family of cpos \(\text{Dom} = (\text{Dom}_a | a \in \text{Kind}_{\ast}^\omega)\) needed for the \(\lambda\omega^\omega\)-model is an object in the category \(\prod_{\text{Kind}_{\ast}^\omega} \text{CPO}\).
The constructions on cpos that are used in the domain equations – FS, GP and GS – are all functors.
8. 2.  CPO-MODEL FOR \( \omega^k \)

8.21 Definition (FS, GP, GS).
The object-parts of the functors \( \text{FS} : \text{CPO}^{OP} \times \text{CPO} \to \text{CPO} \) and \( \text{GP}, \text{GS} : \Pi_I \text{CPO} \to \text{CPO} \)
have been defined in definition 8.7. The morphism-parts are defined below.

- Suppose \( \phi \in \text{Hom}(B', B) \) in \( \text{CPO} \) and \( \psi \in \text{Hom}(C, C') \) in \( \text{CPO} \).
  Then \( \text{FS}(\phi, \psi) \in \text{Hom}(\text{FS}(B, C), \text{FS}(B', C')) \) is the function that maps \( \xi \in \text{FS}(B, C) \) to
  \[ \psi \circ \xi \circ \psi \in \text{FS}(B', C'). \]

- Suppose \( \phi \in \text{Hom}(D, E) \) in \( \Pi_I \text{CPO} \) So \( \phi_i \in \text{Hom}(D_i, E_i) \) in \( \text{CPO} \) for all \( i \in I \).
  Then
  - \( \text{GP}(\phi) \in \text{Hom}(\text{GP}(D), \text{GP}(E)) \) is the function that maps \( \langle d_i | i \in I \rangle \in \text{GP}(D) \) to
    \( \langle \phi_i(d_i) | i \in I \rangle \in \text{GP}(E) \),
  - \( \text{GS}(\phi) \in \text{Hom}(\text{GS}(D), \text{GS}(E)) \) is the function that maps \( \bot \in \text{GS}(D) \) to \( \bot \in \text{GS}(E) \),
    and \( \langle i, d \rangle \in \text{GS}(D) \) to \( \langle i, \phi_i(d) \rangle \in \text{GS}(E) \).

The functor \( \text{FS} \) is contravariant in its first argument. Because of this, it is not possible
to construct the family of cpos \( \text{Dom} = (\text{Dom}_a | a \in \text{Kind}_a) \) needed for an \( \omega^k \)-model as an
initial fixed-point in the category \( \Pi_I \text{CPO} \). There is a standard technique to overcome this
problem, which is described below.

O-categories

The standard technique for solving domain equations that involve the function space functor
or other contravariant functors is described in [LS81] [SP82]. The method is a generalisation
of Scott’s inverse limit construction. We just give a short overview of the method. For a clear
and self-contained presentation we refer to [BH88]. Below we just list all the results that we
will need.

A special kind of category, the so-called O-categories, is used. The crucial property of
these O-categories is is given by theorem 8.23 below. It states that given a functor \( \mathcal{H} \) on an
O-category \( \mathcal{C} \), which is contravariant in one or more of its arguments, there is a corresponding
functor \( \mathcal{F} \) on an associated category \( \mathcal{C}_{PR} \), which is covariant in all its arguments. Moreover,
\( \omega \)-continuity of this functor \( \mathcal{F} \) can be proved relatively easily. But first, some definitions are
needed.

8.22 Definition (O-category) A category is an O-category iff

- every hom-set is a partially ordered set, in which every ascending \( \omega \)-chain has a least
  upper bound,
- composition is \( \omega \)-continuous with respect to the partial order on the hom-sets.

It is trivial to verify that \( \text{CPO} \) and \( \Pi_I \text{CPO} \) are O-categories. The ordering on the hom-sets
of \( \Pi_I \text{CPO} \) is of course defined coordinate-wise.
8.23 Definition (category of embedding-projection pairs).
If \( C \) is an \( \mathcal{O} \)-category, then the associated category of embedding-projection pairs \( C_{PR} \) is the category with

- the same objects as \( C \), i.e. \( \text{Obj}(C_{PR}) = \text{Obj}(C) \),
- as morphisms embedding-projection pairs of morphisms, i.e. \( (\phi, \psi) \in \text{Hom}(a, b) \) in \( C_{PR} \)
  iff \( \phi \in \text{Hom}_C(a, b) \land \psi \in \text{Hom}_C(b, a) \land \phi \psi \subseteq \text{id}_b \land \psi \phi = \text{id}_a \)

The following notion can be used to prove \( \omega \)-continuity of functors on a category \( C_{PR} \)

8.24 Definition (local continuity).
Let \( C \) be an \( \mathcal{O} \)-category, and \( \mathcal{H} : C^{\mathcal{O}P} \times C \rightarrow C \) be a functor, i.e. \( \mathcal{H} \) is a bifunctor on \( C \) that is contravariant in its first and covariant in its second argument.

\( \mathcal{H} \) is called locally continuous if it is continuous with respect to the partial order on hom-sets, i.e. if, for all \( b, b', c, c' \in \text{Obj}(C) \), the mapping

\[
\phi \in \text{Hom}(b', b), \psi \in \text{Hom}(c, c') \mapsto \mathcal{H}(\phi, \psi) \in \text{Hom}(\mathcal{H}(b, c), \mathcal{H}(b', c'))
\]

is continuous w.r.t. the ordering on these hom-sets.

For example, it is easy to check that the functor \( FS \) is locally continuous. We can now state the main property of (functors on) \( \mathcal{O} \)-categories

8.25 Theorem (\( \omega \)-continuity from local continuity).
Let \( \mathcal{H} : C^{\mathcal{O}P} \times C \rightarrow C \) be a locally continuous functor, and \( C \) be an \( \mathcal{O} \)-category which has all \( \omega \)-colimits, i.e. every \( \omega \)-chain has a colimit.

Then the functor \( \mathcal{F} : C_{PR} \rightarrow C_{PR} \) defined by

\[
\mathcal{F}(c) = \mathcal{H}(c, c) \quad \text{for } \text{CPR-objects } c
\]
\[
\mathcal{F}(\phi, \psi) = (\mathcal{H}(\psi, \phi), \mathcal{H}(\phi, \psi)) \quad \text{for } \text{CPR-morphisms } (\phi, \psi)
\]

is an \( \omega \)-continuous functor.

Proof. This theorem is a slightly weakened version of theorem 3 in [SP82]. The difference is that the rather technical condition that \( C \) has "locally determined \( \omega \)-colimits of embeddings" has been replaced with the condition that \( C \) has all \( \omega \)-colimits. By the corollary to theorem 2 in [SP82] this condition is stronger.

This result will be used in the next section to prove \( \omega \)-continuity of a functor, with which we then construct a \( \lambda \omega^\omega \)-model as an initial fixedpoint.

8.2.2 Construction of a CPO-model
Before the family of cpos \( \text{Dom} \) can be constructed, an environment model for the datatype-constructors is needed. Any environment model for the datatype-constructors can be used. We take the simplest possible model, i.e. the term model.
8.26 Definition. \( \text{KINDTERM} = (\text{Kind}, \Phi_{\text{Kind}}, \sqsubseteq, \Pi, \exists, \times, [\mu]) \) is the closed term model for the \( \lambda w^n \)-constructors. \( \Box \)

So \( \text{Kind}_{BC} \) is the collection of \( \beta \)-equivalence classes of closed datatype-constructors of type \( K \), i.e.

\[
\text{Kind}_{BC} = \{ \sigma \mid \sigma \vdash \sigma : K \} / \approx_{\beta},
\]

with the associated bijections \( \Phi_{K_1 \rightarrow K_2} \in \text{Kind}_{K_1 \rightarrow K_2} \rightarrow \text{Kind}_{K_1} \rightarrow \text{Kind}_{K_2} \) defined by

\[
\Phi_{K_1 \rightarrow K_2} = \lambda [\sigma]_{\approx_{\beta}} \in \text{Kind}_{K_1 \rightarrow K_2}, [\tau]_{\approx_{\beta}} \in \text{Kind}_{K_1} . [\sigma \tau]_{\approx_{\beta}} b.
\]

Obviously, \( \text{KINDTERM} \) is an environment model for the datatype-constructors of \( \lambda w^n \). The function \( \sqsubseteq \) \( \in \text{Kind} \rightarrow \text{Kind} \rightarrow \text{Kind} \) is

\[
\sqsubseteq = \lambda [\sigma]_{\approx_{\beta}}, [\tau]_{\approx_{\beta}} \in \text{Kind} . [\sigma \rightarrow \tau]_{\approx_{\beta}}.
\]

The other datatype-constructors are interpreted similarly. We will not distinguish a datatype-constructor \( \sigma \) and the equivalence class \( [\sigma]_{\approx_{\beta}} \) to which it belongs, and we also do not distinguish \( \rightarrow \) and \( \sqsubseteq \), \( \Pi \) and \( \exists \), \( \times \) and \( \ast \), etc..

To complete the \( \lambda w^n \)-model, a family of cpos \( \text{Dom} = (\text{Dom}_a \mid a \in \text{Kind}_*) \) is needed that solves the domain equations. To construct such a \( \text{Dom} \) as the initial fixpoint, an appropriate \( \omega \)-category and \( \omega \)-continuous functor \( \mathcal{F} \) on that category are needed. The category should have families of cpos as objects, and the functor \( \mathcal{F} \) should map an object \( \text{Dom} = (\text{Dom}_a \mid a \in \text{Kind}_*) \) to \( \mathcal{F}(\text{Dom}) = (\mathcal{F}(\text{Dom})_a \mid a \in \text{Kind}_*) \) with

\[
\mathcal{F}(\text{Dom})_{a \rightarrow} = \mathcal{F}S(\text{Dom}_a, \text{Dom}_a)
\]

\[
\mathcal{F}(\text{Dom})_{\Pi F} = \mathcal{G}P(\text{Dom}_{F(a)} \mid a \in \text{Kind}_K)
\]

\[
\mathcal{F}(\text{Dom})_{\exists F} = \mathcal{G}S(\text{Dom}_{F(a)} \mid a \in \text{Kind}_K)
\]

\[
\mathcal{F}(\text{Dom})_{\times F} = \mathcal{G}P(\text{Dom}_{F(l)} \mid l \in L)
\]

\[
\mathcal{F}(\text{Dom})_{+ F} = \mathcal{G}S(\text{Dom}_{F(l)} \mid l \in L)
\]

\[
\mathcal{F}(\text{Dom})_{\mu F} = \text{Dom}_{F(\mu F)}
\]

Then any fixpoint of \( \mathcal{F} \) provides a solution of the domain equations.

As the category we use \( (\prod \text{Kind}_*, \text{CPO})_{PR} \). We will omit the subscript \( \text{Kind}_* \) and just write \( (\prod \text{CPO})_{PR} \) for \( (\prod \text{Kind}_*, \text{CPO})_{PR} \). By definition 8.23 the category \( (\prod \text{CPO})_{PR} \) is the category with

- as objects families of cpos \( (D_a \mid a \in \text{Kind}_*) \),
- as the morphisms from \( D \) to \( E \) projection-embedding pairs \( (\phi, \psi) \), i.e. \( \phi \in \text{Hom}(D, E) \) and \( \psi \in \text{Hom}(E, D) \) in \( \prod \text{CPO} \), and \( \psi \circ \phi = \text{id}_D \) and \( \phi \circ \psi \subseteq \text{id}_E \).

By the definition of product category (def. 8.19), this means that \( \phi \) and \( \psi \) are families of continuous functions, with \( \phi_a \in D_a \rightarrow E_a \) and \( \psi_a \in E_a \rightarrow D_a \) and \( \psi_a \circ \phi_a = \text{id}_{D_a} \) and \( \phi_a \circ \psi_a \subseteq \text{id}_{E_a} \) for all \( a \in \text{Kind}_* \).
Instead of the category $\Pi(CPO)_{PR}$ we could also use the category $\Pi(CPO_{FR})$. In fact, the two categories are isomorphic. They have the same objects. The morphisms in $\Pi(CPO)_{PR}$ are pairs of families of continuous functions $(\phi_a | a \in \text{Kind}_{*})$, $(\psi_a | a \in \text{Kind}_{*})$, and the morphisms in $\Pi(CPO_{FR})$ are families of pairs of continuous functions $(\phi_a, \psi_a) | a \in \text{Kind}_{*})$. Using $\Pi(CPO_{FR})$ instead of $\Pi(CPO)_{PR}$ requires slightly more work.

To apply the initial fixpoint theorem we need.

8.27 Lemma. $\Pi(CPO)_{PR}$ is an $\omega$-category.

Proof. We must show that the category $\Pi(CPO)_{PR}$ has all $\omega$-colimits, i.e., that every $\omega$-chain has a colimit, and that it has an initial element.

The obvious candidate for the initial object is the constant family $\{(\bot) | a \in \text{Kind}_{*}\}$ which maps every index $a$ to the one-point cpo. It can easily be verified that this is indeed an initial element.

The category $CPO_{PR}$ has all $\omega$-colimits (see for instance [SP82]). Then by lemma 8.20 $\Pi(CPO_{PR})$ has all $\omega$-colimits, and this category is isomorphic to $\Pi(CPO)_{PR}$. $\square$

We now define the functor $\mathcal{F}$ on the category $\Pi(CPO)_{PR}$ that is used to construct the family of cpos $\text{Dom} = (\text{Dom}_a | a \in \text{Kind}_{*})$ that solves the domain equations. The definition of the object-part of $\mathcal{F}$ is prescribed by the domain equations. For the definition of the morphism-part we then just have to be careful to give $\mathcal{F}$ the right arguments.

8.28 Definition. The functor $\mathcal{F}$ from $\Pi(CPO)_{PR}$ to $\Pi(CPO)_{FR}$ is defined as follows:

- **Object-part** Suppose $D \in \text{Obj}(\Pi(CPO)_{PR})$. So $D = (D_a | a \in \text{Kind}_{*})$ is a family of cpos. Then $\mathcal{F}(D) = (\mathcal{F}(D)_a | a \in \text{Kind}_{*})$ is defined as follows:

\[
\begin{align*}
\mathcal{F}(D)_a &\Rightarrow FS(D_a, D) \\
\mathcal{F}(D)_{\Pi F} &\Rightarrow GP(D_{\Pi F} | a \in \text{Kind}_{K}) \\
\mathcal{F}(D)_{\Sigma F} &\Rightarrow GS(D_{\Pi F} | a \in \text{Kind}_{K}) \\
\mathcal{F}(D)_{\times F} &\Rightarrow GP(D_{\times F} | l \in L) \\
\mathcal{F}(D)_{+ F} &\Rightarrow GS(D_{+ F} | l \in L) \\
\mathcal{F}(D)_{\mu F} &\Rightarrow D_{\mu F}
\end{align*}
\]

- **Morphism-part** Suppose $(\phi, \psi) \in \text{Hom}(D, E)$ in $\Pi(CPO)_{PR}$. So all $\phi_a$ and $\psi_a$ are continuous functions, $\phi_a \in D_a \rightarrow E_a$ and $\psi_a \in E_a \rightarrow D_a$.

Then $\mathcal{F}(\phi, \psi) = (\phi', \psi')$, with $\phi'_a \in \mathcal{F}(D)_a \rightarrow \mathcal{F}(E)_a$ and $\psi'_a \in \mathcal{F}(E)_a \rightarrow \mathcal{F}(D)_a$ defined as follows:

\[
\begin{align*}
\phi'_{a-} &\Rightarrow FS(\psi'_a, \phi'_a) \\
\phi'_{\Pi F} &\Rightarrow GP(\phi'_{\Pi F} | a \in \text{Kind}_{K}) \\
\phi'_{\Sigma F} &\Rightarrow GS(\phi'_{\Pi F} | a \in \text{Kind}_{K}) \\
\phi'_{\times F} &\Rightarrow GP(\phi'_{\times F} | l \in L) \\
\phi'_{+ F} &\Rightarrow GS(\phi'_{\times F} | l \in L) \\
\phi'_{\mu F} &\Rightarrow \phi'_{\mu F}
\end{align*}
\]

It is easy to verify that $\mathcal{F}$ preserves identities and composition, so that it is indeed a functor.
The domain equations for $\lambda^\mu_5$ can now be stated as $\text{Dom} \cong F(\text{Dom})$. Theorem 8.25 will be used to prove that $F$ is $\omega$-continuous.

8.29 Lemma. $F$ is $\omega$-continuous.

Proof. To apply theorem 8.25 we first have to prove that $\Pi \text{CPO}$ has all $\omega$-colimits. CPO has all $\omega$-colimits (see for instance [SP82]), so then by lemma 8.20 $\Pi \text{CPO}$ has all $\omega$-colimits.

All that is needed now to apply theorem 8.25 is a locally continuous functor $\mathcal{H} : (\Pi \text{CPO})^{\text{op}} \times \Pi \text{CPO} \to \Pi \text{CPO}$ such that

- (Object-part) Suppose $D \in \text{Obj}((\Pi \text{CPO})^{\text{op}})$ and $E \in \text{Obj}(\Pi \text{CPO})$. So $D$ and $E$ are elements of $\Pi \text{CPO}$. Then $\mathcal{H}(D, E)$ is defined by
  \[
  \mathcal{H}(D, E)_{\sigma \rightarrow \tau} = \text{FS}(D_\sigma, E_\tau)
  \]
  \[
  \mathcal{H}(D, E)_{\eta \in \eta'} = \text{GP}(E_{F(\eta)}, \eta \in \text{Kind}_{\text{M}})
  \]
  \[
  \mathcal{H}(D, E)_{\Sigma F} = \text{GS}(E_{F(\eta)}, \eta \in \text{Kind}_{\text{M}})
  \]
  \[
  \mathcal{H}(D, E)_{\mu F} = \text{GP}(E_{F(\mu)}, \mu \in \text{L})
  \]
  \[
  \mathcal{H}(D, E)_{\mu F} = \text{GS}(E_{F(\mu)}, \mu \in \text{L})
  \]

- (Morphism-part) Suppose $\phi \in \text{Hom}(D, E)$ in $(\Pi \text{CPO})^{\text{op}}$, and $\psi \in \text{Hom}(D', E')$ in $\Pi \text{CPO}$. Then $\mathcal{H}(\phi, \psi) \in \text{Hom}(\mathcal{H}(D, E), \mathcal{H}(D', E'))$ in $\Pi \text{CPO}$ is defined by
  \[
  \mathcal{H}(\phi, \psi)_{\sigma \rightarrow \tau} = \text{FS}(\phi_\sigma, \psi_\tau)
  \]
  \[
  \mathcal{H}(\phi, \psi)_{\eta \in \eta'} = \text{GP}(\psi_{F(\eta)}, \eta \in \text{Kind}_{\text{M}})
  \]
  \[
  \mathcal{H}(\phi, \psi)_{\Sigma F} = \text{GS}(\psi_{F(\eta)}, \eta \in \text{Kind}_{\text{M}})
  \]
  \[
  \mathcal{H}(\phi, \psi)_{\mu F} = \text{GP}(\psi_{F(\mu)}, \mu \in \text{L})
  \]
  \[
  \mathcal{H}(\phi, \psi)_{\mu F} = \text{GS}(\psi_{F(\mu)}, \mu \in \text{L})
  \]

Note that $\mathcal{H}$ uses its first argument, which in the definition above is $D$ or $\phi$, in contravariant places, i.e. as first argument of FS. Its second argument is used in all other places. It is easy to check - coordinate-wise - that $\mathcal{H}$ preserves identities and composition, so that it is indeed a functor, and it immediately follows from the definitions of $F$ and $\mathcal{H}$ that $F(D) = \mathcal{H}(D, D)$ and $F(\phi, \psi) = (\mathcal{H}(\psi, \phi), \mathcal{H}(\phi, \psi))$.

It is also easy to prove - again coordinate-wise - that $\mathcal{H}$ is locally continuous. This follows immediately from the fact that FS, GP, and GS are locally continuous.

Then by theorem 8.25 the functor $F$ is $\omega$-continuous.
8.30 Lemma.
Let \((\text{Dom}, (\phi, \psi))\) be the initial fixpoint of \(F\) in the category \((\coprod \text{CPO})_{P^H}\).
Let \(\Phi = \phi_{\text{in}, b}, \Phi_1 = \phi_{\text{out}, b}, \Phi_0 = \phi_0, \Phi = \phi_{\text{in}, b}, \Phi_1 = \phi_{\text{out}, b}, \Phi_0 = \phi_0, \Phi\) is an environment model for \(\lambda \omega^\mu\).

Then \((\text{KIND}_{\text{ERM}}, \text{Dom}, \Phi, \Phi^0, \Phi^1, \Phi^0, \Phi^1, \Phi^0, \Phi^1)\) is an environment model for \(\lambda \omega^\mu\).

**Proof.** We have to prove that \(\Gamma \vdash M : \sigma\) \(\eta\) is defined for all \(\Gamma \vdash M : \sigma\) and \(\eta \models \Gamma^0\).
Recall that this amounts to checking if the ranges of the isomorphisms \(\Phi_{\sigma^0, \sigma}\) and \(\Phi_{\sigma^0, \sigma}\) are large enough. It easy to see that this is the case, because the range of \(\Phi_{\sigma^0, \sigma}\) is the collection of all continuous functions from \(\text{Dom}_\sigma\) to \(\text{Dom}_{\tau}\), and the range of \(\Phi_{\eta, \sigma}\) is the whole product \(\text{GP}(\text{Dom}_{\eta, \sigma}) \mid a \in \text{Kind}_{\eta, \sigma})\)

**Discussion**

The fact that we have used the category CPO is not essential. Other O-categories could be used, for instance the category of directed complete partial orderings (posets with lubs of all directed sets; a set \(S\) is directed if every finite subset of \(S\) has an upper bound in \(S\)) or complete lattices types would then be interpreted as directed complete partial orderings or complete lattices.

A drawback of the model we have described is that the interpretations of the polymorphic datatypes are too big, because the cpo \(\coprod \text{Dom}_{\eta, \sigma}\) is isomorphic with the whole product \(\coprod \text{Dom}_{\eta, \sigma}\). This product contains many functions which are not parametric, i.e., which behave in completely different ways when applied to different arguments. For example, the cpo \(\coprod \text{Dom}_{\sigma, \text{bool}}\) includes functions that return true or false depending on the type-argument that they get, whereas all closed terms of type \((\sigma, \text{bool})\) are constant functions. Alternative models which do not have this drawback are the PER-based models described in [AP90] and [BM92], which provide much smaller interpretation of the polymorphic types. In these models \(\coprod \text{Dom}_{\sigma, \text{bool}}\) does indeed only contain constant functions.

However, an advantage of the model construction we have described is that it is much simpler. All that is needed for the model construction is the collection of domain equations given by the general model definition. It can easily be extended to include other type-constructions on cpos, such as smashed products and coalesced sums. It can also be extended to incorporate subtyping, as is shown in [PHE93].
8.3 The Programming Logic

As in the previous chapters, given any model for the programming language that meets the general model definition, a proof-irrelevance semantics for the associated logic can be defined. In fact, because in $\lambda \omega_L^\mu$ we have the same logical primitives as in $\lambda \omega_L^+$, the proof-irrelevance model for $\lambda \omega_L^\mu$ can be defined exactly as it was defined for $\lambda \omega_L^+$ in section 7.3.

Let

$$M = (KIND, Dom, \Phi^-, \Phi^+, \Phi^D, \Phi^X, \Phi^+, \Phi^\mu)$$

be any environment model for $\lambda \omega_L^\mu$. A model for $\lambda \omega_L^\mu$ in which $M$ is the sub-model for $\lambda \omega_L^\mu$ can be defined as follows:

- the interpretation of the kinds, datatype-constructors and programs is given by $M$, because by lemma 4.37 $\lambda \omega_L^\mu$ has the same kinds, datatype-constructors and programs as $\lambda \omega_L^+$.

- the interpretation of all other expressions is defined as for $\lambda \omega_L^+$ in section 7.3. So the interpretation of the propkinds is as given in definition 6.33, (in $\lambda \omega_L^+$ propkinds were interpreted as in $\lambda \omega_L$), the interpretation for the prop-constructors is as given in definition 7.24, and all proof terms are interpreted as the element of 1.

The properties proved in section 7.3 for the proof-irrelevance model for $\lambda \omega_L^+$ also hold for this proof-irrelevance model for $\lambda \omega_L^\mu$, and they can be proved in exactly the same way.

8.31 THEOREM (Soundness).

1. Let $der_1$ and $der_2$ be derivations of $\Gamma \vdash a : A$ and $\Gamma \vdash a : A'$, respectively. Then

$$[[\Gamma \vdash a : A]]_\eta = [[[\Gamma \vdash a : A']]_\eta,$$

where the meanings are defined using $der_1$ and $der_2$ respectively.

2. If $\Gamma \vdash a : A$, then for all $\eta \models \Gamma$

$$[[\Gamma \vdash a : A]]_\eta \in \begin{cases} \text{Kind}_A & \text{if } \Gamma \vdash A : \Box_s \\
\text{Dom}_{[\Gamma \vdash A : \Box_s]} & \text{if } \Gamma \vdash A : *_s \\
[\Gamma \vdash A : \Box_p]_\eta & \text{if } \Gamma \vdash A : \Box_p \\
[\Gamma \vdash A : *_p]_\eta & \text{if } \Gamma \vdash A : *_p \end{cases}$$

3. If $a \sim_{\beta_1} a'$, $\Gamma \vdash a : A$, and $\Gamma \vdash a' : A$, then $[[\Gamma \vdash a : A]]_\eta = [[[\Gamma \vdash a' : A]]_\eta$ for all $\eta \models \Gamma$. □

As before, the proof-irrelevance model can be used to prove consistency (of contexts)

8.32 COROLLARY (Consistency of $\lambda \omega_L^\mu$).

$\lambda \omega_L^\mu$ is consistent (False is not provable).

8.33 LEMMA (Consistency of $\Gamma_{CPO}$ and $AXIOM^\mu$ in $\lambda \omega_L^\mu$).

False is not provable in the context $\Gamma_{CPO}$ extended with axioms from $AXIOM^\mu$. 
PROOF. \([\text{False} : *_p] = 0\), so we supply a model \(\mathcal{M}\) for the programming language and show that in the resulting \(\omega^\omega_\omega\)-model there is an environment \(\eta\) that satisfies the context \(CPO\) extended with any axioms from \(AXIOM^n\).

As \(\mathcal{M}\) we can take the model constructed in section 8.2.

The environment \(\eta\) should of course interpret the relation \(\subseteq\) \((Ho \star, \alpha \rightarrow \alpha \rightarrow *_p)\) declared in \(CPO\) as (the characteristic function of) the cpo-ordering in the model, i.e.

\[
\eta(\subseteq) = \lambda a \in Kind_a. \lambda x, y \in Dom_a. \begin{cases} 1 & \text{if } x \subseteq_{Dom_a} y \\ 0 & \text{if } x \not\subseteq_{Dom_a} y \end{cases}
\]

With this interpretation of \(\subseteq\), it can then be checked that all the axioms in \(CPO\) and \(AXIOM^n\) are true in the model, i.e. that they are interpreted as 1.

There is an apparent discrepancy between the notion of chain in the model and the notion of chain defined in \(CPO\). In the model \(\mathcal{M}\) constructed in section 8.2 all datatypes are \(\omega\)-cpos, which means that chains of the form \(c_0 \subseteq c_1 \subseteq c_2 \subseteq \ldots\) have a limit. In \(CPO\) (definition 5.16), on the other hand, a chain is defined as a totally ordered set, and the axiom \textit{complete} in \(CPO\) states that all these chains have limits. However, because all cpos in \(\mathcal{M}\) are countable, these two notions of chain are equivalent. \(\Box\)
Chapter 9

Conclusion

We now look back on our work, compare it with related work by others, and point out some directions for future research.

9.1 Evaluation

We have investigated the possibility of using type theory as the basis for a typed functional programming language and an associated external programming logic. Three programming logics have been introduced, for three richly-typed functional programming languages of increasing complexity.

The basic system $\lambda \omega_L$ is a small and homogeneous system, which is defined as a single PTS. It provides one language for datatypes, programs, specifications, propositions and proofs, including proofs that programs meet certain specifications. The programming language and the logic are well-matched, because we use a single formalism for both of them. For every abstraction that is possible in the programming language, the logic provides matching abstractions to reason about it. For instance, in the programs we can abstract over all datatypes – $\lambda x \cdot x$ – and in propositions we can quantify over all datatype – $\forall x \cdot x$.

Mechanical verification of proofs in these systems is possible. Verification of proofs amounts to type-checking. The conversion relation in the programming language has to be kept decidable, which does impose some restrictions when the programming language is extended with recursion in chapter 5 (see discussion 5.10).

The logics are all higher-order logics, providing very powerful quantifications. This is useful for an implementation, because without these quantifications many propositions that can be now expressed inside the system would have to be given as schematic rules, and then it would be up to the implementation to provide ways for dealing with the necessary quantifications. There are however occasions where the abstractions that are possible are still not powerful enough. For instance, it is not possible to quantify over all kinds or over all labelled products and sums.

Advantages of using Pure Type Systems

The notion of PTS has proved very useful. By defining the basic system $\lambda \omega_L$ as a PTS, we obtain a lean and elegant system. Also, PTSs are by nature very orthogonal systems. The different kinds of abstraction that are allowed in a PTS can be freely combined.
CHAPTER 9 CONCLUSION

One advantage of using PTSs is that we get some of the syntactic properties for free. However, as proofs of these properties are easy, but boring, induction proofs, we feel this is not the most important advantage. More important is that the compact specifications of PTSs make it easy to compare and relate different systems. This helped to guide the design of the basic system in chapter 3. It is easy to distinguish the *s* part of the system that provides the programming language, and the *s*p part of the system that provides the logic, and the effect of adding a single PTS-rule can be considered. It also helped to prove strong normalisation of the systems. It is easy to observe that strong normalisation of \( \lambda \omega L \) and its extension with definitions \( \lambda \omega L_\delta \) immediately follows from strong normalisation of \( \lambda C_\delta \), the Calculus of Constructions with definitions.

The fact that DPTSs can easily be compared also provided the key to the proof of strong normalisation for \( \lambda C_\delta \), the Calculus of Constructions with definitions. Strong normalisation of this DPTS can easily be deduced from SN of a related PTS (namely ECC).

Advantages of separating programs from proofs

In the introduction we explained the difference between internal and external programming logics, and we motivated our choice for external programming logics: \( \lambda \omega L \) and its extensions are all external programming logics. Programs and their correctness proofs are separate but related objects, as are datatypes and specifications. The datatypes express as much information as is possible without making typing undecidable or introducing irrelevant proofs objects in programs. In other words, datatypes express structural information but not logical information.

The strict separation between logic and programming language, and the fact that programs and datatypes cannot depend on propositions and proofs, has several advantages.

First, it makes it possible to extend programming language and logic in different directions. For example, in proofs we want to use classical logic and in programs we want to use recursion. The fact that datatypes cannot express logical information makes it possible to have a fixpoint operator for all datatypes without causing inconsistencies in the logic. The fact that we are not interested in the algorithmic content of proofs means we can use classical logic.

An example of a further extension of the programming language, which immediately suggests itself once we have labelled products and sums, is subtyping. Subtyping would provide some much-needed flexibility in the type system. It is more difficult to envisage a notion of subtyping in internal programming logics.

The separation between programming language and logic means that for the larger part of the system the semantics can be kept simple. A simple truth-value interpretation can be used for the propositions and their proofs. It also makes it possible to give a conventional denotational semantics for the programming language \( \lambda \omega L^\delta \), which interprets recursion in the usual way.

Pragmatics

We have shown that although they are separate objects, programs and their correctness proofs can be constructed together. This involves a double bookkeeping, of programs and their datatypes on the one hand, and predicates and correctness proofs on the other.

We have shown that for each language construction in the programming language there is not only a typing rule, but there are also — possibly several — associated proof rules that
are derivable in the logic. These rules can be used for the top-down, or compositional, development of programs together with their correctness proofs.

The pragmatics of program construction still requires a lot more work. Most of the primitives for program construction offered by $\lambda \omega_1$ and its extensions are very primitive indeed. For practical program construction larger building blocks for programs together with their typing and useful proof rules are needed. An example of this is given in chapter 4, where some derivable proof rules are given for a more complicated language construction, namely the abtype-construct for abstract datatypes.

Apart from these compositional rules, we have also given a few examples of provable equalities between programs. These provide correctness-preserving transformations which can be used to prove that a program meets a certain specification. Examples of simple program transformation rules have been given in chapters 4 and 5. Note that any conditions that have to be satisfied for a transformation to be correct can also be given in the systems.

### 9.2 Comparison with related work

Several ways of using type theory as a formal basis for programming logics have been proposed. We now compare some of these approaches with ours.

#### Program Extraction in the Calculus of Constructions

The ongoing work on program extraction from constructive proofs in the Calculus of Constructions and its successor Coq, based on [PM89a], is closely related to our work, at least as far as the type systems that are involved are concerned. In section 3.5 we have already compared the type systems involved in the two approaches. The target language of the extraction procedure is the same programming language provided by $\lambda \omega_1$, namely Girard's system $F^\omega$, or in PTS-terms, the PTS $\lambda \omega$.

From a pragmatic point of view there is a big difference. Instead of constructing a proof from which a program and its correctness proof are then extracted, we directly construct a program with a correctness proof. In the introduction we already motivated our choice for an external rather than an internal programming logic. An advantage is that the program under construction is clearly visible at all times. Design choices can then be made not only on the basis of the specification, but also on the basis of an operational understanding of the algorithm and efficiency considerations. Also, we have the possibility of giving an incomplete specification, or of specifying only a part of the program.

A big difference with our approach is the way recursion is dealt with. As shown in chapter 5, we can extend the programming language with a fixpoint operator for making recursive programs. In the programming language there are then non-terminating programs and partial objects, and domain theory can be used to reason about these.

When the Calculus of Constructions is used as an internal programming logic, a fixpoint operator cannot be added because this would make the system inconsistent. However, the system can be extended with a a well-founded recursor instead of a fixpoint operator. With such a well-founded recursor any recursive program can be constructed, provided a proof is supplied that the recursion is well-founded.

The restriction to well-founded recursion has obvious advantages. All programs terminate and there are no partial functions. For example, this means that there can be a type nat
which contains the natural numbers. If we have a fixpoint operator, any type nat will also contain a bottom element, and a predicate Nat : nat → *p is needed to characterise the natural numbers. A disadvantage is that there is a mismatch between the programs and datatypes in the programming logic and those available in actual programming languages. For example, in the programming logic it is not possible to reason directly about the datatype of possibly infinite lists as available in lazy functional programming languages.

This difference is a fundamental difference between programming logics where programs cannot depend on proofs and programming logics where programs can depend on proofs. In programming logics where programs cannot depend on proofs, partial functions and objects cannot be avoided if we want more flexible forms of recursion than primitive recursion, or similar fixed recursion patterns for which termination is guaranteed (e.g. [Men87])

Deliverables

Burstall has introduced the name deliverable for a pair consisting of a program and a correctness proof. The approach to program construction using deliverables, discussed in [BM90] [McK92], is similar to ours in spirit. In this approach the system ECC, the Extended Calculus of Constructions [Luo94], is used as an external programming logic for programs in the simply typed lambda calculus.

Our difference with our work is the type systems used for the programming language. Instead of the higher order lambda calculus \( \lambda \omega \) that we use, the simply typed lambda calculus \( \lambda \rightarrow \) is used.

The main difference with our approach is that the strong sum types of ECC are used to form pairs of programs and correctness proofs - deliverables - and pairs of datatypes and predicates - specifications - inside the system ECC. This has the advantage that these pairs can then be manipulated inside the system. Some examples of operations on specifications - i.e. pairs of datatypes and predicates - that can be defined in ECC are given in [Luo91].

Manipulating these pairs inside ECC does impose some restrictions on the shape of programs and specifications. Programs have to be given in a combinatorial notation. It is not immediately possible to express a relation between input and output of functions, but this requires so-called second-order deliverables.

Although the intention is the construction of programs in the simply typed lambda calculus, the system ECC is really much too powerful for this purpose. Also, ECC does not separate the world of programs and datatypes from the world of proofs and propositions in the way that \( \lambda \omega_L \) and its extensions do. The type \( *_p \) of all propositions is in fact treated as a datatype. Because of this, it is possible to form programs and datatypes that depend on proofs or propositions. In particular, the specification formed by a datatype and a predicate can again be used as a datatype. This means that the introduction of a fixpoint operator for programs would make the system inconsistent.

ATTT

Hayaishi has introduced the programming logic ATTT [Hay93]. The approach to program construction he suggests is also very close to ours. As far as the programming language is concerned, the only difference is that the second order lambda calculus \( \lambda \omega \) is used instead of the higher order lambda calculus \( \lambda \omega_L \). And as in \( \lambda \omega_L \), in ATTT programs and datatypes
cannot depend on proofs or propositions. Instead of predicates on datatypes, Hayashi uses subsets of datatypes, which are called refinement types.

The possibility of including well-behaved recursive datatypes, such as natural numbers and lists, with primitive recursors is considered. However, the problem of providing more flexible forms of recursion than primitive recursion in the system is not considered.

Using a logical framework

If we are not interested in execution of programs, but just want computer assistance for constructing correct programs, and mechanical verification of proofs, then there is an alternative to implementing \( \lambda \omega_L \) (or, indeed, to implementing any of the other theories mentioned above). This alternative is to use a so-called logical framework, such as AUTOMATH [dB80] or Edinburgh LF [HHP93].

These logical frameworks are type theories in which arbitrary formal systems can be represented, for example the programming logic of our choice. We then do not directly interpret certain types as datatypes and others as propositions. Instead, we define representations for programs, datatypes, propositions, and specifications in the logical framework, and supply all the typing rules for programs and all the inference rules for proving propositions. The logical framework then provides support for proving statements in the formal system that has been given, such as "program \( M \) has datatype \( \sigma \)" or "program \( M \) satisfies predicate \( P \)."

There are several differences between implementing \( \lambda \omega_L \) directly, and encoding it in a logical framework.

The obvious advantage of the second approach is that an existing implementation can be used. However, the price for this is that the encoding of a formal system in the logical framework makes things more difficult to read and write.

Because we are already interested the \( \ast \)-part of \( \lambda \omega_L \) by itself, as a typed programming language, it seems preferable not to use a logical framework. The important disadvantage of using a logical framework is namely is that statements such as "program \( M \) has datatype \( \sigma \)" are expressed inside the framework, and have to be (dis)proved using the type inference rules. The whole point of having a typed programming language is that the verification of statements \( \Gamma \vdash M . \sigma \) is left to the type checker, so that we do not have to prove these ourselves.

**LCF**

In chapter 5 the extension of the programming language with a fixpoint combinator is considered. Domain theory is used as the basis for reasoning about recursive programs in the associated programming logic \( \lambda \omega_L^{\mu} \). This means we reason about recursion in the same way as in LCF [GMW79], [Pan87]. There are however many differences between LCF and \( \lambda \omega_L^{\mu} \).

Like the logical frameworks discussed above, LCF does not offer the possibility of executing programs. Also, LCF is only concerned with verification of programs, i.e. with proving that a given program meets a certain specification, and not with the construction of correct programs.

In \( \lambda \omega_L^{\mu} \) programs are explicitly typed, whereas in LCF programs are implicitly typed, in the style of ML. Finally, \( \lambda \omega_L^{\mu} \) provides a much richer logical language than LCF. LCF is a first order logic, and the original definition of LCF in [GMW79] provides conjunction and universal quantification as the only logical connectives, and \( \subseteq \) as the only predicate symbol.
\( \lambda \omega_L \) is a higher-order logic, in which predicates are first-class citizens. This has the advantage that many interesting predicates can be defined or declared inside the formal system. For example, we have shown that the notion of admissibility can be expressed in \( \lambda \omega_L \), whereas in LCF this is left to the meta-theory. This additional strength of \( \lambda \omega_L \) makes it possible to build more interesting theories inside the system, in which we no longer have to refer to the cpo-ordering \( \sqsubseteq \) and on the bottom element \( \bot \).

9.3 Future Work

As we have already mentioned, a lot more work has to be done to make program construction in \( \lambda \omega_L \) and its extensions practical. The language constructs offered by the type lambda calculi are very primitive, but also very powerful. For practical program construction we want to have more convenient constructs for building programs at our disposal, with associated proof rules. We have given a few simple examples of derived constructs, e.g. if-then-else and abstype, and shown how proof rules for them can be obtained. More of such language constructs are needed, with associated proof rules, and possibly some special syntax to keep things readable. In addition to such compositional rules for these constructs, we also want program transformation rules for them which can be used to prove correctness of programs.

In the system with recursion, it is awkward to have to go back to the cpo-ordering \( \sqsubseteq \) every time we want to prove a property of a recursive program. Many recursive programs will follow more or less standard recursion patterns, such as primitive recursion, or recursion on the natural numbers or lists with all recursive calls on smaller natural numbers or shorter lists. For such common recursion patterns we would like to prove termination once and for all, and obtain associated induction rules that can be used to prove properties of programs using that recursion patterns.

The explicit typing in the programming language and the logic results in a large amount of fairly trivial bookkeeping, which is well-suited for machine support by an implementation of the type systems. Such an implementation would offer a programming environment that combines a syntax-directed editor for well-typed programs and a goal-directed proof assistant. Suitable tactics for the construction of programs with their correctness proofs would have to be investigated.

At several places we have introduced special notation for certain language constructs. Sometimes we also omitted type parameters, arguing that these could easily be reconstructed from the context. An implementation should ideally offer such a flexible syntax, with possibilities to introduce new notations and to suppress some of the type information.

A topic for future research are further extensions of the programming language Cardelli’s language Quest [Car89], which is based on \( \lambda \omega \), suggests several possibilities to extend the programming language. Here it is useful that we have a more or less conventional denotational semantics for the programming language, because it can accommodate many extensions.

One important extension is subtyping. As shown in [PHE93], the cpo-model given in chapter 8 can be extended to interpret subtyping. Subtyping makes the type system more flexible. It makes it easy to extend the datastructures used in a program, and hence supports the evolution of programs. Higher-order lambda calculi such as \( \lambda \omega \) extended with subtyping have been used to mimic some of the features of object-oriented programming languages [PT92] [FM94]. By incorporating subtyping in \( \lambda \omega_L \) and its extensions we could investigate suitable logical rules for reasoning about these features.
Bibliography


Index

→types, 12
\[\rightarrow\], 109
for PER-model, 117
+types, 60–62, 67–69
\[\top\], 130
for PER-model, 137
\[\times\]-types, 59, 65, 67
\[\times\], 130
for PER-model, 137
\[\Pi\]-types, 12
\[\Pi\], 109
for PER-model, 117
\[\Sigma\]-types, 62–65
\[\Sigma\], 130
for PER-model, 137
\[\mu\]-types, 86
\[\mu\], 142
\[\leftrightarrow\], 35
\neg, 36
\forall, 34
\[\forall\], 119
\exists, 35, 76
\[\exists\], 139
\Rightarrow, 34
\[\Rightarrow\], 119
\forall, 35, 76
\[\forall\], 139
\land, 35, 76
\[\land\], 139
\exists, 88
\ast (type of datatypes), 28
\ast_p (type of propositions), 33
\Box, 28
\Box_p, 33
0, 119
1, 119
e (composition), 84
\eta \equiv \Gamma, 126
\eta \equiv \Gamma^\Box, 110
\eta \models \Gamma^\Box\ast, 113
\eta \models \Gamma^\Box\ast D, 124
I_{LOGIC}, 35
I_{CPO}, 95

consistency of, 157

for datatype-constructors of \[\omega_2\], 110
for datatype-constructors of \[\omega_2^+\], 131
for datatype-constructors of \[\omega_2^\ast\], 142
for programs of \[\omega_2\], 114
for programs of \[\omega_2^+\], 134
for programs of \[\omega_2^\ast\], 146
for proofs, 126
for prop-constructors of \[\omega_2\], 124
for prop-constructors of \[\omega_2^+\], 140
for prop-constructors of \[\omega_2^\ast\], 157
for prop-kinds of \[\omega_2\], 123
for prop-kinds of \[\omega_2^+\], 139
for prop-kinds of \[\omega_2^\ast\], 157
\[\triangleright\] (one-step reduction), 11
\[\triangleright\triangleright\] (many-step reduction), 11
\[\triangleright_{\beta}\] (\[\beta\]-reduction), 11
for \[\times\]-types, 59
for +types, 62
for \[\Sigma\]-types, 64
for labelled products, 66
for labelled sums, 69
\[\triangleright_{\delta}\] (\[\delta\]-reduction), 16
\[\triangleright_{\mu}\] (\[\mu\]-reduction), 86
\approx (conversion), 11
\equiv_3, 11
\equiv_2, 16
\equiv_L (Leibniz equality of programs), 41
\[\lambda\] \rightarrow (simply typed \[\lambda\]-calculus), 15
\[\lambda_2\] (second-order typed \[\lambda\]-calculus), 15
\[\lambda\omega\] (higher-order typed \[\lambda\]-calculus), 15
\[\lambda\mathbb{C}\] (Calculus of Constructions), 15
\[\lambda\mathbb{C}_{pa}\], 54

170
for $\lambda \omega^n_{s}$, 146
for datatype-constructors of $\lambda \omega^s$, 110
for datatype-constructors of $\lambda \omega^+_s$, 130
for datatype-constructors of $\lambda \omega^\mu_s$, 142

environment model
for $\lambda \omega$, 114
for $\lambda \omega^+_s$, 136
for $\lambda \omega^\mu_s$, 148
for datatype-constructors of $\lambda \omega_s$, 111
for datatype-constructors of $\lambda \omega^+_s$, 131
for datatype-constructors of $\lambda \omega^\mu_s$, 143

extensionality, 43

$F^n$ (see $\lambda \omega$), 15
false, 71
False, 35, 76
fixpoint induction, 93, 98
FS, 144, 151
functional (D)PTS, 14
functional (D)PTS with $+, \times$ and $\Sigma$, 58
FV (free variables), 10

global definition, 16
GP, 144, 151
GS, 144, 151

if, 71
kind, 28
Kind (kinds), 30

labelled product types, 65–67
labelled sum types, 67–69
Leibniz’ equality, 41
interpretation of, 126
local definition, 16

nat, 88
O-category, 151

PCons (prop-constructor).
for $\lambda \omega_L$, 39
for $\lambda \omega^+_L$, 77

Pkind (prop-kinds), 39
PKkindp, 119
polynat, 32
product category, 150
Prog (programs)

for $\lambda \omega$, 30
for $\lambda \omega^+_s$, 74
for $\lambda \omega^\mu_s$, 88

program, 28
program extraction, 4, 54–56
Proof (proof terms)
for $\lambda \omega_L$, 39
for $\lambda \omega^+_L$, 77

proof, 33
prop-constructor, 33
prop-kind, 33

proposition, 33
PTS, 10 15
PTS with $+, \times$ and $\Sigma$, 58–70

$R$ (PTS-rules), 10
$R^+$, 58
$R^X$, 58
$R^H$, 58
$R^E$, 58

S (PTS-sorts), 10
second-order typed lambda-calculus ($\lambda \omega$), 15
simply typed lambda-calculus ($\lambda \rightarrow$), 15
SN (see strong normalisation), 14

soundness of $\beta$-conversion
for datatype-constructors of $\lambda \omega_s$, 112
for datatype-constructors of $\lambda \omega^+_s$, 131
for datatype-constructors of $\lambda \omega^\mu_s$, 143
for programs of $\lambda \omega_s$, 116
for programs of $\lambda \omega^+_s$, 136
for programs of $\lambda \omega^\mu_s$, 148
for $\lambda \omega_L$, 124–126
for $\lambda \omega^+_L$, 140
for $\lambda \omega^\mu_L$, 157

soundness of type assignment
for datatype-constructors of $\lambda \omega_s$, 111
for datatype-constructors of $\lambda \omega^+_s$, 131
for datatype-constructors of $\lambda \omega^\mu_s$, 143
for programs of $\lambda \omega_s$, 114
for programs of $\lambda \omega^+_s$, 136
for programs of $\lambda \omega^\mu_s$, 148
for $\lambda \omega_L$, 125–126
for $\lambda \omega^+_L$, 140
for $\lambda \omega^\mu_L$, 157

specification, 44
specification of (D)PTS, 10
specification of (D)PTS with +, × and Σ, 58

SR (see subject reduction)

strong normalisation (SN), 14
  for δ-reduction, 18
  of λC, 15
  of λC₆, 20
  of λω₆, 31
  of λω₆ₐ, 40
  of λω₆ₘ, 75
  of λω₆ₘ⁺, 78
  of λω₆ₘ⁻, 90

strong sum types, 64–65

subject reduction (SR)
  for PTS, 14
  for PTS with +, × and Σ, 70
  for DPTS, 19
  for DPTS with +, × and Σ, 70
  for λω₆, 89
  for λω₆ₘ, 90

T (pseudoterms), 10
true, 71

unit, 71

uniqueness of types (UT)
  for PTS, 14
  for PTS with +, × and Σ, 70
  for DPTS, 19
  for DPTS with +, × and Σ, 70
  for λω₆, 89
  for λω₆ₘ, 90

unit, 71

UT (see uniqueness of types)

weak sum types, 62–65

where, 64
Curriculum Vitae

14 juli 1967 geboren te Haarlemmermeer

mei 1985 diploma International Baccalaureate,
International School of London

september 1985 aanvang studie Informatica

april 1990 doctoraal examen Informatica, met lof,
Technische Universiteit Eindhoven

1990 – 1994 onderzoeker in opleiding aan de Technische Universiteit Eindhoven,
in het kader van NWO/SION-project "Getypeerde Lambda Calculi",
onder leiding van dr.ir. C. Hemerik

vanaf oktober 1994 EuroFocs Fellowship aan INRIA Rocquencourt, Frankrijk

huidig adres Faculteit Wiskunde en Informatica
Technische Universiteit Eindhoven
Postbus 513
5600 MB Eindhoven
Nederland

e-mail erik@win.tue.nl
STELLINGEN

behorende bij het proefschrift

A Programming Logic Based on Type Theory

van

Erik Poll

1 De uitbreiding van het Bruce-Meyer-Mitchell model met \( \Sigma \)-types is niet zo eenvoudig als wordt beweerd in [1] (zie hoofdstuk 7 van dit proefschrift).


2 De in [2] beschreven (co)inductieve types met (co)recursoren kunnen in de polymorfe lambda-calculus \( \lambda \Sigma \) uitgebreid met inductieve types en primitieve recursoren (beschreven in bijv. [3]) worden gecodeerd.


3 In [4] wordt opgemerkt dat voor de type-constructor "\( \rightarrow \)" subtypering contravariantie vereist om recursieve types covariantie, en dat de combinatie van subtypering en recursieve types hierdoor problematisch lijkt. Bij niet-standaard gebruik van de inverse-limitconstructie zijn deze schijnbaar tegenstrijdige eisen aan een eenvoudig te verenigen (zie [5]).


4. In typcrings-algoritmen voor Pure Type Systems is het handig om naast het type van een term tegelijkertijd ook het type van dat type te bepalen.


5. Aangezien er meer ervaring is met het ontwikkelen van grote programma's dan met het ontwikkelen van grote formele bewijzen, is het nuttig om bij het ontwerp van bewijsten te kijken naar fachterten die programmeertalen bieden.

6. Bewijzen met inductie zijn vaak goede voorbeelden van bewijzen met intimidatie.

7. Promotie aan een universiteit zou niet direct gevolgd mogen worden door een aanstelling aan dezelfde universiteit.

8. Tekstverwerkers maken het aanbrengen van veranderingen te eenvoudig.

9. Het is jammer dat zowel Griekse hoofdletters ook Latijnse hoofdletters zijn.

10. Het nadeel van computernetwerken is dat het aftrekken van agressie op de apparatuur op je bureau minder bevredigend is.