Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow

Jan Zwanenburg
Object-Oriented Concepts and Proof Rules:
Formalization in Type Theory and
Implementation in Yarrow
Zwanenburg, Jan

Object-oriented concepts and proof rules : formalization in type
theory and implementation in Yarrow / by Jan Zwanenburg. -
Proefschrift. - ISBN 90-386-0731-8
NUGI 852
Subject headings : lambda calculus / abstract data types / object
oriented programming languages
CR Subject Classification (1998): F.4.1, D.3.1, D.3.3, D.3.2

The work in this thesis has been carried out under the auspices of the research school IPA
(Institute for Programming research and Algorithmics).

IPA dissertation series 1999-12
Printed by University Press Facilities, Eindhoven
Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 8 december 1999 om 16.00 uur

door

Jan Zwanenburg

geboren te Drachten
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. F.E.J. Kruseman Aretz
en
prof.dr. H.P. Barendregt

Copromotor:
dr.ir. C. Hemerik
Acknowledgements

First of all, I thank Kees Hemerik, my co-promotor, for giving me this Ph.D. position, and introducing me to the beautiful field of type theory. Furthermore, during numerous discussions he taught me to do things one at a time, and to keep them as simple as possible (although we often varied in opinion on what is simple). Two years ago, he delegated parts of the supervision to F.E.J. Kruseman Aretz and Erik Poll.

I thank F.E.J. Kruseman Aretz, my first promotor, for the careful reading of various versions of this thesis, and pointing out many errors, inconsistencies, and things unclear for people not totally immersed in type theory. His being formally retired only increases my gratitude.

My thanks go to Erik Poll for the pleasant collaboration, which led to Chapter 6, and the time he spent on reading many of my manuscripts, correcting my English, suggesting better explanations, and indicating technical problems.

I thank my second promotor, Henk Barendregt, and the other member of the manuscript committee, Gerard Renardel de Lavalette, for their comments and suggestions. I am grateful to Adriana Compagnoni for reading Chapter 7, revealing a few important errors and suggesting improvements, and to Eric Rajmakers for the pleasant collaboration leading to the graphical user interface of Yarrow.

Being member of the type-theoretical group has been stimulating. For that I thank Roel Bloo, Tijn Borghuis, Rob Nederpelt, and Martijn Oostdijk in general, Herman Geuvers, Twan Laan and Paula Severi for the fruitful collaboration, and Michael Franssen for being my roommate and putting up with my smoking.

Last of all, I thank life itself, because it is very interesting.

Jan Zwanenburg
October 4, 1999
Contents

Acknowledgements 5

1 Introduction 11
1.1 Subject of this Thesis 11
1.2 Pure Type Systems 13
1.3 The Programming Logic \( \lambda \omega_L \) 13
1.4 Research Objective 14
1.5 Contributions of this Thesis 15
1.6 Structure of this Thesis 15
1.7 Presentation of Theory 18

I Type Theory and Yarrow 19

2 Pure Type Systems 21
2.1 \( PTSs \) 22
2.1.1 Standard Typing Rules 24
2.1.2 Meta-theory for \( \Gamma_B \) 25
2.1.3 Examples of \( PTSs \) 27
2.2 A Different Presentation of the Typing Rules 27
2.2.1 Meta-theory for \( \vdash \) 28
2.2.2 Type-Checking Algorithm 30
2.3 \( PTSs \) with Definitions 30
2.4 Interpretation of a \( PTS \) as a Logic 33

3 Yarrow 35
3.1 A User's Perspective 36
3.1.1 Main Mode 36
3.1.2 Proof Mode 37
3.2 A Type-Theoretical Perspective 42
3.2.1 Formal Treatment of the Proof Mode 42
3.2.2 Why Holes in Goals are not Allowed 52
3.2.3 Syntactic Sugar: Infix Notation and Implicit Arguments 52
3.2.4 Flags 53
3.2.5 Related Work 54
3.3 Software Architecture 56
3.3.1 Design Issues 56
CONTENTS

3.3.2 Monads .......................................................... 57
3.3.3 System Architecture ........................................... 59
3.3.4 Top Level User Interface ...................................... 60
3.3.5 Engine .......................................................... 63
3.3.6 Service Routines ............................................... 66
3.3.7 Conclusion ...................................................... 67

4 $\lambda\omega_L$, a Programming Logic ............................... 71
   4.1 $\lambda\omega$ as a Programming Language : $\lambda\omega_\delta$ .... 72
   4.2 $\lambda\omega$ as a Logic : $\lambda\omega_p$ .......................... 76
   4.3 The Programming Logic $\lambda\omega_L$ ......................... 79
   4.4 Semantics ...................................................... 85
   4.5 Axioms ......................................................... 86
   4.6 Library ......................................................... 89
   4.7 Example: Sorting .............................................. 96
   4.8 Comparison with Program Extraction in $\lambda C$ ........... 98

II Object-Oriented Concepts and Proof Rules ..................... 101

5 Introduction to Objects ........................................... 103
   5.1 What is OOP .................................................. 103
   5.2 An Example OO Program ...................................... 104
   5.3 Modelling OOP ................................................. 105
   5.4 The Structure of the Rest of this Thesis ................. 106
   5.5 Related Work ................................................ 107

6 $\lambda\omega_L^+$ and Modelling Abstract Datatypes ................. 109
   6.1 Syntax ...................................................... 111
   6.1.1 Records .................................................. 111
   6.1.2 Existential Types ....................................... 112
   6.1.3 Meta-Theory .............................................. 114
   6.2 Axioms ........................................................ 115
   6.3 Example: Stack ............................................... 116
   6.3.1 Hiding without Existential Types ....................... 116
   6.3.2 Hiding using Existential Types ......................... 120
   6.4 Parametricity ............................................... 125
   6.4.1 Observational Behaviour and Equality of Implementations 125
   6.4.2 Simulations .............................................. 126
   6.4.3 Parametricity for Existential Types ................... 131
   6.4.4 General Parametricity .................................. 131
   6.5 Example: Bags ............................................... 135
   6.5.1 Implementation 1 ........................................ 136
   6.5.2 Implementation 2 ........................................ 147
   6.5.3 Implementation 3 ........................................ 156
   6.6 General Interfaces and Specifications ..................... 165
   6.6.1 General Interfaces ..................................... 165
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6.2</td>
<td>Sensible Specifications</td>
<td>167</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Well-Behaved Specifications</td>
<td>169</td>
</tr>
<tr>
<td>6.6.4</td>
<td>General Proof Rules for ADTs</td>
<td>175</td>
</tr>
<tr>
<td>6.7</td>
<td>Formalization in Yarrow</td>
<td>179</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Preliminaries</td>
<td>179</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Parametricity</td>
<td>181</td>
</tr>
<tr>
<td>6.7.3</td>
<td>First-Order Interfaces</td>
<td>183</td>
</tr>
<tr>
<td>6.7.4</td>
<td>Quotients and Subsets</td>
<td>185</td>
</tr>
<tr>
<td>6.7.5</td>
<td>Sensible and Well-Behaved Specifications</td>
<td>187</td>
</tr>
<tr>
<td>6.7.6</td>
<td>The Implementor's and the User's Specification</td>
<td>190</td>
</tr>
<tr>
<td>6.8</td>
<td>Axiomatizing Quotients and Subsets</td>
<td>192</td>
</tr>
<tr>
<td>6.8.1</td>
<td>First-Order Algebras</td>
<td>192</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Higher-Order Algebras</td>
<td>194</td>
</tr>
<tr>
<td>6.8.3</td>
<td>Summary of Axiomatizing</td>
<td>200</td>
</tr>
<tr>
<td>6.9</td>
<td>Conclusions</td>
<td>201</td>
</tr>
<tr>
<td>6.9.1</td>
<td>Related Work</td>
<td>201</td>
</tr>
<tr>
<td>6.9.2</td>
<td>Future Work</td>
<td>203</td>
</tr>
<tr>
<td>6.9.3</td>
<td>Formalizing ADTs in (\lambda \omega_{L}^{+}) and Yarrow</td>
<td>205</td>
</tr>
</tbody>
</table>

7 PTSs with Subtyping  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Syntax and Typing Rules</td>
<td>211</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Syntax</td>
<td>211</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Typing Rules</td>
<td>213</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Examples of (PTS^{\leq_{s}})</td>
<td>216</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Alternatives for Rules</td>
<td>222</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Summary of Meta-Theoretical Results</td>
<td>224</td>
</tr>
<tr>
<td>7.2</td>
<td>Meta-Theory</td>
<td>225</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Restricted Subtyping Rules</td>
<td>226</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Generation Properties for Subtyping</td>
<td>230</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Equivalence of Liberal and Restricted Subtyping Rules</td>
<td>234</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Subject Reduction</td>
<td>239</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Minimal Typing</td>
<td>244</td>
</tr>
<tr>
<td>7.2.6</td>
<td>Relation to PTSs</td>
<td>251</td>
</tr>
<tr>
<td>7.3</td>
<td>Type-Checking Algorithm</td>
<td>252</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Completeness is Problematic</td>
<td>254</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Syntax-Directed Rules</td>
<td>255</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Soundness</td>
<td>260</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Completeness</td>
<td>263</td>
</tr>
<tr>
<td>7.3.5</td>
<td>From Derivation Rules to Algorithms</td>
<td>266</td>
</tr>
<tr>
<td>7.3.6</td>
<td>Termination</td>
<td>269</td>
</tr>
<tr>
<td>7.3.7</td>
<td>Summary of Type-Checking</td>
<td>273</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusions</td>
<td>275</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Related Work</td>
<td>275</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Summary and Conclusion</td>
<td>278</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Subject of this Thesis

The subject of this thesis is most concisely expressed in the title: "Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow". Let us explain the various notions used in this title.

Formalizing programming languages

Formalizing a programming language means developing a theory for the language in which the various concepts of the language get a formal and hence precise meaning. We distinguish three aspects of such theories: syntax, semantics, and proof rules. The syntax of a language determines which programs are well-formed. A semantics describes the meaning of well-formed programs. Proof rules are logical rules for reasoning about correctness of programs. These proof rules are expressed in certain logical systems which are called programming logics. Traditionally, programming logics consisted of three quite separate subsystems, one for programs, one for logical specifications, and one for correctness of programs with respect to specifications (e.g. Hoare logic). This was caused by the differences between the logical language and the programming language.

What is the point of formalization? One purpose is a precise description and a good understanding of the language, and the fundamental concepts it invokes. This is called the descriptive purpose. Another purpose is then to come up with improved languages; the normative purpose. The criteria for how good language features are are orthogonality, compositionality, generality, uniformity, and simplicity. For example, it may turn out that certain constructs are very specialized (not general), that constructs have much in common with other constructs (the set of constructs is not orthogonal), or that certain constructs are hard to formalize, which may indicate that these constructs are too complex.

Type theory

Type theory has proved to be a successful framework for formalizing fundamental concepts of programming languages. Important achievements of type theory are proper formalizations of polymorphism [Rey74], data abstraction [MP84], and subtyping [CW85]. The normative effects are clearly visible in the development of functional programming languages. For the
time being, the effect on imperative languages is mainly descriptive. Type theory has been so successful because it is orthogonal, general and uniform.

Type theory has also had a large impact on the formalization of logic. The orthogonality made it possible to set up new logical systems, more powerful than before. Furthermore, type theory gives a uniform formalization of not only propositions and predicates, but also of proofs. This paved the way for the development of a class of proof assistants, which are implementations of type systems suitable for the interactive construction of proofs. During this interactive construction, proof assistants maintain the bookkeeping and check validity of the proof steps proposed by the user.

The relevance of type theory to both logic and programming languages has been beneficial in several ways, of which we mention two. First, it gives rise to a cross-fertilization between the two areas; for example the current understanding of data abstraction has been found by carrying over the logical concept of existential quantification to the field of programming. Second, the field of programming logics made a large advancement by type theory (e.g. the system Coq [B+97]). Because of the relation of type theory to both programming and logic, programming logics can be defined concisely as a type system in which both programs and datatypes on one hand, and proofs and propositions on the other hand can be expressed uniformly. This stands in contrast to traditional programming logics. Erik Poll defined in his Ph.D. thesis [Pol94] the programming logic $\lambda \omega_L$, which is the starting point of our work. The system $\lambda \omega_L$ is concerned with reasoning about the powerful functional language $F^\omega$ [Gir72] and is based on the Pure Type Systems ($PTS$s, see [Bar92]), which form an important framework within type theory. The orthogonality and flexibility of $PTS$s make it possible to define $\lambda \omega_L$ as a concise system, in which programs, their specifications and proof rules can be expressed.

Object-oriented programming languages

The last couple of decades saw a gradual shift from traditional imperative languages to Object-Oriented Programming (OOP) languages. The popularity of the OOP paradigm has several reasons. OOP allows – more than other paradigms – the reuse of software components. It makes it easy to extend programs. In OOP certain parts of programs can be hidden from other parts, which encourages modular programming. However, many concepts in this paradigm are not clear and there is an abundance of apparently similar concepts. As examples we mention the confusion between subtyping and inheritance [CHC90], and the discussion on which languages are object-oriented, e.g. C++ [Str86] versus Smalltalk [GR83]. So it is necessary to understand and clean up the concepts of this paradigm by formalization. By the descriptive effects of formalization a better understanding of the OOP concepts should be achieved, and by the normative effects the concepts should be made cleaner, more uniform, and more orthogonal.

Type theory, object-oriented programming, and Yarrow

Because of the success of type theory for traditional programming languages, type theory seems a promising approach to gain more insight in OOP (e.g. see [GM94]). Concerning syntax, this promise has been partially fulfilled by the work of Pierce and Turner [PT94]. In this work OOP concepts of a small example language are modelled by well-understood type-theoretical notions. This model can also be thought of as providing a semantics for OOP,
by composing the model with a semantics for the type-theoretical notions used. The goal of our work is to find proof rules for OOP by combining the proof rules for the various type-theoretical ingredients of the model. The type theory we use is (an extended version of) $\lambda\omega_L$. We intend to use formalization in $\lambda\omega_L$ both descriptively and normatively, i.e. we intend to clarify OOP concepts and improve them. We provide machine support for our programming logic in the form of Yarrow, a newly developed proof assistant for arbitrary PTSs.

In the rest of this chapter, we first discuss PTSs and the programming logic $\lambda\omega_L$. Then we will give a more extensive account of our research objective, which will be followed by a description of the main contributions of this thesis. This chapter concludes with an overview of the rest of this thesis, and a brief explanation how theory developed in the programming logic is presented.

### 1.2 Pure Type Systems

The Pure Type Systems (PTSs) form a family of typed $\lambda$-calculi (e.g. [Bar92]). The central notion in this framework is that of the typing judgment. A typing judgment is of the form $\Gamma \vdash a : A$, and expresses that term $a$ has type $A$ in context $\Gamma$. Terms and types belong to the same set of $\lambda$-terms. PTSs have a very concise and elegant definition, but offer a wide range of systems through various settings of the parameters of the framework. An important feature of the framework of PTSs is that many meta-theoretical properties can be proved for large classes of PTSs, so there is no need to prove these properties separately for each particular PTS.

Some PTSs can be seen as typed programming languages. For this a typing judgment $\Gamma \vdash a : A$ is interpreted as "program $a$ has datatype $A$ in context $\Gamma$", and a context $\Gamma$ is interpreted as a set of declarations of primitive types and programming constructs available in the language. So both programs and datatypes are expressed as $\lambda$-terms.

Other PTSs can be seen as logical systems. For this a typing judgment $\Gamma \vdash a : A$ is interpreted as "$a$ is a proof of proposition $A$", and a context $\Gamma$ is interpreted as a set of declarations of propositions and axioms in the logic. So both proofs and propositions are expressed as $\lambda$-terms. Hence proofs are formal objects in this interpretation. Since the steps in proofs are tiny, proof objects are usually huge and unreadable.

Well-known PTSs are system $F$, the second-order or polymorphic $\lambda$-calculus [Gir72, Rey74], $F^\omega$, the higher-order $\lambda$-calculus [Gir72], and $\lambda C$, the Calculus of Constructions [ChI88]. System $F^\omega$ can be interpreted both as a very powerful programming language, and as a higher-order proposition logic.

### 1.3 The Programming Logic $\lambda\omega_L$

The system $\lambda\omega_L$ is a particular PTS, which is used as a programming logic. This system is defined and motivated in Erik Poll's Ph.D. thesis [Pol94]. Here we give a brief summary of the properties of $\lambda\omega_L$.

- $\lambda\omega_L$ is a PTS, and a refinement of $\lambda C$, the Calculus of Constructions, so it has a lot of nice syntactical properties.
- $\lambda\omega_L$ contains a programming language — $F^\omega$ — and a logic for reasoning about these programs and writing specifications for them. Programs, datatypes, proofs and propo-
sitions (in particular, specifications) are all $\lambda$-terms in $\lambda\omega_L$. This is one of the main features of $\lambda\omega_L$; despite its concise definition as a PTS, all four kinds of constructs are expressed in the same system.

- The programming language of $\lambda\omega_L$ contains few but powerful primitives. Most importantly, it contains existential datatypes as in [MP84], which allow abstract datatypes. However, $\lambda\omega_L$ is still not quite enough to do OOP.

One of the main distinctions between $\lambda\omega_L$ and other programming logics based on type theory (e.g. [Moh86]) is that datatypes are not intended as complete specifications, but as partial specifications, which should be easily readable and (mechanically) verifiable. Other advantages of using datatypes as partial specifications are that for large programs often only partial specifications are feasible, and that the use of datatypes in $\lambda\omega_L$ corresponds with their use in the practice of programming (in typed programming languages).

Unfortunately, the bare system $\lambda\omega_L$ is not expressive enough to model object-oriented programming in it.

1.4 Research Objective

Our ultimate objective is to extend the programming logic $\lambda\omega_L$ to an object-oriented programming logic. So we should be able to express object-oriented programs and their specifications in this extended logic, and we should find general proof rules for OO programs.

We have chosen to use $\lambda\omega_L$ as basis for an OO programming logic, because of the merits of $\lambda\omega_L$ given above, and because the programming part of $\lambda\omega_L$ is $F^\omega$, which is a suitable basis to model OOP ([PT94], see below).

The approach to our objective

One aspect of extending $\lambda\omega_L$ with OOP features is the extension of the programming language. Pierce and Turner show in [PT94] that in order to model the most important OOP features it suffices to extend the programming language $F^\omega$ (which is the programming part of $\lambda\omega_L$) with the following four constructs:

1. records,
2. existential types,
3. subtyping, and
4. the fixed point combinator.

The combination of records and subtyping facilitate the reuse of software components and easy extension of programs. Existential types provide the hiding mechanism present in OOP languages. The reason for the fixed point combinator is more technical in nature, and although it is important, it will not be treated in this thesis. In Chapter 5 we explain OOP in much more detail and indicate how these extensions can model OOP features.

Our approach is as follows. First we treat the constructs in isolation, including the development of proof rules for each construct. Then we combine the constructs, as indicated by [PT94], and by corresponding combination of the proof rules we hope to obtain proof rules for OOP.
1.5 Contributions of this Thesis

Proof rules for abstract datatypes

The proof rules for existential types given in Poll's thesis do not properly reflect the hiding achieved by these types. One contribution of this thesis is a proper proof rule for existential types. This rule generalizes and formally justifies the folk principle [GHM78] for Abstract Datatypes (ADTs), which are the most pure applications of existential types.

In [MP84], where existential types are introduced, the use of existential types to prove natural properties about ADTs is already mentioned as future work. In [PA93] a start has been made by the so-called theory of parametricity, but only we complete this line of research.

PTSs with subtyping

Extending the syntax of $\lambda \omega_L$ with subtyping is difficult. The problems lie in showing that the extended type system satisfies the usual soundness properties, such as Subject Reduction. The solutions we found for these problems are not only applicable to $\lambda \omega_L$ with subtyping, but to any PTS extended with subtyping. Our solution is sticking to a rigorous and novel design decision, namely to define the rules for subtyping independently of those for typing.

The definition of PTSs with subtyping, including the development of the meta-theory, forms another contribution of this thesis. This is an important contribution, since this new framework includes not only a number of existing systems with subtyping, but also some new systems with subtyping, interesting as programming language or logic.

The proof assistant Yarrow

We claimed above that non-trivial proofs given as $\lambda$-terms are large and unreadable. Therefore, machine support is indispensable for formal reasoning in $\lambda \omega_L$ and its extensions. Computer programs that offer this support are called proof assistants. The third contribution of this thesis is the proof assistant Yarrow, which handles a wide range of PTSs, including $\lambda \omega_L$. We have used Yarrow throughout this thesis in order to ensure correctness of statements in the logic $\lambda \omega_L$.

These are the three contributions of this thesis. We have not achieved the original goal of obtaining proof rules for OOP, because the work on subtyping and ADTs took too much time. In Chapter 9 we do show how the programming constructs of records, subtyping, and existential types are used to model OOP, but we do not combine the proof rules for these constructs to OOP proof rules.

1.6 Structure of this Thesis

This thesis is divided into two parts. The first part serves as basis for the second part by giving formal tools (PTSs, $\lambda \omega_L$) and the software we need (Yarrow). In the second part we extend $\lambda \omega_L$ to model OOP. Figure 1.1 sketches the technical dependencies between the chapters. This figure may be useful for readers interested in certain subjects only; the descriptions of the chapters below make the dependencies more precise.
CHAPTER 1. INTRODUCTION

![Diagram](image)

Figure 1.1: Technical dependencies between chapters

Contents of Part I

In part I we present the basis of this thesis, which is the programming logic $\lambda \omega_L$ and software support for $\lambda \omega_L$ by Yarrow. This part starts with Chapter 2, describing $PTS$s. Chapter 3 describes the use, the theory and the implementation of Yarrow, which is a proof assistant supporting a wide range of $PTS$s. In Chapter 4 we show how $\lambda \omega_L$ is defined as a $PTS$, how it is interpreted as programming logic, and we develop basic theory and some simple examples with the aid of Yarrow. Next, we give a more extensive abstract of these chapters.

Chapter 2 reviews the theory of Pure Type Systems ($PTS$s), and is largely copied from [Pol94]. This chapter starts with presenting the syntax of the terms, the typing rules, and a number of meta-theoretical properties, which are all standard. Then an alternative formulation of the $PTS$s is given, which makes it easier to extend them with new concepts, such as subtyping. The chapter concludes with the extension of $PTS$s with definitions, which are essential for practical use of these systems.

Chapter 3 discusses the proof assistant Yarrow. First, we discuss why software support is necessary for developing theory in a $PTS$, and why we implemented our own proof assistant instead of using existing software. Then we consider Yarrow from three perspectives. First we take the user's perspective: how Yarrow is operated and how proofs are developed interactively in Yarrow. Second, we consider the theoretical perspective, and in particular discuss the theory of interactive proof development in $PTS$s. The third perspective concerns the software architecture of Yarrow, where the main points are Yarrow's treatment of I/O in the purely functional language Haskell, and the communication between the user interface and the engine. The latter is important since both a textual and a graphical interface have been implemented.

It is not necessary to read Chapter 3 to understand the rest of this thesis. However, Yarrow has been used throughout this thesis to develop and present theory in our programming logic (see Section 1.7 below).

Chapter 4 describes the programming logic $\lambda \omega_L$. It is given as a $PTS$, which is described in three parts, namely first the programming language and the logic separately, and then the entire programming logic. We give a list of axioms and the library of general constructions developed in Yarrow. This library includes some datatypes, some general programming constructs (e.g. concatenation of lists), some predicates (e.g. the $\leq$ ordering on numbers) and general lemmas (e.g. $\leq$ is transitive). With the use of this library, we specify and implement a small example program, and prove its correctness. This chapter also relates $\lambda \omega_L$ to other
programming logics. Apart from the library and the example, most of Chapter 4 is copied from Poll's thesis.

Contents of Part II

In Part II of this thesis we extend $\lambda\omega_L$ to model OOP. This part starts with Chapter 5, which gives a brief introduction to OOP, and indicates the extensions of $\lambda\omega_L$ necessary to model OOP: records, existential types, and subtyping, amongst others. Records and existential types are added to $\lambda\omega_L$ in Chapter 6, resulting in system $\lambda\omega_L^+$. We have chosen not only to extend $\lambda\omega_L$, but the whole framework of PTSs with subtyping. This leads to the framework of $\text{PTS}^L$, a family of PTSs with subtyping (Chapter 7). Then we define in Chapter 8 $\lambda\omega_L^{+\leq}$ as a member of this framework extended with records and existential types. Chapter 9 models OOP using $\lambda\omega_L^{+\leq}$. We give the conclusions of the whole thesis in Chapter 10. We now describe the chapters in Part II in some more detail.

Chapter 5 briefly introduces OOP by explaining a small example OO program, and lists the OOP features we will model (in Chapter 9), and the type-theoretical extensions necessary to model these features. So Chapter 5 motivates the extensions elaborated in Chapters 6 and 7, but understanding OOP is only necessary in Chapter 9. Chapter 9 also gives a comparison with related work on theory of OOP.

Chapter 6 adds records and existential types to $\lambda\omega_L$. The extension of the syntax is rather straightforward. But Chapter 6 shows it is quite an enterprise to develop proper proof rules. We need parametricity to obtain a richer notion of equality on existential types, that properly reflects the hiding achieved by these types. However, this is not enough to obtain proper proof rules for Abstract Datatypes (ADTs), which are the simplest applications of existential types. We need two additional axioms, stating the existence of quotient and subset types respectively, to get proof rules for ADTs which are generalized versions of the folklore proof rules. Only by the use of Yarrow the necessity of the additional axioms became apparent, and we develop the complicated theory.

Chapter 7 adds subtyping to the framework of Pure Type Systems. This chapter is only concerned with extending the syntax, and the proof of a number of meta-theoretical properties. However, this is quite hard. We succeed by sticking to a rigorous and novel design decision, namely to define the rules for subtyping independently of those for typing. This introduces other difficulties, but these are not insurmountable. Most of this chapter is devoted to the meta-theory, which is not necessary to understand the other chapters. On the other hand, no knowledge of $\lambda\omega_L$ is necessary to understand Chapter 7; only the introductory motivation and some minor remarks assume some knowledge of this system.

In Chapter 8 we define $\lambda\omega_L^{+\leq}$ as member of the $\text{PTS}^L$ framework, extended with records and existential types. Here we also give the proof rules for subtyping, which are subtle, but not complicated, and give a small example program which uses subtyping, with its specification and proof of correctness.

Chapter 9 shows how the ingredients of records, existential types and subtyping can be used to model OOP. Our model, based on [PT94], is presented stepwise so that OOP features are modelled one at a time. For some steps we need constructs not discussed before, but which are fairly innocent and are introduced when needed. We speculate on proof rules for objects, because we had insufficient time to combine the proof rules for the ingredients in the manner indicated by the model.
1.7 Presentation of Theory

Virtually all of the theory in $\lambda\omega_L$ (and its extensions) we present in this thesis has been developed interactively in Yarrow, so this theory is mechanically verified, and hence correct and consistent. In order to make this theory readable, we added features to Yarrow to present definitions, theorems and proofs in \LaTeX format rather than plain ASCII. For example:

\[
\text{double} := \lambda x:\text{Nat}. x + x
\]
\[
: \text{Nat} \to \text{Nat}
\]
\[
\text{even\_\_double} := \ldots : \forall x:\text{Nat}. \text{even (double x)}
\]

The first two lines define the \texttt{double} function. Here we see that all identifiers referring to programs and datatypes are printed in the \texttt{ttyle} font. The last line indicates that we have proved a lemma with name \texttt{even\_\_double}: the dots "\ldots" indicate that the large \texttt{\lambda} term representing the proof is left out. All identifiers referring to proofs and predicates are printed in the \texttt{italics} font. The notational conveniences (syntactic sugar) implemented in Yarrow (described in Section 3.2.3) are also used in this thesis.

Normally, we show the lemmas from the theory without proofs for reasons of brevity, but some \texttt{\lambda} terms representing proofs are presented as a deduction in so-called flag style format [Ned90], e.g.

1 \[
\begin{array}{l}
  x:\text{Nat} \\
\end{array}
\]
\text{hyp}

2 \[
\begin{array}{l}
  \text{double } x = x + x \\
\end{array}
\]
\text{def. double}

3 \[
\begin{array}{l}
  \exists y:\text{Nat}. \text{double } x = y + y \\
\end{array}
\]
\text{def. double, 2}

4 \[
\begin{array}{l}
  \text{even (double } x) \\
\end{array}
\]
\text{def. even, 3}

5 \[
\begin{array}{l}
  \forall x:\text{Nat}. \text{even (double } x) \\
\end{array}
\]
\text{def. even, 3, hyp, 2, 4}

These presentations have also been generated by Yarrow, as explained in Section 3.2.4.

In many fragments of theory and deductions some minor modifications have been made by hand, in particular concerning line breaking.
Part I

Type Theory and Yarrow
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 generalization of the systems in Barendregt's lambda cube [Bar92]. The notion of PTS is useful for two reasons. First, some meta-theoretical properties can be proved for all PTSs or for large classes of PTSs at the same time. Second, many type systems can be described as PTSs, and these compact but 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 4.

The "pureness" of PTSs lies in the fact that there is only one type constructor, namely the dependent product Π (which generalizes the function space →), and one reduction rule, namely β-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 [B+97], LEGO [LP92], and CONSTRUCTOR [Hei91], do provide a definition mechanism, even though the underlying type systems do not have them. 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.

This preliminary chapter reviews the theory of PTSs, and is included to make this thesis self-contained. It is divided into three sections. In Section 2.1, the definition of PTSs and their most important properties are given. Section 2.2 presents an alternative — but equivalent — formulation of PTSs which is the basis for the systems in the following chapters. In Section 2.3, the definition of DPTSs and their most important properties are given. Section 2.4 shows briefly how a PTS can be interpreted as a logic. Most of the text of Sections 2.1 and 2.3 is copied from [Pol94].
2.1 P Ts

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

A P Ts is a typed lambda calculus. It can be defined as a 4-tuple consisting of a set of pseudoterms, a set of pseudocontexts, a reduction relation on pseudoterms, and – most importantly – a typing relation. This typing relation is defined by a set of inference rules for deriving judgments 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 \( \Gamma \)".

**Definition 2.1.1** A specification of a P Ts 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 P Ts 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 \( \ast \) for the universe of all types, a sort \( \ast_p \) for the universe of all datatypes, or a sort \( \ast_y \) 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 P Ts specified by \( S = (S, A, R) \) is denoted by \( \lambda S \). The pseudoterms of a P Ts 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 P Ts and a more ad-hoc definition.

**Definition 2.1.2 (Pseudoterms)** The pseudoterms \( T \) of a P Ts \( \lambda(S, A, R) \) are defined by

\[ T ::= V \mid S \mid (T T) \mid (\lambda V : T. T) \mid (\Pi V : T. T) \]

where \( V \) is the set of variables (identifiers). □

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 \( A \equiv B \) if \( A \) and \( B \) are equal up to renaming of bound variables, \( A[x := B] \) for \( A \) with \( B \) substituted for the free occurrences of \( x \), and \( A[x_1 := B_1, \ldots, x_n := B_n] \) for \( A[x_1 := B_1] \ldots [x_n := B_n] \).

**Definition 2.1.3 (Pseudocontexts)** The pseudocontexts \( C \) of a P Ts \( \lambda S \) are defined by

- \( \epsilon \in C \)
- \( \Gamma, x : A \in C \) if \( \Gamma \in C, A \in T, x \in V \) and \( x \not\in \text{FV}(\Gamma) \cup \text{FV}(A) \).

Here \( \epsilon \) is the empty context, and \( \text{FV}(\epsilon) = \emptyset \) and \( \text{FV}(\Gamma, x : A) = \text{FV}(\Gamma) \cup \{x\} \cup \text{FV}(A) \). □
The prefix "pseudo" is used because only the pseudoterms and pseudocontexts that are in the typing relation (i.e. that are well-typed) will ultimately be of interest. These pseudoterms and pseudocontexts will be called the terms and the contexts (Definition 2.2.1.1).

Our notion of pseudocontexts is more restricted than usual. We have chosen to do so, because this is necessary for certain extensions of the PTSs, namely DPTSs (Section 2.3) and PTSXs (Chapter 7). In Chapter 7, we explain in more detail why this restriction is necessary.

Convention: We use meta-variables as follows:

- $a, b, c, d, A, B, C, D$ range over (pseudo)terms,
- $s$ ranges over sorts,
- $x$ ranges over variables,
- $\Gamma$ ranges over (pseudo)contexts.
- Meta-variables may be decorated with subscripts and primes.

Convention: 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.
- Repeated $\lambda$- or $\Pi$-abstractions over the same type may be abbreviated by one abstraction over a list of variables, e.g. $(\lambda x, y: A. b)$ abbreviates $(\lambda x: A. \lambda y: A. b)$.

Definition 2.1.4 (Reduction) The $\beta$-reduction relation $\Rightarrow_\beta \subseteq T \times T$ is defined by

$$(\lambda x: A. b) \ a \ \Rightarrow_\beta \ b[x := a]$$

and the usual compatibility rules.

The reflexive and transitive closure of $\Rightarrow_\beta$ is denoted by $\Rightarrow_\beta$. The reflexive, transitive and symmetric closure of $\Rightarrow_\beta$ is denoted by $\equiv_\beta$. □

Theorem 2.1.5 (Church-Rosser, CR for short)
If $a \equiv_\beta b$ then there is a $c$ such that $a \Rightarrow_\beta c$ and $b \Rightarrow_\beta c$.

Proof sketch: Nowadays, the easiest proof of this property is by showing a PTS is a orthogonal Combinatory Reduction System [KOR93]. □

Definition 2.1.6

1. $x: A \in \Gamma$ if $\Gamma \equiv \Gamma_0, x: A, \Gamma_1$ for some $\Gamma_0, \Gamma_1$.
2. $\Gamma \subseteq \Gamma'$ if $x: A \in \Gamma \Rightarrow x: A \in \Gamma'$ for all $x, A$. □
2.1.1 Standard Typing Rules

In this section we give the usual presentation of the typing rules, as in [Bar92]. In order to
distinguish typing judgments in this standard presentation from the presentation we use in
Section 2.2, the turnstile symbol is subscripted with a $B$ (for "Barendregt").

**Definition 2.1.1.1 (B-Typing)** The typing relation $\Gamma \vdash B : s$ in a PTS $\lambda(S,A,R)$ is the
smallest relation closed under the following type inference rules:

- **(axiom)**
  \[
  \begin{array}{c}
  \varepsilon \vdash_B s_1 : s_2 \\
  \hline
  \end{array}
  \quad (s_1 : s_2) \in A
  
- **(var)**
  \[
  \begin{array}{c}
  \Gamma \vdash A : s \\
  \hline
  \Gamma, x : A \vdash_B x : A
  \end{array}
  
- **(weaken)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B b : B \quad \Gamma \vdash_B A : s \\
  \hline
  \Gamma, x : A \vdash_B x : B
  \end{array}
  
- **(II-form)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B A : s_1 \quad \Gamma, x : A \vdash_B B : s_2 \\
  \hline
  \Gamma \vdash_B (\Pi x : A. B) : s_3
  \end{array}
  \quad (s_1, s_2, s_3) \in R
  
- **(II-intro)**
  \[
  \begin{array}{c}
  \Gamma, x : A \vdash_B b : B \quad \Gamma \vdash_B (\Pi x : A. B) : s \\
  \hline
  \Gamma \vdash_B (\lambda x : A. b) : (\Pi x : A. B)
  \end{array}
  
- **(II-elim)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B b : (\Pi x : A. B) \quad \Gamma \vdash_B a : A \\
  \hline
  \Gamma \vdash_B b[a/x] : B
  \end{array}
  
- **(conv)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B b : B \quad \Gamma \vdash_B B' : s \quad B =_B B' \\
  \hline
  \Gamma \vdash_B b : B'
  \end{array}
  
\[\square\]

Note that by the definition of the set of pseudocontexts $C$ the variable $x$ is $\Gamma$-fresh in the rules
(var), (weaken), (II-form) and (II-intro).

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

- **(→-form)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B A : s_1 \quad \Gamma \vdash_B B : s_2 \\
  \hline
  \Gamma \vdash_B A \rightarrow B : s_3
  \end{array}
  \quad (s_1, s_2, s_3) \in R
  
- **(→-intro)**
  \[
  \begin{array}{c}
  \Gamma, x : A \vdash_B b : B \quad \Gamma \vdash_B A \rightarrow B : s \\
  \hline
  \Gamma \vdash_B (\lambda x : A. b) : A \rightarrow B
  \end{array}
  
- **(→-elim)**
  \[
  \begin{array}{c}
  \Gamma \vdash_B b : A \rightarrow B \quad \Gamma \vdash_B a : A \\
  \hline
  \Gamma \vdash_B b[a/x] : B
  \end{array}
  
In a given PTS there may be rules $(s_1, s_2, s_3) \in R$ for which the rule (II-form) is more general
than needed, and just the rule (→-form) would suffice.

**Convention:** The $\rightarrow$ is right-associative, and binds weaker than application. For example,
$(A \rightarrow B \rightarrow C D)$ means $(A \rightarrow (B \rightarrow (C D)))$. 

2.1.2 Meta-theory for $\Gamma_B$

Below we list some properties of $PTS$s. We give sketches of the proofs, for easy future reference in Chapter 7. Full proofs of these properties can be found in [GN91] or [Bar92]. In this section we omit the $B$ in $\Gamma_B$.

The first lemma states that a variable may be replaced by a concrete expression of the same type.

Lemma 2.1.2.1 (Substitution)
If $\Gamma, x : A, \Gamma' \vdash b : B$ and $\Gamma \vdash a : A$ then $\Gamma, \Gamma'[x := a] \vdash b[x := a] : B[x := a]$

Proof sketch: By induction on the derivation of $\Gamma, x : A, \Gamma' \vdash b : B$. □

The next two lemmas state that the context behaves as expected. Here we use the condition $\Gamma \vdash b : B$ to express that pseudocontext $\Gamma$ is well-formed.

Lemma 2.1.2.2 (Start) If $\Gamma \vdash b : B$ then
- if $(s_1:s_2) \in A$ then $\Gamma \vdash s_1 : s_2$.
- if $x : A \in \Gamma$ then $\Gamma \vdash x : A$.

Proof sketch: Both by easy induction on the derivation of $\Gamma \vdash b : B$. □

Lemma 2.1.2.3 (Weakening) If $\Gamma \vdash a : A$ and $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash b : B$ then $\Gamma' \vdash a : A$

Proof sketch: By induction on the derivation of $\Gamma \vdash a : A$. □

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

Lemma 2.1.2.4 (Generation)
- if $\Gamma \vdash s : C$ then $C =^B s'$ for some $(s:s') \in A$
- if $\Gamma \vdash x : C$ then $C =^B A$ for some $x : A \in \Gamma$
- if $\Gamma \vdash \Pi x : A. B : C$ then $C =^B s_3$ and $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$ for some $(s_1,s_2,s_3) \in \mathcal{R}$
- if $\Gamma \vdash \lambda x : A. b : C$ then $C =^B \Pi x : A. B$ and $\Gamma, x : A \vdash b : B$ for some $B$.
- if $\Gamma \vdash b a : C$ then $C =^B B[x := a]$ and $\Gamma \vdash b : \Pi x : A. B$ and $\Gamma \vdash a : A$ for some $x, A, B$.

Proof sketch: Each case can be proved by induction on the derivation. The rule (conv) can be handled collectively. □

If a term is inhabited then it is a sort or it is typable with a sort:

Lemma 2.1.2.5 (Correctness of Types, CT for short)
If $\Gamma \vdash a : A$ then $A \equiv s$ or $\Gamma \vdash A : s$ for some $s$. 
Proof sketch: By induction on the derivation. The only interesting case is (II-elim), which uses Generation (Lemma 2.1.2.4) and Substitution (Lemma 2.1.2.1).

The following theorem states that reduction preserves types. This is an important soundness property for a PTS. If a PTS is interpreted as a programming language, it means that the execution of a well-typed program does not result in run-time type errors.

**Theorem 2.1.2.6 (Subject Reduction, SR for short)** Suppose \( a \rightarrow^* b \).

- If \( \Gamma \vdash a : A \) then \( \Gamma \vdash b : A \).

- If \( \Gamma, x : a, \Gamma' \vdash c : C \) then \( \Gamma, x : b, \Gamma' \vdash c : C \).

Proof sketch: By mutual induction on the derivation. The only interesting case is the rule (II-elims) for the first part. This case uses Generation (Lemma 2.1.2.4), Correctness of Types (Lemma 2.1.2.5) and Substitution (Lemma 2.1.2.1).

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

**Definition 2.1.2.7** A (specification of a) PTS is functional if

\[
(s : s') \in \mathcal{A} \text{ and } (s : s'') \in \mathcal{A} \implies s' \equiv s''
\]

\[
(s_1, s_2, s_3) \in \mathcal{R} \text{ and } (s_1, s_2, s_3') \in \mathcal{R} \implies s_2 \equiv s_3'
\]

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

**Theorem 2.1.2.8 (Uniqueness of Types (UT) for functional PTSs)**
If \( \Gamma \vdash a : A \) and \( \Gamma \vdash a : A' \) then \( A \rightarrow^* A' \).

Proof sketch: By induction on the structure of \( a \), using Generation (Lemma 2.1.2.4).

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.

**Definition 2.1.2.9** A PTS is strongly normalizing (SN) if all its terms are, i.e. if \( \Gamma \vdash a : A \) implies \( a \) and \( A \) are strongly normalizing, 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.
2.2. A DIFFERENT PRESENTATION OF THE TYPING RULES

2.1.3 Examples of PTSs

Definition 2.1.3.1

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

$$\mathcal{S} = \{*, \Box\}, \mathcal{A} = \{(*:\Box)\}, \mathcal{R} = \{(*, *)\}.$$  

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

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

$$\mathcal{S} = \{*, \Box\}, \mathcal{A} = \{(*:\Box)\}, \mathcal{R} = \{(*, *, (\Box, *))\}.$$  

$\lambda 2$ is essentially 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

$$\mathcal{S} = \{*, \Box\}, \mathcal{A} = \{(*:\Box)\}, \mathcal{R} = \{(*, *, (\Box, *), (\Box, *))\}.$$  

$\lambda \omega$ is essentially the higher-order typed lambda calculus, also known as system $F^\omega$, introduced in [Gir72].

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

$$\mathcal{S} = \{*, \Box\}, \mathcal{A} = \{(*:\Box)\}, \mathcal{R} = \{(*, *, (\Box, *), (\Box, *), (\Box, *))\}.$$  

$\lambda C$ is essentially 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 4 the system $\lambda \omega$ and a refinement of the system $\lambda C$ will be discussed in depth.

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

Theorem 2.1.3.2 $\lambda C$ is Strongly Normalizing.

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

2.2 A Different Presentation of the Typing Rules

We prefer a different presentation of the typing rules, denoted by $\vdash$, over $\vdash_P$. In this presentation there are two kinds of judgments, one to express that a context $\Gamma$ is well-formed, denoted by $\Gamma \vdash ok$, and one to express that in $\Gamma$ a term $a$ has type $A$. These judgments are mutually dependent. Although our presentation has two kinds of judgments instead of one, and 8 derivation rules instead of 7, we prefer this presentation. The reasons are:

1. This presentation is more flexible; it is easier to extend with new kinds of judgments, as will happen in Chapters 7 and 9.
2. It is closer to efficient type-checking algorithms, that first check the well-formedness of a context, and then calculate the type of a term in this context.

3. A separate $\Gamma \vdash ok$ judgment is more intuitive.

**Definition 2.2.1** (Typing) The typing relation $\vdash$ and the well-formedness relation $\vdash ok$ in the PTS $\lambda(S, A, R)$ are the smallest relations closed under the following inference rules:

- **(C-empty)** \[ \epsilon \vdash ok \]
- **(C-var)** \[ \Gamma \vdash A : s \quad \Gamma, x : A \vdash ok \]
- **(var)** \[ \Gamma \vdash ok \quad \Gamma \vdash x : A \quad x : A \in \Gamma \]
- **(axiom)** \[ \Gamma \vdash ok \quad \Gamma \vdash s_1 : s_2 \quad (s_1, s_2) \in A \]
- **(II-form)** \[ \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \Gamma \vdash (\Pi x : A. B) : s_3 \quad (s_1, s_2, s_3) \in R \]
- **(II-intro)** \[ \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \quad \Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B) \]
- **(II-elim)** \[ \Gamma \vdash b : (\Pi x : A. B) \quad \Gamma \vdash a : A \quad \Gamma \vdash b a : B[x := a] \]
- **(conv)** \[ \Gamma \vdash b : B \quad \Gamma \vdash b' : s \quad B =_{\beta} B' \quad \Gamma \vdash b : B' \]

\[ \square \]

### 2.2.1 Meta-theory for $\vdash$

In this section we will establish the relation between our formulation of PTSs and the standard formulation (see [Geu93]).

**Definition 2.2.1.1**

1. A pseudoterm $A$ is a *term* if $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ for some $\Gamma, B$.

2. A pseudocontext $\Gamma$ is a *context* if $\Gamma \vdash ok$.

3. A term $A$ is *inhabited* in context $\Gamma$ if $\Gamma \vdash a : A$ for some $a$.

4. A term $a$ is *typable* in context $\Gamma$ if $\Gamma \vdash a : A$ for some $A$.

5. A term $A$ is *well-typed* in context $\Gamma$ if $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ for some $B$. So a term is well-typed if it is inhabited or typable. \[ \square \]
2.2. A DIFFERENT PRESENTATION OF THE TYPING RULES

We first need three lemmas that state that the context behaves as expected.

**Lemma 2.2.1.2 (Start)** If $\Gamma \vdash a : A$ then $\Gamma \vdash ok$. Furthermore, the derivation for $\Gamma \vdash ok$ is shorter than the derivation for $\Gamma \vdash a : A$.

*Proof:* By straightforward induction on the structure of the derivation.

**Lemma 2.2.1.3 (Weakening)** If $\Gamma' \vdash ok$ and $\Gamma \subseteq \Gamma'$ and $\Gamma \vdash a : A$ then $\Gamma' \vdash a : A$.

*Proof:* By induction on the length of the derivation of $\Gamma \vdash a : A$. We do the case (II-intro) here. All other cases are similar or straightforward.

We have $\Gamma, x : A \vdash b : B$ and $\Gamma \vdash \Pi x : A. B : s$, and we have to show $\Gamma' \vdash \lambda x : A. b : \Pi x : A. B$. We may assume $x$ is fresh in $\Gamma'$. By Lemma 2.2.1.2 we have $\Gamma, x : A \vdash ok$, so $\Gamma \vdash A : s$, so by IH $\Gamma' \vdash A : s$ and using our assumption we have $\Gamma', x : A \vdash ok$. By using the IH on $\Gamma, x : A \vdash b : B$ we now have $\Gamma', x : A \vdash b : B$. Directly by IH, we have $\Gamma' \vdash \Pi x : A. B : s$. Using the (II-intro) rule we obtain $\Gamma' \vdash \lambda x : A. b : \Pi x : A. B$.

**Lemma 2.2.1.4** If $\Gamma, \Gamma' \vdash ok$ then $\Gamma \vdash ok$.

*Proof:* By straightforward induction on the length of $\Gamma'$.

The following two theorems express that derivable $\vdash_B$ judgments are derivable $\vdash$ judgments, and vice versa.

**Theorem 2.2.1.5** If $\Gamma \vdash_B a : A$ in $\lambda S$, then $\Gamma \vdash a : A$ in $\lambda S$.

*Proof:* By induction on the structure of the derivation. We treat the case (weaken) here, all other cases are straightforward.

We know $\Gamma \vdash_b b : B$ and $\Gamma \vdash_B A : s$ and $x$ fresh in $\Gamma$, and we have to show $\Gamma, x : A \vdash b : B$. Using the IH we have $\Gamma \vdash b : B$ and $\Gamma \vdash A : s$, from which follows $\Gamma, x : A \vdash ok$. Using Lemma 2.2.1.3 we derive $\Gamma, x : A \vdash b : B$.

**Theorem 2.2.1.6** If $\Gamma \vdash a : A$ in $\lambda S$ then $\Gamma \vdash_B a : A$ in $\lambda S$.

*Proof:* We strengthen the theorem with

- If $\Gamma \vdash ok$ and $(s_1 : s_2) \in \mathcal{A}$, then $\Gamma \vdash_B s_1 : s_2$,
- If $\Gamma \vdash ok$ and $x : A \in \Gamma$ then $\Gamma \vdash_B x : A$.

We proceed by mutual induction on the structure of the derivation. The cases fall into two classes.

- The ordinary typing rules. For (var) and (axiom) we need the IH for $\Gamma \vdash ok$, the rest is straightforward.
- The well-formedness rules for contexts. For case (C-var) we use lemma 2.1.2.2, the case (C-empty) is straightforward.

By Theorems 2.2.1.5 and 2.2.1.6 both presentations of the typing derivation are equivalent. So the theorems in Section 2.1.2, viz. subject reduction, uniqueness of typing for functional PTSs and strong normalization for $\lambda C$, carry over to our presentation.
2.2.2 Type-Checking Algorithm

The type checking problem \( \Gamma \vdash a : A \) is the problem of deciding whether \( \Gamma \vdash a : A \) is derivable for given \( \Gamma \), \( a \) and \( A \). The type inference problem \( \Gamma \vdash a : ? \) is the problem of deciding if a term \( a \) 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 normalizing 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 [BJMP94] and [Pol93]. The class of PTSs handled by the algorithm in [Pol93] are the so-called bijective PTSs.

Definition 2.2.2.1 A (specification of a) PTS is bijective if it is functional and

\[
(s_1, s_2, s_3) \in R \text{ and } (s_1, s_2', s_3) \in R \implies s_2 = s_2'
\]

All PTSs that we consider are bijective. More specifically, all systems in Barendregt’s cube are bijective.

An important property of the algorithm for bijective PTSs is that it includes sorts in judgments, i.e., judgments have the form \( \Gamma \vdash a : A : s \), which is short for \( \Gamma \vdash a : A \) and \( \Gamma \vdash A : s \). This algorithm is extended in Section 7.3 to handle subtyping as well.

2.3 PTSs with Definitions

This section gives a short summary of DPTSs, as introduced in [SP94]. DPTSs are PTSs extended with a definition mechanism. Such a mechanism is indispensable for practical use of PTSs, and Yarrow is based on these DPTSs. However, for understanding the rest of the thesis, it is not necessary to be familiar with DPTSs; an informal knowledge of definitions should suffice. We do not give the formal properties of DPTSs, which can be looked up in [SP93, SP94].

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 pseudoterms, a set of pseudcontexts, 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 \( \lambda S \).

Definition 2.3.1 (Pseudoterms) The set of pseudoterms \( T \) of a PTS \( \lambda S \) is extended by

\[
T ::= \ldots \mid \text{let } V := T : T. T
\]

Like \( \lambda \) and \( \Pi \), let binds a variable: in \( (\text{let } x := a : A. b) \) occurrences of \( x \) in \( b \) are bound. The notions of free variables, substitution, and \( \beta \)-reduction are extended to this larger collection of pseudoterms in the obvious way. For definitions the same conventions for omitting parentheses apply as for \( \lambda \) and \( \Pi \)-abstractions. So \( (\text{let } x := a : A. b c) \) is \( (\text{let } x := a : A. (b c)) \), and not \( ((\text{let } x := a : A. b) c) \).
Definition 2.3.2 (Pseudocontexts) The set of pseudocontexts \( \Gamma \) of a DPTS \( \lambda S \) is defined by

- \( \epsilon \in \mathcal{C} \)
- \( \Gamma, x : A \in \mathcal{C} \) if \( \Gamma \in \mathcal{C}, A \in T, x \in V \) and \( x \notin \text{FV}(\Gamma) \cup \text{FV}(A) \).
- \( \Gamma, x := a : A \in \mathcal{C} \) if \( \Gamma \in \mathcal{C}, a, A \in T, x \in V \) and \( x \notin \text{FV}(\Gamma) \cup \text{FV}(a) \cup \text{FV}(A) \).

Here \( \text{FV}(\epsilon) = \emptyset \), \( \text{FV}(\Gamma, x : A) = \text{FV}(\Gamma) \cup \{ x \} \cup \text{FV}(A) \), and \( \text{FV}(\Gamma, x := a : A) = \text{FV}(\Gamma) \cup \{ x \} \cup \text{FV}(a) \cup \text{FV}(A) \).

Definitions in pseudocontexts, e.g. \( \Gamma, x := a : A, \Gamma' \), are called global definitions. Definitions in pseudoterm, e.g. (let \( x := a : A. b \)), are called local definitions, sometimes also called let-expressions.

Definition 2.3.3 (\( \delta \)-reduction) The \( \delta \)-reduction relation \( \vdash \subseteq \Gamma \times T \times T \) is the smallest relation such that

- \( \Gamma_1, x := a : A, \Gamma_2 \vdash x \triangleright_{\delta} a \)
- \( \Gamma \vdash \text{let } x := a : A. b \triangleright_{\delta} b \text{ if } x \notin \text{FV}(b) \)

and that is closed under the following compatibility rules

- if \( \Gamma, x := a : A \vdash b \triangleright_{\delta} b' \text{ then } \Gamma \vdash (\text{let } x := a : A. b) \triangleright_{\delta} (\text{let } x := a : A. b') \)
- if \( \Gamma, x : A \vdash b \triangleright_{\delta} b' \text{ then } \Gamma \vdash (\Pi x : A. b) \triangleright_{\delta} (\Pi x : A. b') \)
  \hspace{1cm} \Gamma \vdash (\lambda x : A. b) \triangleright_{\delta} (\lambda x : A. b') \)
- if \( \Gamma \vdash a \triangleright_{\delta} a' \text{ then } \Gamma \vdash (\text{let } x := a : A. b) \triangleright_{\delta} (\text{let } x := a' : A. b) \)
  \hspace{1cm} \Gamma \vdash (\text{let } x := a : A. b) \triangleright_{\delta} (a' b) \)
  \hspace{1cm} \Gamma \vdash (b a) \triangleright_{\delta} (b a') \)
  \hspace{1cm} \Gamma \vdash (\Pi x : a. b) \triangleright_{\delta} (\Pi x : a'. b) \)
  \hspace{1cm} \Gamma \vdash (\lambda x : a. b) \triangleright_{\delta} (\lambda x : a'. b) \)

We write \( \Gamma \vdash a \triangleright_{\beta\delta} b \) if \( \Gamma \vdash a \triangleright_{\delta} b \) or \( a \triangleright_{\delta} b \). For \( \rho \in \{ \delta, \beta\delta \} \), \( \Gamma \vdash \triangleright_{\rho} \) is the reflexive and transitive closure of \( \Gamma \vdash \triangleright_{\rho} \text{, and } \Gamma \vdash \equiv_{\rho} \text{ is the reflexive, transitive and symmetric closure of } \Gamma \vdash \triangleright_{\rho} \text{.} \)

Theorem 2.3.4 (Church-Rosser) If \( \Gamma \vdash a \equiv_{\beta\delta} b \) then there is a \( c \) such that \( \Gamma \vdash a \triangleright_{\beta\delta} c \) and \( \Gamma \vdash b \triangleright_{\beta\delta} c \).  

Our definition of the typing rules for definitions is adapted from the rules in [SP94], so as to match with the other rules.

Definition 2.3.5 (Typing) The typing relation \( \vdash \) and the well-formedness relation \( \vdash ok \) in the DPTS \( \lambda S \) are the smallest relations closed under the type inference rules listed in Definition 2.2.1 plus the following rules:
The terms and contexts of λS₆ and the notions of being well-typed, typable and inhabited in a context in λS₆ are defined for the DPTS₆ as for PTS₆ (see Definition 2.2.1.1). Without further proof, we state that all the properties of DPTS₆ as defined in [SP94] also hold for our presentation of the typing rules.

Lemma 2.3.6 (Correctness of Types) If Γ ⊢ a : A then A ⊳ s or Γ ⊢ A : s for some s.

Theorem 2.3.7 (Subject Reduction) Suppose Γ ⊢ a ⊳ₜ b.
- If Γ ⊢ a : A then Γ ⊢ b : A.
- If Γ, x : a, Γ' ⊢ c : C then Γ, x : b, Γ' ⊢ c : C.

Theorem 2.3.8 (Uniqueness of Types for functional DPTS₆) If Γ ⊢ a : A and Γ ⊢ a : A' then Γ ⊢ A ⊳ₜ A'.

Theorem 2.3.9 The DPTS λC is strongly βδ-normalizing.

Unfortunately, it is not known whether adding definitions preserves strong normalization for each PTS₆. Theorem 2.3.9 is proved by using strong normalization of a bigger PTS.

The δ-normal form of a term A with respect to context Γ is denoted by δnfΓ(A). This notion is extended straightforwardly to contexts. Using δnf(·), we can express that every judgment in a DPTS can be translated to a judgment of δ-normal forms in the corresponding PTS.

Theorem 2.3.10 Γ ⊢ a : A in λS₆ ⊳ δnf(Γ) ⊢ δnfΓ(a) : δnfΓ(A) in λS.

An immediate consequence of this theorem is that λS₆ is a conservative extension over λS in the logical sense, because every λS context and λS term are a λS₆ context and λS₆ term in δ normal form.

Theorem 2.3.11 (Conservativity of λS₆ over λS) Let S be a functional specification, and let A be well-typed in context Γ in λS. Then

A is inhabited in Γ in λS₆ ⇔ A is inhabited in Γ in λS.
A useful extension of definitions are parametric definitions. In this extension, definitions can have parameters. A parametric definition over a variable \( x : A \) is mainly useful when we cannot form abstractions over \( x : A \). DPTSs with parameters have been studied by Laan, Severi and Zwanenburg (see [Laa97]).

### 2.4 Interpretation of a PTS as a Logic

Many PTSs can be interpreted as a logic, by the Curry-Howard-de Bruijn isomorphism (e.g. see [Gen93]). We give here a short overview how this interpretation works in general; in Chapter 4 we discuss a particular PTS and its use as a logic in more detail. The Curry-Howard-de Bruijn isomorphism interprets types as propositions and terms as proofs; a term is considered a proof of a type if and only if it is an inhabitant of that type.

A type of the form \( P \rightarrow Q \) (i.e. \( \Pi x : P.Q \) with \( x \notin \text{FV}(Q) \)) is interpreted as the implication \( P \rightarrow Q \). This interpretation is sensible, since the introduction and elimination rules for \( \rightarrow \) correspond with the introduction and elimination rules for \( \rightarrow \). Take the elimination rule:

\[
\frac{\Gamma \vdash q : P \rightarrow Q \quad \Gamma \vdash p : P}{\Gamma \vdash q \, p : Q}
\]

By erasing the terms left of the colon we obtain the \( \rightarrow \) elimination rule (if \( \Gamma \vdash P \rightarrow Q \) and \( \Gamma \vdash P \) then \( \Gamma \vdash Q \)). The proof \( q \, p \) of \( Q \) is the application of the proof \( q \) of \( P \rightarrow Q \) to the proof \( p \) of \( P \). Take the introduction rule:

\[
\frac{\Gamma, x : P \vdash q : Q \quad \Gamma \vdash P \rightarrow Q : s}{\Gamma \vdash (\lambda x : P. q) : P \rightarrow Q}
\]

Ignoring the second premise and the terms left of the colon, we obtain the \( \rightarrow \) introduction rule (if \( \Gamma, P \vdash Q \) then \( \Gamma \vdash P \rightarrow Q \)). The proof \( (\lambda x : P. q) \) of \( P \rightarrow Q \) is built-up with the proof \( q \) of \( Q \) in which \( x \) (standing for an arbitrary proof of \( P \)) occurs.

A type of the form \( \Pi x : A. Q \) is interpreted as the quantification \( \forall x : A. Q \). Again, this interpretation is sensible, because of the introduction and elimination rules. Rule II-elim is:

\[
\frac{\Gamma \vdash p : (\Pi x : A. Q) \quad \Gamma \vdash a : A}{\Gamma \vdash p \, a : Q[x := a]}
\]

By erasing the terms \( p \) and \( p \, a \), we obtain the \( \forall \) elimination rule (if \( \Gamma \vdash \forall x : A. Q \) and \( \Gamma \vdash a : A \) then \( \Gamma \vdash Q[x := a] \)). The proof \( p \, a \) of \( Q[x := a] \) is the application of the proof \( p \) to term \( a \), which does not stand for a proof, but either for a mathematical object such as a number, or for a proposition (in higher-order logics). Note the application can stand for both \( \rightarrow \) and \( \forall \) elimination. The introduction rule for \( \Pi \) is:

\[
\frac{\Gamma, x : A \vdash q : Q \quad \Gamma \vdash (\Pi x : A. Q) : s}{\Gamma \vdash (\lambda x : A. q) : (\Pi x : A. Q)}
\]

Ignoring the second premise and erasing the terms \( q \) and \( (\lambda x : A. q) \) we obtain the \( \forall \) introduction rule: if \( \Gamma, x : A \vdash Q \) then \( \Gamma \vdash \forall x : A. Q \) (this rule is usually formulated slightly differently). The proof \( \lambda x : A. q \) of \( x : A. Q \) is the abstraction of \( q \) over \( x : A \). So an abstraction can stand for \( \rightarrow \) or \( \forall \) introduction.

For example, consider the PTS \( \lambda \rightarrow \) and the context \( \Gamma \equiv P : *, Q : *, R : *, a : P \rightarrow Q, b : Q \rightarrow R \). We interpret \( P, Q \) and \( R \) as propositions, so \( P \rightarrow Q \) may be read as the implication \( P \rightarrow Q \). Now the term \( \lambda x : P. b \, (a \, x) \) has type \( P \rightarrow R \), and is interpreted as a proof of \( P \rightarrow Q \).
Chapter 3

Yarrow

As shown in Section 2.4, \( \lambda \)-terms can represent proofs, and a term is a proof of a given type (interpreted as proposition) if and only if the term is an inhabitant of that type. So proof-checking amounts to type-checking. This is an important reason for using PTSs: a type-checker, which is just a small program, can check proofs in powerful logics. However, any non-trivial proof will be a large and unreadable \( \lambda \)-term (e.g. see Figure 3.11 on page 68), since the term constructs (application and abstraction) represent tiny steps in a proof. It is not reasonable to demand from a human to give proofs as \( \lambda \)-terms; we need machine support to aid the user in constructing proofs as \( \lambda \)-terms. Computer programs that give this support are called proof assistants. During a proof, the proof assistant always shows the propositions yet to be proved, and the user indicates which logical step to do next. By this interaction, the user can focus on the logical aspects of the proof, while the machine keeps track of the proof term under construction. Some well-known proof assistants of this kind are Coq [Coq97], LEGO [LP92], and Alfa [Hal97]. But we implemented our own proof assistant, called Yarrow. We did this because of the following reasons.

- None of the existing programs has the PTSs as underlying theory. By making a proof assistant based on PTSs, we offer users the opportunity to experiment with calculi of different logical strengths. Furthermore we can use Yarrow for the particular PTS we are interested in, \( \lambda \omega_L \), as defined in the next chapter. We point out that \( \lambda \omega_L \) is a subsystem of the calculi offered by Coq and LEGO, so we could develop the proofs in the next chapter in those systems. But we would not be sure whether we stayed within the borders of \( \lambda \omega_L \).

- It is hard to extend Coq or LEGO with the features we need, such as subtyping.

- It is fun to make such a program, and we were dissatisfied with some deficiencies of Coq and LEGO. That is, we were interested how a proof assistant could be made, and why certain features were not present in existing assistants. One such a feature, called forward reasoning, is described in Section 3.1.2.

The three sections of this chapter describe different aspects of Yarrow. Section 3.1 gives the user's perspective on Yarrow by way of some concrete examples of its use. In Section 3.2 we highlight the type-theoretical aspects of our proof assistant, including the theoretical basis of interactive proof construction, some mechanisms improving readability of terms, and the presentation of proofs in the flag-style format. Also a technical comparison with other proof assistants is given here.
Yarrow offers both a graphical and a textual interface, and has been coded entirely in Haskell [Tho06, Pet97], making use of the Fudgets library [HC95] for the graphical interface. In Section 3.3 we concentrate on the software architecture of Yarrow, in particular the use of monads, the coupling of user interface and proof engine, polymorphic output routines, and flexible representations of lambda terms. This section is an adaptation of a technical report [Zwa98].

3.1 A User’s Perspective

Just as any proof assistant, Yarrow maintains a global context, which is the context used for typing and reduction. In Yarrow’s main mode, the user can select a PTS, add declarations and definitions to the global context, load a piece of context stored in a file, ask for the type and normal form of a term with respect to the global context, and so on. The most essential commands available in the main mode are described in Section 3.1.1.

Furthermore, the user can start proof tasks from the main mode. Such a task consists of the interactive construction of an inhabitant of a given type, which is usually interpreted as lemma or theorem. During a task, Yarrow is in proof mode, which is described in Section 3.1.2. If the task is finished, the constructed term is stored in a definition in the global context, so that future proofs can use this theorem, and Yarrow returns to main mode. The proof mode is mainly used when the selected PTS is interpreted as a logic. This stands in contrast to the main mode, which is used regardless of the interpretation of the selected PTS.

3.1.1 Main Mode

One important aspect of Yarrow is that the user can choose the PTS he wants to work in, viz. with the command System. For example,

> System (*,#), (*:#), (((*,#),(*,#),(#,*)#,(*,#))

selects the PTS \( \lambda C \) ("\( > \)" is the prompt in main mode). All bijective PTSs with a finite specification can be selected; in fact Yarrow uses DPTSs, i.e. PTSs extended with definitions (see Section 2.3). The bijective PTSs are defined in [Pol93], and include all the systems of the \( \lambda \)-cube (see Definition 2.1.3.1), and the system \( \lambda \omega_I \), which will form the basis of the rest of this thesis. The reason for permitting the bijective systems is that Yarrow uses the type-checking algorithm of [Pol93], which is only proved sound and complete for bijective PTSs.

Initially, the global context is empty. Declarations are added with the Var command, e.g.

> Var N : *
N : *
> Var O : N
O : N
> Var S : N -> N
S : N->N
>

declares the variable \( N \) (which we interpret as the set of natural numbers), \( O \) (zero) and \( S \) (successor). The main differences between Yarrow’s ASCII and conventional mathematical
3.1. A USER'S PERSPECTIVE

<table>
<thead>
<tr>
<th>ASCII</th>
<th>conventional notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>\lambda</td>
</tr>
<tr>
<td>0</td>
<td>\forall</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;</td>
</tr>
<tr>
<td>#</td>
<td>□</td>
</tr>
</tbody>
</table>

Figure 3.1: Syntax for terms in Yarrow

notation are given in Figure 3.1. Each declaration and definition is type-checked before it is added to the global context, so that the global context remains well-formed. Definitions are added with the `Def` command, e.g.

> `Def one := S 0 : N`
> `one := ... : N`
> `Def id := \n:N. n`
> `id := ... : N->N`
> `Def wrong := id id`
> `11`  
> ```
> `expects a N but is applied to ` with type N->N (type error)`
>
In Yarrow's response to a well-typed definition, the definients is abbreviated to two dots, only the type is printed in full.

The command `bdRed` print the $\beta\delta$-normal form of a term in the global context, after it has been successfully type-checked, e.g.

> `bdRed id one`
> `S 0`
> `3 reductions.``

Finally, the command `Prove` brings Yarrow into proof mode, which we discuss next.

3.1.2 Proof Mode

In proof mode, a proof assistant should assist the user in finding an inhabitant (proof term) of a given type (the goal). Yarrow does this by maintaining a partial proof term. A partial proof term is a pseudoterm with holes of the form `?;` in it. Initially, the partial proof term is just a hole.

The user indicates with a certain set of commands, called tactics, how this proof term should be built up; each application of a tactic replaces a hole in the partial proof by some pseudoterm, which may have again holes in it. Yarrow keeps track of the partial proof, and the type of each hole, i.e. the type that a term to be substituted for the hole should have.

This type is called the subgoal, and since there may be several holes, there may be several subgoals. Associated with each hole and subgoal is a local context. The local context is the list of all declarations and definitions occurring in the partial proof term on the path to the corresponding hole. So for making a proof of a subgoal, we can not only use the global context, but also the corresponding local context. The combination of local context, hole and subgoal is called a task. So the proof term is constructed top-down, each tactic refines a subgoal to zero or more subgoals, which should be easier to prove. This means that we
work in a goal-directed or backward fashion, i.e. we work backward from the (sub)goals to the assumptions.

If at some moment the proof term is complete, i.e. it does not contain any holes, the process of proving is finished. The proof term is stored in a definition that is added to the global context, after we have checked that the proof term is an inhabitant of the goal. If the implemented tactics are free from errors, then this final type checking of the proof term is redundant. But we do not want to rely on this. Type checking the final proof term guarantees that correctness of the theorem prover only depends on correctness of the type checker, and not on correctness of the tactics.

An example proof in Yarrow

Let us illustrate this with an example; in Section 3.2.1 we give a more formal account. Suppose we work in the PTS AC and the goal is $\forall P,Q,R:* (P \rightarrow Q \rightarrow R) \rightarrow (Q \rightarrow P \rightarrow R)$. In this example, we give the input to and output from Yarrow (with the textual interface) verbatim.

In Yarrow we use the command Prove $x:A$ to indicate we want to find an inhabitant of type $A$. The variable $x$ should be fresh, and will be used for storing the inhabitant in the global context when we are finished.

> Prove thm3 : @P,Q,R:* (P->Q->R)->(Q->P->R)

Yarrow responds with:

Proofterm = ?1

---------------------------------------------------------------------

?1 : @P,Q,R:* (P->Q->R)->Q->P->R

$ $

So the partial proof term is just a hole $?_1$. The output $?1 : @P,Q,R:* \ldots$ indicates that the term we will substitute for this hole should be an inhabitant of our goal. The dollar sign $\$$ is the prompt in proof mode.

We indicate with the tactic Intro $x$ that our proof term should be an abstraction over $x$. This is sensible, because our goal is a $\Pi$-type, and from the form of the $\Pi$-type Yarrow deduces what the type of $x$ and the type of the body should be.

$ \$ Intro P
?1 := \l P:* . ?3

Proofterm = \l P:* . ?3

P : *

---------------------------------------------------------------------

?3 : @Q,R:* (P->Q->R)->Q->P->R

The first line of the output indicates what pseudoterm is substituted for hole $?_1$, and the second line gives the resulting partial proof term. Under the dashed line Yarrow states that the hole $?_3$ should have type $@Q,R:* \ldots$, this type is the subgoal. The declaration above the line gives the local context of the subgoal, i.e. the declarations of the variables we may use in a proof for the subgoal. This combination of a local context, a hole and a subgoal is the task.
To solve this task, we again use the Intro tactic, now without a parameter; Yarrow chooses the name of the variable to abstract over, which will typically be the variable used in the II-type in the subgoal.

$ Intro
?3 := \Q : *. ?4
Proofterm = \P, \Q : *. ?4

P : *
Q : *

?4 : \R : *. (P → Q → R) → Q → P → R

Recalling that an implication → is a form of II-type, we could use the tactic Intro four times, to obtain subgoal R. For convenience, the tactic Intros repeats Intro as often as possible (i.e. as long as the subgoal is a II-type), giving the same result.

$ Intros
Proofterm = \P, \Q, \R : *. \H : P → Q → R, \H1 : Q, \H2 : P. ?9

P : *
Q : *
R : *
H : P → Q → R
H1 : Q
H2 : P

?9 : R

Now we need to make an inhabitant of R. Most mathematicians and logicians would reason as follows: "From H and H2 it follows that Q → R (the proof is H H2). From this we know by H1 that R holds (the proof is H H2 H1). Q.e.d." This is an example of forward reasoning, i.e. working from the assumptions (or previously proved lemmas) towards the conclusion. In Yarrow, this kind of reasoning is strongly supported, but we first consider the backward reasoning that is commonly used in proof assistants, so we work from the subgoal to the assumptions.

Our subgoal is R, and we can prove this with H, since the conclusion of its type is R; we will consider later how to prove the premises of its type. More precisely, the application of H to two terms, the first of type P and the second of type Q, is of type R. Such an application is generated by the Apply tactic, which has as parameter a term, H in this case.

$ Apply H
Proofterm = \P, \Q, \R : *. \H : P → Q → R, \H1 : Q, \H2 : P. H ?13 ?11

2 tasks

P : *
Q : *
R : *
H : P→Q→R
H1 : Q
H2 : P

?13 : P

2) ?11 : Q

The two arguments of H are unknown, so two holes are used as arguments. So we have two
tasks, one to find a term of type P to substitute for ?13, and one to find a term of type Q
to substitute for ?11. Each task has its own local context (which may differ in general), but
only the local context for the first task is printed.

All tactics always work on the first task, but there is a command to change the order of
the tasks, so no order of proving is forced upon the user. We are satisfied with the current
order, so we have to find an inhabitant of P. Such an inhabitant is available in the local
context, viz. the term H2. The Exact tactic has a term as parameter, and uses this as proof
term for the current subgoal (provided the term has the correct type).

$ Exact H2
?13 := H2
Proofterm = \P, Q, R:* > H:F->Q>R.\H1:Q.\H2:P. H H2 ?11

P : *
Q : *
R : *
H : P→Q→R
H1 : Q
H2 : P

?11 : Q

We now have one task, which is again solved with Exact.

$ Exact H1
?11 := H1
Proofterm = \P, Q, R:* > H:F->Q>R.\H1:Q.\H2:P. H H2 H1

Goal proved!

The proof term is now complete, since it contains no holes. We exit from proof mode and
store the found proof term in the global context by the Exit command.

$ Exit
Prove thm3 : @P, Q, R:* (P→Q→R)→Q→P→R
Intro P
Intro
Intros
Yarrow gives a recapitulation of the commands (called the proof script) used to state and prove this simple theorem. The variable given as parameter to the Prove command, viz. \texttt{thm3}, is defined as the proof term (here abbreviated to \ldots), with as type the goal we started with. By storing the definition in the global context, we can from now on use \texttt{thm3} in other proofs.

Let us come back to the subject of forward and backward reasoning. As we have stated before, existing type-theoretic proof assistants give only strong support to backward reasoning. We feel that a proof assistant should also give good support to forward reasoning, for the following reasons. Written mathematical proofs are usually presented in a forward manner ("from \(A\) and \(B\) we know \(C\), and by lemma \(D\) we obtain \(E\), which by \(F\) implies the required property"). So when using a proof assistant for the formalization of a given informal proof, the user can follow this proof in the given order only if it supports forward reasoning; if only backward reasoning is available, the user has to read the informal proof backwards. If the user wants to give a formal proof without an informal proof at hand, the proof assistant should support the order of reasoning the user does in his mind. Our own experience is that this is a mixture of both forward and backward reasoning ("by \(A\) and \(B\) we know \(C\). By lemma \(D\) used with \(C\) our goal reduces to \(E\)" and so on). Therefore good support for forward reasoning is important. Existing proof assistants based on type theory such as Coq [Coq97] and LEGO [LP92] have a set of tactics that mainly support backward reasoning, and forward reasoning is awkward. So the user is more or less forced to use the backward approach. In Yarrow we have implemented a tactic, \texttt{Forward}, that supports forward reasoning. This tactic is defined — in a formal form — on page 47.
3.2 A Type-Theoretical Perspective

In this section we discuss several type-theoretical aspects of Yarrow. The most essential part of a proof assistant is the type-checking algorithm. In Yarrow, we used the algorithm for bijective PTSs, described in [Pol93] (and later, its extension with subtyping, given in Section 7.3 of this thesis).

The most interesting part is the type-theoretical foundation of the proof mode, which we describe in Section 3.2.1. In particular, we formally describe the state of Yarrow in proof mode, show how tactics change the state, and give a formal description of the most important tactics.

In Yarrow there is an important restriction on the form that subgoals can have. This restriction, and our reasons for introducing it, are explained in Section 3.2.2.

Yarrow offers two mechanisms that improve readability of λ-terms, which are both described in Section 3.2.3. The syntactic sugaring provided by these mechanisms will be used throughout this thesis.

The user may wish to inspect previous proofs. Since the proof as λ-term is unreadable in all but the most trivial cases, Yarrow has the feature to present constructed proofs in a natural deduction format, namely in flag style, which are much better readable, though still formal and very precise. This feature is used in several places in this thesis. Proofs in flag style are discussed in Section 3.2.4.

We compare Yarrow with other proof assistants based on type theory in Section 3.2.5.

3.2.1 Formal Treatment of the Proof Mode

Now let us consider the proof mode more formally. First, we have to introduce the holes as pseudoterm, and define the notions of the holes occurring in a pseudoterm, and the substitution of a pseudoterm for a hole.

**Definition 3.2.1.1** The definition of pseudoterms is extended with holes of the form $?_i$ where $i \in \mathbb{N}$.

$$T ::= ... \mid ?_i \in \mathbb{N}$$

The notion $HL(a)$, the holes of pseudoterm $a$, is simply defined as the set of all holes occurring in $a$. Bound holes do not exist, so $HL(\lambda x : A. b) = HL(A) \cup HL(b)$. The substitution of pseudoterm $b$ for hole $?_i$ in pseudoterm $a$, notated with $a(?_i := b)$ simply replaces all occurrences of $?_i$ by $b$ in $a$. This substitution does not avoid capture of bound variables, e.g. $(\lambda x : A. ?_3)(?_3 := x) \equiv (\lambda x : A. x)$, whereas $(\lambda x : A. y)(y := z) \equiv (\lambda x' : A. x)$. We abbreviate $a(?_{h_1} := b_1) \ldots (?_{h_n} := b_n)$ to $a(?_{h_1} := b_1, \ldots, ?_{h_n} := b_n)$.

It is essential that a substitution for a hole does not avoid capture of bound variables, since if it did, we could not replace a hole by a variable from the local context. There are no typing rules for holes, so any typable term does not contain any holes.

Next, we define the state in the proof mode. It consists of the global context, the partial proof term, the goal and the set of tasks. Here, each task consists of a hole, a subgoal and the total context for this hole; the total context is the concatenation of the global context with the local context. Before we only had a local context in a task, but in the formal treatment it is convenient to combine them.
Definition 3.2.1.2 A state is a quadruple of the form

\[(\Gamma, a, A, \{(\Gamma_1, ?_{h_1}, A_1), \ldots , (\Gamma_n, ?_{h_n}, A_n)\})\]

where \(\Gamma\) is the global context, \(a\) is the partial proof term, \(A\) is the goal, each triple \((\Gamma_i, ?_{h_i}, A_i)\) is a task, where \(\Gamma_i\) is the total context of the task, \(?_{h_i}\) is the hole of the task, and \(A_i\) is the subgoal of the task. We admit only states with

1. \(\Gamma \vdash A : s\) for some \(s\).
   The goal must be a type.
2. \(\forall i. \exists s. \Gamma_i \vdash A_i : s\).
   So every subgoal must be a type in the corresponding context. In particular, subgoals may not contain holes. The reasons for this important restriction are given in Section 3.2.2.
3. \(\forall i, j. h_i = h_j \implies i = j\).
   Every task must correspond with a different hole. Hence the set of tasks can also be considered as a mapping of holes to total contexts and subgoals.
4. \(\text{HL}(a) = \{ ?_{h_1}, \ldots , ?_{h_n} \}\).
   Every hole in the proof term corresponds to a task and vice versa.
5. \(\forall a_1 \ldots a_n\) with \((\forall i. \Gamma_i \vdash a_i : A_i)\) we have \(\Gamma \vdash a(?_{h_1} := a_1, \ldots , ?_{h_n} := a_n) : A\).
   So, given proofs \(a_1 \ldots a_n\) for all the tasks, substituting these proofs in the partial proof term yields a proof of the goal. \(\square\)

Given a context \(\Gamma\) and a goal \(A\) with \(\Gamma \vdash A : s\), the initial state is

\[(\Gamma, ?_1, A, \{(\Gamma, ?_1, A)\})\].

It is easy to verify that this initial state is admitted. We call a state final when \(n = 0\). By conditions 4 and 5, this implies that the proof term is complete (i.e. it does not contain any holes), and that it is an inhabitant of the goal.

Now we specify the effect of a tactic applied to task \(i\) \((\Gamma_i, h_i, A_i)\). If a tactic is applied to this task, it gets as parameters \(\Gamma_i\), \(A_i\) and the set \(\{h_1, \ldots , h_n\}\) of used holes, and delivers a pair

\[(b, \{(\Gamma'_1, ?'_{h'_1}, A'_1), \ldots , (\Gamma'_m, ?'_{h'_m}, A'_m)\})]\n
where \(b\) is the partial proof term for the task, and each \((\Gamma'_j, h'_j, A'_j)\) is a new task. The new state is

\[(\Gamma, a(?_{h_i} := b), A, \{(\Gamma_1, ?_{h_1}, A_1), \ldots , (\Gamma_n, ?_{h_n}, A_n)\} \setminus \{(\Gamma_i, h_i, A_i)\} \cup \{(\Gamma'_1, ?'_{h'_1}, A'_1), \ldots , (\Gamma'_m, ?'_{h'_m}, A'_m)\})],\]

so the partial proof term for the task is substituted for the corresponding hole, the old task is removed, and the new tasks are added.

In order for the new state to be admitted, the result of the tactic should satisfy

1. \(\forall i. \exists s. \Gamma_i \vdash A'_i : s\).
2. \(\forall i, j. h'_i = h'_j \implies i = j\).
3. \( \{h'_1, \ldots, h'_m\} \cap \{h_1, \ldots, h_n\} = \emptyset. \)

The new holes must be different from the old holes.

4. \( \text{HL}(b) = \{?_{h'_1}, \ldots, ?_{h'_m}\}. \)

5. \( \forall a'_1 \ldots a'_m \text{ with } (\forall j. \Gamma'_j \vdash a'_j : A'_j) \) we have \( \Gamma \vdash b(?_{h'_1} := a'_1, \ldots, ?_{h'_m} := a'_m) : A_i. \)

So \( b \) should be a partial proof term for the subgoal.

Next, we will discuss the implementation of the most essential tactics. In the following, we assume that some task has been selected, and that \( \Gamma \) stands for the total context of this task (above: \( \Gamma_i \)), \( A \) is the subgoal of this task (above: \( A_i \)), and \( U \) is the set of used holes (above: \( \{h_1, \ldots, h_n\} \)).

In examples we use the \( \text{PTS} \lambda C \) and the context

\[
\]

where we interpret \( N \) as the set of natural numbers, \( O \) as zero, \( S \) as successor, \( P \) and \( Q \) as predicates on numbers, and \( R \) as a proposition.

**The Exact tactic**

The exact tactic is the simplest tactic, since it simply delivers a given term as proof term with no new tasks. So the tactic \texttt{Exact} \( a \) is implemented as follows.

- Check \( \Gamma \vdash a : A \),
  
  So \( a \) must be a proof of the subgoal. In particular, \( a \) may not contain any holes.

- Deliver \((a, \emptyset)\).

A variant of \texttt{Exact} is the \texttt{Assumption} tactic, which has no arguments. This tactic looks for a variable \( x \) declared in the local context of which the type is convertible to the subgoal, and uses this variable as proof term.

**The Introduction tactic**

The introduction tactic results in an abstraction as partial proof for the subgoal. The tactic \texttt{Intro} \( x \) is implemented as:

- Check \( x \notin \text{FV}(\Gamma) \).

- Check \( A \equiv \Pi x : B. C \) for some \( B \) and \( C \) (performing \( \alpha \)-conversion if necessary).

- Let \( h \notin U \).
  
  Since the \texttt{Intro} tactic delivers a new goal, we have to find a free hole.

- Deliver \( (\lambda x : B. ?_h, \{((\Gamma, x : B), ?_h, C)\}) \).
3.2. A TYPE-THEORETICAL PERSPECTIVE

The Apply tactic (simple version)

We explain the Apply tactic in two steps. Here we give a simple version, which is a special case of the general implementation of Apply given below. Both versions give as partial proof an application of a given term \( b \) to a number of arguments.

The simple version of the tactic Apply \( b \) is implemented as:

- Check \( \Gamma \vdash b : B_1 \to \ldots B_n \to A \) for some \( B_1, \ldots, B_n \).
  
So the conclusion of the type of \( b \) should be the selected subgoal.

- Choose different \( h_1, \ldots, h_n \) such that \( \{h_1, \ldots, h_n\} \cap U = \emptyset \).

  Since Apply delivers \( n \) new goals, we have to find \( n \) free holes.

- Deliver \((b \ ?_1 \ldots \ ?_n, \{\Gamma, \ ?_{h_1}, B_1\}, \ldots, \{\Gamma, \ ?_{h_n}, B_n\})\).

For an example we refer to the use of Apply in the example proof on page 39.

This simple version of Apply is quite restricted in its use, since it can handle only those cases where the type of its parameter has the form \( B_1 \to \ldots B_n \to A \) and \( A \) is the current subgoal. Often, the parameter \( b \) has the more general type \( \Pi x_1 : B_1 \ldots \Pi x_n : B_n \cdot C \) where a number of the variables \( x_1 \ldots x_n \) occur in \( C \). In this case, the application \( b \ b_1 \ldots b_n \) has type \( C[x_1 := b_1, \ldots, x_n := b_n] \) (provided that \( b_i \)'s have the right types). Since we want to make an inhabitant of \( A \), this type \( C[x_1 := b_1, \ldots, x_n := b_n] \) should be convertible to \( A \). This demand usually determines the values of the \( x_i \) occurring in \( C \), so a number of arguments \( b_i \) are determined. Other variables \( x_i \) may not occur in \( C \), and hence the values for these \( b_i \)'s are not determined; we will use a hole for each of these \( b_i \).

For example, suppose that \( b : \Pi x_1 : N. \Pi x_2 : P \ x_1. \ Q \ x_1 \) (i.e. for all naturals \( x_1 \) we have that \( P \ x_1 \) implies \( Q \ x_1 \)), and that we want to prove the subgoal \( Q \ b \) by applying \( b \). The application \( b \ b_1 b_2 \) has type \( Q \ b_1 \) provided that \( b_1 : N \) and \( b_2 : P \ b_1 \). The type \( Q \ b_1 \) is only equal to our subgoal if \( b_1 \) is \( O \). So, the constraint that the type of the application should be equal to the subgoal determines \( b_1 \), but does not determine \( b_2 \). The partial proof term delivered by the Apply tactic should be \( b \ O ?_2 \), with as new goal \( ?_2 : P \ O \).

The problem of finding values \( b_1 \ldots b_n \) such that \( C[x_1 := b_1, \ldots, x_n := b_n] \) is convertible to \( A \) is a matching problem, which we will denote by \( \Gamma; x_1 : B_1 \ldots x_n : B_n \vdash C \equiv_m A \). We will first formalize this notion before defining the implementation of the general Apply tactic as indicated above.

**Intermezzo: Matching**

**Definition 3.2.1.3** A matching problem is a quadruple \((\Gamma; \Gamma'; p; \ b)\) with \( \Gamma, \Gamma' \vdash p : P \) and \( \Gamma \vdash b : B \) for some \( P \) and \( B \), and where \( \Gamma' \) may not contain definitions.

Usually, we write such a problem in the form \( \Gamma; \Gamma' \vdash p \equiv_m b \). Here \( p \) is the pattern that should match with \( b \), and \( \Gamma' \) declares the variables to be substituted for. Since \( \Gamma \vdash b : B \), \( b \) cannot contain variables from \( \Gamma' \). This distinguishes matching problems from unification problems (e.g. see [Rob65]), where \( b \) may also contain variables from \( \Gamma' \).

A solution to a matching problem is a substitution with certain properties.

**Definition 3.2.1.4** A substitution \( \sigma \) is a mapping of some set of variables, called the domain \( \text{dom}(\sigma) \), to terms. The application \( \sigma(a) \) of the substitution \( \sigma \equiv [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] \) to term \( a \) is defined as \( a[x_1 := t_1, \ldots, x_n := t_n] \).
Definition 3.2.1.5 A solution to a matching problem $\Gamma; x_1 : B_1, \ldots, x_n : B_n \vdash p \overset{2}{=} b$ is a substitution $\sigma$ with

1. $\Gamma \vdash \sigma(p) =_{\beta^s} b$.
   So the application of the substitution to the pattern should give a term convertible to $b$.

2. $\text{dom}(\sigma) \subseteq \{x_1, \ldots, x_n\}$.
   Substitute only for variables that may be substituted for.

3. $\forall x_i \in \text{dom}(\sigma). \Gamma \vdash \sigma(x_i) : \sigma(B_i)$.
   The type of each variable $x_i$ must match with the type of the expression we substitute.
   See Example 2 below. \qed

We call a solution $\sigma$ with $\sigma(p) \equiv b$ a first-order solution, and other solutions higher-order.

Let us give three examples.

1. The matching problem $\Gamma; x_1 : N, x_2 : P \vdash x_1 \overset{2}{=} Q \overset{2}{=} O$ has as (first-order) solution $[x_1 \mapsto O]$.

2. Consider the matching problem $\Gamma; X : *, x : X \vdash x \overset{2}{=} O$. The substitution $[x \mapsto O]$ is not a solution, since by demand 3 we then should have $\Gamma \vdash O : X$. The substitution $[X \mapsto N, x \mapsto O]$ is a (first-order) solution.

3. Consider the matching problem $\Gamma; f : N \rightarrow N \vdash f \overset{2}{=} O \overset{2}{=} S \overset{2}{=} O$. Some solutions are $[f \mapsto S]$, $[f \mapsto \lambda x : N. S \mapsto x]$ and $[f \mapsto \lambda x : N. S \mapsto O]$. Only the first one is a first-order solution.

A solution $\sigma$ is more general than solution $\tau$ if $\text{dom}(\sigma) \subseteq \text{dom}(\tau)$ and $\Gamma \vdash \sigma(x) =_{\beta^s} \tau(x)$ for all $x \in \text{dom}(\sigma)$. As the last example shows, there is not such a thing as a most general solution for a matching problem, although we have most general first-order solutions. In most PTSs, including $\lambda 2$ and bigger systems, the question whether a matching problem has a solution or not is undecidable [Dow93].

In Yarrow, we implemented a matching routine that given a matching problem either delivers one solution or fails; if a first-order solution to the problem exists, the routine will deliver the most general first-order solution, if there is not a first-order solution, the routine sometimes still finds some solution. In this thesis we will not describe the implementation of this matching routine.

The Apply tactic (general version)

The tactic Apply $b$ is implemented as:

- Check $\Gamma \vdash b : \Pi x_1 : B_1, \ldots, x_n : B_n. C$, where $x_1 \ldots x_n$ are different and do not occur in $\Gamma$.
   Since $C$ may be a $\Pi$-type, there may be several choices for $n$.

- Find a solution $\sigma$ for the matching problem $\Gamma; x_1 : B_1, \ldots, x_n : B_n \vdash \overset{2}{=} C \overset{2}{=} A$.

- Choose different $h_1 \ldots h_n$ such that $\{h_1, \ldots, h_n\} \cap U = \emptyset$. 

3.2. A TYPE-THEORETICAL PERSPECTIVE

- Define \( c_i = \begin{cases} \sigma(x_i) & \text{if } x_i \in \text{dom}(\sigma) \\ \text{?}_{h_i} & \text{otherwise} \end{cases} \)

Some \( c_i \) are holes, as in the simple version of \texttt{Apply} given above, whereas others are determined by the constraint that \( C \) and \( A \) should match.

- Define \( C_i = B_i[x_1 := c_1, \ldots, x_i := c_{i-1}] \) for all \( i \) with \( x_i \notin \text{dom}(\sigma) \).

These \( C_i \)'s will be the new subgoals.

- Check \( \text{HL}(C_i) = \emptyset \) for all defined \( C_i \).

This check makes sure that the new goals will not contain any holes, as illustrated in Example 3 below.

- Deliver \( (b, c_1, \ldots, c_n, [(\Gamma, ?_{h_i}, C_i) \mid x_i \notin \text{dom}(\sigma)]) \).

There may be several combinations of \( n \) and \( \sigma \) possible. Yarrow takes a combination with the least \( n \).

We give three examples.

1. Take as context \( \Gamma \equiv \Gamma_e, H : \forall x_1 : N.\ P \ x_1 \rightarrow Q \ x_1 \) (i.e. \( H : \Pi x_1 : N.\ \Pi x_2 : P \ x_1.\ Q \ x_1 \)), as subgoal \( A \equiv Q \ O \) and as used holes \( U = \{1, 2\} \). Then \texttt{Apply} \( H \) executes as follows.

The matching routine for problem \( \Gamma; x_1 : N, x_2 : P \ x_1 \vdash Q \ x_1 \equiv_m Q \ O \) gives substitution \([x_1 \mapsto O]\). Let us say \( h_1 = 3, h_2 = 4 \). This gives \( c_1 = O, c_2 = ?_4 \), and \( C_2 = P \ O \). We have \( \text{HL}(C_2) = \emptyset \), so we proceed and deliver \( (H \ O \ ?_4, \{\Gamma, ?_4, P \ O\}) \).

2. This version of \texttt{Apply} indeed generalizes the simple version give before. Suppose that \( \Gamma \vdash b : B_1 \rightarrow \ldots B_n \rightarrow A \), i.e. \( \Gamma \vdash b : \Pi x_1 : B_1.\ \Pi x_2 : B_2.\ \ldots \Pi x_n : B_n.\ A \) with \( \{x_1, \ldots, x_n\} \cap (\text{FV}(B_1) \cup \ldots \cup \text{FV}(B_n)) = \emptyset \). The most general substitution for the matching problem is \( [\] \), i.e. the empty substitution. So \( c_i = ?_{h_i} \) and \( C_i = B_i \) for all \( i \) (since no \( x_i \)'s occur in \( B_j \)), giving the same result as the simple version of \texttt{Apply}.

3. Take as context \( \Gamma \equiv \Gamma_e, H : \forall x_1 : N.\ P \ x_1 \rightarrow R, \) as subgoal \( A \equiv R \) and as used holes \( U = \{1\} \). Then \texttt{Apply} \( H \) executes as follows. The matching routine gives the empty substitution. Let us say \( h_1 = 2, h_2 = 3 \). This gives \( c_1 = ?_2, c_2 = ?_3 \), and \( C_1 = ?_2 \) and \( C_2 = P \ ?_2 \). The tactic now fails, because \( \text{HL}(C_2) = \{?_2\} \neq \emptyset \). If this check were omitted, we would have had a hole in a subgoal.

The Forward tactic (simple version)

The \texttt{Apply} tactic supports backward reasoning, and its power comes from the matching algorithm that finds certain parts of the proof term. Now we present two versions of the \texttt{Forward} tactic, which supports forward reasoning. The simple version given first clarifies the principles, but its practical use is limited. Below we give a powerful version, which uses the matching algorithm.

The simple version of \texttt{Forward} has one parameter, say \( b \) with type \( B \), and the effect of the tactic is that \( B \) is added to the local context (assumptions). This is achieved by constructing the proof term \( (\lambda y : B.\ ?_h)\ b \); so the local context for \( ?_h \) has as additional declaration \( y : B \). (Alternatively, we could use as proof the local definition let \( y := b : B.\ ?_h \). In practice there are no strong arguments for choosing one above the other.) Effectively, we emulate bottom-up construction of the proof term: a subtree of type \( B \) is available for \( ?_h \), and given a complete term for \( ?_h \) we can reduce the redex so that all occurrences of \( y \) are replaced by \( b \).
Why do we say Forward supports forward reasoning? It does so if \( b \) is an application \( c \) \( d \). E.g. suppose that \( d : D \) and \( c : D \rightarrow C \). Then the effect of Forward is that the propositions \( D \rightarrow \Gamma \) and \( D \) are combined into the new assumption \( C \) (which is the type of \( b \)). Moreover, suppose that \( d : D \) and \( c : \Pi x : D.C \), and we consider the type of \( c \) as a universal quantification \( \forall x : D.C \), then the effect is that we instantiate the quantification with \( d \) to the new assumption \( C \{ x := d \} \).

For example, consider the example proof at the point just before Apply is used (page 39). We have the task \((\Gamma, ?, R)\) with \( \Gamma = P, Q, R : s, H : P \rightarrow Q \rightarrow R, H1 : Q, H2 : P \). The tactic Forward \( H H2 \) gives a new task \((\Gamma, H3 : Q \rightarrow R, ?, R)\), so the subgoal is left unchanged and the assumptions are extended with \( Q \rightarrow R \). Forward delivers \((\lambda H3 : Q \rightarrow R. ?_{10}) (H H2)\) as proof term.

The tactic Forward \( b \) is implemented as:

- Check \( \Gamma \vdash b : B \) for some \( B \).
- Let \( h \not\in U \).
- Check \( \Gamma \vdash B \rightarrow A : s \) for some \( s \).
- The proof term \((\lambda y : B. ?_{h}) b\) is only typable if the type of the abstraction, \( B \rightarrow A \), is admitted in the current \( PTS \). In practice, this \( \rightarrow \)-type is always admitted, since it corresponds with the implication \( \rightarrow \) present in all useful \( PTSs \).
- Deliver \( ((\lambda y : B. ?_{h}) b, ((\Gamma, y : B), ?_{h}, A)) \), where \( y \) is chosen so that it does not occur in \( \Gamma \).

In Coq, this simple version of Forward is available by the Coq tactic Generalize. However, the current version of Coq (V6.2.4) does not support the general version of the Forward tactic presented next.

The Forward tactic (general version)

After we implemented this simple version of Forward, it turned out that its practical use could be improved. We illustrate this with an example.

Suppose \( H : \Pi x_{1} : N, P \ x_{1} \rightarrow Q \ x_{1} \) and \( H1 : P \ (S O) \) (compare with example 2 of the general Apply tactic), and we want to add the assumption \( Q \ (S O) \), which is valid by \( H \) and \( H1 \). In the simple version of Forward, we achieve this by the command Forward \( H (S O) \) \( H1 \). The crucial point is that, intuitively, the specification of the argument \( (S O) \) is superfluous, since it is clear by \( H1 \) that we should take for \( x_{1} \) the term \( (S O) \). So we will improve the Forward tactic so that it automatically determines such superfluous terms. Forward now demands two parameters \( b \) and \( d \) separated by the keyword On, and tries to find the obvious arguments \( c_{1}, \ldots, c_{m} \) such that the application \( b \ c_{1} \ldots c_{m} \ d \) is well-typed. E.g. in our example the tactic Forward \( H On \) \( H1 \) achieves the desired effect. In general, this is a big improvement, since the arguments that may be left out can be a lot larger than the \( (S O) \) of our example, and hence the omission of these obvious arguments can make a lot of difference in the amount of text the user has to type.

So the idea of Forward \( b On d \) is to find arguments \( c_{1}, \ldots, c_{m} \) such that the application \( b \ c_{1} \ldots c_{m} \ d \) is well-typed, say with type \( E \). This type is added to the local context, in the same way as in the simple version of Forward. Some \( c_{i} \) may be found by matching, whereas others are a hole, giving additional subgoals. This determination of \( m \) and the \( c_{i}s \) proceeds...
3.2. A TYPE-THEORETICAL PERSPECTIVE

roughly as follows. Determine the type \( D \) of \( d \) and the type \( \Pi x_1 : B_1 . . . . \Pi x_n : B_n . C \) of \( b \). Try to match \( D \) with \( B_1 \). If this succeeds we have \( m = 0 \) since the application \( b \) is well-typed. If this fails, we try to match \( D \) with \( B_2 \). If this succeeds, say with substitution \( \sigma \), we have \( m = 1 \), and there are two cases. If \( \sigma \) determines \( x_1 \) (as in the example above), we have \( c_1 = \sigma(x_1) \), so \( b \circ (x_1) \) is the proof of the new assumption \( (\Pi x_3 : B_3 . . . . \Pi x_n : B_n . C)[x_1 := \sigma(x_1), x_2 := d] \).

If \( \sigma \) does not determine \( x_1 \), we have \( c_1 = ?_{h_1} \), so \( b \circ ?_{h_1} \) is the partial proof of the new assumption \( (\Pi x_3 : B_3 . . . . \Pi x_n : B_n . C)[x_1 := ?_{h_1}, x_2 := d] \), and as additional subgoal we have \( ?_{h_1} : B_1 \). If matching \( D \) with \( B_2 \) does not succeed, we proceed with \( B_3 \), and so on to \( B_n \).

The tactic \texttt{Forward} \( b \) on \( d \) is implemented as:

- Check \( \Gamma \vdash b : \Pi x_1 : B_1 . . . . \Pi x_n : B_n . C \) and \( \Gamma \vdash d : D \), where \( x_1 . . . . x_n \) are different and do not occur in \( \Gamma \), and \( C \) may not be a \( \Pi \)-type.

Since \( C \) is not a \( \Pi \)-type, \( n \) is determined by the type of \( b \).

- Find the least \( m \geq 0 \) for which there is a solution \( \sigma \) for the matching problem

\[ \Gamma; x_1 : B_1 , . . . , x_m : B_m \vdash B_{m+1} \equiv_D \]

- Choose different \( h, h_1 . . . . h_m \) such that \( \{ h, h_1 . . . . h_m \} \cap U = \emptyset \).

- Define for \( 1 \leq i \leq m \) the pseudoterm \( c_i = \begin{cases} \sigma(x_i) & \text{if } x_i \in \text{dom}(\sigma) \\ ?_{h_i} & \text{otherwise} \end{cases} \)

Some \( c_i \) are holes, others are determined by the constraint that \( B_{m+1} \) and \( D \) should match.

- Define \( C_i = B_i[x_1 := c_1, . . . , x_{i-1} := c_{i-1}] \) for all \( i \) with \( 1 \leq m \) and \( x_i \notin \text{dom}(\sigma) \).

These \( C_i \) will be new, additional, subgoals.

- Define \( E = (\Pi x_{m+2} : B_{m+2} . . . . \Pi x_n : B_n . C)[x_1 := c_1, . . . , x_m := c_m, x_{m+1} := d] \).

This is the type of the application \( b \circ c_1 . . . . c_m \), and is the "main" subgoal.

- Check \( \text{HL}(E) = \emptyset \) and \( \text{HL}(C_i) = \emptyset \) for all defined \( C_i \).

This check makes sure that the new goals will not contain any holes.

- Check \( \Gamma \vdash E \rightarrow A : s \) for some \( s \).

- Deliver \( ((\lambda y : E. ?_h)(b \circ c_1 . . . . c_m \circ d), \{ (\Gamma, y : E, ?_h, A) \} \cup \{ (\Gamma, ?_{h_i}, C_i) \mid x_i \notin \text{dom}(\sigma) \}) \), where \( y \) is chosen so that it does not occur in \( \Gamma \).

Let us give three examples of how \texttt{Forward} works.

1. The example given above: take \( \Gamma \equiv \Gamma_e, H : \Pi x_1 : N. P \ x_1 \rightarrow Q \ x_1, H1 : P \ (S \ O) \), and as subgoal some \( A \). Then \texttt{Forward} \( H \) on \( H1 \) executes as follows. For \( m = 0 \), we match \( N \) with \( P \ (S \ O) \), which fails, so we proceed with \( m = 1 \). Matching now succeeds, giving \( \sigma = [x_1 \mapsto S \ O] \). We assume \( ?_{19} \) and \( ?_{20} \) are unused holes, and define \( c_1 = S \ O \). There are no \( C_i \)s defined, and \( E = Q \ (S \ O) \), and we check \( \text{HL}(E) = \emptyset \). Since we are in \( \lambda C \), the type \( E \rightarrow A \) is well-formed, so we deliver as proof term \( (\lambda H2 : Q \ (S \ O), ?_{19} \ (H \ (S \ O) \ H1) \) and as task \( ((\Gamma, H2 : Q \ (S \ O), ?_{19}, A) \).

2. Take \( \Gamma \equiv \Gamma_e, T : N \rightarrow \ast, H : \Pi x_1 : N. T \ x_1 \rightarrow P \ x_1 \rightarrow Q \ x_1, H1 : P \ O \) and as subgoal some \( A \). \texttt{Forward} \( H \) on \( H1 \) delivers as proof term \( (\lambda H2 : Q \ O, ?_{19} \ (H \ O ?_{20} H1) \) and as tasks \( ((\Gamma, H2 : Q \ O), ?_{19}, A) \) and \( (\Gamma, ?_{20}, T \ O) \).
3. Take

\[ \Gamma \equiv \Gamma', L:*, \text{cons}:N \rightarrow L \rightarrow L, eq:L \rightarrow L \rightarrow *, \]
\[ \text{lem5}: \forall a, b:N. \forall x, y:L. eq (\text{cons a x}) (\text{cons b y}) \Rightarrow eq x y, \]
\[ H: eq (\text{cons m p}) (\text{cons n q}) \]

for some terms \( m, n: N \) and \( p, q: L \). Intuitively, \( L \) is the type of lists, \( eq \) is the equality on lists, and \( \text{lem5} \) indicates that if two composed lists are equal, their tails are equal. Take as subgoal some \( A \). Forward \( \text{lem5} \) on \( H \) delivers as proof term

\[ (\lambda H2: eq p q, ?_{19}) (\text{lem5} \ n \ m \ p \ q \ H), \]

and as task \((\Gamma, H2: eq p q), ?_{19}, A)\). The matching algorithm has found the four arguments \( m, n, p \) and \( q \), which would have to be given by hand were the simple version of Forward used. If these arguments are large terms, the advantage of this general version becomes even more apparent.

In the current version of Yarrow, we have generalized Forward even further, by permitting a list of parameters \( d_1, \ldots, d_n \) instead of just one parameter \( d \) behind "On". Supposing \( d_1: D_1 \), we first find an \( m_1 \) such that \( B_{m_1} \) matches with \( D_1 \), say with substitution \( \sigma_1 \), then we find an \( m_2 > m_1 \) such that \( \sigma_1(B_{m_2}) \) matches with \( D_2 \) and so on.

In Yarrow we have also implemented an extended version of the Apply tactic, with syntax Apply \( b \) On \( d \), which is roughly the combination of Forward \( b \) On \( d \) and Apply \( H \), where \( H \) is the name of the new assumption introduced by Forward.

**Special tactics**

In Yarrow we have a fixed set of tactics for the usual introduction and elimination principles in predicate logic with equality, e.g. \( \text{orIL} \), \( \text{orIR} \) and \( \text{orE} \) for the disjunction. We call these tactics special, since they can be used only after the user has indicated the axioms or lemmas that give the introduction and elimination principles.

For instance, if the global context contains a variable \( \text{or}: s \rightarrow s \rightarrow s \) and lemmas

\[ \text{lem1} : \forall P, Q: s. P \rightarrow or P Q, \]
\[ \text{lem2} : \forall P, Q: s. Q \rightarrow or P Q, \]
\[ \text{lem3} : \forall P, Q: s. \forall R: s'. \text{or} P Q \rightarrow (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R, \]

for some \( s, s' \), the user indicates with the commands

\[ > \text{Use orIL lem1} \]
\[ > \text{Use orIR lem2} \]
\[ > \text{Use orE lem3} \]

that he wants to consider \( or \) as a connective for disjunction, with introduction and elimination principles given by \( \text{lem1} \), \( \text{lem2} \) and \( \text{lem3} \). After Yarrow has checked whether the lemmas have the proper form for disjunction, the user can use the tactics \( \text{orIL} \) and so on whenever appropriate. For example, in \( \lambda C \) with lemmas as indicated above \( (s, s' = *) \), here follows a part of the proof of symmetry of \( or \).

\[ P: * \]
\[ Q: * \]
3.2. A TYPE-THEORETICAL PERSPECTIVE

H : or P Q

?5 : or Q P

$ OrE H
?5 := lem3 P Q (or Q P) H (\land P. ?15) (\land Q. ?17)

2 tasks

P : *
Q : *
H : or P Q
H1 : P

?15 : or Q P

2) ?17 : or Q P

$ OrIR
?15 := lem2 Q P ?19

2 tasks

P : *
Q : *
H : or P Q
H1 : P

?19 : P

2) ?17 : or Q P

$

The special tactics do not depend on the particular PTS which is used, and it does not matter whether the introduction and elimination principles are axioms (declarations) or lemmas (definitions). In rich PTSs they usually are lemmas (since the connectives can be defined with their second-order encoding), but in some systems, the principles have to be introduced as axioms.

The implementation of the special tactics for the logical connectives \( \land, \lor, \neg \) and False and the existential quantification \( \exists \) is straightforward (recall that implication \( \to \) and universal quantification \( \forall \) are directly available by \( \Pi \) -types). The implementation for rewriting tactics (i.e. elimination of a Leibniz’ equality), was complicated by two features. First, we allow the equality to be universally quantified. For example, given the subgoal \( P (\text{plus} \ (S \ m) \ O) \) and lemma \( eqO : \forall z : N. \ eq \ (\text{plus} \ x \ O) \ z \), the execution of \texttt{Rewrite eqO} gives the new subgoal \( P (S \ m) \); by traversing the original subgoal and matching each subterm with the left-hand side \( \text{plus} \ x \ O \), Yarrow has found that for \( x \) the term \( (S \ m) \) should be taken. Second, there
may be several instances of the left-hand side in the subgoal, and we allow the user to specify which instance should be rewritten.

3.2.2 Why Holes in Goals are not Allowed

By our definition of state we do not allow holes in (sub)goals (sometimes called dependent goals). It is very useful to have holes in goals, because it allows e.g. the derivation of programs from the specification, as proposed in [Pol94]. But it also introduces huge problems. For a detailed discussion see [Mun97b], here we give just a sketch of the problems related to holes in goals and their possible solutions.

These problems are caused by the wish or need to manipulate and calculate with subgoals, e.g. perform $\beta$-reduction on a subgoal or to calculate the type of a (subterm of a) subgoal. In order to still allow typing of subgoals, we need to adapt the typing rules, so that holes are typed. In order to allow $\beta$-reduction of subgoals, we need to introduce explicit substitutions (see e.g. [Blo97]). For example, the redex $\lambda x : A. ?_1 x$ is contracted to $?_1[x := a]$, which cannot be simplified any further, since at some point we can replace $?_1$ by a term in which $x$ occurs. So $?_1[x := a]$ must be considered as term, and substitution is a syntactical construction inside the system rather than an operation on terms defined outside the system. So we would have to consider PTSs extended with holes and explicit substitutions, which is beyond the scope of this thesis. For the proof assistant Alfa [Hal97] such an approach is used, but with a different type system (and it is not clear on which theory Alfa is based).

An easier way to allow a limited form of holes in goals is to restrict the manipulation of subgoals in which holes occur. This approach is used in Coq and LEGO, but decreases the usefulness of holes in goals. Therefore we chose in Yarrow for the simple approach not to allow holes in goals; holes are only permitted in proof terms, and since those are not manipulated (except replacing a hole by a pseudoterm), no problems occur.

3.2.3 Syntactic Sugar: Infix Notation and Implicit Arguments

If we stick to the standard syntax of PTSs, large terms often become unreadable. In Yarrow we implemented two mechanisms for improving readability. First, we allow infix notation, in a similar way as in Haskell [Tho96]. In this mechanism a number of identifiers (e.g. $+$, $\equiv$, $\land$ and $\sim$) are designated to be infix operators, and expressions using these are parsed accordingly, e.g. the expression $a + b$ is parsed as the application of $+$ to $a$ and $b$. If infix operators are not used infix, they must be bracketed, e.g. $(+) a$ is the application of $+$ to $a$. Infix notation is purely a matter of parsing and printing, and has nothing to do with type theory.

Second, we have a mechanism for implicit arguments, so that "obvious" arguments can be suppressed, both in input and output. This mechanism is explained in detail below. The mechanisms work very well together, for example if we have a polymorphic equality $\text{eq} : \Pi A. A \to A \to \ast$ and the addition $\text{plus} : N \to N \to N$, the proposition $\text{eq} N (\text{plus} x O) x$ can now be written as $x + O = x$ (where the equality has name $=$ instead of $\text{eq}$, and addition $+$ instead of $\text{plus}$).

The mechanism of implicit arguments allows "obvious" arguments to be suppressed. But if the input leaves some arguments implicit, Yarrow has to determine these arguments, since internally these arguments must be present. First of all, only arguments to global variables may be implicit, and only after the user has indicated how many arguments he wishes to
be left implicit for the variable. For instance, the command `Implicit 1 eq` allows the first argument of `eq` (as declared above) to be left implicit. But it is not mandatory to leave arguments implicit, e.g. we may both write `eq O O` and `eq N O O`. In order for Yarrow to detect whether arguments are left implicit or not, the parameter $n$ in the command `Implicit n` $x$ has to satisfy the following property. Suppose

$$\Gamma \vdash x : \Pi x_1 : A_1. \ldots \Pi x_m : A_m. C,$$

$$\Gamma \vdash A_i : s_i \text{ for all } i,$$

then $n < m$ and $s_{n+1} \neq s_1$. Given input $x b_1 \ldots b_k$, Yarrow determines the sort of $b_1$, and if this is equal to $s_1$, there are apparently no implicit arguments; if the sort of $b_1$ is equal to $s_{n+1}$, Yarrow assumes there are implicit arguments. Since $s_1$ and $s_{n+1}$ are different, only one of the situations can occur, so it is clear whether there are implicit arguments in a given input or not.

Now we discuss how to calculate these implicit arguments, suppose $\Gamma \vdash b_i : B_i$ for all $i$. Yarrow tries to find a solution $\sigma$ for all of the following $k$ matching problems.

$$\Gamma; x_1 : A_1, \ldots, x_n : A_n \vdash A_{n+1} \equiv_m B_1$$

$$\vdots$$

$$\Gamma; x_1 : A_1, \ldots, x_{n+k-1} : A_{n+k-1} \vdash A_{n+k} \equiv_m B_k.$$

Assuming this succeeds with substitution $\sigma$, the term with explicit arguments is $x \sigma(x_1) \ldots \sigma(x_n) b_1 \ldots b_k$. This term is used internally, after it is type-checked so that possible bugs in the matching routine do not affect consistency.

For example, consider `comp : \Pi X, Y, Z. (Y \rightarrow Z) \rightarrow (X \rightarrow Y) \rightarrow (X \rightarrow Z)` (e.g. the composition of functions), and $B : \ast, E : N \rightarrow B$ (e.g. $E$ is a boolean function on numbers). The only implicit declaration allowed for `comp` is `Implicit 3 comp`. The input `comp E S` gives the matching problems

$$\Gamma; X : \ast, Y : \ast, Z : \ast \quad \vdash Y \rightarrow Z \equiv_m N \rightarrow B$$

$$\Gamma; X : \ast, Y : \ast, Z : \ast, H : Y \rightarrow Z \quad \vdash X \rightarrow Y \equiv_m N \rightarrow N$$

for which the substitution $[X \leftarrow N, Y \leftarrow N, Z \leftarrow B]$ is a solution. This gives the term `comp N N B E S`. The input `comp E` without further arguments results in an error, since $X$ cannot be determined.

In the output, Yarrow simply does not print the first $n$ arguments of $x$, if $x$ has more than $n$ arguments. So the term `comp N N B` is printed as such, the term `comp N N B E` is printed as `comp E`, and `comp N N B E S` is printed as `comp E S`. As we see in the second case, this simple minded printing approach is not ideal, since the term `comp E` cannot be used as input, but in practice this problem rarely occurs.

### 3.2.4 Flags

Yarrow has the ability to print proofs in the flag-style format [Ned90]. This is a formal notation for proofs which makes it clear which hypotheses are valid at each point in the proof. Every hypothesis is written in a box. Connected to this box is a "flagpole", which indicates the scope of this hypothesis. The justification for every proposition is written behind it; a justification typically consists of a logical construct (e.g. $\land$, $\forall$), a letter that indicates
whether this construct is Introduced or Eliminated, and references to the lines or theorems which the current line depends on. The justification $\equiv m, n$ indicates the current line is obtained by replacing in line $n$ the right-hand side of the equality on line $m$ by the left-hand side, and similarly for $\equiv m, n$. For a simple example of this style, see Figure 3.2. For a more interesting example, see Figure 3.12 on page 69, which proves in a certain context that the \texttt{insert} function keeps ordered lists ordered. We prefer this more formal notation over a textual presentation, because the flag-style format is clearer, more concise, and the propositions are not embedded within English “prose”. A very similar notation is used in Jape [SB96], where this layout is used to build proofs interactively. For comparison, we give the list of tactics used in Yarrow to make the proof of \texttt{Ordered_insert} in Figure 3.10 (on page 68), and Figure 3.11 shows the resulting $\lambda$-term.

The algorithm that produces this presentation of a proof from a proof-object is quite similar to the ones described in [CKT95] and [Cos96], although they produce proofs in pseudo natural language. The basic algorithm is natural and simple: the presentation of a proof term $p$ is a composition of the presentations of the subterms of $p$. However, this produces quite lengthy proofs. A big improvement is the combination of similar steps into one step, e.g. in line 24 of Figure 3.12, two steps are contracted into one ($\forall E$ of line 12 with term $b$, and $\equiv E$ of the result with line 23). Up to this improvement, the algorithm in [Cos96] and ours are similar. One difference is that their algorithm works for the Calculus of Inductive Constructions, whereas ours works for $PTSs$.

Yarrow has the option to print the proofs in \LaTEX format (by default, ASCII format is used). All flag-style proofs in this thesis are generated in this way, and only a few have undergone manual editing to combine some small steps into a bigger step.

3.2.5 Related Work

Table 3.1 gives a comparison between existing proof assistants based on type theory and Yarrow. The most important point of difference is the underlying type system. Whereas existing assistants have a fixed system (or the choice of 4 systems in LEGO), Yarrow can handle any bijective $PTS$ with a finite specification, which makes Yarrow ideal for experi-
### Table 3.1: Comparison of related systems with Yarrow

<table>
<thead>
<tr>
<th>System</th>
<th>Type system(s)</th>
<th>Support for forward reasoning</th>
<th>Automated theorem proving</th>
<th>Holes in goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq</td>
<td>Calculus of Inductive Constructions</td>
<td>Weak</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>LEGO</td>
<td>Logical Framework, Calculus of Constructions, Generalized Calculus of Constructions, Unified Theory of Dependent Types</td>
<td>None</td>
<td>No</td>
<td>Limited</td>
</tr>
<tr>
<td>Alfa</td>
<td>Martin-Löf type theory</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yarrow</td>
<td>Bijective PTSs</td>
<td>Strong</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The implementation with logics (PTSs) of different strengths. On the other hand, all other systems support inductive definitions, which are very useful when formalizing mathematics or proving correctness of programs. In this thesis we can manage without those inductive definitions, although sometimes awkwardly.

As we have shown in Section 3.1.2, Yarrow gives good support to forward reasoning, which existing systems do not. We think they can easily (and should) be extended to support this. Coq is the only system that has some form of automated theorem proving, so that (some) easy parts of a proof can be given by the machine, which is very useful. In Yarrow we have not implemented this because of lack of time. The last point of comparison is whether holes in goals are admitted. We have chosen not to, which keeps things simple (see the end of Section 3.2.1). In Coq and LEGO holes in goals are available in a limited form, which makes them unsuitable for deriving programs from specifications too. Only Alfa supports holes in goals in their full generality. Note that the underlying theory needs major extensions to cope with holes in goals; for instance explicit substitutions. For Alfa it is not clear what this theory is.
3.3 Software Architecture

Yarrow has been coded entirely in the purely functional language Haskell [Tho96, Pet97], making use of the Fudgets library [HC95] for the graphical interface. In this section we discuss various interesting aspects of the Yarrow software architecture:

1. Monads to mimic impure features in a purely functional language. The Yarrow system consists of several layers, such as the type checker, which needs error handling, the proof engine which needs a notion of state, and the user interface on top which requires IO. Error handling, state and IO can all be handled by means of monads [WB89]. So we defined three layers of monads: error monads, state monads and a combination of IO and state monads. By using in each layer of the program exactly the monad that is necessary, every layer gets exactly the resources it is entitled to.

2. Support for multiple interfaces. The Yarrow proof engine has been coded entirely in Haskell and can run on many different platforms. It has been designed in such a way that it can cooperate with various user interfaces. Two such interfaces have been developed. First, a simple command line interface (CLI), which can be used on any platform supporting Haskell, because this interface is based on the standard IO monads [Tho96]. Second, a graphical user interface (GUI) based on the Fudgets library, which is a Haskell library available for a limited number of platforms. The coupling between user interface and engine is very thin, consisting of just a single function.

3. Generic output routines. In the command line interface, strings representing composite terms are composed from strings representing the parts. In the graphical user interface, views representing composite terms are composed from views representing the parts. These two representations are generated by means of a single set of polymorphic output routines.

4. Flexible term representations. An important aspect of proof assistants is the implementation of terms with bound variables and related notions such as substitution. The particular implementation used in Yarrow is such that binding and substitution are handled in a basic layer of the system, independent of the kind of binder. This makes it easy to add new binding constructs, such as existential types (Chapter 6) or bounded quantifications (Chapter 7).

In Section 3.3.1 we give the design issues in more detail. Section 3.3.2 is devoted to the definition of the monads, and the advantages of this approach. In Section 3.3.3 we describe the global architecture of Yarrow, which consists of three main components. Section 3.3.4 treats the top level user interface, and the communication with the engine, which is itself described in Section 3.3.5. The third component consists of the service routines, including parsing and printing, and is described in Section 3.3.6. We give the conclusion in Section 3.3.7.

3.3.1 Design Issues

Yarrow is designed to be a flexible proof assistant. In this section we make this notion more precise, and identify the problems we have to solve to achieve this.

A proof assistant is an interactive system. For every interactive system implemented in a purely functional language, a major design issue is how to handle IO, state and errors. We
elaborate a bit on this subject. Errors — also known as exceptions — may occur in many places in the implementation. The whole program will benefit from a uniform approach. The middle and upper levels of a proof assistant have to be able to work with a state, that represents amongst other the global context (theorems that have already been proved), and the proof that is currently worked on. The top level of a proof assistant must be able to perform input and output (IO), to handle the keyboard, the screen, the file system, and, in case of a GUI, the mouse.

Now we will specify what we mean with flexible. We consider the following aspects concerning the interface:

- The system should be set up in such a way, that it is possible to have several user interfaces available, that all work together with the same kernel.
- It should be easy to change notation, independently of the actual user interface.

We also wish flexibility in the type system that is implemented.

- The user can select the \(PTS\) he wants to use at run-time. It is not necessary to allow every \(PTS\), but the common ones, e.g. the simply typed lambda-calculus, the second-order lambda calculus (system \(F\)), and other \(PTS\)s in the lambda-cube [Bar92] should be available.
- It should be easy to add other term constructions.

3.3.2 Monads

The problems concerning errors, state and IO can be solved by the use of monads, see [Wad92] for an extensive introduction. A monad is a type-constructor \(M\) with a pair of polymorphic functions,

\[
\text{unit}\_M :: a \rightarrow M\ a
\]

\[
\text{bind}\_M :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b
\]

Intuitively, a value of type \(M\ T\) is a calculation which yields a value of type \(T\) as result. It may have some additional effects, depending on the actual monad \(M\). The expression \(\text{unit}\_M\ x\) denotes a calculation which just yields \(x\) as result, without any additional effects. The expression \(\text{bind}\_M\ x\ f\) denotes the combination of \(x\) and \(f\): First \(x\) is calculated, which delivers a result \(r\). Then the calculation of \(f\ r\) takes place. Typically, a monad has some additional functions that introduce the additional effects we spoke about.

Now we introduce three particular monads that play an important role in the implementation of Yarrow.

The simplest monad is the error (exception) monad \(\text{Err}\). A value in such a monad is either a value of some type, if no error has occurred, or an error message. Apart from the usual functions on monads, the error monad has two other functions. First, a function \(\text{error}\_E\) that, given an error message, produces a value in this error monad. Second, a function \(\text{handle}\_E\) that allows the programmer to handle (capture, catch) errors and leave the monad.

\[
\text{data}\ \text{Err}\ a = \text{Success}\ a \mid \text{Error}\ \text{ErrorMessage}
\]

\[
\text{--\ Err\ is\ a\ monad}
\]

\[
\text{unit}\_E\ x = \text{Success}\ x
\]
bindE \ (Error \ m) \ f = Error \ m \\
bindE \ (Success \ r) \ f = f \ r \\
errE \ : : \ ErrorMessage \rightarrow \ Err \ a \\
handleE \ : : \ Err \ a \rightarrow (ErrorMessage \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow b

The next monad is the state and error monad State. From now on we will abbreviate this to state monad. A value \( v \) in such a monad is a state transformer: Given an initial value, \( v \) produces a pair of a new state and a value in the error monad. In the definition of State, we have parametrized over the type of the state. So \( \text{State \ } s \) is a monad with states of type \( s \). Apart from the usual functions on monads, the state monad has functions for reading and updating the state. It also has a function that generates errors and one that handles errors, but the latter has a slightly different type than its \( \text{Err} \) counterpart, since we do not want to lose the state if an error occurs.

\[
\begin{align*}
type \text{State} \ s \ a &= s \rightarrow (s, \text{Err} \ a) \\
\text{unitS} &: : a \rightarrow \text{State} \ s \ a \\
\text{bindS} &: : \text{State} \ s \ a \rightarrow (a \rightarrow \text{State} \ s \ b) \rightarrow \text{State} \ s \ b \\
\text{fetchS} &: : \text{State} \ s \ s \\
\text{updateS} &: : (s \rightarrow s) \rightarrow \text{State} \ s \ s \\
\text{errS} &: : \text{ErrorMessage} \rightarrow \text{State} \ s \ a \\
\text{handleS} &: : \text{State} \ s \ a \rightarrow (\text{ErrorMessage} \rightarrow \text{State} \ s \ b) \rightarrow (a \rightarrow \text{State} \ s \ b) \rightarrow \text{State} \ s \ b
\end{align*}
\]

The last monad contains IO, state and errors. It is a straightforward combination of Haskell's built-in IO monads and the state monads given above.

\[
\begin{align*}
type \text{Imp} \ s \ a &= s \rightarrow IO \ (s, \text{Err} \ a) \\
\text{-- For every } s, \ (\text{Imp} \ s) \text{ is a monad} \\
\text{-- It has functions} \\
\text{-- } \text{unitI}, \ \text{bindI}, \\
\text{-- } \text{fetchI}, \ \text{updateI}, \ \text{errI}, \ \text{handleI}, \\
\text{-- } \text{getLineI}, \ \text{putStrLn}, \ \text{readFileI}, \ \text{writeFileI}
\end{align*}
\]

From now on, when we use the notion IO monad, we mean this \text{Imp}.

We have used Haskell's type classes [WB89, Jon95] to define overloaded functions \text{unit} and \text{bind} so that all monads can be used in exactly the same way (this makes a slight modification of State and Imp necessary). In fact, we could follow this approach even further, by making overloaded functions \text{fetch} and \text{update} for monads that have a state (State and Imp).

Why did not we only define the Imp monad, since everything we can do with \text{Err} and State can also be done with Imp? One big advantage of this layered approach is security. We give each routine just the monad it needs, and not something more powerful. In this way a routine cannot access resources (i.e. IO or state) that it is not entitled to. We give a few examples of this. The typing routine uses the \text{Err} monad, so it cannot change the state, nor can it do any IO. The implementations of tactics use the State monad, because they should change the state concerning a proof, but may not do IO; IO is implemented only in the user interface. So by considering the resources a routine is entitled to, we decide which monad a routine will use. During the implementation, we shifted only a few, auxiliary, routines from
one monad to another, so the distinction between different kinds of monads gave rise to only little additional effort.

Another advantage of the layered approach is the possibility to use monads without IO within a non-monadic function. This is quite important, because the graphical user interface is based on the Fudgets system, which does not work with IO monads at all. Therefore, it is insistent that the proof-engine does not use IO monads, but at most state monads, which can be converted to ordinary functions (with the state as argument and as result). In this way the graphical interface can invoke the proof-engine. The different layers of monads are sketched in Figure 3.3. An arrow from block A to block B means that a function living in A can invoke functions living in B. In other words, a function in B can be converted to one in A.

In the following sections, we show that monads are used throughout the implementation of Yarrow. We will see that higher layers in the implementation use higher level monads.

3.3.3 System Architecture

Figure 3.4 shows the architecture of Yarrow. Each block is a module with a certain functionality, and each arrow indicates a dependency. The block labelled “Engine” is the part that defines the objects of our system (terms, contexts, specification of PTSs) and the functions that manipulate these objects, like the typing routine, the tactics, and the routines that extend the context.

The block labelled “Service routines” consists of all sorts of routines needed for the user interface without actually performing any IO. This includes printing and parsing routines, and the displaying of help texts. The service routines depend on the engine in a rather trivial way. They only use the representation of the objects and some elementary functions on these representations.

The block labelled “Top level user interface” (TLI) contains the main loop of the program. This handles input by the user, sends the appropriate messages to the engine, and presents the results of these messages on the screen. The TLI uses several service routines, but only
one function of the engine. The combination of the TLI with the service routines forms the user interface. We have split the user interface in these two parts because of modularity; every TLI and every set of service routines can be combined into a user interface.

The following three sections each describe one of the blocks and the connection with other blocks.

3.3.4 Top Level User Interface

In Section 3.3.4.1 we describe the communication between the TLI and the engine. Currently, there are two top level interfaces available. The command line interface (CLI) is implemented straightforwardly using the IO monads defined in Section 3.3.2. The CLI is used in a similar way as Coq and LEGO, see Section 3.1.2 for an example. The graphical user interface is described in Section 3.3.4.2.

3.3.4.1 Communication

In this section we describe how the communication between the top level user interface and the Yarrow engine is implemented. From the viewpoint of the TLI, the engine is just a database to which queries can be sent, which will return a certain result. All possible queries are packed into a datatype called Query, and all possible results into Result. The only function from the engine available to the TLI is doQuery, which handles all queries. So the communication between the TLI and the engine can be visualized as in Figure 3.5.

Since we work in a purely functional language, the engine does not own a state. How can the engine then change the context, for example? The answer is that doQuery is a state transformer. Concretely:

\[ \text{doQuery} :: \text{Query} \rightarrow \text{State EngineState Result} \]

So another way of viewing the information flow is that a pair (query, engineState) is sent to the engine, which responds with a pair (result, engineState'). It is the responsibility of the TLI that the next query is paired with this new engineState'.

Now we will consider the datatypes Query and Result in more detail. They are set up in such a way, that the TLI never has to perform manipulations on terms, so that all computational functionality resides in the engine. A stylized subset of the queries and results is depicted in Figure 3.6, in total there are about 30 queries and 20 results. They are grouped around five subjects. The following list gives these subjects and one or two representative queries with their associated results.
3.3. SOFTWARE ARCHITECTURE

![Diagram of communication between top level user interface, query, engine, and result]

Figure 3.5: Communication

1. Proof-tasks. The query **QProveVar** \((\nu, t)\) starts a new proof-task, where \(t\) is the goal, and \(\nu\) is the name of the goal. The engine gives as result **RProofTask taskId toProve**. The variable **taskId** contains the identification of the proof-task (there may be several proof-tasks), and **toProve** contains all information about the proof-task, i.e. the proof term, and all the subgoals. The query **QTactics taskId tacticTerm** performs the tactic associated with **tacticTerm** on proof-task **taskId**, with result **RTactic toProve**.

2. Loading and saving of modules (a module is a group of related definitions). The query **QLoadModule name contents** request the loading of module **name**. Since the engine cannot do any IO, the TLI also gives as parameter the contents of the file associated with **name**. Usually, this leads to the result **RModulesAre** which indicates that loading is done, and gives the new list of currently loaded modules.

   But if the module imports other modules **mods**, a result **RLoadList mods** is returned, to which the TLI reacts by issuing **QLoadModule** queries. Here, the narrow communication channel and the inability of the engine to perform IO make the implementation of loading of modules awkward. This is severed by implementing the printing of status messages, that indicate which module is currently being loaded. All of this is forced through the narrow channel, but the resulting code is not elegant.

3. The global context. The query **QDeclareVars** adds one or more declarations to the context, and **QGiveGlobContext** requests the current context. Both queries result in **RGlobContextIs**.

4. The parameters of the system. The query **QSetTypingSystem** asks for a change in **PTS**, and this is answered with **RTypingSystemOk**.

5. Calculation of normal forms or types. For example, **QGiveType t** requests calculation of the type of **t**, and **RTypeIs** returns **t** with its type and sort.

So each query is associated with a small number of results (usually one). Apart from that, the result **RError** is always allowed.

The big advantage of having only this narrow communication channel between the TLI and the kernel is modularity. This reveals itself in three ways:

- The connection between the engine and the TLI is very sharply defined.
- The TLI uses IO monads or fudgets (both with state) in order to perform IO. Since there is only one communication channel, the conversion between the ordinary functional
CHAPTER 3. YARROW

```plaintext
data Query =
  QProveVar (Var,Term) |
  QTacticTaskId TacticTerm |
  QLoadModule ModuleName String |
  QDeclareVar ([Var],Term) |
  QGiveGlobContext |
  QSetTypingSystem System |
  QGiveType Term |
  ...
data Result =
  RProofTaskId TaskId ToProve |
  RTactic ToProve |
  RModulesAre [ModuleName] |
  RLoadList [ModuleName] |
  RGlobContextIs GlobContext |
  RTypingSystemOk |
  RTypeIs (Term,Term,Sort) |
  RError ErrorMessage |
  ...
```

Figure 3.6: A subset of the queries and results

types of the engine on one hand, and the IO monads or fudgets on the other hand, is limited to one place in the program.

- Since different queries can have the same sort of result, the output for these queries will be uniform.

3.3.4.2 Graphical User Interface

The Fudgets library [HC95] is used to implement the graphical user interface. Fudgets offers mechanisms to construct and combine windows, buttons, menus and other window gadgets. The implementation of the graphical user interface is described in detail in [Rai97].

We treat again the example given in Section 3.1.2, but now in the graphical user interface. The starting point is shown in Figure 3.7. There are two windows, one for the global context (on the left), and one for the proof we are working on (on the right). For this example, all actions take place in the latter. The main area of this window is divided into three parts. The top part shows the local context of the current subgoal, the middle part shows the current subgoal itself, and the bottom part shows the other subgoals. On the right-hand side of the window are several buttons for invoking commonly used tactics. A complete list of tactics can be found under the menu bar entries Tactics and Special. The far bottom of the window consists of a command line, where the user can type in commands, and below that, a status line which displays possible errors.

The user invokes the \( \rightarrow \)-introduction tactic by clicking on the button labelled Intros. This results in an immediate change in the main area of the window. The user then selects the variable \( \mathcal{H} \) in the local context by clicking on it, and clicks on the button labelled Apply. These actions form the \(-\)-elimination tactic. The same effect can be achieved in several other ways. First, by clicking Apply without having selected a term. This causes a pop-up window to appear, in which \( \mathcal{H} \) can be typed in. Second, by selecting the Apply tactic from the menu-bar entry Tactics. Third, by selecting \( \mathcal{H} \), summoning the pop-up menu, and selecting the Apply \( \mathcal{H} \) option from this menu. Last, by typing Apply \( \mathcal{H} \) in the command line on the bottom of the screen.

The graphical user interface has the usual advantages of GUls over CLls. One particular example for Yarrow is that unfolding one occurrence of a defined variable (e.g. in a subgoal)
3.3. SOFTWARE ARCHITECTURE

is achieved by clicking on this occurrence and clicking the Unfold-button. The user of Yarrow with the CLI has to count which occurrence he wants to unfold, and type in this number as parameter to the Unfold tactic.

3.3.5 Engine

The engine is subdivided into four blocks, as illustrated in Figure 3.8, where the arrows symbolize dependency. The blocks on the right define the datatypes representing terms, contexts and PTSs (on the lower level) and representing tactics (on the upper level). The blocks on the left define the algorithms working on these datatypes; the typing and reduction routines etc. on the lower level, and the implementation of tactics and other commands on the upper level. So the lower level concerns terms, and the upper level concerns commands. We now treat each of the block in more detail, and state which monads are used.

We treat the representation of PTSs, terms and contexts separately in Section 3.3.5.1.

All routines working on terms are non-monadic or use the error monad. Often some part of the state (e.g. the current PTS) is given as argument to these routines. The most important routine is the typing algorithm. Essentially, we have used Poll's algorithm for bijective PTSs [Pol93], since this class of bijective PTSs includes all systems in the lambda-cube [Bar92].

The block defining the representation of tactics is necessary because of two reasons. First, it allows all tactics to be covered by one query (see Section 3.3.4.1). Second, tactics are printed in a summary, when a proof is finished.

Finally, the implementations of tactics and other commands use mostly state monads. The function doQuery (see Section 3.3.4.1) is also of this form; it acts as a sort of a traffic warden. It inspects the query and calls the appropriate command.
3.3.5.1 Representation of Terms

A major design issue when implementing a proof assistant is how to represent terms. The most important decision to be made, is how bound variables must be handled. The most common possibilities are:

**Named variables** Annotate each binder with the bound variable, and use identical representations for bound and free variables. Care must be taken for name clashes and substitutions, but these subtleties are hidden in a few routines.

**de Bruijn indices** Represent a bound variable by an index, which is the number of binders between the occurrence of the variable and its binder.

First we observe that names have to be retained anyway, since the clearest way to print bound variables is using names. That means \( \alpha \)-conversion is still needed, in order to print terms in an unambiguous way. This is of crucial importance, since the user assigns meaning to the term as it is printed, and not as it is represented internally.

So the simplest way of representing bound variables is using names (it is not necessarily the most efficient). Therefore we have chosen this approach in Yarrow.

Another issue is how to represent terms, so that they may easily be extended with other term constructions than those of PTSs, e.g. records and existential types. We chose to set up our datatype representing terms in a way reminiscent of Combinatory Reduction System [KOR93]. We consider three *categories* of terms.

- The *basic category* contains terms that do not have any subterms. For any Combinatory Reduction System, variables belong to this category. For PTSs, sorts also belong to this category.

- The *non-binder category* contains terms that have a number of subterms, and do not bind any variables in those subterms. For PTSs, only applications belong to this category. Applications have two subterms.
The *binder category* contains terms that bind one variable of some type in one subterm called the body. Furthermore, they may have a number of subterms in which the variable is not bound. For $DPTS$s, abstractions, $\Pi$-types and local definitions belong to this category. The definens in a local definition is an example of a subterm in which the variable is not bound.

This definition of binder category could be generalized so that multiple variables may be bound, but this is not necessary for our purposes.

The small disadvantage that this division is slightly less efficient in execution space and time, is hugely offset by the advantage that critical routines like substitution, determination of free variables, and changing a bound variable ($\alpha$-conversion) are written once and for all, and never need to be adapted. Furthermore, routines like matching and checking for convertibility of terms become more uniform.

This is the actual Haskell code that defines terms:

```haskell

-- Var with type and sort
type Item = (Vari,Term,Sort)

data Term = Basic BasicCat | Nonb NonbCat [Term] | Bind BindCat [Term] Item Term

-- No subterms
instance NonbCat = Vr Vari |
instance Srt Sort |
instance Hole Hnum |

-- Two subterms
instance NonbCat = App |

-- No additional subterms
instance BindCat = Abs |
instance All |
instance Delta |

We always annotate a variable with its sort (the type of its type), hence Item is a triple. We have a basic construct called Hole which is used for the construction of proof terms.

**Example 3.3.5.1.1** The term let \(x\) := \(\text{three} : \text{Nat}\). even \(x\) is encoded as:

\[
\text{Bind Delta [Basic (Vr three)] (x, Basic (Vr Nat), *)}
\]

\[
\text{(Nonb App [Basic (Vr even), Basic (Vr x)])}
\]

where we assume \(\text{three}, \text{Nat}, \text{even}\) and \(x\) are of type \(\text{Vari}\), and * is of type Sort.

For routines like type-checking, that are specific for $PTS$-terms (instead of being uniform for general terms with bounded variables), we supply constructor and destructor functions for each form of $PTS$ terms. E.g., the constructor for application takes two terms \(t\) and \(u\) and returns the application Nonb App \([t,u]\). The corresponding destructor takes a term \(t\) and delivers a boolean stating whether \(t\) is an application, and the subterms of \(t\) if so.

Contexts are represented in a similar vein as terms, so that it is easy to move a binder into the context.

```haskell

type ContextElement = (Item,ContCat,[Term])

data ContCat = Decl | Def

-- No other arguments
-- One other argument
```
3.3.6 Service Routines

The service routines are divided in six modules, as illustrated in Figure 3.9. We give a short explanation of the modules.

The module "Representation of commands" defines the representation of the textual commands, so that these commands are available to all top level interfaces. In fact, the GUI uses this module, so that the user may also type in commands, instead of using the mouse.

The two modules that define the parser routines are quite straightforwardly implemented. The parser routines are written in an imperative style, using state monads, although parser combinators may be more elegant (e.g. [Hut92]).

The remaining modules "Printing terms", "Printing tactics" and "Help texts" are all concerned with output, and are given a separate treatment in Section 3.3.6.1, in which we explain how these routine are made independent of the actual TLI.

3.3.6.1 Polymorphic Printing Routines

Although the various interfaces differ greatly in outward appearance, they actually have many structural similarities which can be exploited by making the output routines polymorphic. For instance, when outputting a term as a value of some type a (e.g. String or Graphics), we need two functions. One function turns an atomic piece of text into an a, so this function has type String -> a. It is used for outputting variables or symbols (like \ and ->). The other function concatenates a number of a's together, so it has type [a] -> a. It is used to print composite terms. So, a representation using type a is characterized by a value of tuple type (String->a, [a]->a), hence the print routine for terms is declared as

\[
\text{printTerm :: Display a -> Term -> a}.
\]

A similar approach has been followed for generation of output in the help system. In this case the print routine is declared as:

\[
\text{printHelp :: Display a -> HelpText -> a}.
\]

Actually, both printing routines are a bit more complicated, to cater for selection of subterms and clicking on links in the help texts.
3.3.7 Conclusion

The complete proof assistant has been written in Haskell. The suitability of this language varies over different parts of the program. The engine and service routines benefit from Haskell features like polymorphism, higher-order functions, and especially user-defined datatypes. For the top level user interfaces, these features do not play such an important role. The command line interface with its IO monads is imperative in style and could, in itself, equally well be written in an imperative language. The graphical user interface uses the Fudgets library, which allows easy construction of windows from components like buttons and text-fields. However, Fudgets offers only a very basic functionality; some essential features, like saving files, and many fancy features, like dragging, are absent. Furthermore, it is slow and memory demanding. In our experience, the concepts of the Fudgets library made programming a GUI relatively easy and enjoyable, but the library is not ripe enough to be used in a "real-world" application.

Monads are introduced in [Wad92] to mimic impure features, like errors, state and IO. These features are necessary in a proof assistant, and therefore we used monads. This turned out to be a good decision: monads allow a flexible, elegant and uniform treatment of errors, state and IO. However, in a big program, such as Yarrow, we cannot just use one monad throughout the program, since it would give the whole program access to all resources (state, IO). Therefore we defined different layers of monads, and each layer of the program gets exactly the monad it needs. For example, the typing routine uses error monads, and cannot make any changes in the state, nor perform any IO. Haskell’s type classes [Jon95] make a consistent use of monads possible. We expect that this approach with different layers of monads can be applied to many big programs written in Haskell.

Another important issue is how to design the proof assistant in such a way that several user interfaces can be used with one engine. We decided to create a very narrow communication channel of queries and results, that are handled by one function of the engine. This approach is successful. It helps to separate the tasks of the total program, and allowed us to implement the user interfaces quite independently from the engine. Furthermore, it promotes uniformity, e.g. similar commands always give the same format of output. The strict adherence to this discipline made the implementation of a few commands with a lot of IO (viz. loading and saving of modules) complicated, but the other commands are well-suited to this approach.

We have claimed Yarrow is a big program. Let us make this more precise. The engine consists of about 7600 lines of code. About half of this is for the kernel, which includes representation of terms, reduction, typing and matching. The service routines are implemented in 2500 lines of Haskell. The command line interface is about 1000 lines of code, which is rather small compared with the graphical user interface, consisting of about 5000 lines. This brings the total program size to 11000 to 15000 lines. The most complicated routines lie in the kernel of the program, particularly an efficient reduction routine and a matching routine. Also much effort has gone into the Apply and Rewrite tactics.

All in all, the coding of a complete proof assistant in Haskell has been a successful experiment. The extension of Yarrow with records (which are described in Chapter 6), was quite easy, because of our flexible representation of terms. The extension with subtyping (Chapter 7) was more difficult, because of the adaptations necessary to the tactics and the matching routine. We intend to extend the graphical user interface with proof by pointing.

Yarrow with a textual interface is electronically available on the world wide web (see page 321).
Figure 3.10: The Yarrow tactics for the proof of Ordered_insert.

Prove Ordered_insert : ∀m:Nat. ∀l:List Nat. Ordered l →

Ordered (insert m l)

Intro
Apply indlist
Intro
Rewrite insert_nil
Apply Ordered_singleton
Intros
AndE Ordered_cons_ On H1
OrE Le_Or_Gt On m, a
Rewrite Le_insert Then Try Assumption
Apply Ordered_cons
Assumption
Intro
Apply Le_trans On H6
OrE Elem_cons_ On H7
Rewrite H9
Apply Le_refl
Apply H3 Then Assumption
Rewrite Gt_insert Then Try Assumption
Apply Ordered_cons
Apply H Then Assumption
Intro
OrE Elem_insert_ On H7
Rewrite H9
Apply m_Lt_n_m_Le_n
Assumption
Apply H3
Assumption
Exit

Figure 3.11: The proof of Ordered_insert as λ-term

\[ \lambda m:\text{Nat}. \text{indlist Nat} (\lambda a:\text{List Nat}. \text{Ordered as} \rightarrow \text{Ordered (insert m as)}) (\lambda H: \text{Ordered (nil Nat)} \cdot \text{is_elim_1 (List Nat) (insert m (\text{nil Nat})) (singleton m)} (\lambda H1: \text{List Nat.Ordered H1) (insert_nil m) (Ordered_singleton m)}) (\lambda a:\text{Nat. did: List Nat. \text{Ordered}} \rightarrow \text{Ordered (insert m as). \text{is_elim_2 (List Nat) (insert m as).)} (\lambda H2:\text{(\forall b: Nat. Elem b as \rightarrow a < b)} \cdot \text{Ordered as. (\lambda H5: m \leq a \lor a < m. \text{Or_e (m \leq a)} (a < m)} (\text{Ordered (insert m (a; as))) H5 \cdot \text{Le inserted m a as H6) (Ordered_cons a as H1)} (\lambda b: \text{Nat. H7: Elem b (a; as)} \cdot \text{Le_trans m a b H6 (\lambda H8: b = a \lor \text{Elem b as. Or_e (b = a)} (\text{Elem b as)} (a < b)} H8 (\lambda H9: b = \text{is_elim_1 Nat b a (H10: Nat .a \leq H10) H9 (Le_refl a)} (\lambda H9: \text{Elem b as. H9 b H9)} (\text{Elem_conc Nat b a as H7))) (\lambda H6: a < m. \text{is_elim_1 (List Nat) (insert m (a; as)) (a; insert m as)}} (\lambda H7: \text{List Nat.Ordered H7) (Gt_insert m a as H6) (Ordered_cons a (insert m as) (H H4)} (\lambda b: \text{Nat. H7: Elem b (insert m as). (\lambda H8: b = m \lor \text{Elem b as. Or_e (b = m)} (\text{Elem b as)} (a < b)} H8 (\lambda H9: b = \text{is_elim_1 Nat b m (H10: Nat.a \leq H10) H9 (m.Lt_n_m_Le_n a m H6)} (\lambda H9: \text{Elem b as. H9 b H9)} (\text{Elem_insert b m as H7)))) (\text{Le_Or_Gt m a)}) (\text{And_er (\forall b: Nat. Elem b as \rightarrow a \leq b)} (\text{Ordered as) H2)} (\text{And_el (\forall b: Nat. Elem b as \rightarrow a \leq b)} (\text{Ordered as) H2)} (\text{Ordered_cons a as H1})) \]
Figure 3.12: The proof of Ordered.insert in flag-style

```
1  m : Nat
2  Ordered (nil Nat)
3  insert m (nil Nat) = singleton m
4  Ordered (singleton m)
5  Ordered (insert m (nil Nat))
6  Ordered (nil Nat) => Ordered (insert m (nil Nat))
    a : Nat
7  as : List Nat
8  Ordered as => Ordered (insert as)
9  Ordered (a, as)
10
11  (forall Nat. ElemNat b as => a <= b) \land Ordered as
12  forall Nat. ElemNat b as => a <= b
13  Ordered as
14  m <= a \land a < m
15  m <= a
16
17  insert m (a, as) = m, a, as
18  b : Nat
19  ElemNat b (a, as)
20  b = a \lor ElemNat b as
21  a <= a
22  a <= b
23  ElemNat b as
24  a <= b
25  a <= b
26  m <= b
27  forall Nat. ElemNat b (a, as) => m <= b
28  Ordered (m, a, as)
29  Ordered (insert m (a, as))
30  a < m
31
32  insert m (a, as) = a, insert m as
33  Ordered (insert m as)
34  b : Nat
35  ElemNat b (insert m as)
36  b = m \lor ElemNat b as
37  a <= m
38  a <= b
39  ElemNat b as
40  a <= b
41  a <= b
42  forall Nat. ElemNat b (insert m as) => a <= b
43  Ordered (a, insert m as)
44  Ordered (insert m (a, as))
45  Ordered (insert m (a, as))
46 forall Nat.forall List Nat. (Ordered as => Ordered (insert m as)) =>
47  Ordered (a, as) => Ordered (insert m (a, as))
48  forall List Nat. Ordered l => Ordered (insert m l)
49  forall Nat.forall List Nat. Ordered l => Ordered (insert m l)
```
Chapter 4

\(\lambda\omega_L\), a Programming Logic

In this chapter the pure 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 datatypes 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 \(F^\omega\). 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 [Car91] 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 strong language. In the following chapters, more type 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 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 seen 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 4.1 and 4.2, and then in Section 4.3 the programming logic \(\lambda\omega_L\) is defined. Section 4.4 reflects on consistency and semantics of \(\lambda\omega_L\). Up to this point, we follow the ideas of [Pol94]. Indeed, much of the text is copied from that source. Section 4.5 summarizes the definitions and axioms, which we use to give a library of lemmas of \(\lambda\omega_L\) in Section 4.6. In Section 4.7 we use this library to specify a program, give its implementation and prove it correct.
4.1 $\lambda\omega$ as a Programming Language: $\lambda\omega_\delta$

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:

**Definition 4.1.1** $\lambda\omega_\delta$ is the (D)PTS specified by

$$S = \{s, \Box\}, \quad A = \{(*_s, \Box_s)\}, \quad R = \{(*_s, *_s), (\Box_s, *_s), (\Box_s, \Box_s)\}.$$

This very compact definition of $\lambda\omega_\delta$ is not very enlightening. Below we discuss the different kinds of terms that can be distinguished in $\lambda\omega_\delta$, and the different forms of abstractions that are allowed.

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

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 $\times$ that maps two datatypes to their product type, or a function $\text{List}$ that maps a datatype $A$ to the datatype of $A$-lists.

An important difference between the datatypes and the other datatype-constructors is that the former can be inhabited – by programs –, whereas the latter cannot. For instance, a function $\text{List}$ 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

$$IK ::= *_s \mid IK \rightarrow IK.$$  

$*_s$ is the type of all datatypes, $*_s \rightarrow *_s$ is the type of all functions from datatypes to datatypes, and so on. E.g., $\text{List}$ has type $*_s \rightarrow *_s$, and $\times$ has type $*_s \rightarrow *_s \rightarrow *_s$.

There is a fourth level in $\lambda\omega_\delta$, which consists just of the symbol $\Box_s$. $\Box_s$ is the type of all the kinds, so $*_s, *_s \rightarrow *_s$, etc. all have type $\Box_s$. It plays an important role in the definition of $\lambda\omega_\delta$ 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:

**Definition 4.1.2**

1. If $\Gamma \vdash IK : \Box_s$ for some context $\Gamma$, then $IK$ is a *kind*.

2. If $\Gamma \vdash A : IK$ for some kind $IK$ and context $\Gamma$, then $A$ is a *datatype-constructor*.

3. If $\Gamma \vdash A : *_s$ for some context $\Gamma$, then $A$ is a *datatype* (and a datatype-constructors).

4. If $\Gamma \vdash a : A$ for some datatype $A$ and context $\Gamma$, then $a$ is a *program*. □
4.1. $\lambda \omega$ AS A PROGRAMMING LANGUAGE : $\lambda \omega_s$

1. \[ f : A \rightarrow B \]
   - hyp
2. \[ g : B \rightarrow C \]
   - hyp
3. \[ x : A \]
   - hyp
4. \[ f \, x : B \]
   - $\rightarrow E \ 1,3$
5. \[ g \,(f \, x) : C \]
   - $\rightarrow E \ 2,4$
6. \[ \lambda x : A \, g \,(f \, x) : A \rightarrow C \]
   - $\rightarrow I \ 3-5$
7. \[ \lambda g : B \rightarrow C \, . \lambda x : A \, g \,(f \, x) : (B \rightarrow C) \rightarrow A \rightarrow C \]
   - $\rightarrow I \ 2-6$
8. \[ \lambda f : A \rightarrow B \, . \lambda g : B \rightarrow C \, . \lambda x : A \, g \,(f \, x) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C \]
   - $\rightarrow I \ 1-7$

Figure 4.1: Typing derivation in flag-style

**Theorem 4.1.3** $\lambda \omega_s$ is $SN_{B_S}$.

*Proof:* $\lambda \omega_s$ is a subsystem of $\lambda C_s$, and $\lambda C_s$ is $SN_{B_S}$ (Theorem 2.3.9). \qed

**Basic datatypes and datatype-constructors**

---

4.1. $\lambda \omega$ AS A PROGRAMMING LANGUAGE : $\lambda \omega_s$

Contexts can declare variables as datatype-constructors and as programs. For example, the context

\[
\text{List} : * , \ s \rightarrow * , \ \text{Nat} : * , \ n : \text{Nat} , f : \text{Nat} \rightarrow \text{Nat} , A : * , \ x : A
\]

declares a datatype-constructor List, datatypes Nat and A, and programs (data) n, f and x.

**Convention:** We use meta-variables as follows:

- $IK$ ranges over kinds.
- $A, B$ range over datatypes and the other datatype-constructors.
- $X$ ranges over $V$ (the set of variables), when used as datatype-constructor.
- $a, b, f, g$ range over programs.
- $x, y, z$ range over $V$, when used as programs.
- Meta-variables may be decorated with subscripts and primes.

For actual variables (elements of $V$) we have the following conventions:

- Datatype-constructors are written in the teletype-font, and start with a capital, e.g. Nat and List.
- Programs are written in the teletype-font, and start with a small letter, e.g. true and reverse.

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 $(\square_s, \square_s)$ provides a form of abstraction in datatype-constructors. The rules
The same levels can be distinguished as in $\lambda\omega_s$, but now their interpretation is different. The different levels of $\lambda\omega_p$-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 soon become 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

   $$IP ::= *_p | IP \to IP .$$

   $*_p$ is the type of all propositions, $*_p \to *_p$ is the type of all functions from propositions to propositions, and so on. For example, $\neg$ has type $*_p \to *_p$, and $\land$ has type $*_p \to *_p \to *_p$.

   The fourth level in $\lambda\omega_p$ consists of just the symbol $\square_p$. $\square_p$ is the type of all the prop-kinds, e.g. $*_p, *_p \to *_p$, etc. all have type $\square_p$.

An example of such an abstraction is $(\lambda A:*_. A \to A) : *_s \to *_s$, the datatype-constructor that maps a datatype $A$ to the datatype $A \to A$.

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 characterize the different axes.

- $(*_s, *_s)$ gives programs that depend on programs.
  For example, the program $(\lambda x:A. b)$ a depends on the program $a$.

- $(\square_s, *_s)$ gives programs that depend on datatypes.
  For example, the program $(\lambda X:*_. b) A$ depends on the datatype $A$.

- $(\square_s, \square_s)$ gives datatypes that depend on datatypes.
  For example, the datatype $List A$ depends on the datatype $A$.

A nice way to present typing derivations is to use the flag-style format [Ned90], in which "flags" denote the scope of declarations. As an example, we consider in Figure 4.1 the typing derivation for the composition of functions, defined here as $\lambda f:A \to B, \lambda g:B \to C, \lambda x:A.g(fx)$ of type $(A \to B) \to (B \to C) \to A \to C$.

This is the only type derivation for a program in this thesis, since it is easy enough to check that our programs are well-typed, because they will all be fairly small.

$\lambda\omega_s$ and its extension with definitions $\lambda\omega_{s6}$ are strongly normalizing:
Definition 4.2.2

1. If $\Gamma \vdash iP : \Box_p$ for some context $\Gamma$, then $iP$ is a \textit{prop-kind}.

2. If $\Gamma \vdash P : iP$ for some prop-kind $iP$ and some context $\Gamma$, then $P$ is a \textit{prop-constructor}.

3. If $\Gamma \vdash P : *_p$ for some context $\Gamma$, then $P$ is a \textit{proposition} (and a prop-constructor).

4. If $\Gamma \vdash p : P$ for some proposition $P$ for some context $\Gamma$, then $p$ is a \textit{proof}. \hfill \Box

Contexts can declare prop-constructors and proofs. For example, the context

$$\neg : *_p \rightarrow *_p , P : *_p , p : \neg P$$

declares prop-constructors $\neg$ and $P$ and a proof $p$. The declaration $p : \neg P$ in the context is an \textit{assumption} (or an \textit{axiom}), viz. the assumption that there is a proof $p$ of the proposition $\neg P$.

Convention:

- $iP$ ranges over propkinds.
- $P, Q, R, S, T$ range over propositions and the other prop-constructors.
- $p, q$ range over proofs.

We do not reserve special meta-variables ranging over proposition-variables or proof-variables (assumptions). There is no notational difference between actual variables for propositions and proofs and the corresponding meta-variables.

The rules $(*_p, *_p)$ and $(\Box_p,*_p)$ provide two ways of forming propositions. The rule $(*_p,*_p)$ allows the formation of implication:

$$(\iff\text{-form}) \quad \frac{\Gamma \vdash P : *_p \quad \Gamma \vdash Q : *_p}{\Gamma \vdash P \iff Q : *_p}$$

Here $\iff$ is just another notation for $\rightarrow$ that reflects its intended interpretation. The associated introduction and elimination rules are

$$(\iff\text{-intro}) \quad \frac{\Gamma, p : P \vdash q : Q \quad \Gamma \vdash P \iff Q : *_p}{\Gamma \vdash (\lambda p : P. q) : P \iff Q}$$

$$(\iff\text{-elim}) \quad \frac{\Gamma \vdash q : P \iff Q \quad \Gamma \vdash p : P}{\Gamma \vdash q p : Q}$$

A proof of $P \iff Q$ 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 $p, q,$ $(\lambda p : P. q)$ and $q p$ leaves the usual introduction and elimination rules for implication in natural deduction.

The rule $(\Box_p,*_p)$ allows higher-order universal quantification

$$(\forall\text{-form}) \quad \frac{\Gamma \vdash iP : \Box_p \quad \Gamma, P : iP \vdash Q : *_p}{\Gamma \vdash (\forall P : iP. Q) : *_p}$$
Here \( \forall \) is just another notation for \( \Pi \). If \( \mathcal{P} \) is \( \star_p \), this results in quantification over all propositions, as in second-order logic. This makes it possible for example to form the proposition \((\forall P: \star_p, \neg P \implies P)\).

Finally, the rule \((\square_p, \square_p)\) allows the formation of prop-kinds of the form \( \mathcal{P}_1 \to \mathcal{P}_2 \):

\[
(\to\text{-intro}) \quad \frac{\Gamma \vdash \mathcal{P}_1 : \square_p \quad \Gamma \vdash \mathcal{P}_2 : \square_p}{\Gamma \vdash \mathcal{P}_1 \to \mathcal{P}_2 : \square_p}
\]

For example, it provides the prop-kind \( \star_p \to \star_p \), the type of \( \neg \), and \( \star_p \to \star_p \to \star_p \), the type of \( \vee \) and \( \land \). It also makes it possible to abstract over a prop-kind in propositions and prop-constructors. E.g., the propositional connective \( \neg \) can be defined as \((\lambda P: \star_p, P \implies \text{False})\) of type \( \star_p \to \star_p \), where False is the proposition representing falsehood.

Again, we present a typing derivation using these rules in flag-style, namely \( \lambda p: A \implies B \). \( \lambda q: B \implies C \). \( \lambda r: A \). \( q(p r) \) which has type (is a proof of) \((A \implies B) \implies (B \implies C) \implies A \implies C \).

\[
\begin{array}{c|l}
1 & p: A \implies B \\
2 & q: B \implies C \\
3 & \ \\
4 & r: A \\
5 & \rightarrow E 1,3 \\
6 & p r: B \\
7 & \ \\
8 & \lambda r: A. q(p r): A \implies C \\
9 & \ \\
10 & \lambda q: B \implies C. \lambda r: A. q(p r): (B \implies C) \implies A \implies C \\
11 & \ \\
12 & \lambda: A \implies B. \lambda q: B \implies C. \lambda r: A. q(p r): (A \implies B) \implies (B \implies C) \implies A \implies C \\
13 & \ \\
\end{array}
\]

Note the similarity with the derivation on page 75. In contrast with programs, we are not interested in the terms, which soon become quite long, so we usually leave these out. All essential information is in the types and in the justification written behind the types. For an example of a large proof term and its presentation in flags, see Figures 3.11 and 3.12 on pages 68 and 69.

**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.

**Definition 4.2.3** LOGIC is the context

\[
\begin{array}{c}
\text{False} & := & \forall P: \star_p, P \\
& : & \star_p \\
\text{True} & := & \forall P: \star_p, P \implies P \\
& : & \star_p \\
\end{array}
\]
4.3. THE PROGRAMMING LOGIC $\lambda \omega_L$

\[\neg := \lambda P:*_p, P \rightarrow False\]
\[\wedge := \lambda P, Q:*_p, \forall R:*_p, (P \rightarrow Q \rightarrow R) \rightarrow R\]
\[\vee := \lambda P, Q:*_p, \forall R:*_p, (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R\]

This context defines the propositional constants truth ($\text{True}$) and falsehood ($\text{False}$) and the logical connectives negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). From now on, we always assume LOGIC is part of the context.

For a propkind $IP$ we can define

\[\exists_F := \lambda P:IP \rightarrow *_p, (\forall R:*_p, (\forall X:IP. (P X) \rightarrow R) \rightarrow R)\]

Then for a term $(\lambda S:IP. P) : IP \rightarrow *_p$, the term $\exists_F(\lambda S:IP. P)$ represent the higher-order existential quantification "$\exists S:IP. P$". The definition of $\exists_F$ is parameterized by a propkind $IP$. We cannot abstract over $IP$ and define $\exists$ as follows

\[\exists := \lambda P:*_p, \exists_F\]
\[\Pi IP:*_p, (IP \rightarrow *_p) \rightarrow *_p,\]

because the abstraction $\lambda IP:*_p, \ldots$ is not allowed in $\lambda \omega_p$. One way to consider $\exists$ as a single constant, is to allow parametric definitions [Laa97].

In contrast to the encoding of datatypes in $\lambda \omega_s$, this encoding of logical connectives in $\lambda \omega_p$ is quite satisfactory. First, we are not interested in their reduction behaviour. Second, all their intuitionistic properties, i.e. their introduction and elimination principles, are provable in $\lambda \omega_p$. For example, in $\lambda \omega_p$ we can prove $(\forall P:*_p, \text{False} \rightarrow P), (\forall P, Q:*_p, P \rightarrow (P \vee Q))$, and $(\forall P, Q:*_p, (P \wedge Q) \rightarrow P)$. Note that the quantification over propositions makes it possible to express these introduction and elimination principles inside the system $\lambda \omega_p$.

4.3 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 2.1.3.1. In fact, it is a subsystem of one of the variants of the Calculus of Constructions defined in [PM89].
Before defining $\omega_L$, we consider which $PTS$-rules are required for reasoning about $\omega_L$-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 $A$ is $A \rightarrow *_p$. Similarly, a relation or binary predicate on a datatype $A$ is of type $A \rightarrow A \rightarrow *_p$. 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 $A$ is a relation $\equiv : A \rightarrow A \rightarrow *_p$. Like $*_p$, the terms $A \rightarrow *_p$ and $A \rightarrow A \rightarrow *_p$ are prop-kinds, i.e. they have type $\square_p$. Their formation requires the $PTS$-rule $(*_s, \square_p)$:

\[
\frac{\Gamma \vdash A : *_s \quad \Gamma \vdash IP : \square_p}{\Gamma \vdash A \rightarrow IP : \square_p}
\]

(→-intro)

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

\[
\frac{\Gamma \vdash IK : \square_s \quad \Gamma, X : IK \vdash IP : \square_p}{\Gamma \vdash (\Pi X : IK. IP) : \square_p}
\]

(Π-form)

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

\[
\frac{\Gamma \vdash A : *_s \quad \Gamma, x : A \vdash P : *_p}{\Gamma \vdash (\forall x : A. P) : *_p}
\]

(∀-form)

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 $(\square_s, *_p)$ is needed:

\[
\frac{\Gamma \vdash IK : \square_s \quad \Gamma, X : IK \vdash P : *_p}{\Gamma \vdash (\forall X : IK. P) : *_p}
\]

(∀-form)

By taking $IK \equiv *_s$, this rule allows universal quantification over all datatypes. For example, this is used in the proposition $\forall A : *_s, \forall x : A. (\equiv A x x)$, which expresses reflexivity of the relation $\equiv$ for all datatypes $A$.

We have now discussed all the rules of $\omega_L$:

**Definition 4.3.1** $\omega_L(\mathcal{S})$ is the (D)PTS specified by

\[
\begin{align*}
\mathcal{S} & = \{*_s, \square_s, *_p, \square_p\} \\
\mathcal{A} & = \{(*_s : \square_s), (*_p : \square_p)\} \\
\mathcal{R} & = \{\square_s, \square_p, (*_s, *_p), \square_s, *_p, (*_s, *_p), \square_p, *_p, (*_s, *_p), \square_p, *_p, (*_s, *_p), \square_p, *_p, (*_s, *_p)\}.
\end{align*}
\]
4.3. THE PROGRAMMING LOGIC $\lambda\omega_L$

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 4.1.2 and 4.2.2.

Observe that when $\mathcal{R}$ 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: $\{(\Box_s, \Box_s), (\Box_s, \star_s), (\star_s, \star_s)\}$,
- $\lambda\omega_p$ for the propositions and their proofs: $\{(\Box_p, \Box_p), (\Box_p, \star_p), (\star_p, \star_p)\}$,
- all possible dependencies of propositions and proofs on programs and types, provided by all rules of the form $\langle \star_s, \star_p \rangle$: $\{(\Box_s, \Box_p), (\Box_s, \star_p), (\star_s, \Box_p), (\star_s, \star_p)\}$.

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

- $\langle \star_s, \Box_s \rangle$.
  As a result, there are no datatypes that depend on programs. An example of a datatype depending on a program is the type $\langle \text{less} \text{than } a \rangle$ that consists of all natural numbers smaller than $a$. 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 Section 1.3 we already discussed why we do not want such a very expressive type system for programs.

- $\langle \star_p, \Box_p \rangle$.
  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.

- $\langle \Box_p, \Box_s \rangle$. As a result, there are no programs or datatypes that depend on propositions or proofs.

Table 4.1 summarizes all ten rules, with examples of how they are used, and their interpretation. We assume that $\vdash A : s_1$ and $\vdash b : B : s_2$.

Because there are no rules of the form $\langle \Box_p, \star_s \rangle$, $\lambda\omega_L$ is a conservative extension of the programming language $\lambda\omega_s$, i.e. in $\lambda\omega_L$ we have the same kinds, datatype-constructors and programs as in $\lambda\omega_s$:

**Theorem 4.3.2 (Conservativity of $\lambda\omega_L$ over $\lambda\omega_s$)**

Suppose $\Gamma \vdash a : A$ and $a$ is a program, datatype-constructor or a kind (i.e. $\Gamma \vdash_{\lambda\omega_L} A : *_s$, $\Gamma \vdash_{\lambda\omega_L} A : \Box s$, or $A \equiv \square_s$), and $\Gamma$ is a $\lambda\omega_s$ context.

Then $\Gamma \vdash_{\lambda\omega_s} a : A$.

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

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

**Lemma 4.3.3** If $\Gamma \vdash a : A$ in $\lambda\omega_L(\Box)$, then $\Gamma \vdash |a| : |A|$ in $\lambda C(\Box)$, where $|z|$ is $z$ with $\star$ substituted for $\star_s$ and $\star_p$ and $\Box$ substituted for $\Box_s$ and $\Box_p$. 
Table 4.1: The ten rules of $\lambda\omega_L$, with their use and interpretation

Proof: Induction on the derivation. In fact, it suffices to observe that erasing the subscripts in the specification of $\lambda\omega_L(\delta)$ produces the specification of $\lambda C(\delta)$.

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 4.3.3 it follows immediately from the fact that $\lambda C(\delta)$ is strongly normalizing that $\lambda\omega_L$ and its extension with definitions $\lambda\omega_L \delta$ are:

**Theorem 4.3.4** $\lambda\omega_L(\delta)$ is $\text{SN}_{\bar{\beta}(\delta)}$.

So $\lambda\omega_L$ can be embedded in $\lambda C$. On the other hand, system $\lambda\text{PRE}D\omega$ [Geu93], corresponding with traditional higher-order predicate logic, can be embedded in $\lambda\omega_L$. The main difference between $\lambda\omega_L$ and $\lambda\text{PRE}D\omega$ is that the latter has a very restricted "programming language": there are no rules $(\ast_1, \ast_3)$, $(\square_3, \ast_3)$, and $(\square_4, \square_3)$ in $\lambda\text{PRE}D\omega$.

**Basic propositions and prop-constructors.**

In $\lambda\omega_L$ there are three forms of existential quantification: over a prop-kind $\mathcal{P}$, over a datatype $A$ and over a kind $\mathcal{K}$, i.e. of the form $(\exists x : \mathcal{P}. P)$, $(\exists x : A. P)$ and $(\exists x : \mathcal{K}. P)$, respectively. In Definition 4.2.3 it was shown how in $\lambda\omega_L$ the first form of existential quantification can be defined in terms of $\forall$ and $\rightarrow$. The other two can be defined similarly:
4.3. THE PROGRAMMING LOGIC $\lambda\omega_L$

**Definition 4.3.5** Existential quantification over a datatype ($\exists$) and over a kind $\mathcal{K}$ ($\exists_{\mathcal{K}}$) are defined by

\[
\exists := \lambda A : * , \lambda P : A \to * . \forall R : * . (\forall x : A . (P x) \implies R) \implies R
\]

\[
\exists_{\mathcal{K}} := \lambda P : \mathcal{K} \to * . \forall R : * . (\forall x : \mathcal{K} . (P x) \implies R) \implies R
\]

for all $\mathcal{K}$.

In the first definition we have abstracted over all possible datatypes $A$. In the second definition we cannot abstract over all possible kinds $\mathcal{K}$, 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:

**Definition 4.3.6 (Leibniz’ Equality)** \textit{LEIBNIZ} is the context

\[
= := \lambda A : * , \lambda x, y : A . \forall P : A \to * . (P x) \implies (P y)
\]

\[
: \Pi A : * , A \to A \to *
\]

This context defines Leibniz’ equality for programs. $= \!$ will usually be written infix, with its first argument left implicit, since it is equal to the type of the other arguments. So $(= : A \ a \ b)$ is written as $a = b$ (see Notation 4.5.1 below). From now on, we always assume that \textit{LEIBNIZ} is part of the context, so that we can always use $\! = \!$ to express equality of programs in $\lambda\omega_L$.

Despite its asymmetric definition, the predicate $\! = \!$ has all the required properties:

- It is reflexive, because all $\beta\delta$-convertible programs are Leibniz’ equal, as shown below.

Suppose $\Gamma \vdash a : A$, $\Gamma \vdash b : A$ and $\Gamma \vdash a =_{\beta\delta} b$. Then there is an inhabitant – i.e. a proof – of $a = b$ in context $\Gamma$. As an illustration of a formal proof in the logic, this proof and its type derivation are given below in flag-style notation.

\[
\begin{array}{c}
1 \quad P : A \to *
\end{array}
\]

hyp

\[
\begin{array}{c}
2 \quad p : P a
\end{array}
\]

hyp

\[
\begin{array}{c}
3 \quad p : P b
\end{array}
\]

$\beta\delta$-conv 2

\[
\begin{array}{c}
4 \quad (\lambda p : P a . p) : (P a \implies P b)
\end{array}
\]

$\implies I$ 2-3

\[
\begin{array}{c}
5 \quad (\lambda P : A \to * . \lambda p : P a . p) : (\forall P : A \to * . P a \implies P b)
\end{array}
\]

$\forall I$ 1-4

\[
\begin{array}{c}
6 \quad (\lambda P : A \to * . \lambda p : P a . p) : (a = b)
\end{array}
\]

$\beta\delta$-conv 5

- We can substitute equals for equals in propositions, because

\[
\forall A : * , \forall x, y : A . x = y \implies (\forall Q : A \to * . Q x \implies Q y)
\]
is provable in $\lambda \omega^\delta$, as is shown below. Using this property it can be proved that $=$ is symmetric and transitive.

\[
\begin{array}{c}
A : *_1 \\
x : A \\
y : A \\
p : x = y \\
Q : A \rightarrow *_p \\
q : Q x \\
p : (\forall P : A \rightarrow *_p, P x \rightarrow P y) \\
p Q q : Q y \\
\lambda A : *_1, \lambda x, y : A. \lambda p : x = y. \lambda Q : A \rightarrow *_p, \lambda q : Q x. p q : \\
(\forall A : *_1, \forall x, y : A. x = y \rightarrow (\forall Q : A \rightarrow *_p, Q x \rightarrow Q y)) \\
\end{array}
\]

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 judgments. Instead of $\Gamma \vdash p : P$, we then write $\Gamma \vdash \ldots : P$.

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 used as assumptions. For example, we can assume that two programs $a$ and $b$ of type $A$ are Leibniz’ equal by introducing an assumption $p : a = b$ in the context. We will introduce axioms for Leibniz’ equality that make it an extensional equality. As a consequence, Leibniz’ equality for functions, e.g. $f = g$, 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.

So we want to assume that equality of functions is extensional, i.e. that

\[
f = f' \iff (\forall x : A. f x = f' x) \\
g = g' \iff (\forall X : \bar{K}. g X = g' X)
\]

for all $f, f' : A \rightarrow B$ and $g, g' : (\Pi X : \bar{K} A X)$. 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.

**Definition 4.3.7** \(EXT\) is the set of axioms containing

\[
\begin{align*}
is\_arrow & : \forall A, B : *_1, \forall f, g : A \rightarrow B. (\forall x : A. f x = g x) \rightarrow f = g \\
is\_pi\_K & : \forall A : \bar{K} \rightarrow *_1, \forall f, g : (\Pi X : \bar{K} A X). (\forall X : \bar{K}. f X = g X) \rightarrow f = g
\end{align*}
\]

for all $\bar{K}$.

We lack the necessary quantifications to express axioms in \(EXT\) in a finite number of $\lambda \omega_L$-axioms, since we cannot quantify over all kinds $\bar{K} : \Box_1$ in is\_pi\_K.

Besides Leibniz’ equality on programs ($=$), we also introduce Leibniz’ equality on prop-constructors ($\iff$), to allow easy reasoning with prop-constructors, in particular with propositions and predicates and relations on datatypes. Leibniz’ equality on prop-constructors is defined in a similar way as the equality on programs.
Definition 4.3.8 (Leibniz’ Equality for prop-constructors)

PROP-LEIBNIZ is the set of definitions containing

\[
\leftrightarrow_p := \lambda P, Q : IP. \forall R : IP \rightarrow \ast_p, (R P) \iff (R Q)
\]

\[
\vdash IP \rightarrow IP \rightarrow \ast_p
\]

for every IP : ⊢p.

From now on, we always assume PROP-LEIBNIZ is part of the context.

For example, \(\leftrightarrow \ast_p\) is Leibniz’ equality for propositions, and \(\leftrightarrow \lambda \rightarrow \ast_p\) is Leibniz’ equality for predicates on datatype \(\emptyset\). The equality \(\leftrightarrow_F\) is reflexive, symmetric and transitive, and equal prop-constructors can be substituted for equals in propositions. This is proved similarly to the proofs of these properties for Leibniz’ equality on programs =. Unfortunately, we cannot derive that equivalent propositions \(P\) and \(Q\) — i.e. \(P \iff Q\) and \(Q \iff P\) — are Leibniz’ equal; we need to introduce this property as an axiom. Furthermore, we have to introduce an axiom scheme expressing extensionality of predicates on programs, i.e. if \(\forall A : \ast, P, Q : A \rightarrow IP\) and \(\forall a : A. P a \iff \forall a : A. Q a\) then \(P \iff \lambda a \rightarrowIP Q\), similar to extensionality of functions in Definition 4.3.7 above.

Definition 4.3.9 PROP-EXT is the set of axioms containing

\[
equiv_{\text{prop}} : \forall P, Q : \ast_p, (P \iff Q) \implies (Q \iff P) \implies (P \iff \ast_p Q)
\]

\[
equiv_{\text{pred}} : \forall A : \ast, \forall P, Q : A \rightarrow IP, (\forall a : A. P a \iff \forall a : A. Q a) \implies (P \iff \lambda a \rightarrowIP Q)
\]

for all IP : ⊢p.

Using \(\equiv_{\text{prop}}\), we can derive, for example, False \(\iff \ast_p \neg\) True. Using also \(\equiv_{\text{pred}}\), we can derive \((\lambda a : A. a = a) \iff \lambda \rightarrow \ast_p (\lambda a : A. \text{True})\) for any datatype \(A\). We could also introduce extensionality of prop-constructors with respect to datatypes or prop-constructors, e.g. if \(P, Q : \ast_s \rightarrow \ast_p\) and \(\forall X : \ast_s. P X \iff \ast_p Q X\) then \(P \iff \ast_s \rightarrow \ast_p Q\), but we refrained from doing so, because we do not need these extensionality principles. For \(\iff\) the same conventions hold as for =, see Notation 4.5.1 below.

The distinction between \(\ast_s\) and \(\ast_p\) is very important for introducing axiom \(\equiv_{\text{prop}}\): without this distinction \(\equiv_{\text{prop}}\) would make the system inconsistent [Geu89].

Our set of axioms is completed with the axiom of classic logic.

Definition 4.3.10 AXIOM is the union of EXT and PROP-EXT with the following axiom.

\[
\text{classic} : \forall P : \ast_p, \neg (\neg P) \implies P
\]

4.4 Semantics

When we use \(\lambda_{\omega_L}\) as a programming logic, we are interested in the question whether it is consistent. \(\lambda_{\omega_L}\) is consistent, in the sense that not all propositions are provable. This is equivalent with saying that False, as defined in 4.2.3, is not provable. The definition mechanism does not affect consistency: by Theorem 2.3.11 \(\lambda_{\omega_L}\) is consistent if \(\lambda_{\omega_L}\) 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 Theorem 2.3.10 we can relate consistency of a context in \( \lambda \omega_{L_6} \) and \( \lambda \omega L \): a context is consistent in \( \lambda \omega_{L_6} \) if its \( \delta \)-normal form is consistent in \( \lambda \omega L \).

**Theorem 4.4.1 (Consistency of \( \lambda \omega L \))** \( \text{False} \) is not provable in \( \lambda \omega_{L_6} \) in context LOGIC.

By Theorem 2.3.10 this follows from that \( (\forall P : \ast \cdot P) \) is not provable in \( \lambda \omega L \) in the empty context. There are two ways to prove this theorem. The syntactic way is to use the fact that \( \lambda \omega L \) is SN. From SN it follows that \( \text{False} \) is not provable: if it were, it would have a proof in normal-form, and a simple syntactic argument shows that no inhabitant of \( \text{False} \) can be in normal form. A semantic proof can be given using the PER model of \( \lambda \omega L \) (see [Pol94]). The advantage of the second proof is that it can easily be extended to prove consistency of certain contexts, for example:

**Theorem 4.4.2 (Consistency of AXIOM in \( \lambda \omega L \))**

\( \text{False} \) is not provable in a context containing only axioms from AXIOM.

*Proof:* See [Pol94]. Actually, the proof in [Pol94] does not cater for the axioms equiv prop and equiv pred, but the proof can easily be extended so that it does. \( \square \)

So it is safe to use classical logic in \( \lambda \omega L \), and to assume that equality of programs and equality of prop-constructor constructors are extensional.

### 4.5 Axioms

This section gives all definitions and axioms of \( \lambda \omega L \). They are all type-checked and formatted by Yarrow, except for the axioms and definitions using subscripts (e.g. \( \exists F \)), see Remark 4.5.2 below. In order to improve readability, we use the following typographical conventions for different kinds of variables.

**Convention:**

- Datatype-constructors are written in the teletype-font, and start with a capital, e.g. \texttt{Nat} and \texttt{List}.
- Programs are written in the teletype-font, and start with a small letter, e.g. \texttt{true} and \texttt{reverse}.
- Propositions and other prop-constructors are written in italic, and start with a capital, e.g. \texttt{Ordered}.
- Proofs are written in italic, and start with a small letter or have underscore characters in them, e.g. \textit{classic} and \texttt{Or.sym}.

This convention holds also for the library given in the next section. There we set out the guidelines for the naming of axioms and lemmas, which also apply here for a few axioms.

The axioms are divided into four groups. We start with the axioms for predicate logic (that have been given before, in Sections 4.2 and 4.3), followed by a group for each primitive datatype: booleans, natural numbers and lists.
Predicate logic

This is the context that contains the definitions and axioms for predicate logic:

\[\text{False} \quad ::= \quad \forall P: *_p. P \]
\[\text{True} \quad ::= \quad \forall P: *_p. P \implies P \]
\[\neg \quad ::= \quad \lambda P: *_*p. P \implies \text{False} \]
\[\land \quad ::= \quad \lambda P, Q: *_*p. \forall R: *_*p. (P \implies Q \implies R) \implies R \]
\[\lor \quad ::= \quad \lambda P, Q: *_*p. \forall R: *_*p. (P \implies R) \implies (Q \implies R) \implies R \]
\[\exists \quad ::= \quad \lambda A: *_*s. \lambda P: A \implies *_*p. \forall Q: *_*p. \forall x: A. P x \implies Q \implies Q \]
\[\exists P \quad ::= \quad \lambda P: IP \implies *_*p. \forall R: *_*p. (\forall X: IP. (P X) \implies R) \implies R \]
\[\exists K \quad ::= \quad \lambda P: IK \implies *_*p. \forall R: *_*p. (\forall X: IK. (P X) \implies R) \implies R \]
\[= \quad ::= \quad \lambda A: *_*s. \lambda x, y: A. \forall P: A \implies *_*p. P x \equiv P y \]
\[\text{classic} \quad ::= \quad \forall P: *_*p. \neg \neg P \implies P \]
\[\text{isarrow} \quad ::= \quad \forall A, B: *_*s. \forall f, g: A \implies B. (\forall x: A. f x = g x) \implies f = g \]
\[\text{ispik} \quad ::= \quad \forall A: IK \implies *_*s. \forall f, g: (\forall X: IK. A X). (\forall X: IK. f X = g X) \implies f = g \]
\[\iff \quad ::= \quad \lambda P, Q: *_*p. (P \implies Q) \implies (Q \implies P) \implies (P \iff Q) \]
\[\equiv \quad ::= \quad \lambda A: *_*s, \forall P, Q: A \implies IP. (\forall a: A. P a \iff \forall P. Q a) \iff (P \iff a \rightarrow P. Q) \]

for all \(IK\) and \(IP\).

**Notation 4.5.1** The connectives \(\land, \lor, =\) and \(\iff\) are normally used infix. If they are not used infix, they are enclosed within parentheses. We often write datatype arguments of polymorphic predicates (e.g. Leibniz' equality =) as a subscript. If the subscript of \(=\) or \(\iff\) is obvious from the context, we leave this subscript implicit. So if \(a\) and \(b\) have type \(A\), then \((=)\; A\; a\; b\) and \((=)_A\; a\; b\) and \(a = b\) are all notations for the same term. Finally, we write \(\exists, \exists_P\) and \(\exists_K\) applied to an abstraction in the standard notation, e.g. \(\exists P: *_*p. P \implies \text{False}\) abbreviates \(\exists P: *_*p. \lambda P: *_*p. P \implies \text{False}\).

These conventions increase readability. Furthermore, they can be translated straightforwardly to the standard PTS syntax. For the infix notation, this is just a matter of parsing. Implicit arguments can be determined by the types of the explicit arguments, such as the type of \(a\) and \(b\) in \((a = b)\). In Yarrow these conventions are implemented (see Section 3.2.3), so formulas generated by Yarrow also follow these conventions.
Remark 4.5.2 In Yarrow we cannot enter infinite sets of axioms or definitions, such as \( \text{is\_pi}_{\ast} \) and \( \exists_P \). We avoid this problem by only introducing these axioms or definitions as needed. E.g. if we use \( \text{is\_pi}_{\ast} \) only for \( \ast \equiv \ast_{\ast} \) and \( \ast \equiv \ast_{\ast} \rightarrow \ast_{\ast} \), we just define

\[
\begin{align*}
\text{is\_pi}_{\ast} : & \forall A : \ast_{\ast} \rightarrow \ast_{\ast}, \forall f, g : (\Pi X : \ast_{\ast}, A X). \\
& (\forall X : \ast_{\ast}, f X = g X) \rightarrow f = g \\
\text{is\_pi}_{\ast} \rightarrow \ast_{\ast} : & \forall A : (\ast_{\ast} \rightarrow \ast_{\ast}) \rightarrow \ast_{\ast}, \forall f, g : (\Pi X : \ast_{\ast} \rightarrow \ast_{\ast}, A X). \\
& (\forall X : \ast_{\ast} \rightarrow \ast_{\ast}, f X = g X) \rightarrow f = g.
\end{align*}
\]

This works especially well for kinds, since their structure is so simple and only few kinds are used. Since prop-kinds can contain datatypes (e.g. \( \text{Nat} \rightarrow \ast_p \)), they are far more numerous, and we have to use a more refined approach, namely by defining them as polymorphic as possible in \( \lambda_{\omega} \). E.g. in this thesis we use \( \exists_P \) only for propositions \( (P \equiv \ast_p) \), for predicates \( (P \equiv A \rightarrow \ast_p \) for datatypes \( A \) and for relations \( (P \equiv A \rightarrow B \rightarrow \ast_p \) for datatypes \( A \) and \( B \)), so we just define:

\[
\begin{align*}
\exists_p : & \lambda P : \ast_p \rightarrow \ast_p, \forall R : \ast_p, (\forall X : \ast_p, (P X) \rightarrow R) \rightarrow R \\
: & (\ast_p \rightarrow \ast_p) \rightarrow \ast_p \\
\exists'_{p} : & \lambda A : \ast_p, \lambda P : (A \rightarrow \ast_p) \rightarrow \ast_p, \forall R : \ast_p, (\forall X : A \rightarrow \ast_p, P X) \rightarrow R) \rightarrow R \\
: & \Pi A : \ast_p, ((A \rightarrow \ast_p) \rightarrow \ast_p) \rightarrow \ast_p \\
\exists''_p : & \lambda A, B : \ast_p, \lambda P : (A \rightarrow B \rightarrow \ast_p) \rightarrow \ast_p, \forall R : \ast_p, (\forall X : A \rightarrow B \rightarrow \ast_p, P X) \rightarrow R) \rightarrow R \\
: & \Pi A, B : \ast_p, ((A \rightarrow B \rightarrow \ast_p) \rightarrow \ast_p) \rightarrow \ast_p
\end{align*}
\]

This approach is not ideal, but quite manageable in practice.

Booleans

Now we introduce the booleans as a \( \lambda_{\omega} \) context. In general, the introduction of a datatype consists of the declaration of the datatype and its constructors, a primitive recursor on that datatype and axioms that define the properties of this recursor, an induction lemma and axioms that express that the constructors are different injections.

For the booleans, the constructors are true and false, and the primitive recursor is if.

\[
\begin{align*}
\text{Bool} : & \ast_p \\
\text{true} : & \Pi A : \ast_p, \text{Bool} \rightarrow A \rightarrow A \\
\text{false} : & \Pi A : \ast_p, \text{Bool} \rightarrow A \rightarrow A \\
\text{if\_true} : & \forall A : \ast_p, \forall x, y : A \rightarrow P \rightarrow P \rightarrow (\forall b : \text{Bool}, P b) \\
\text{if\_false} : & \forall A : \ast_p, \forall x, y : A \rightarrow P \rightarrow P \rightarrow (\forall b : \text{Bool}, P b) \\
\text{indbool} : & \forall P : \text{Bool} \rightarrow \ast_p, P \rightarrow P \rightarrow (\forall b : \text{Bool}, P b) \\
\text{true\_is\_false} \rightarrow & \neg (\text{true} = \text{false})
\end{align*}
\]

We leave the first type argument of if implicit.

Natural numbers

The declaration of the natural numbers follows the structure outlined above. However, we do not need axioms which express that 0 and S are different injections, because this is derivable (see Section 4.6).
4.6. LIBRARY

Nat : *,
0 : Nat
S : Nat → Nat
primrecnat : \Pi B:*_. B → (Nat → B → B) → Nat → B
primrecnat.O : \forall B:*_. \forall nv:B. \forall sv:Nat → B → B. primrecnat nv sv 0 = nv
primrecnat.Sm : \forall B:*_. \forall nv:B. \forall sv:Nat → B → B. \forall m:Nat.
primrecnat nv sv (S n) = sv m (primrecnat nv sv m)
indnat : \forall P:Nat → *_. P 0 \Longrightarrow (\forall m:Nat. P m \Longrightarrow P (S m)) \Longrightarrow (\forall m:Nat. P m)

We leave the first type argument of primrecnat implicit.

Lists

For the declaration of polymorphic lists the same notes as for the naturals hold.

List : *_ → *
nil : \Pi A:*_. List A
(): \Pi A:*_. A → List A → List A
primreclist : \Pi A,B:*_. B → (A → List A → B → B) → List A → B
primreclist.nil : \forall A,B:*_. \forall nv:B. \forall sv:A → List A → B → B.
primreclist sv (nil A) = nv
primreclist_cons : \forall A,B:*_. \forall nv:B. \forall sv:A → List A → B → B. \forall head:A. \forall tail:List A.
primreclist sv (head; tail) =
sv head tail (primreclist sv sv tail)
indlist : \forall A:*_. \forall P:List A → *_. P (nil A) \Longrightarrow
(\forall a:A. P a:List A. a as \Longrightarrow P (a; as)) \Longrightarrow
P:List A. a as

We leave the first argument of the infix cons operation (;) implicit, so 0;(nil Nat) is
written for (; Nat 0 (nil Nat)). The first two arguments of primreclist are left implicit.

4.6 Library

In this section we present a small library of definitions and lemmas in the form of a \( \lambda \omega \_ \) context, in order to treat some more advanced examples of programs and their correctness proofs. It gives an idea how much work has to be done before interesting examples of programs
and proofs can be given.

Again, this library is checked by Yarrow. Every proof term abbreviated to "..." has been
constructed in Yarrow. Every lemma that is used in such a proof has to be referred to by
name. Since there are so many lemmas in the library, it is important that lemmas have clear
names, which are easy to remember. Therefore we use the following guideline concerning the
naming of lemmas.
CONVENTION:

1. If the lemma is characterized by its conclusion, it is named after the conclusion. The basic rule is to replace in the textual representation of the conclusion all spaces (applications) by underscore characters to obtain the name. E.g. \( O.Le.m : \forall m : \text{Nat}. \ 0 \leq m \). We replace infix operators (e.g. \( \leq \)) by a proper name (e.g. \( Le \)).

2. If the lemma cannot be characterized by its conclusion, but an hypothesis is informative enough, the lemma is named after this hypothesis, and is suffixed with an underscore character. E.g. \( Sm.is.Sm._n : \forall m, n : \text{Nat}. \ S \ m = S \ n \rightarrow m = n \).

3. If rules 1 and 2 both do not work, we name the lemma after an assumption and the conclusion, separated by double underscore characters. E.g. \( m.Lt.n.Sm.Le.n : \forall m, n : \text{Nat}. \ m < n \rightarrow S \ m \leq n \).

4. We often consider negation as implication of falsehood, so rule 2 applies. E.g. \( Sm.is._O._n : \forall m : \text{Nat}. \ \neg \neg(S \ m = 0) \).

5. When the lemma is named after an equality, it often suffices to use only the left hand side. E.g. \( \text{pred.Sm} : \forall m : \text{Nat}. \ \text{pred}(S \ m) = m \).

6. Often we leave non-informative variable names out. E.g. \( \text{if.true} : \forall a : *, \forall x, y : A. \ \text{if true} x y = x \).

7. The conventions set out above are guidelines, and we use them mainly when there is not a more abstract name. E.g. \( Le._\text{refl} : \forall m : \text{Nat}. \ n \leq m \).

This convention is already used in Section 4.5, but only here it becomes really important.

This section is subdivided into four parts, concerning predicate logic, the booleans, the natural numbers and lists. In general, the comments that we give in these parts also apply to the parts thereafter.

PREDICATE LOGIC

Here we give some standard lemmas about classical predicate logic with Leibniz’ equality.

\hspace{1cm} \text{ex.falso} \quad \equiv \quad \forall P : *_{P}. \ \text{False} \rightarrow P

\hspace{1cm} \text{Not.i} \quad \equiv \quad \forall P : *_{P}. \ (P \rightarrow \text{False}) \rightarrow \neg P

\hspace{1cm} \text{Not.e} \quad \equiv \quad \forall P : *_{P}. \ \neg P \rightarrow P \rightarrow \text{False}

\hspace{1cm} \text{true.True} \quad \equiv \quad \forall a

\hspace{1cm} \text{And.i} \quad \equiv \quad \forall P, Q : *_{P}. \ P \rightarrow Q \rightarrow P \land Q

\hspace{1cm} \text{And.el} \quad \equiv \quad \forall P, Q : *_{P}. \ P \land Q \rightarrow P

\hspace{1cm} \text{And.er} \quad \equiv \quad \forall P, Q : *_{P}. \ P \land Q \rightarrow Q

\hspace{1cm} \text{And.sym} \quad \equiv \quad \forall P, Q : *_{P}. \ P \land Q \rightarrow Q \land P

\hspace{1cm} \text{And.assoc} \quad \equiv \quad \forall P, Q, R : *_{P}. \ P \land (Q \land R) \rightarrow (P \land Q) \land R

\hspace{1cm} \text{Or.IL} \quad \equiv \quad \forall P, Q : *_{P}. \ P \rightarrow P \lor Q

\hspace{1cm} \text{Or.ir} \quad \equiv \quad \forall P, Q : *_{P}. \ Q \rightarrow P \lor Q

\hspace{1cm} \text{Or.e} \quad \equiv \quad \forall P, Q, R : *_{P}. \ (P \lor Q) \rightarrow (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R

\hspace{1cm} \text{Or.sym} \quad \equiv \quad \forall P, Q : *_{P}. \ P \lor Q \rightarrow Q \lor P

\hspace{1cm} \text{Or.assoc} \quad \equiv \quad \forall P, Q, R : *_{P}. \ (P \lor (Q \lor R)) \rightarrow (P \lor Q) \lor R
4.6. LIBRARY

\[ Ex_i \quad := \quad \forall A: \ast_p, \forall x:A. \forall P:A \to \ast_p. P x \implies \exists P \]

\[ Ex_e \quad := \quad \forall A: \ast_p, \forall P:A \to \ast_p. \forall R: \ast_p. \exists P \implies (\forall x:A. P x \implies R) \implies R \]

\[ is_refl \quad := \quad \forall A: \ast_p, \forall x:A. x = x \]

\[ is_elim_r \quad := \quad \forall A: \ast_p, \forall x, y:A. \forall P:A \to \ast_p. x = y \implies P x \implies P y \]

\[ is_elim_l \quad := \quad \forall A: \ast_p, \forall x, y:A. \forall P:A \to \ast_p. x = y \implies P y \implies P x \]

\[ is_sym \quad := \quad \forall A: \ast_p, \forall x, y:A. x = y \implies y = x \]

\[ is_trans \quad := \quad \forall A: \ast_p, \forall x, y, z:A. x = y \implies y = z \implies x = z \]

\[ exm \quad := \quad \forall P: \ast_p. P \lor \neg P \]

\[ contrapos \quad := \quad \forall P, Q: \ast_p. (\neg P \implies Q) \implies \neg Q \implies P \]

\[ impl_Or \quad := \quad \forall P, Q: \ast_p. (\neg P \implies Q) \implies P \lor Q \]

\[ add_negat \quad := \quad \forall P: \ast_p. (\neg P \implies P) \implies P \]

Many of these lemmas are used for Yarrow’s special tactics (see Section 3.2.1 on page 50).

Booleans

\[ false_is_true_ \quad := \quad \ldots : \neg (false = true) \]

\[ exh_bool \quad := \quad \forall b: \text{Bool}. b = true \lor b = false \]

Apart from the axiom true_is_false_, we also have the lemma false_is_true_, because it is awkward in a proof-assistant to prove this lemma from the axiom every time we need it.

The lemma exh_bool expresses the exhaustion property of booleans.

Naturals

Ordinary mathematics textbooks largely consist of definitions, followed by lemmas (theorems) and proofs. In a \(\lambda_w\) context, this is exactly the same. After a definition, we give a few simple lemmas that follow directly from the definition, followed by more complicated lemmas and theorems. We omit most of the proofs, since they are quite trivial.

Since the \(\lambda_w\) framework is quite bare, often definitions have to be given in an awkward way, i.e. using a primitive recursive to define functions on datatypes, or using higher-order quantification to give an inductive definition of a relation. In the basic lemmas that follow these definitions are much easier to read and give a better understanding of these definitions. Indeed, the lemmas completely determine the function or relation most of the time.

A more elegant way to define functions or relations inductively, is to extend the calculus with inductive types (e.g. the Calculus of Inductive Constructions [PPM90], used in Coq). However the definition and meta-theory of such a calculus is much more complicated, and since our encoding of objects does not need inductive types, we ignore them.

The library on natural numbers starts with the definition of the predecessor function.

\[ \text{pred} \quad := \quad \text{primrecnat0} (\lambda m, n:\text{Nat}. m) \]

\[ \quad \quad \quad : \text{Nat} \to \text{Nat} \]

\[ \text{pred}_0 \quad := \quad \ldots : \text{pred} 0 = 0 \]

\[ \text{pred}_{S m} \quad := \quad \ldots : \forall m: \text{Nat}. \text{pred}(S m) = m \]
Using the predecessor and the axiom $true_{is}.false_{.}$ we prove the following lemmas, which say that $0$ and $S$ are different injections.

\[
\begin{align*}
O_{is}.Sm_{} & := \ldots : \forall m: \text{Nat}. \rightarrow (0 = S m) \\
Sm_{is}.O_{} & := \ldots : \forall m: \text{Nat}. \rightarrow (S m = 0) \\
Sm_{is}.Sn_{} & := \ldots : \forall m, n: \text{Nat}. S m = S n \implies m = n \\
m_{is}.Sm_{} & := \ldots : \forall m: \text{Nat}. \rightarrow (m = S m)
\end{align*}
\]

Some functions over naturals can be simply defined as an iteration, and do not need the full power of primrecnat. Therefore we define the iterator over natural numbers.

\[
\begin{align*}
\text{internat} & := \lambda B: \ast_p, \lambda \text{nv}: B. \lambda \text{sv}: B \rightarrow B. \text{primrecnat} \text{nv} (\lambda \text{dummy}: \text{Nat}. \text{sv}) \\
& \quad : \Pi B: \ast_p, B \rightarrow (B \rightarrow B) \rightarrow \text{Nat} \rightarrow B \\
\text{internat}_O & := \ldots : \forall B: \ast_p, \forall \text{nv}: B. \forall \text{sv}: B \rightarrow B. \text{internat} \text{nv} 0 = \text{nv} \\
\text{internat}_S m & := \ldots : \forall B: \ast_p, \forall \text{nv}: B. \forall \text{sv}: B \rightarrow B. \forall m: \text{Nat}. \forall n: \text{Nat}. \text{internat} \text{nv} \text{sv} (S m) = \text{sv} (\text{internat} \text{nv} \text{sv} m)
\end{align*}
\]

Next we introduce a lemma which helps us prove lemmas about $\leq$, which will be defined below.

\[
\begin{align*}
nat\_double\_ind & := \ldots : \forall R: \text{Nat} \rightarrow \text{Nat} \rightarrow \ast_p. \\
& \quad (\forall m: \text{Nat}. R 0 m) \implies \\
& \quad (\forall m: \text{Nat}. R (S m) 0) \implies \\
& \quad (\forall m, n: \text{Nat}. R m n \implies R (S m) (S n)) \implies \\
& \quad (\forall m, n: \text{Nat}. R m n)
\end{align*}
\]

The relation $(m \leq)$ is the smallest relation closed under $m \leq m$ and $m \leq p \implies m \leq S p$. In more formal terms, $(m \leq)$ is the intersection of all predicates $R: \text{Nat} \rightarrow \ast_p$ with

- $R m$ and
- $R p \implies R (S p)$ for all $p$.

So $m \leq n$ holds iff $R n$ for all these predicates. This formulation is easily converted into type theory.

\[
\begin{align*}
(\leq) & := \lambda m, n: \text{Nat}. \forall R: \text{Nat} \rightarrow \ast_p. R m \implies (\forall p: \text{Nat}. R p \implies R (S p)) \implies R n \\
& \quad : \text{Nat} \rightarrow \text{Nat} \rightarrow \ast_p \\
\text{Le}.\_\text{refl} & := \ldots : \forall m: \text{Nat}. m \leq m \\
\text{m}.\_\text{Le}.\text{Sn} & := \ldots : \forall m, n: \text{Nat}. m \leq n \implies m \leq S n \\
\text{Le}.\_\text{ind} & := \ldots : \forall m: \text{Nat}. \forall P: \text{Nat} \rightarrow \ast_p. P m \implies \\
& \quad (\forall n: \text{Nat}. m \leq n \implies P n \implies P (S n)) \implies \\
& \quad (\forall n: \text{Nat}. m \leq n \implies P n) \\
\text{O}.\_\text{Le}.\_m & := \ldots : \forall m: \text{Nat}. 0 \leq m \\
\text{Sm}.\_\text{Le}.\_\text{Sn} & := \ldots : \forall m, n: \text{Nat}. m \leq n \implies S m \leq S n \\
\text{Le}.\_\text{trans} & := \ldots : \forall m, n, p: \text{Nat}. m \leq n \implies n \leq p \implies m \leq p \\
\text{m}.\_\text{Le}.\_\text{Sm} & := \ldots : \forall m: \text{Nat}. m \leq S m \\
\text{predm}.\_\text{Le}.\_m & := \ldots : \forall m: \text{Nat}. \text{pred} m \leq m \\
\text{Sm}.\_\text{Le}.\_\text{Sn} & := \ldots : \forall m, n: \text{Nat}. S m \leq S n \implies m \leq n \\
\text{Sm}.\_\text{Le}.\_\text{n} & := \ldots : \forall m, n: \text{Nat}. S m \leq n \implies m \leq n
\end{align*}
\]
The relation $<$ is defined in terms of $\leq$, so that we can prove properties of $<$ using the properties of $\leq$.

\[
(\triangleleft) := \quad \lambda m, n : \text{Nat}. \quad \text{S } m \triangleleft n : \text{Nat} \rightarrow \text{Nat} \rightarrow *_p
\]

\[
m_\text{Lt}. n_\text{Sm}. \text{Le}. n := \ldots : \forall m, n : \text{Nat}. m \triangleleft n \implies \text{S } m \leq n
\]

\[
\text{Sm}. \text{Le}. m_\text{Lt}. n := \ldots : \forall m, n : \text{Nat}. \text{S } m \leq n \implies m \triangleleft n
\]

\[
m_\text{Lt}. S m := \ldots : \forall m : \text{Nat}. m < S m
\]

\[
m_\text{Lt}. Sn := \ldots : \forall m, n : \text{Nat}. m < n \implies m < S n
\]

\[
O_\text{Lt}. S m := \ldots : \forall m : \text{Nat}. \text{S } m < S m
\]

\[
S_\text{Lt}. S n := \ldots : \forall m, n : \text{Nat}. m < n \implies \text{S } m < S n
\]

\[
m_\text{Lt}. O_\text{Lt}. m := \ldots : \forall m : \text{Nat}. \text{S } m \leq S m
\]

\[
m_\text{Lt}. m_\text{Lt}. S n := \ldots : \forall m, n : \text{Nat}. m < S n \implies m < n
\]

\[
m_\text{Lt}. m_\text{Lt}. m := \ldots : \forall m : \text{Nat}. \text{S } m < m
\]

\[
O_\text{Lt}. m_\text{Lt}. m := \ldots : \forall m : \text{Nat}. \text{O } m \implies 0 < m
\]

\[
\text{Lt}. \text{n} := \ldots : \forall m, n : \text{Nat}. m < n \implies n (m = n)
\]

\[
\text{Lt}. \text{trans} := \ldots : \forall m, n, p : \text{Nat}. m < n \implies n < p \implies m < p
\]

\[
\text{Lt}. \text{Le}. \text{trans} := \ldots : \forall m, n, p : \text{Nat}. m < n \implies n < p \implies m < p
\]

\[
\text{Le}. \text{Lt}. \text{trans} := \ldots : \forall m, n, p : \text{Nat}. m < n \implies n < p \implies m < p
\]

\[
\text{Le}. \text{Lt}. \text{Or}. \text{is} := \ldots : \forall m, n : \text{Nat}. m < n \implies m < n \lor m = n
\]

\[
\text{Le}. \text{Or}. \text{Lt} := \ldots : \forall m, n : \text{Nat}. m < n \lor n < m
\]

\[
\text{Le}. \text{Not}. \text{Lt} := \ldots : \forall m, n : \text{Nat}. m < n \implies m \triangleleft 0 (n < m)
\]

\[
\text{Not}. \text{Lt}. \text{Le} := \ldots : \forall m, n : \text{Nat}. \text{S } m \leq n \implies m \triangleleft n
\]

The relation $\leq$ given above is a prop-constructor; given two values a and b, a $\leq$ b is a proposition. We cannot use it in a program, since programs may not depend on propositions. Therefore we introduce the program $\leq$, which given two values returns a boolean, which may be used in programs. The definition of $\leq$ is preceded by the definition of a test for zero. The lemmas for $\leq$ just associate it with $\leq$.

\[
iszero := \quad \text{internat true} \quad \lambda b : \text{Bool}. \text{false}
\]

\[
iszero \_o := \ldots : \text{iszero } O = \text{true}
\]

\[
iszero \_s m := \ldots : \forall m : \text{Nat}. \text{iszero } (\text{S } m) = \text{false}
\]

\[
\leq := \quad \lambda m, n : \text{Nat}. \text{internat } (\lambda m : \text{Nat}. \text{iszero } m)
\]

\[
(\lambda \text{le.Pn} : \text{Nat} \rightarrow \text{Bool}. \lambda m : \text{Nat}. \text{le.Pn} (\text{pred } m)) \quad n m
\]

\[
: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool}
\]
\textbf{CHAPTER 4. }\lambda\omega_1, \text{ A PROGRAMMING LOGIC}

\begin{align*}
\text{leq}_\text{O}_m & := \ldots : \forall m: \text{Nat}. \text{leq } 0 m = \text{true} \\
\text{leq}_\text{Sm}_0 & := \ldots : \forall m: \text{Nat}. \text{leq } (S m) 0 = \text{false} \\
\text{leq}_\text{Sm}_n & := \ldots : \forall m, n: \text{Nat}. \text{leq } (S m) (S n) = \text{leq } m n \\
\text{leq}_\text{true}_\text{Le} & := \ldots : \forall m, n: \text{Nat}. \text{leq } m n = \text{true} \Rightarrow m \leq n \\
\text{Le}_\text{leq}_\text{true} & := \ldots : \forall m, n: \text{Nat}. m \leq n \Rightarrow \text{leq } m n = \text{true} \\
\text{Gl}_\text{leq}_\text{false} & := \ldots : \forall m, n: \text{Nat}. n < m \Rightarrow \text{leq } m n = \text{false} \\
\text{leq}_\text{false}_\text{Gl} & := \ldots : \forall m, n: \text{Nat}. \text{leq } m n = \text{false} \Rightarrow n < m \\
\end{align*}

From this piece of the library, we give the proof of \textit{Sm}_\text{Le}, \text{Sn}_n in "flag"-style.

\begin{verbatim}
1 \begin{array}{ll}
  m: \text{Nat} & \text{hyp} \\
  n: \text{Nat} & \\
  S m \leq S n & \\
  \end{array}
2 \begin{array}{ll}
  \text{pred } (S n) = n & \forall E \text{ pred }_\text{Sm} \\
  \text{pred } (S m) = m & \forall E \text{ pred }_\text{Sm} \\
  \forall R: \text{Nat} \to *_p. R (S m) \Rightarrow (\text{Vp: Nat. R p} \Rightarrow R (S p)) \Rightarrow R (S n) & \beta\delta\text{-conv 3} \\
  \text{pred } (S m) \leq \text{pred } (S m) & \forall E \text{ Le}_\text{refl} \\
  \end{array}
3 \begin{array}{ll}
  p: \text{Nat} & \\
  \text{pred } (S m) \leq \text{pred } p & \text{hyp} \\
  \end{array}
4 \begin{array}{ll}
  \text{pred } (S p) = p & \forall E \text{ pred }_\text{Sm} \\
  \text{pred } p \leq p & \forall E \text{ pred }_\text{Sm} \\
  \text{pred } p \leq \text{pred } (S p) & \Rightarrow 10,11 \\
  \text{pred } (S m) \leq \text{pred } (S p) & \forall E \text{ Le}_\text{trans},9,12 \\
  \forall p: \text{Nat}. \text{pred } (S m) \leq \text{pred } p \Rightarrow \text{pred } (S m) \leq \text{pred } (S p) & \forall I 8-13 \\
  \text{pred } (S m) \leq \text{pred } (S n) & \forall E 6,7,14 \\
  m \leq \text{pred } (S n) & \Rightarrow 5,15 \\
  m \leq n & \Rightarrow 4,16 \\
  \forall m, n: \text{Nat. } S m \leq S n \Rightarrow m \leq n & \forall I 1-17
\end{verbatim}

**Lists**

Just as for the natural numbers, we define iteration as a simple form of recursion. This function \textit{iterlist} is known as \texttt{foldr} in functional languages.

\begin{align*}
\text{iterlist} & := \lambda A, B:\ast, \lambda \text{nv}: B. \lambda s v: A \to B. \text{ primrec list } (\lambda a: A. \lambda \text{dummy: List } A. s v a) \\
                  & : \Pi A, B:\ast. B \to (A \to B \to B) \to \text{ List } A \to B \\
\text{iterlist}_\text{nil} & := \ldots : \forall A, B:\ast, \forall \text{nv}: B. \forall s v: A \to B. \text{ iterlist } (\text{nv sv } (\text{nil } A)) = \text{nv} \\
\text{iterlist}_\text{cons} & := \ldots : \forall A, B:\ast, \forall \text{nv}: B. \forall s v: A \to B. \forall h e a d: A. \forall t a i l: \text{ List } A. \\
                             & \text{ iterlist } (\text{nv sv } \text{ (head, tail)}) = \\
                             & \text{ sv head (iterlist } (\text{nv sv tail})
\end{align*}
The functions head and tail which are partial in functional languages, have to be made total in \( \lambda w \). For head we achieve this by having a default argument as parameter.

\[
\text{tail} \quad := \quad \lambda A: * . \text{primereclist} (\text{nil}\ A) \ (\lambda \text{hd}: A . \text{tl}, p : \text{List } A . \text{tl}) \\
\quad : \quad \Pi A : * . \text{List } A \rightarrow \text{List } A
\]

\[
\text{tail}\_\text{nil} \quad := \quad \forall A : * . \text{tail} (\text{nil}\ A) = \text{nil}\ A
\]

\[
\text{tail}\_\text{cons} \quad := \quad \forall A : * . \forall A : \text{List } A . \text{tail} (a ; 1) = 1
\]

\[
\text{head} \quad := \quad \lambda A : * . \lambda A : \text{List } A . \text{iterlist} a (\lambda a : , \text{prev}: A . a') \\
\quad : \quad \Pi A : * . A \rightarrow \text{List } A \rightarrow A
\]

\[
\text{head}\_\text{nil} \quad := \quad \forall A : * . \forall A : \text{head} a (\text{nil}\ A) = a
\]

\[
\text{head}\_\text{cons} \quad := \quad \forall A : * . \forall A : b : A . \forall A : \text{List } A . \text{head} a (b ; 1) = b
\]

Using the head and tail functions and the axiom true \_ false \_ we prove the following lemmas, which say that nil and \( \text{cons} \) are different injections.

\[
\text{nil}\_\text{is}\_\text{cons}_\_ \quad := \quad \forall A : * . \forall A : \text{List } A . \forall A : A . \rightarrow (\text{nil}\ A = a ; 1)
\]

\[
\text{cons}\_\text{is}\_\text{nil}_\_ \quad := \quad \forall A : * . \forall A : \text{List } A . \forall A : A . \rightarrow (a ; 1 = \text{nil}\ A)
\]

\[
\text{cons}\_\text{is}\_\text{cons}_\_ \quad := \quad \forall A : * . \forall A : 11, 12 : \text{List } A . \forall A : a, a' : A . \\
\quad \rightarrow a = 11 = 12 = a = a' = 11 = 12
\]

Next we introduce concatenation of lists, denoted by the infix symbol \( + \).

\[
(+) \quad := \quad \lambda A : * . \lambda A : \text{List } A . \text{iterlist} a (\rightarrow)
\]

\[
(\lambda \text{hd} : A . \lambda \text{concat}\_\text{tail} : \text{List } A . \text{hd} ; \text{concat} \_\text{tail} ) \rightarrow
\quad : \quad \Pi A : * . \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A
\]

\[
\text{nil}\_\text{concat} \quad := \quad \forall A : * . \forall A : \text{List } A . \text{nil} A = A = A
\]

\[
\text{cons}\_\text{concat} \quad := \quad \forall A : * . \forall A : \text{List } A . \forall A : A . \rightarrow (a ; 1 = \text{nil}\ A)
\]

\[
\text{concat}\_\text{assoc} \quad := \quad \forall A : * . \forall A : \forall A : \text{List } A . \forall A : A (a ; 1) = a = a (a ; 1)
\]

\[
\text{concat}\_\text{nil} \quad := \quad \forall A : * . \forall A : \forall A : \text{List } A . \forall A : A . \forall A : \text{nil} A (a ; 1)
\]

\[
\text{singleton} \quad := \quad \lambda A : * , \lambda A : \text{List } A . \text{nil} A
\quad : \quad \Pi A : * . A \rightarrow \text{List } A
\]

The relation \( (\text{Elem}_A \ a) \) is the smallest relation closed under \( (\text{Elem}_A \ a (a ; m)) \) and \( (\text{Elem}_A \ a m \rightarrow (\text{Elem}_A \ a (b ; m))) \). We write the datatype argument of \( \text{Elem} \) subscript.

\[
\text{Elem} \quad := \quad \lambda A : * . \lambda A : \text{List } A . \forall P : \text{List } A \rightarrow * _p .
\quad (\forall m : \text{List } A . P (a ; m) \rightarrow P (a ; m)) \rightarrow
\quad (\forall m : \text{List } A . \forall a : A . P m \rightarrow P (b ; m)) \rightarrow
\quad P 1
\quad : \quad \Pi A : * . A \rightarrow \text{List } A \rightarrow * _p
\]

\[
\text{Elem} \_a . \text{a } \_\text{cons} \quad := \quad \forall A : * . \forall A : \forall A : \text{List } A . \text{Elem}_A \ a (a ; 1)
\]

\[
\text{Elem} \_a . \text{b } \_\text{cons} \quad := \quad \forall A : * . \forall A : b : A . \forall A : \text{List } A . \text{Elem}_A \ a 1 \rightarrow \text{Elem}_A \ a (b ; 1)
\]

\[
\text{Elem} \_\text{strong } \_\text{ind} \quad := \quad \forall A : * . \forall A : \forall A : \text{List } A . \text{Elem}_A \ a 1 \rightarrow
\quad (\forall P : \text{List } A \rightarrow * _p . (\forall m : \text{List } A . P (a ; m)) \rightarrow
\quad (\forall m : \text{List } A . \forall a : A . \text{Elem}_A \ a m \rightarrow P m \rightarrow P (b ; m)) \rightarrow
\quad P 1)
\]
\[ \text{Elem\_exh} := \ldots : \forall A:*, \forall a:A. \forall l:\text{List}\ A. \text{Elem}\ A\ a \leftarrow l \rightarrow \]
\[ (\exists m:\text{List}\ A. l = a;m) \lor \]
\[ (\exists m:\text{List}\ A. \exists b:A. l = b;m \land \text{Elem}\ A\ a;m) \]
\[ \]
\[ \text{Elem\_nil} := \ldots : \forall A:*, \forall a:A. \neg (\text{Elem}\ A\ a (\text{nil}\ A)) \]
\[ \text{Elem\_cons} := \ldots : \forall A:*, \forall a,b:A. \forall l:\text{List}\ A. \text{Elem}\ A\ a (b;l) \rightarrow a = b \lor \text{Elem}\ A\ a \]

The library presented in this section is relatively long, even though it is not complete in even the most modest sense (e.g. we did not define addition). This is a consequence of having to start from scratch. In the next section we show that using this library, the development of a program with its correctness proof can be relatively short.

### 4.7 Example: Sorting

Let us consider a simple example: an algorithm that sorts a list of natural numbers. In order to formulate the specification for this algorithm, we first need to define what it means for a list to be ordered. The relation \textit{Ordered} is the smallest predicate closed under:

- \textit{Ordered} \textit{nil}

- \textit{Ordered} \textit{m} \land (\forall b: \text{Nat}. \text{Elem}\ b\ m \rightarrow a \leq b) \rightarrow \textit{Ordered}\ (a;m)

This particular formulation of \textit{Ordered} was chosen because it follows the structure of lists, which make the proofs easier.

In \(\Lambda\omega_L\), \textit{Ordered} is encoded as follows:

\[
\text{Ordered} := \lambda l:\text{List}\ \text{Nat}. \forall P:\text{List}\ \text{Nat} \rightarrow *_p,
\]
\[ P (\text{nil}\ \text{Nat}) \rightarrow \]
\[ (\forall a: \text{Nat}. \forall m:\text{List}\ \text{Nat}. P\ m \rightarrow)
\]
\[ (\forall b: \text{Nat}. \text{Elem}\ \text{Nat}\ b\ m \rightarrow a \leq b) \rightarrow P\ (a;m)) \rightarrow
\]

\[
P_1 := \text{List}\ \text{Nat} \rightarrow *_p\]

\[\text{Ordered\_nil} := \ldots : \text{Ordered}\ (\text{nil}\ \text{Nat})\]

\[\text{Ordered\_cons} := \ldots : \forall a: \text{Nat}. \forall m:\text{List}\ \text{Nat}. \text{Ordered}\ m \rightarrow
\]
\[ (\forall b: \text{Nat}. \text{Elem}\ \text{Nat}\ b\ m \rightarrow a \leq b) \rightarrow \]
\[ \text{Ordered}\ (a;m)\]

\[\text{Ordered\_singleton} := \ldots : \forall m: \text{Nat}. \text{Ordered}\ (\text{singleton} m)\]

\[\text{Ordered\_exh} := \ldots : \forall l:\text{List}\ \text{Nat}. \text{Ordered}\ l \rightarrow
\]
\[ l = \text{nil}\ \text{Nat} \lor
\]
\[ (\exists a: \text{Nat}. \exists m:\text{List}\ \text{Nat}. l = a;m \land
\]
\[ (\forall b: \text{Nat}. \text{Elem}\ \text{Nat}\ b\ m \rightarrow a \leq b) \land \text{Ordered}\ m)\]

\[\text{Ordered\_cons} := \ldots : \forall a: \text{Nat}. \forall m:\text{List}\ \text{Nat}. \text{Ordered}\ (a;m) \rightarrow
\]
\[ (\forall b: \text{Nat}. \text{Elem}\ \text{Nat}\ b\ m \rightarrow a \leq b) \land \text{Ordered}\ m\]

Furthermore, a sorting algorithm should deliver a permutation of the original list. We define the relation \textit{Perm} as the smallest reflexive and transitive relation closed under swapping two
4.7. EXAMPLE: SORTING

consecutive elements in a list. In $\lambda \omega_L$, we encode this as follows:

\[
\text{Perm} \quad := \quad \lambda \text{A} : \ast, \lambda \text{as}, \text{bs} : \text{List A.} \forall P : \text{List A} \to \text{List A} \to \ast_p
\]
\[
(\forall \text{x}s : \text{List A.} \ P \text{x}s \text{x}s) \implies
(\forall \text{x}s, \text{ys}, \text{zs} : \text{List A.} \ P \text{x}s \text{ys} \implies P \text{ys} \text{zs} \implies P \text{xs} \text{zs}) \implies
(\forall \text{x}s, \text{ys} : \text{List A.} \ \forall \text{x}, \text{y} : \text{A.} \ P (\text{xs}++(\text{x}; \text{ys})) (\text{xs}++(\text{y}; \text{x}; \text{ys}))) \implies
P \text{ as bs}
\]
\[
\Pi \text{A} : \ast, \text{List A} \to \text{List A} \to \ast_p
\]
\[
\text{Perm_refl} \quad := \quad \forall \text{A} : \ast, \forall \text{x}s : \text{List A.} \ \text{Perm}_A \text{x}s \text{x}s
\]
\[
\text{Perm_trans} \quad := \quad \forall \text{A} : \ast, \forall \text{x}s, \text{ys}, \text{zs} : \text{List A.}
\]
\[
P\text{erm}_A \text{x}s \text{ys} \implies \text{Perm}_A \text{ys} \text{zs} \implies \text{Perm}_A \text{x}s \text{zs}
\]
\[
\text{Perm_swap} \quad := \quad \forall \text{A} : \ast, \forall \text{x}s, \text{ys} : \text{List A.} \ \forall \text{x}, \text{y} : \text{A.}
\]
\[
P\text{erm}_A (\text{x}s++(\text{x}; \text{ys})) (\text{xs}++(\text{y}; \text{x}; \text{ys}))
\]
\[
\text{Perm_cons} \quad := \quad \forall \text{A} : \ast, \forall \text{A} : \text{A}, \forall \text{l}, \text{m} : \text{List A.} \ \text{Perm}_A \text{l} \text{m} \implies \text{Perm}_A (\text{a}; \text{l}) (\text{a}; \text{m})
\]
\[
\text{Perm_sym} \quad := \quad \forall \text{A} : \ast, \forall \text{x}s, \text{ys} : \text{List A.} \ \text{Perm}_A \text{x}s \text{ys} \implies \text{Perm}_A \text{ys} \text{x}s
\]

Another option is to define the transitive and reflexive closure of an arbitrary relation and apply this to the relation \text{Swap}, which expresses that two consecutive elements in a list are interchanged. For reasons of brevity, we have chosen a more direct approach here.

Before we can define the sorting algorithm, we define an auxiliary function, which inserts one element in a sorted list, so that the result is again sorted.

\[
\text{insert} \quad := \quad \lambda \text{n} : \text{Nat. primereclist (singleton n)}
\]
\[
(\lambda \text{head} : \text{Nat.}, \lambda \text{tail}, \text{insert.tail} : \text{List Nat.}
\]
\[
\text{if} (\text{leq n head}) (\text{n;} \text{head}; \text{tail}) (\text{head}; \text{insert.tail})
\]
\[
\text{Nat} \to \text{List Nat} \to \text{List Nat}
\]
\[
\text{insert.nil} \quad := \quad \forall \text{m} : \text{Nat. insert m (nil Nat) = singleton m}
\]
\[
\text{leq.insert} \quad := \quad \forall \text{m}, \text{n} : \text{Nat.} \ \forall \text{l} : \text{List Nat.} \ \text{m} \leq \text{n} \implies \text{insert m (n; l) = m; n; l}
\]
\[
\text{gt.insert} \quad := \quad \forall \text{m}, \text{n} : \text{Nat.} \ \forall \text{l} : \text{List Nat.} \ \text{m} < \text{n} \implies \text{insert m (n; l) = n; insert m l}
\]
\[
\text{Elem.insert} \quad := \quad \forall \text{m}, \text{n} : \text{Nat.} \ \forall \text{ns} : \text{List Nat.}
\]
\[
\text{Elem}_\text{Nat} \text{m (insert n ns) = m} \vee \text{Elem}_\text{Nat} \text{m ns}
\]
\[
\text{Ordered.insert} \quad := \quad \forall \text{m} : \text{Nat.} \ \forall \text{l} : \text{List Nat.} \ \text{Ordered l} \implies \text{Ordered (insert m l)}
\]
\[
\text{Perm.insert} \quad := \quad \forall \text{m} : \text{Nat.} \ \forall \text{l} : \text{List Nat.} \ \text{Perm}_\text{Nat} \text{ (insert m l) (m; l)}
\]

We have presented the proof of \text{Ordered.insert} in flag-style in Figure 3.12 on page 69.

Finally, we can straightforwardly define the sorting algorithm — insertion sort — and prove it correct.

\[
\text{sort} \quad := \quad \text{primereclist (nil Nat)}
\]
\[
(\lambda \text{head} : \text{Nat.}, \lambda \text{tail}, \text{sort.tail} : \text{List Nat.}
\]
\[
\text{insert head sort.tail}
\]
\[
\text{: List Nat} \to \text{List Nat}
\]
\[
\text{sort.nil} \quad := \quad \text{sort (nil Nat) = nil Nat}
\]
\[
\text{sort.cons} \quad := \quad \forall \text{m} : \text{Nat.} \ \forall \text{l} : \text{List Nat.} \ \text{sort (m; l) = insert m (sort l)}
\]
\[
\text{Ordered.sort} \quad := \quad \forall \text{l} : \text{List Nat.} \ \text{Ordered (sort l)}
\]
\[
\text{Perm.sort} \quad := \quad \forall \text{l} : \text{List Nat.} \ \text{Perm}_\text{Nat} \text{ (sort l) l}
\]
These proofs are simple; we show \texttt{Ordered.sort} in flag-style.

\begin{align*}
1 & \quad \text{Ordered (sort (nil Nat))} & = \leftarrow \text{sort.nil, Ordered.nil} \\
2 & \quad \text{a : Nat} & \quad \text{hypothesis} \\
3 & \quad \text{as : List Nat} & \\
4 & \quad \text{Ordered (sort as)} & \\
5 & \quad \text{sort (a; as) = insert a (sort as)} & \forall \text{E sort.cons} \\
6 & \quad \text{Ordered (insert a (sort as))} & \forall \text{E Ordered.insert,4} \\
7 & \quad \text{Ordered (sort (a; as))} & = \leftarrow 5,6 \\
8 & \quad \forall a : \text{Nat. } \forall \text{as : List Nat. } \text{Ordered (sort as)} \implies \text{Ordered (sort (a; as))} & \forall \text{E indlist,1,8} \\
9 & \quad \forall \text{a : List Nat. } \text{Ordered (sort (a; as))} & \\
\end{align*}

This example is relatively simple. Still, it could not have been developed without Yarrow (or an other proof-assistant based on type theory). Such machine assistance becomes even more imperative in the following chapters, where the type system and the examples become more complicated.

\section{4.8 Comparison with Program Extraction in $\lambda C$}

In \cite{Moh86} Paulin-Mohring describes an approach to program construction that uses program extraction. This approach is quite different from the one we propose. However, it involves closely related type systems, which is why we discuss it here in a bit more detail.

Program extraction is used in so-called 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 $\forall x : A. \exists y : B. P x y$ contains an algorithm which, given a term $x$ returns a term $y$ and a proof $p_y$ of $P x y$ (together, this witness $y$ and proof $p_y$ form a proof of $(\exists y : B. Q x y)$). 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 programs are constructed in a very indirect way. There is no direct control over the program, and it is often difficult to read. Furthermore, the most straightforward proofs often lead to inefficient programs; better programs can be obtained at the cost of less natural proofs.

Another 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, there is much "junk" in the program. A solution to this problem, presented in \cite{PM89}, is to make a syntactical distinction between informative and non-informative propositions. Only the informative parts of a proof are used to produce the program.

The total amount of work in both approaches is approximately equal. In the extraction approach, the terms $y$ and $p_y$ are given indirectly by the term $p$. In $\lambda \omega_L$, we construct two
4.8. COMPARISON WITH PROGRAM EXTRACTION IN $\lambda C$

separate terms $y$ and $p_y$ simultaneously [Pol94]. Unfortunately, Yarrow does not support this simultaneous construction because for this "holes in goals" are necessary, which present severe technical difficulties (see Section 3.2.2). So in Yarrow we have to give first $y$ and then $p_y$ separately.
Part II

Object-Oriented Concepts and Proof Rules
Chapter 5

Introduction to Objects

Object-Oriented Programming (OOP) started with the introduction of the programming language Simula [BDMN73]. Since then, OOP has gained enormously in popularity. Important milestones were the introduction of the languages Smalltalk [GR83], C++ [Str86] and Java [AG96]. In this introductory chapter we explain the most important notions of OOP by way of a small example program, show how these notions can be modeled in $\lambda\omega$, with certain extensions, indicate what extensions are treated in the rest of this thesis, and relate to other work on OOP.

5.1 What is OOP

The most important notions of Object-Oriented Programming are object, class, method and inheritance. An object is a single entity consisting of a state and a collection of methods that operate on the state. This stands in contrast with traditional languages, such as Fortran, C, and Pascal, where data and algorithms are always separate entities. This combining of state and methods is called aggregation.

The programmer does not give objects directly, but defines classes, which can generate objects. A definition of a class consists of a set of methods, the type of the state, and an initial state. In this way, a class generates an initial object. By the application of a method from an object the state may change, but the methods and the type of state remain invariant. The state consists of a number of instance variables. Instance variables of an object are comparable to fields of a record in more traditional languages.

A class may be defined from scratch, but also as an extension of an existing class. The new class is called a subclass of the old class, the superclass. A subclass may have more instance variables or more methods, and typically has both. The subclass only has to specify what it has more than the superclass, the rest of both state and methods is inherited. Methods can be inherited because objects of a subclass have all instance variables of objects of the superclass, so methods that work for objects of a superclass will also work for objects of the subclass. Sometimes a subclass redefines existing methods, i.e. methods can be overridden. By inheritance a whole hierarchy of classes can be defined. Related to inheritance is subtyping. By this mechanism, objects generated by a subclass may be used in a program whenever objects of the original class are expected. Intuitively, this is sound, because objects of the subclass have more methods.

Another important concept is hiding or encapsulation: the state of an object cannot be
class Point is
    var x : Nat = 0
with
    getX = state.x,
    setX = \n:Nat. state{z:n},
    bump = self.setX state (S (self.getX state))
end

class ColPoint from Point is
    var c : Colour = red
with
    getC = state.c,
    setC = \c:Colour. state{z:c},
    setX = \n:Nat. let state’ := super.setX state n in
        self.setC state’ blue
end

Figure 5.1: An example of object-oriented programming

accessed directly, but only through its methods. OOP has this feature in common with Abstract Datatypes (ADTs). The last major concept is that of self reference: a method applied to an object may invoke other methods that are defined on this object. This encourages separation of concerns inside a class. There are several variants for the precise meaning of self reference, as we will see later on.

5.2 An Example OO Program

Figure 5.1 shows a small object-oriented program in a functional OOP language, derived from [PT94]. Virtually all object-oriented languages are imperative, but a functional variant is better suited to be modelled in $\lambda$<sub>i</sub> and its extensions. We will clarify the concepts introduced above with this example.

The first lines define the class Point. An object in this class will represent a point on the line, that can be moved or bumped. The second line declares that there is just one instance variable, $x$, with initial value 0.

The rest of the class describes the methods of Point. The first method, $\text{getX}$, delivers the $x$-component of the state of a point. Since every method operates on the state, the programmer does not have to specify the state as argument to each method; the state is an implicit argument accessible by the keyword state. In real, imperative OOP languages even this keyword may be left out and a part of the state may be accessed by just the name of the corresponding instance variable. However, in this functional style it is necessary to specify which state is used, which we will illustrate later on.

The second method, $\text{setX}$, changes the $x$-coordinate of the point. The new value of $x$ is given as argument $n$ (read "\" as "\lambda") to the method. In this functional setting this means $\text{setX}$ returns a new point object with a modified $x$-coordinate. The new point has the same methods as the old point. In an imperative OOP language we would not deliver a new point, but just change the instance variable $x$.

The third method bumps a point, i.e. the $x$ coordinate is increased by one. It does so
by invoking other methods defined on points. This self reference is denoted by prefixing the
method’s name by “self.”, and is applied to a state.

We give a simple example of the use of this class. We assume that the program text of
Figure 5.1 is already read in by a (non-existing) interpreter. The expressions given to the
interpreter are preceded by #: the output is given right under each expression.

# (newPoint >> setX (S 0)) >> getX
S 0 : Nat

We explain the input from left to right. The value newPoint is implicitly defined when the
class Point was declared, and delivers a new Point object with initial state as specified in
the class. The symbol >> indicates that we invoke a method; the expression p >> m is the
invocation of method m from object p, on the state of the same object p. So here we invoke
method setX from object newPoint on the state of this object, with argument S 0. Since
setX gives a new state, the result of this invocation is another object with a different state.
From this object we invoke the getX method which delivers the x component of the state.
The encapsulation mechanism automatically provided by classes makes it impossible to access
the state directly, only by methods we can access parts of the state.

The class ColPoint consists of coloured points. We define it as a subclass of Point, so it
inherits the instance variable x, and the methods getX and bump. It adds an instance variable
c, and two new methods. The setX method is redefined (overridden), but it still makes use
of the old version by calling super.setX. In this definition we see why we have to specify on
which state methods must act.

The setX method applied to a point in the plane will change both the coordinate and the
colour:

# (newColPoint >> setX (S 0)) >> getC
blue : Colour

Here we see clearly the setX for coloured points is used and not the implementation of setX
for ordinary points.

Now a question arises around the self reference in the definition of bump. When this
method is applied to a ColPoint, will self.setX refer to the setX of class Point (and not
change the colour), or to the setX of ColPoint (and change the colour to blue)? In every
object-oriented language this question appears. Most languages (e.g. Smalltalk) have chosen
the last option as this is the most flexible. Our more advanced models will also model this
so-called late binding, illustrated by the following example.

# (newColPoint >> bump) >> getC
blue : Colour

So self.setX in the definition of bump refers here to the setX of coloured points. This
completes our discussion of the example program.

5.3 Modelling OOP

Traditional, non-OOP languages have a strong formal underpinning. Unfortunately, the the-
etoretical foundations of OOP languages, including sound typing rules, formal semantics and
proof rules, are rather underdeveloped, despite the large interest by the formal computer
science community. Our intention is to (help) develop these theoretical aspects of OOP languages, in particular the proof rules. We do this by modelling objects in type theory, using a variant of the existential model [PT94], which uses the type system $F^\leq$, the higher-order lambda calculus with subtyping, records and existential types. Our intention is that by encoding OOP features with these well-known building blocks, the theoretical aspects of those OOP features can be understood by combining the theoretical aspects of the building blocks.

The existential model consists of the following 6 steps.

1. **Aggregation** The *instance variables* and *methods* are combined (aggregated) into one value, which forms an *object*. The initial values for the instance variables and the implementation of the methods, together with the corresponding types form a *class*. In order to model aggregation we need records (labelled products).

2. **Encapsulation** We hide the state of the object, so that the state is only accessible through the methods. Encapsulation is achieved by using existential types. We pack the type of the state and the aggregated value (of step 1) into another value, which will henceforth represent an object. From this package, only the methods are accessible, while the type of the state is abstract. The type of such object is an existential type.

3. **Subtyping** This is the possibility to use an object generated by a subclass whenever objects of the superclass are expected. We need the subtyping mechanism as provided by system $F^\leq$ to model this.

4. **Inheritance** We define methods in such a way that a subclass can inherit methods from a superclass. We also show how method *overriding* is modelled. In order to model inheritance we need a special form of subtyping called width-subtyping. Only in this respect our model deviates from [PT94]: they "emulate" width-subtyping by standard $\lambda$-calculus constructs, at the cost of a more complex model.

5. **Self reference** We introduce the fixed point combinator to model that the implementation of one method can use other methods of the same class.

6. **Late binding** By applying the fixed point combinator (introduced in step 5) at an other place we model late binding.

5.4 The Structure of the Rest of this Thesis

So our model uses 5 extensions: records, existential types, subtyping, width-subtyping and the fixed point combinator. Since we want not only to model OO programs, but also to reason about them, we need to extend our programming logic $\lambda\omega_L$ with these features. For each extension two aspects can be considered: a proper definition of the extended syntax, including preservation of the meta-theoretical properties, and the proof rules for the new features. The first three extensions are the subjects of Chapters 6 through 8.

In Chapter 6 we treat records and existential types. It is easy to extend the syntax and meta-theory of $\lambda\omega_L$ with these constructs, but the proof rules for existential types are not so simple. The bulk of this chapter is devoted to proof rules for abstract datatypes, which are simple applications of existential types.

For subtyping (notated with $\leq$), the situation is the reverse. The proof rules for subtyping are quite simple, but it is quite hard to extend the meta-theory of $\lambda\omega_L$ with subtyping.
Therefore we devote a separate chapter, Chapter 7, to the framework of \(PTS\)s extended with subtyping (\(PTS^{\leq}\), pronounce as \(PTS\)-subs). We consider the extension of general \(PTS\)s instead of just \(\lambda_{\omega_L}\), since the definition of \(\lambda_{\omega_L}\) with subtyping (\(\lambda_{\omega_L}^{\leq}\)) as a \(PTS^{\leq}\) is much more concise than an ad hoc definition of \(\lambda_{\omega_L}^{\leq}\). Also general meta-theory is more concise than theory for the one specific instance \(\lambda_{\omega_L}^{\leq}\). Furthermore, the general framework of \(PTS^{\leq}\)s can be useful for other applications than just our programming logic, e.g., for programming languages and for formalization of mathematics. In Chapter 8 we define \(\lambda_{\omega_L}^{+\leq}\) as a particular \(PTS^{\leq}\), extended with the records and existential types of Chapter 6, and we give the proof rules for subtyping in this programming logic.

In Chapter 9 we will give the technical details of the model according to the 6 steps above, introducing the extensions of width-subtyping and the fixed point combinator as needed. Unfortunately, we have not enough time to develop proof rules for objects.

## 5.5 Related Work

There are a few groups of OOP languages. The majority of the OOP languages is *class-based*, and our explanation of OOP concerns these class-based languages. Also for the rest of this thesis, we will focus on class-based languages. Another group of OOP languages (e.g., Self [US87]) is called *object-based* (or *delegation based*). These languages do not have the notion of class. Instead, objects are defined directly, and objects themselves can be extended to form new objects. The notion of encapsulation plays a smaller role here, but subtyping remains an important feature.

In order to develop the theory of OOP, we have chosen to model OOP features in type theory. We distinguish two other strategies.

The first is to consider objects as primitives. A major contribution to this strategy is given by [AC96a]. This article defines object calculi with primitives for constructing an object, invoking the method of an object, and redefinition of a method, similar in spirit to \(\lambda\)-calculus with as primitives abstraction and application. Reduction rules for these calculi are given, and the usual properties of reduction, such as Church-Rosser, are proved. From the theoretical aspects we mentioned above, typing rules and semantics are treated, but proof rules are not. Although these object calculi are object-based, concepts such as classes can be encoded in these calculi. However, data abstraction is not present in [AC96a].

A second strategy for developing theory for OOP is to model classes as co-algebras, which are the dual of algebras. This strategy, which emerged only recently [Rei95, HHJT98], focuses on specifications for classes; actual definitions of classes play a submissive role here. Consequently, the theory does not cope with inheritance. Typing rules are not an important issue in this strategy (as they are not for ordinary algebras), and a semantics for inheritance is not (yet) developed.

There exist several models of OOP in type theory. We used the existential model, where objects are inhabitants of existential types. Another well-known model is the recursive record model (e.g., [CHC90]), where objects are inhabitants of recursive record types. The type system used is the second-order lambda calculus with subtyping, records and recursive types. Although this recursive record model is semantically very similar to the existential model [BCP97], recursive types have more complicated proof rules than existential types, so we have chosen the existential model as basis to find proof rules for OOP.
Chapter 6

$\lambda \omega^+_L$ and Modelling Abstract Datatypes

Like all PTSs, $\lambda \omega_L$ provides only very few primitives for term and type formation. In this chapter it is extended with more type constructors. In addition to the dependent product type constructor $\Pi$, we introduce records (labelled cartesian products) and existential types ($\Sigma$-types or weak dependent sum types).

Records allow us to combine several terms into one term. This feature, called aggregation, is a corner stone of object-oriented programming. Existential types provide abstract datatypes. These are used to encapsulate certain aspects of a data structure, so that they cannot be seen from the outside world. This encapsulation, also known as data hiding or data abstraction, is another corner stone of OOP.

Records and existential types are added to $\lambda \omega_L$, the programming part of $\lambda \omega_L$. The additional axioms for records and existential types can be expressed without new primitives in the logical part of $\lambda \omega_L$, so the logic remains unchanged.

The strength of the type systems is not really increased by the new type constructors. There are encodings of the new types in the original system. There are several reasons for introducing records and existential types 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 records and existential types are primitives, they can be implemented more efficiently.

The bulk of this chapter is devoted to using records and existential types for Abstract Datatypes (ADTs, see e.g. [LSAS77]). In particular, we give proof rules for ADTs; these are interesting in their own right, but also as a stepping stone to the encoding of objects that will follow in Chapter 9. Abstract datatypes serve two purposes. First, the use and the implementation of an ADT are separated, by providing encapsulation (or hiding), i.e. only a selected set of operations are visible to the outside world (the user of the ADT). In particular, the concrete representation type is not visible, hence the name abstract datatype. Second, a datatype representing the abstract type and a set of operations on that datatype are combined into a single value. This is a special feature of the use of existential types for ADTs, discussed in [MP84]. The development of an ADT consists of four steps, and we treat the use of the ADT as the fifth step:
1. The interface of the ADT indicates which operations are visible to the outside world, and the signature of these operations.

2. The specification of the ADT gives the properties the visible operations should satisfy.

3. The implementation consists of a representation type and implementations for that representation type of the operations that are visible. Records are used to aggregate the operations into one term, and existential types combine the representation type and the record of operations into one value, the implementation.

4. The correctness proof shows that the implementation satisfies the specification. The main focus on our treatment of ADTs is supplying proof rules to facilitate correctness proofs of ADTs.

5. The ADT is used as building block of a program, and that program can then be proved correct using correctness of the implementation.

Abstract datatypes are particularly useful if several persons are working on one (large) program. We distinguish two hypothetical persons. First, the user of the ADT, who gives the interface and the specification of the ADT and who uses the ADT as building block in a program. The user of the ADT is not concerned with the actual representation, and has no access to it. Second, the implementor of the ADT, who is given the interface and the specification, and who delivers an implementation and a correctness proof. The implementor is not concerned with how the ADT is used.

The focus of this chapter lies on step 4: proof rules for ADTs. Although the very elegant formalization of ADTs using existential types introduced in [MP84] has been around for some time, it has never been used as a solid basis for proof rules for ADTs. Plotkin and Abadi [PA93] have shown that the principle of parametricity is useful for reasoning about ADTs. However, we will see that this principle is not sufficient to derive practical proof rules. Only with the aid of two additional axioms, stating the existence of so-called quotient and subset algebras, we can derive practical proof rules.

In Section 6.1 we give the syntax of $\lambda\omega_L^+$. Section 6.2 gives the most evident axioms of $\lambda\omega_L^+$; we will introduce some more axioms later on (in Sections 6.4, 6.5 and 6.6). Section 6.3 presents the traditional example of an ADT: the stack. This section explains how ADTs are treated in $\lambda\omega_L^+$, following the five step approach given above. This example, including correctness proof, proceeds straightforwardly. Section 6.4 explains the principle of parametricity and its relation to observational behaviour. In Section 6.5 we treat a more complicated example of an ADT, namely bags. Here we encounter a problem during step 4, the correctness proof. The problem is that the implementor cannot prove that his implementation satisfies the specification, although the program is correct. Here the correctness depends essentially on the hiding achieved through the existential types. We solve this problem by using parametricity and assuming the existence of quotients and subsets.

In Section 6.6 we abstract over the example chosen in Section 6.5 and give a general theory for ADTs. Section 6.7 gives the complete theory as it is formalized in Yarrow. In Section 6.8 we consider alternative axioms for introducing quotients and subsets. Finally, Section 6.9 concludes this chapter.
6.1 Syntax

In this section \( \lambda \omega_L \) is extended to include more types than just the dependent product types \((\Pi x : A. B)\), namely

- record types \( \{l_1 : A_1, \ldots, l_n : A_n\} \), and
- existential types \( \Sigma X : \mathcal{K}. A \), also known as \( \Sigma \)-types, dependent sum types or weak sum types. Existential types are usually written as \( \exists X : \mathcal{K}. A \), but we prefer the notation with \( \Sigma \) since it prevents confusion with the existential quantification.

Adding new types comes down to

- extending the set of pseudoterms with new language constructs, viz. a type construct, a term construct for making terms of these types, and a term construct for eliminating terms of these types,
- adding new reduction rules, and
- adding new type inference rules.

6.1.1 Records

Record types are of the form \( \{l_1 : A_1, \ldots, l_n : A_n\} \). It is the cartesian product of the types \( A_1, \ldots, A_n \), where every component \( A_i \) is tagged with a label \( l_i \). Hence record types are also known as labelled products. The elements of the type \( \{l_1 : A_1, \ldots, l_n : A_n\} \) are record values \( \{l_1 = a_1, \ldots, l_n = a_n\} \), where each component \( a_i \) has type \( A_i \). The component with label \( l_i \) is extracted from a record value \( b \) by the field selection \( b \cdot l_i \).

Since the components of a product are labelled, the order of the fields "\( l_i : A_i \)" and "\( l_i = a_i \)" in the type \( \{l_1 : A_1, \ldots, l_n : A_n\} \) and the term \( \{l_1 = a_1, \ldots, l_n = a_n\} \) 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.

This marks one advantage of records over cartesian products: components of records can be referred to by a name — in the form of a label — instead of by a position for cartesian products. Another advantage of records over products is that records allow for clear subtyping rules (see Chapter 8), which is a great asset when encoding objects with records. This is the main motivation for choosing records over cartesian products.

**Definition 6.1.1.1** The set of pseudoterms is extended with:

\[
T ::= \ldots | \{\mathcal{L} : T_1, \ldots, \mathcal{L} : T\} | \{\mathcal{L} = T_1, \ldots, \mathcal{L} = T\} | T \cdot \mathcal{L}
\]

where \( \mathcal{L} \) is a set of labels.

**Convention:** Meta-variables \( l \) and \( m \) range over labels. Actual labels are written in the teletype-font.

**Definition 6.1.1.2** The reduction rules are extended with:

\[
\{l_1 = a_1, \ldots, l_n = a_n\} \cdot l_i \triangleleft a_i
\]
And the typing rules are extended with:

\[ \Gamma \vdash \text{ok} \quad \text{for all } i \mid \Gamma \vdash A_i : \ast \quad \frac{}{\Gamma \vdash \{ l_1 : A_1, \ldots, l_n : A_n \} : \ast} \quad n \geq 0, \ l_i = l_j \Rightarrow i = j \]

\[ \text{(Rec-intro)} \quad \frac{\Gamma \vdash \{ l_1 : A_1, \ldots, l_n : A_n \} : \ast \quad \text{for all } i \mid \Gamma \vdash a_i : A_i}{\Gamma \vdash \{ l_1 = a_1, \ldots, l_n = a_n \} : \{ l_1 : A_1, \ldots, l_n : A_n \}} \]

\[ \text{(Rec-elim)} \quad \frac{\Gamma \vdash a : \{ l_1 : A_1, \ldots, l_n : A_n \}}{\Gamma \vdash a \cdot _i l_i : A_i} \]

**Discussion** Just as the \( \Pi \) formation rule is parametrized by a set \( \mathcal{R} \), we could parametrize the record formation rule by a set \( \mathcal{R}^{\text{Rec}} \), as follows.

\[ \frac{}{\Gamma \vdash \text{ok} \quad \text{for all } i \mid \Gamma \vdash A_i : s \quad s \in \mathcal{R}^{\text{Rec}} \quad n \geq 0, \ l_i = l_j \Rightarrow i = j} \]

where we also adapt the introduction rule (replace \( \ast \) by \( s \)). Our actual rules could then be obtained by choosing \( \mathcal{R}^{\text{Rec}} = \{ \ast \} \), but we would have the freedom to add more elements to \( \mathcal{R}^{\text{Rec}} \). The most sensible addition would be \( \ast' \), which would introduce labelled products of propositions. The introduction and elimination rules indicate such a product should be interpreted as the conjunction of these propositions. For that we already have \( \land \), through a totally satisfactory encoding (see Definition 4.2.3). The advantage of labelled conjunctions is that the individual conjuncts can be identified by name (label).

In this case, we have chosen to keep the rules as simple as possible, instead of making them more general.

### 6.1.2 Existential Types

Existential types are of the form \( (\Sigma X : \mathcal{K}. B) \). Inhabitants of \( (\Sigma X : \mathcal{K}. B) \) are essentially pairs \( (A, b) \) consisting of a type-constructor \( A : \mathcal{K} \) and a program \( b : B[X := A] \).

There are a number of variants of the existential types; these variants are discussed at the end of this section.

Existential types can be used as abstract datatypes, as explained in [MP84]. Examples of this are given in Sections 6.3 and 6.5. The rules for existential types look most familiar if \( \Sigma \) is read as \( \exists \); the introduction and elimination rule for \( \Sigma \) are then the usual introduction and elimination rule for an existential quantification.

**Definition 6.1.2.1** The set of pseudoterms is extended with:

\[ T ::= \ldots \mid (\Sigma V : T.T) \mid (\text{pack } \langle T, T \rangle \text{ in } T) \mid (\text{unpack } T \text{ as } \langle V, V \rangle \text{ in } T) \]

The reduction rules are extended with:

\[ \text{unpack } (\text{pack } \langle A, b \rangle \text{ in } B) \text{ as } (X, x) \text{ in } c \Rightarrow_{\beta} c[X := A][x := b] \]

And the typing rules are extended with:
6.1. SYNTAX

\[(\Sigma\text{-form}) \quad \Gamma \vdash \emptyset : \emptyset, \Gamma, X : \emptyset \vdash B : \ast \quad \Gamma \vdash (\Sigma X : \emptyset, B) : \ast\]

\[(\Sigma\text{-intro}) \quad \Gamma \vdash A : \emptyset \quad \Gamma \vdash b : B[X := A] \quad \Gamma \vdash \Sigma X : \emptyset, B : \ast \quad \Gamma \vdash (\text{pack} (A, b) \text{ in } \Sigma X : \emptyset, B) : (\Sigma X : \emptyset, B)\]

\[(\Sigma\text{-elim}) \quad \Gamma, X : \emptyset, x : B \vdash c : C \quad \Gamma \vdash a : \Sigma X : \emptyset, B \quad \Gamma \vdash C : \ast \quad \Gamma \vdash (\text{unpack } a \text{ as } (X, x) \text{ in } c) : C\]

Discussion There are three variations on existential types as we have introduced them here.

- Our existential types are weak \(\Sigma\)-types, in contrast to the strong \(\Sigma\)-types used in Martin-Löf’s type theories and in the extended calculus of constructions [Luo89]. Inhabitants of a strong \(\Sigma\)-type \((\Sigma X : \emptyset, B)\) are pairs \((A, b)\) with \(A : \emptyset\) and \(b : B[X := A]\), for which we do have direct access to the components \(A\) and \(b\). There are two projections \(\pi_1\) and \(\pi_2\), with the following typing rules:

  \[(\text{strong-}\Sigma\text{-elim}1) \quad \Gamma \vdash a : \Sigma X : \emptyset, B \quad \Gamma \vdash \pi_1 a : \emptyset \]

  \[(\text{strong-}\Sigma\text{-elim}2) \quad \Gamma \vdash a : \Sigma X : \emptyset, B \quad \Gamma \vdash \pi_2 a : B[X := \pi_1 a]\]

There are two reasons why we do not want strong \(\Sigma\)-types. First, they do not provide any encapsulation, which we need for modelling abstract datatypes or object-oriented programming. Second, the first projection \(\pi_1\) introduces dependency of datatypes on programs (e.g. \(\pi_1 a\) is a datatype if \(a\) is a program). In Section 4.3 we explained why we do not want such dependent types.

- The existential types are a particular kind of \(\Sigma\)-types, just as the polymorphic datatypes are a particular kind of \(\Pi\)-types. We obtain the generic \(\Sigma\) formation rule by parametrizing over a set \(\mathcal{R}^\Sigma\), as follows.

  \[(\Sigma\text{-form}) \quad \Gamma \vdash A : s_1, \Gamma, x : A \vdash B : s_2 \quad \Gamma \vdash (\Sigma x : A, B) : s_2 \quad (s_1, s_2) \in \mathcal{R}^\Sigma\]

where we also adapt the introduction and elimination rules. The rules for existential types (Definition 6.1.2.1) correspond with taking \(\mathcal{R}^\Sigma = (\emptyset, \ast, \ast)\). But there are other sensible choices for elements of \(\mathcal{R}^\Sigma\), namely \((\ast, \ast), (\emptyset, \ast),\) and \((\emptyset, \ast)\). These introduce existential quantifications over programs, over datatypes, and over propositions as logical primitives in the system. (Note the difference between existential types and existential quantifications; the former belong to the programming language, the latter to the logic.) Existential quantifications can be encoded totally satisfactorily in \(\lambda \omega_L\) (see Definitions 4.3.5 and 4.2.3), so there is no need to introduce them as primitives.

- We could allow a more liberal \(\Sigma\) elimination rule (called strong elimination in [PM93]):

  \[(\Sigma\text{-elim}) \quad \Gamma, X : \emptyset, x : B \vdash c : C \quad \Gamma \vdash a : \Sigma X : \emptyset, B \quad \Gamma \vdash C : s \quad \Gamma \vdash (\text{unpack } a \text{ as } (X, x) \text{ in } c) : C\]
This rule is more general, since \( C \) may live in any sort \( s \), instead of just in \(*_s\). In other words, \( C \) does not have to be a datatype. However, this rule is inconsistent with the theory of parametricity (Section 6.4) as we will explain there.

Now that we have explained the new typing rules we formally define the extended programming language \( \lambda \omega_L^+ \) and the extended programming logic \( \lambda \omega_L^+ \).

**Definition 6.1.2.2 (\( \lambda \omega_L^+ \))** System \( \lambda \omega_L^+ \) is \( \lambda \omega_L \) extended with records and existential types, according to Definitions 6.1.1.1, 6.1.1.2 and 6.1.2.1.

**Definition 6.1.2.3 (\( \lambda \omega_L^+ \))** System \( \lambda \omega_L^+ \) is \( \lambda \omega_L \) extended with records and existential types, according to Definitions 6.1.1.1, 6.1.1.2 and 6.1.2.1.

**Convention:** We typically omit the second type parameter of pack, writing \( \text{pack} (A, b) \) in \( \Sigma X : \mathcal{K}. B \) whenever the \( \Sigma \)-type is clear from the context.

### 6.1.3 Meta-Theory

Extending \( \lambda \omega_L \) with records and existential types preserves all its properties, as listed in Chapters 2 and 4: CR\( \beta \), CR\( \beta \delta \), CT, SR\( \beta \), UT\( \beta \), SR\( \beta \delta \), UT\( \beta \delta \). All these properties can be proven in a similar way as for ordinary PTSs. We prove SN for \( \lambda \omega_L^+ \) from SN for \( \lambda \omega_L \) as follows.

**Theorem 6.1.3.1** The system \( \lambda \omega_L^+(\beta) \) is SN\( \beta(\beta) \).

**Proof:** This follows from the fact that \( \lambda \omega_L(\beta) \) is SN\( \beta(\beta) \), because there exists a mapping \( \lfloor . \rfloor \) which maps reduction sequences in \( \lambda \omega_L(\beta) \) to longer reduction sequences in \( \lambda \omega_L(\beta) \). Existential types are mapped to their encodings which were given in Section 4.5 as encodings of \( \exists \). The mapping of records is not so direct. First observe that we can translate the binary cartesian product to \( \lambda \omega_L \) in a similar way as we encode \( \wedge \). Second, N-ary products can be treated as repeated binary ones, i.e. \( |\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n| = |\sigma_1 \times (\sigma_2 \times (\ldots \times \sigma_n))| \). Finally, labelled products (records) 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.

The problem of type checking in \( \lambda \omega_L^+ \) is not essentially more difficult than in \( \lambda \omega_L \); the additional constructs are treated in a similar way as II-types, \( \lambda \)-abstractions and applications.
6.2 Axioms

The following axioms express the induction properties of records and existential types.

**Definition 6.2.1** *LOLPLUS* is *AXIOM* extended with the set of axioms containing

\[
\text{ind_rec}_{\{l_1 : A_1, \ldots, l_n : A_n\}} : \forall P : \{l_1 : A_1, \ldots, l_n : A_n\} \rightarrow *_p,
\]

\[
\exists a_1 : A_1, \ldots, a_n : A_n. P \{l_1 = a_1, \ldots, l_n = a_n\} \implies \forall a : \{l_1 : A_1, \ldots, l_n : A_n\}. P a
\]

\[
\text{ind_sigma}_{\mathcal{K}} : \forall I : \mathcal{K} \rightarrow *_s. \forall P : (\Sigma X : \mathcal{K}. I X) \rightarrow *_p,
\]

\[
\exists a : I. \forall a : I A. P \text{ (pack } \langle A, a \rangle \text{ in } \Sigma X : \mathcal{K}. I X) \implies \forall z : (\Sigma X : \mathcal{K}. I X). P z
\]

for every datatype \(\{l_1 : A_1, \ldots, l_n : A_n\}\) and for every kind \(\mathcal{K}\).

The first axiom actually states that every inhabitant \(a\) of \(\{l_1 : A_1, \ldots, l_n : A_n\}\) is of the form \(\{l_1 = a_1, \ldots, l_n = a_n\}\) for some \(a_1, \ldots, a_n\); this axiom is equivalent to the following exhaustion property:

\[
\forall a : \{l_1 : A_1, \ldots, l_n : A_n\}. \exists a_1 : A_1, \ldots, a_n : A_n. a = \{l_1 = a_1, \ldots, l_n = a_n\}
\]

The second axiom states that every inhabitant \(z\) of \(\Sigma X : \mathcal{K}. I X\) is of the form pack \(\langle A, a \rangle\) for some \(A\) and \(a\); it is equivalent to the following exhaustion property:

\[
\forall I : \mathcal{K} \rightarrow *_s. \forall z : (\Sigma X : \mathcal{K}. I X). \exists A : I. z = \text{pack } \langle A, a \rangle
\]

We prefer the formulations of these axioms in Definition 6.2.1 above the formulations as exhaustion properties because these axioms are more similar to the induction principles for the basic datatypes *Bool*, *Nat* and *List* (see 4.5) and because they are easier to use in Yarrow.

**Remark 6.2.2** Just as in Section 4.5 we cannot reduce this infinite set of axioms to a finite set (Remark 4.5.2). This is awkward, because we can enter only a finite number of axioms in Yarrow. We "solve" this problem for existential types by introducing only those axioms that we need, e.g. for kinds \(*_s\) and \(*_s \rightarrow *_s\).

\[
\text{ind_sigma1} : \forall I : *_s \rightarrow *_s. \forall P : (\Sigma X : *_s, I X) \rightarrow *_p,
\]

\[
\forall a : *_s. \forall a : I A. P \text{ (pack } \langle A, a \rangle \text{ in } \Sigma X : *_s, I X) \implies \forall z : (\Sigma X : *_s, I X). P z
\]

\[
\text{ind_sigma2} : \forall I : (*_s \rightarrow *_s) \rightarrow *_s. \forall P : (\Sigma X : *_s \rightarrow *_s, I X) \rightarrow *_p,
\]

\[
\forall a : *_s \rightarrow *_s. \forall a : I A. P \text{ (pack } \langle A, a \rangle \text{ in } \Sigma X : *_s \rightarrow *_s, I X) \implies \forall z : (\Sigma X : *_s \rightarrow *_s, I X). P z
\]

This approach is acceptable, since in practice we need \(\text{ind_sigma}_{\mathcal{K}}\) for only very few kinds \(\mathcal{K}\).

Unfortunately, this approach is not usable for records, since we cannot formally quantify over labels. Therefore we just introduce these axioms in Yarrow as we need them, however awkward it is. (For practical use of Yarrow in verification, the axioms should be built in.)

These axioms for records are all we need in order to prove properties involving records. However, we will see in Section 6.3.2 that our axioms for existential types are not sufficient. Currently, we can prove \(\text{pack } \langle A, a \rangle = \text{pack } \langle B, a \rangle\) only if \(A\) and \(B\) are convertible and \(a\) is equal to \(b\). We will see that we need a more liberal notion of equality for existential types. This will be given in Section 6.4.
6.3 Example: Stack

This section explores how ADTs are encoded in type theory, using the traditional example of stacks. First we show how stacks are modelled without existential types (Section 6.3.1), recapitulating the ideas of [Rey74]. This way of modelling has its disadvantages, so we proceed with modelling the ADT with existential types (Section 6.3.2), following the ideas of [MP84]. We show how to formulate specifications for ADTs and give a proof principle for proving correctness of programs that make use of ADTs.

6.3.1 Hiding without Existential Types

Our first example will be an ADT of stacks of numbers, which provides a type Stack with 5 operations: creating the empty stack, pushing an element on the stack, popping an element off the stack, taking the top element, and examining whether the stack is empty:

\[
\begin{align*}
\text{empty} & \quad : \quad \text{Stack} \\
\text{push} & \quad : \quad \text{Nat} \to \text{Stack} \to \text{Stack} \\
\text{pop} & \quad : \quad \text{Stack} \to \text{Stack} \\
\text{top} & \quad : \quad \text{Stack} \to \text{Nat} \\
is\text{Empty} & \quad : \quad \text{Stack} \to \text{Bool}
\end{align*}
\]

Rather than providing 5 separate programs, we use the aggregation provided by records to give one single program ops of the following type.

\[
\text{ops} : \quad \emptyset \quad \begin{cases}
\text{empty} : \text{Stack}, \\
\text{push} : \text{Nat} \to \text{Stack} \to \text{Stack}, \\
\text{pop} : \text{Stack} \to \text{Stack}, \\
\text{top} : \text{Stack} \to \text{Nat}, \\
is\text{Empty} : \text{Stack} \to \text{Bool}
\end{cases}
\]

Now each of the operations can be accessed by field selection. This record type leads to the interface of the ADT of stacks, by abstracting over the type Stack.

\[
\text{StackI} := \lambda \text{Rep} : \ast, \emptyset \quad \begin{cases}
\text{empty} : \text{Rep}, \\
\text{push} : \text{Nat} \to \text{Rep} \to \text{Rep}, \\
\text{pop} : \text{Rep} \to \text{Rep}, \\
\text{top} : \text{Rep} \to \text{Nat}, \\
is\text{Empty} : \text{Rep} \to \text{Bool}
\end{cases}
\]

\[
: \ast, \to \ast
\]

This interface gives the signature of the operations that should be accessible from the outside world. Sometimes this whole interface is called the signature of the ADT.

A natural implementation of the ADT of stacks is given by the representation type Rep1 and corresponding operations ops1:

\[
\text{Rep1} := \quad \text{List Nat}
\]

\[
: \ast, \to \ast
\]
6.3. EXAMPLE: STACK

ops1 := {empty = nil Nat,
push = (\text{Nat}),
pop = tail Nat,
top = head Nat 0,
isEmpty = null Nat }
: StackI Rep1

The term head Nat 0 delivers the first element of a list of naturals, or 0 if the list is empty; for a complete definition of List and Nat and their operations we refer to Sections 4.5 and 4.6.

Now we show how this implementation of stacks can be used in a program without exposing the actual representation type. We achieve this in two steps. First, we abstract in a program body over the representation type X and over the operations ops. Second, we apply the result to Rep1 and ops1. So we get:

$$(\lambda X:*_s, \lambda \text{ops:StackI } X, \text{body}) \text{Rep1 ops1} \quad (i)$$

which $\beta$-reduces to

$$\text{body}[X := \text{Rep1}][\text{ops := ops1}] .$$

However, in body (as subterm of (i)) we cannot use the fact that X corresponds to Rep1; by the typing rules (i) is only well-typed in $\Gamma$ if

$$\Gamma, X:*_s, \text{ops:StackI } X \vdash \text{body} : B .$$

for some type B. So when typing body we only know that X is some datatype; we achieve hiding (data abstraction) by the typing rules. E.g. taking

$$\text{body} \equiv \text{ops.isEmpty (nil Nat)}$$

makes (i) untypable, whereas

$$\text{body} \equiv \text{ops.isEmpty (ops.empty)}$$

makes (i) well-typed.

Now let us turn towards correctness proofs. First, we define the following specification for stacks.

$$\text{Spec} := \lambda \text{Rep:*}_s, \lambda \text{ops:StackI Rep.}$$

$$\text{ops.isEmpty} \text{ops.empty} =_{\text{Bool}} \text{true} \land$$

$$(\forall s:\text{Rep.} \forall n:\text{Nat.} \text{ops.isEmpty (ops.push n s)} =_{\text{Bool}} \text{false}) \land$$

$$(\forall s:\text{Rep.} \forall n:\text{Nat.} \text{ops.top (ops.push n s)} =_{\text{Nat}} n) \land$$

$$(\forall s:\text{Rep.} \forall n:\text{Nat.} \text{ops.pop (ops.push n s)} =_{\text{Rep}} s)$$

: $\Pi \text{Rep:*}_s, \text{StackI Rep} \rightarrow *_p$

and it is straightforward to show that our implementation satisfies the specification

$$\text{Spec.ops1} := \ldots : \text{Spec Rep1 ops1}$$
CHAPTER 6. \( \lambda \omega^+_L \) AND MODELLING ABSTRACT DATATYPES

Now consider the following program.

\[
\text{prog1} := (\lambda x:*_s, \lambda o:StackIX. \\
\quad \text{ops isEmpty} \ (\text{ops} \cdot \text{pop} \ (\text{ops} \cdot \text{push} \ 0 \ \text{ops} \cdot \text{empty})) \text{Rep1 ops1} \\
\quad : \text{Bool})
\]

Let us say the specification for prog1 is

\[
\text{prog1 = Bool true}.
\]

In this particular case, we can prove correctness of the program by reflexivity, since \( \text{prog1} =_{\beta S} \text{true} \). A more modular and general approach would break the correctness proof of prog1 down in two parts:

- a proof that the implementation of the ADT meets the specification \( \text{Spec} \) (this proof is \( \text{Spec ops1} \) above), and
- a proof that for any \( X \) and ops that meets \( \text{Spec} \), the main program \( body \) equals true.

We express the general version of this principle formally as the following inferred deduction rule, where the last two premises correspond to the two parts of the proof above.

\[
\begin{align*}
\Gamma, X:*_s, \text{ops:StackIX} & \vdash body : B \\
\Gamma & \vdash Q : B \to *_p \\
\Gamma & \vdash \ldots : \text{Spec Rep1 ops1} \\
\Gamma, X:*_s, \text{ops:StackIX} & \vdash \ldots : Q body \\
(\text{principle}) \quad \Gamma & \vdash \ldots : Q ((\lambda x:*_s, \lambda o:StackIX. body) \text{Rep1 ops1})
\end{align*}
\]

This principle is clearer as a deduction rule in flag-style:

\[
\begin{array}{cccc}
m & \text{Spec Rep1 ops1} \\
\vdots \quad \vdots \\
n & X:*_s \\
n + 1 & \text{ops:StackIX} \\
n + 2 & \text{Spec X ops} \\
\vdots \\
p & Q body \\
p + 1 & Q ((\lambda x:*_s, \lambda o:StackIX. body) \text{Rep1 ops1})
\end{array}
\]

This proof principle is quite trivial, since

\[
(\lambda x:*_s, \lambda o:StackIX. body) \text{Rep1 ops1} \Rightarrow_{\beta} \text{body}[X := \text{Rep1}[ops := \text{ops1}]].
\]

We give this proof principle because it abstracts from the specific modelling of hiding we have here; in the next section we will have a similar proof principle for the encoding with existential types. Formally, this principle is expressed as follows. Here we formalize that \( X \) and \( ops \) may
occur in the meta-variable body by using the ordinary variable body which abstracts over \( X \) and ops, so \( \text{body} \equiv \lambda X:*_s. \lambda \text{ops}:\text{StackI} X. \text{body} \).

**principle** := \( \ldots : \forall \text{Rep1}:*_s. \forall \text{Ops1}:\text{StackI Rep1. Spec Rep1 ops1} \implies \\
\forall A:*_s. \forall \text{body}:(\Pi X:*_s. \text{StackI} X \rightarrow A). \forall Q:A \rightarrow *_p. \\
(\forall X:*_s. \forall \text{Ops}:\text{StackI} X. \text{Spec} X \text{ ops} \implies Q (\text{body} X \text{ ops})) \implies \\
Q ((\lambda X:*_s. \lambda \text{Ops}:\text{StackI} X. \text{body} X \text{ ops}) \text{ Rep1 ops1}) \)

Now we return to our example \( \text{prog1} \), and prove it correct using this principle. We use the principle in flag-style, with \( Q := (\lambda X:\text{Bool} X =_{\text{Bool}} \text{true}) : \text{Bool} \rightarrow *_p \), and \( \text{body} := \text{ops-isEmpty} (\text{ops-pop} (\text{ops-push} 0 \text{ ops-empty})) : \text{Bool} \).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>hyp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X:*_s )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \text{ops}:\text{StackI} X )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( \text{Spec} \text{ X} \text{ ops} )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( \text{ops-isEmpty} \text{ ops-empty} =_{\text{Bool}} \text{true} )</td>
<td>( \wedge E ) 3, def Spec</td>
</tr>
<tr>
<td>5</td>
<td>( \forall s:X. \forall n:\text{Nat}. \text{ops-isEmpty} (\text{ops-push} n s) =_{\text{Nat}} \text{false} )</td>
<td>( \wedge E ) 3, def Spec</td>
</tr>
<tr>
<td>6</td>
<td>( \forall s:X. \forall n:\text{Nat}. \text{ops-top} (\text{ops-push} n s) =_{\text{Nat}} n )</td>
<td>( \wedge E ) 3, def Spec</td>
</tr>
<tr>
<td>7</td>
<td>( \forall s:X. \forall n:\text{Nat}. \text{ops-pop} (\text{ops-push} n s) =_{\text{Nat}} s )</td>
<td>( \wedge E ) 3, def Spec</td>
</tr>
<tr>
<td>8</td>
<td>( \text{ops-pop} (\text{ops-push} 0 \text{ ops-empty}) =_{\text{Nat}} \text{ops-empty} )</td>
<td>( \forall E ) 7</td>
</tr>
<tr>
<td>9</td>
<td>( \text{ops-isEmpty} (\text{ops-pop} (\text{ops-push} 0 \text{ ops-empty})) =_{\text{Bool}} \text{true} )</td>
<td>( \equiv ) 8,4</td>
</tr>
<tr>
<td></td>
<td>( ((\lambda X:*<em>s. \lambda \text{ops}:\text{StackI} X. \text{Spec} \text{ X} \text{ ops}) \text{ Rep1 ops1}) =</em>{\text{Bool}} \text{true} )</td>
<td>principle 1 (-) 9, Spec \text{ops1}</td>
</tr>
<tr>
<td>10</td>
<td>( \text{prog1} =_{\text{Bool}} \text{true} )</td>
<td>( \beta\delta)-conv 10</td>
</tr>
</tbody>
</table>

**Overview of the 5 steps**

Let us review this encoding of hiding in good order, i.e. according to the 5 steps mentioned in the introduction.

1. The *user* gives the interface of the ADT.

\[
\text{StackI} := \ldots : *_s \rightarrow *_s
\]

2. The *user* gives the specification \( \text{Spec} \) of the ADT.

\[
\text{Spec} := \ldots : \Pi \text{Rep}:*_s. \text{StackI Rep} \rightarrow *_p
\]

3. The *implementor* gives an implementation of the ADT, i.e. a representation type and the operations.

\[
\text{Rep1} := \ldots : *_s, \\
\text{ops1} := \ldots : \text{StackI Rep1}
\]
4. The implementor proves that this implementation meets the specification, i.e. that Rep1 and ops1 satisfy Spec.

\[
\text{Spec_ops1} := \ldots := \text{Spec Rep1 ops1}
\]

5. The user of the ADT proves the correctness of the main program under the assumption that the ADT satisfies the specification. Formally, we have the following definition.

\[
\text{principle} := \begin{align*}
\forall & \text{Rep1: } \star. \forall \text{ops1: StackI Rep1. Spec Rep1 ops1} \Rightarrow \\
& \forall A: \star. \forall \text{body: (IX: } \star, \text{ StackI X } \rightarrow A) \forall Q: A \rightarrow \star. \\
& (\forall X: \star, \forall \text{ops: StackI X. Spec X ops} \Rightarrow Q (\text{body X ops})) \Rightarrow \\
& Q ((\lambda X: \star, \lambda \text{ops: StackI X. body X ops} \text{ Rep1 ops1}))
\end{align*}
\]

We have written [generic] for the proof term, since this proof is generic in the definitions given above (viz. in StackI and Spec). So this is not a proof obligation, and therefore we do not write "\ldots ", which we did for \text{Spec_ops1}, which is a proof obligation.

One important disadvantage of the way we modelled an ADT in this section is that the representation type Rep1 and the operations ops1 are two separate values, although they conceptually belong together. [MP84] showed how existential types can be used to formally combine Rep1 and ops1 into one single value, which can be treated just like any other value. In the next section we explain how this works.

### 6.3.2 Hiding using Existential Types

Using existential types, we can combine the representation type Rep and operations ops into a single value, which we call an implementation of the ADT. Such an implementation is a first-class citizen, so we can use it just as any other value, e.g. as argument or result of functions.

The implementation is essentially a pair of the representation type Rep1 and the operations ops1, formed with the pack construction:

\[
\text{impl} := \text{pack (Rep1, ops1) in } \Sigma \text{Rep: } \star. \text{StackI Rep} \\
\quad := \Sigma \text{Rep: } \star. \text{StackI Rep}
\]

It is convenient to define an abbreviation for this type of implementations:

\[
\text{StackImp} := \Sigma \text{Rep: } \star. \text{StackI Rep} \\
\quad := \star
\]

The user can retrieve the elements of this implementation by unpacking it, as follows.

\[
\text{prog1'} := \text{unpack impl as } (X, \text{ops}) \in \text{ops is Empty (ops pop (ops push 0 ops empty)}) \\
\quad := \text{Bool}
\]

The typing rule for unpack ensures the actual representation type Rep1 is not visible in the body of this program; the body is typed under the assumptions X: \star, and ops: StackI X. So now hiding (data abstraction) is achieved using existential types.
6.3. EXAMPLE: STACK

We say that $\textbf{StackImp}$ and $\textbf{imp1}$ form the user's side of the ADT, since the actual representation is not accessible from $\textbf{imp1}$. The type $\textbf{Rep}$ and the operations $\textbf{ops}$ form the implementor's side of the ADT.

The implementor can supply another implementation, e.g. one that is more efficient for some applications. For the example of stacks, we give an implementation that also uses lists, but where the top element of the stack corresponds with the last element of the list.

\[
\begin{align*}
\textbf{Rep2} & := \text{List Nat} \\
& \quad * \\
\textbf{ops2} & := \{ \text{empty = nil Nat,} \\
& \quad \text{push = snoc Nat,} \\
& \quad \text{pop = init Nat,} \\
& \quad \text{top = last Nat 0,} \\
& \quad \text{isEmpty = null Nat} \} \\
& \quad : \text{StackI Rep2} \\
\textbf{imp2} & := \text{pack (Rep2, ops2)} \\
& \quad : \text{StackImp}
\end{align*}
\]

where $\text{snoc}$ adds an element at the end of a list, $\text{init}$ delivers the list except for the last element (or $\text{nil}$ if the list is empty), and $\text{last A a}$ delivers the last element of a list (if the list is empty, it delivers $a$).

The nice thing about existential types is that the implementations are first-class citizens, so we can use implementations just as any other value. For example, we can choose (at run-time) which implementation we use, for instance define $\textbf{imp3} := \text{if } \delta \text{ imp1 imp2 for some condition } \delta$, and use $\textbf{imp3}$ in $\textbf{prog1'}$. Such constructions are not possible without existential types.

Now let us turn to the specification of ADTs. Since the user considers the implementation as a single value of type $\textbf{StackImp}$, the user's specification should be a predicate on this type. Hence we define

\[
\begin{align*}
\textbf{UserSpec} & := \lambda \textbf{imp} : \textbf{StackImp}. \exists \textbf{Rep} : *, \exists \textbf{ops} : \textbf{StackI Rep.} \\
& \quad \textbf{imp} =\textbf{StackImp pack (Rep, ops)} \wedge \textbf{Spec Rep ops} \\
& \quad : \textbf{StackImp} \rightarrow *
\end{align*}
\]

Note that we are forced to "unpack" the ADT in this indirect way, since rule ($\Sigma$-elim) allows only direct unpacking in programs, and not in propositions (see the discussion on strong elimination in Section 6.1.2).

The implementor can easily satisfy the user by giving a proof that $\textbf{imp1}$ satisfies $\textbf{UserSpec}$.

\[
\begin{align*}
\textbf{UserSpec. imp1} & := \ldots : \textbf{UserSpec imp1}
\end{align*}
\]

which follows directly from $\textbf{Spec.ops1}$.

In order for the user to use this $\textbf{UserSpec}$, we give a proof principle, similar to the principle of the previous section. Suppose that an implementation $\textbf{imp}$ satisfies $\textbf{UserSpec}$, and $\textbf{body : A}$ is a program that contains $X$ and $\textbf{ops}$, and $\textbf{Q : A \rightarrow *}$ is the specification of $\textbf{body}$. Then the user can prove that unpack $\textbf{imp}$ as $(X, \textbf{ops})$ in $\textbf{body}$ satisfies $\textbf{Q}$ by showing that $\textbf{body}$ satisfies $\textbf{Q}$
under the assumption of $\text{Spec X ops}$. This principle is clearer as a deduction rule in flag-style:

$$
\begin{array}{l}
\begin{array}{c}
m \\
\textit{UserSpec imp}
\end{array}
\end{array}
\begin{array}{c}
n \\
\begin{array}{c}
X : \ast, \\
\text{ops} : \text{StackIX}
\end{array}
\end{array}
\begin{array}{c}
n + 1 \\
\text{Spec X ops}
\end{array}
\begin{array}{c}
p \\
\text{body}
\end{array}
\begin{array}{c}
p + 1 \\
Q (\text{unpack imp as } (X, \text{ops}) \text{ in body})
\end{array}
\begin{array}{c}
\text{principle } m, n - p
\end{array}
$$

Formally, the principle is expressed as follows.

\[
\text{principle} := \text{[generic]}
\forall \text{imp} : \text{StackImp}. \text{UserSpec imp} \implies \\
\forall A : \ast, \forall \text{body} : (\forall X : \ast, \text{StackIX} \rightarrow A). \forall Q : A \rightarrow \ast, \\
(\forall X : \ast, \forall \text{ops} : \text{StackIX}. \text{Spec X ops} \implies Q (\text{body X ops})) \implies \\
Q (\text{unpack imp as } (X, \text{ops}) \text{ in body X ops})
\]

Now, let us assume the user wants to prove $\text{prog1}^\prime = \text{bool true}$. With our proof principle, the $\text{prog1}^\prime$ is proved correct as follows.

$$
\begin{array}{l}
\begin{array}{l}
1 \\
2 \\
3
\end{array}
\begin{array}{l}
X : \ast, \\
\text{ops} : \text{StackIX}
\end{array}
\begin{array}{l}
\text{Spec X ops}
\end{array}
\begin{array}{l}
\text{hyp}
\end{array}
\begin{array}{l}
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11
\end{array}
\begin{array}{l}
\text{ops-isEmpty ops-empty } = \text{bool true} \\
\forall s : X. \forall n : \text{Nat}. \text{ops-isEmpty (ops-push n s) } = \text{bool false} \\
\forall s : X. \forall n : \text{Nat}. \text{ops-top (ops-push n s) } = \text{Nat n} \\
\forall s : X. \forall n : \text{Nat}. \text{ops-pop (ops-push n s) } = s \\
\text{ops-pop (ops-push 0 ops-empty) } = \text{ops-empty} \\
\text{ops-isEmpty (ops-pop (ops-push 0 ops-empty)) } = \text{bool true} \\
(\text{unpack imp1 as } (X, \text{ops}) \text{ in} \\
\text{ops-isEmpty (ops-pop (ops-push 0 ops-empty)) } = \text{bool true} \\
\text{prog1}^\prime = \text{bool true}
\end{array}
\begin{array}{l}
\land E 3, \text{def Spec} \\
\land E 3, \text{def Spec} \\
\land E 3, \text{def Spec} \\
\land E 3, \text{def Spec} \\
\forall E 7 \\
\Leftarrow 8, 4 \\
\text{principle 1} - 9, \\
\text{UserSpec.imp1} \\
\beta \delta \text{-conv 10}
\end{array}
$$

Overview of the 5 steps

Let us review the ADT of stacks according to the 5 steps mentioned in the introduction.
6.3. EXAMPLE: STACK

1. The user gives the interface of the ADT.
   \[ \text{StackI} \ := \ \ldots \ := \ *_s \rightarrow *_s \]
   This determines the corresponding existential type.
   \[ \text{StackImp} \ := \ \Sigma_{\text{Rep:*_s}} \text{StackI Rep} \]
   \[ \ := \ *_s \]

2. The user gives the specification \( Spec \) of the ADT,
   \[ \text{Spec} \ := \ \ldots \ := \ \Pi_{\text{Rep:*_s}} \text{StackI Rep} \rightarrow *_p \]
   This determines the definition of the user's specification.
   \[ \text{UserSpec} \ := \ \lambda_{\text{imp:StackImp}} \exists_{\text{Rep:*_s}} \exists_{\text{ops:StackI Rep}} \]
   \[ \text{imp} = \text{StackImp \ pack \ (Rep, ops)} \land \text{Spec \ Rep \ ops} \]
   \[ \ := \ \text{StackImp} \rightarrow *_p \]

3. The implementor gives a representation type and the operations. (The implementor might give several implementations; for each of them step 3 and 4 should be followed).
   \[ \text{Rep1} \ := \ \ldots \ := \ *_s \]
   \[ \text{ops1} \ := \ \ldots \ := \ \text{StackI Rep1} \]
   These determine the implementation value.
   \[ \text{imp1} \ := \ \text{pack \ (Rep1, ops1)} \]
   \[ \ := \ \text{StackImp} \]

4. The implementor proves that this implementation meets the specification, i.e. that \text{Rep1} and \text{ops1} satisfy \text{Spec}.
   \[ \text{Spec.ops1} \ := \ \ldots \ := \ \text{Spec \ Rep1 \ ops1} \]
   This determines a proof of \text{UserSpec \ imp1}.
   \[ \text{UserSpec.imp1} \ := \ \text{[generic]} \]
   \[ \ := \ \text{UserSpec \ imp1} \]
   Again, we have written \text{[generic]} for the last proof term, since this proof is completely generic in the definitions given above, just as \text{StackImp} is completely determined by \text{StackI}, \text{UserSpec} is completely determined by \text{StackI} and \text{Spec} and so on. We do not spell out this trivial proof because it is long. So this is not a proof obligation, and therefore we do not write "...".

5. The user of the ADT proves the correctness of the main program under the assumption that the ADT satisfies the specification. Formally, we have the following definition.
   \[ \text{principle} \ := \ \text{[generic]} \]
   \[ \ := \ \forall_{\text{imp:StackImp}} \text{UserSpec \ imp} \rightarrow \]
   \[ \forall A:*_s, \forall \text{body:(\Pi X:*_s \text{StackI X} \rightarrow A).} \forall Q:A \rightarrow *_p, \]
   \[ (\forall X:*_s, \forall \text{ops:StackI X. Spec \ X \ ops} \rightarrow Q (\text{body} X \ \text{ops})) \rightarrow \]
   \[ Q (\text{unpack \ imp as \ (X, \text{ops}) \ in \ body} X \ \text{ops})) \]
Figure 6.1: Relation between concrete and abstract values for the implementation of stacks.

Summary

Let us review this example of stacks. First, note that we could exploit the features of $\lambda\omega_L$ by making this ADT polymorphic in the type of the elements contained in the stack. Second, this example is a bit uninteresting, because no information is hidden; every concrete value (list) corresponds with exactly one abstract value (stack). In the transition of concrete type to abstract type no information is lost. This situation is illustrated in Figure 6.1. Later on, we will give more interesting examples of implementations where there is no one-to-one correspondence (Section 6.5).

So far we have named two advantages of using existential types over the $\lambda\omega_L$ approach in Section 6.3.1.

- Conceptual: The representation type and the operations, conceptually belonging together, can be packed together in one value using existential types.

- Practical: Since implementations of ADTs are first-class citizens, we can manipulate these values just as any other value, e.g. decide at run-time which ADT we use.

There are also logical advantages of existential types. We will see that we can enhance our logic so that we have an interesting notion of equality on existential types. Furthermore, we will show how the more interesting examples of ADTs are proved correct, using this notion of equality (Section 6.5).
6.4 Parametricity

In Section 6.4.1 we explain the need for an axiom that gives a richer notion of equality on existential types: implementations should be considered equal, if they have the same observational behaviour. The notion of observational behaviour is formalized using simulations (Section 6.4.2). Section 6.4.3 gives the axiom, which is an instantiation of the theory of parametricity [PA93]. Section 6.4.4 explains this general theory, from which some results will be used in Section 6.7.5.

6.4.1 Observational Behaviour and Equality of Implementations

For the sake of explanation, we return to the example of Section 6.3, the ADT of stacks. We gave two implementations for this ADT, both with lists, but in imp1 the top element of the stack is represented by the head of the list, and in imp2 the top is represented by the last element of the list. It is remarkable that, although the concrete operations ops1 and ops2 are quite different, the user cannot distinguish between the two implementations pack (Rep1, ops1) and pack (Rep2, ops2). This is the virtue of data hiding provided by existential types. Let us explain this fact in more detail.

The only way for the user to observe the list representing a stack is to use the top and isEmpty operations. Such a list can only be constructed by a number of applications of the push and pop operations on the empty stack. (These limitations are enforced by the hiding mechanism through the typing rules for unpack.) The only way the user could distinguish imp1 and imp2 would be by observing different behaviour. Because of our choice of ops1 and ops2, the implementations imp1 and imp2 have exactly the same behaviour. Let us consider this more formally.

We say that an element of Rep1 corresponds with an element of Rep2 if both represent the same stack. E.g. [1, 3] under implementation imp1 represents the stack (1), and [3, 1] under implementation imp2 represents the same stack, so [1, 3] and [3, 1] correspond. It should be clear that in general a list l1 : Rep1 under operations ops1 corresponds with a list l2 : Rep2 under operations ops2 if l2 is the reversal of l1. So let us define the relation ≃ as follows.

\[ (\simeq) := \lambda l1:Rep1. \lambda l2:Rep2. \text{reverse} l1 \equiv_{\text{List Nat}} l2. \]

This ≃ is a so-called simulation relation, i.e. it relates corresponding elements, because of the following five properties. (In other areas of computer science, ≃ would perhaps be called a bisimulation relation, but we will use that word for a slightly different notion, in Definition 6.4.2.3 below.)

1. ops1·empty and ops2·empty correspond:
   \[ \text{ops1·empty} \simeq \text{ops2·empty}. \]

2. For every number m, the operation push m preserves ≃:
   \[ \forall m: \text{Nat}. \forall l1:Rep1. \forall l2:Rep2. l1 \simeq l2 \implies \text{ops1·push} m l1 \simeq \text{ops2·push} m l2. \]

3. The operation pop preserves ≃:
   \[ \forall l1:Rep1. \forall l2:Rep2. l1 \simeq l2 \implies \text{ops1·pop} l1 \simeq \text{ops2·pop} l2. \]
CHAPTER 6. $\lambda \omega_1^1$ AND MODELLING ABSTRACT DATATYPES

Figure 6.2: $\simeq$ is a simulation between ops1 and ops2.

4. The operation top delivers the same number if applied to corresponding lists:

$$\forall 11: \text{Rep1}. \forall 12: \text{Rep2}. 11 \simeq 12 \Rightarrow \text{ops1.top} \ 11 =_{\text{nat}} \text{ops2.top} \ 12.$$

5. The operation isEmpty delivers the same boolean if applied to corresponding lists:

$$\forall 11: \text{Rep1}. \forall 12: \text{Rep2}. 11 \simeq 12 \Rightarrow \text{ops1.isEmpty} \ 11 =_{\text{bool}} \text{ops2.isEmpty} \ 12.$$

Every operation from ops1 is simulated by its brother from ops2 (and vice versa), and $\simeq$ gives the relation between both sides. Figure 6.2 illustrates this. So we formalized that imp1 and imp2 have the same observable behaviour by giving a simulation $\simeq$ between ops1 and ops2.

Since imp1 and imp2 have the same behaviour, it would be nice to have $\text{imp1} =_{\text{StackImp}} \text{imp2}$. This would be in analogy with extensionality for functions: if two functions $f$ and $g$ behave exactly the same — $f \ x = g \ x$ for all $x$ — we say that $f = g$. This principle is a great asset for doing program transformation. However, the major reason for wanting this richer notion of equality on existential types is that we need it for proving correctness of the ADTs in Section 6.5. Note that we can express this equality only because we have combined the representation type and the operations into a single value.

Unfortunately, with the current set of axioms we cannot prove $\text{imp1} =_{\text{StackImp}} \text{imp2}$ (see also Section 6.2). Therefore we will introduce an axiom (called parametricity) that allows us to prove two implementations to be equal, if there is a simulation between the respective operations. In particular, we will prove $\text{imp1} =_{\text{StackImp}} \text{imp2}$ (Section 6.4.3). First, we will need to give a general and formal definition of simulation.

6.4.2 Simulations

We have introduced the notion of simulation by means of an example for the ADT of stacks. We define the general notion of simulation by induction on the interface of the ADT. The
6.4. PARAMETRICITY

reader might have expected this connection; the interface of stacks is the record with 5 operations, and for each of the operations we listed one property. Each property had a very rough similarity with the type of the respective operation.

However, we give this principle only for the so-called simply-typed interfaces, in order to keep the definition of simulation as simple as possible. The simply-typed interfaces correspond roughly to the types of the simply typed λ-calculus. More precise: an interface of the form \( \lambda X : \star, A \) is simply-typed if \( A \) is built by the constructors for function types and record types from \( X \) and closed type expressions. This excludes many second-order interfaces, like \( \lambda X : \star, \{ I \in (\Pi Y : \star, Y \rightarrow X) \} \), but includes non-traditional interfaces like \( \lambda X : \star, (X \rightarrow X) \rightarrow X \). First we make this predicate “simply-typed” on interfaces formal. The easiest definition of this predicate is that all type-constructors of the form \( \lambda X : \star, \) Simply\( T \) are simply-typed, with Simply\( T \) defined as follows.

\[
\text{Simply} \; T := X \mid T \text{ with } X \notin \text{FV}(T) \mid \text{Simply} \; T \rightarrow \text{Simply} \; T \mid \{ l_1 : \text{Simply} \; T, \ldots, l_n : \text{Simply} \; T \}
\]

However, we prefer the following definition, since it is better suited to formalization in \( \lambda \omega \uparrow \).

**Definition 6.4.2.1** The predicate *simply-typed* on type-constructors of type \( \star, \rightarrow \star \), is the smallest predicate closed under:

- \( \lambda X : \star, X \) is simply-typed.
- \( \lambda X : \star, T \) is simply-typed if \( X \notin \text{FV}(T) \).
- \( \lambda X : \star, I_1 X \rightarrow I_2 X \) is simply-typed if \( I_1 \) and \( I_2 \) are simply-typed and \( X \notin \text{FV}(I_1) \cup \text{FV}(I_2) \).
- \( \lambda X : \star, \{ l_1 : I_1 X, \ldots, l_n : I_n X \} \) is simply-typed if for each \( i \) \( I_i \) is simply-typed and \( X \notin \text{FV}(I_i) \).

Of course, we consider type-constructors up to \( \beta \delta \)-convertibility. ⊓⊔

For example, the type-constructor \( \lambda X : \star, X \rightarrow \text{Nat} \) is simply-typed because it is convertible to \( \lambda X : \star, (\lambda X : \star, X) \rightarrow (\lambda X : \star, \text{Nat}) X \), which is simply-typed. The interface Stack\( I \) is also simply-typed; in fact all traditional interfaces are simply-typed. We will use the word “interface” for any type-constructor of type \( \star, \rightarrow *, \) and “simply-typed interface” for any simply-typed type-constructor.

Now we introduce the general definition of simulation. In fact, our definition of simulation (Definition 6.4.2.2 below) is the usual lifting of a mapping \( I \) on types to a mapping \( \text{Sim}_I \) on relations. For readers not familiar with this lifting, we will use the definition of simulation for Stack\( I \) as guideline to the general definition of simulation.

We generalize from the particular interface Stack\( I \) to arbitrary simply-typed interfaces \( I \), and from the particular operations \( \text{ops} : \text{Stack} \rightarrow \text{Rep} \) and \( \text{ops} : \text{Stack} \rightarrow \text{Rep} \) to arbitrary operations \( y : I \rightarrow \text{Rep} \) and \( z : I \rightarrow \text{Rep} \). Given such an interface \( I \) and operations \( y \) and \( z \), the proposition \( \text{Sim}_I \) \( y z \) will express that \( (\simeq) \) is a simulation between \( y : I \rightarrow \text{Rep} \) and \( z : I \rightarrow \text{Rep} \); so the five properties listed in Section 6.4.1 will be concisely expressed as \( \text{Sim}_{\text{Stack} \; I} \); \( \text{ops} \); \( \text{ops} \). We define \( \text{Sim}_I \) by induction on the simply-typed interface \( I \), so we have four cases for \( I \). This gives three guidelines for finding the definition of \( \text{Sim}_I \).

1. \( \text{Sim}_{\text{Stack} \; I} \); \( \text{ops} \); \( \text{ops} \) = “the conjunction of the 5 properties listed in Section 6.4.1”.
2. $\text{Sim}_I y z$ is well-typed if $y : I \text{Rep}_1$ and $z : I \text{Rep}_2$.

3. $\text{Sim}_I$ is defined by induction on the simply-typed interface $I$.

Now we consider the individual cases. For reasons of presentation, we handle them in a different order than in Definition 6.4.2.1.

**Case** $I = \lambda X : * \rightarrow \{ \{ l_1 : I_1 X, \ldots, l_n : I_n X \} \}$

The interface $\text{Stack}_I$ has this form, namely $\text{Stack}_I = \lambda X : * \rightarrow \{ \emptyset : \text{Stack}_I X, \ldots, \text{isEmpty} : \text{Stack}_I X \}$ where e.g. $\text{Stack}_I_1 = \lambda X : *, X$ and $\text{Stack}_I_5 = \lambda X : *, X \rightarrow \text{Bool}$. We want to have

$$\text{Sim}_{\text{Stack}_I} \text{ops}_1 \text{ops}_2 = \text{"property 1" } \land$$

$$\vdots$$

$$\text{"property 5"}$$

By inspection of property 1, we see it is a relation $R_1$ between $\text{ops}_1 \cdot \text{empty}$ and $\text{ops}_2 \cdot \text{empty}$ (only the empty fields of $\text{ops}_1$ and $\text{ops}_2$ are used in property 1). Similarly, property 2 is a relation $R_2$ between $\text{ops}_1 \cdot \text{push}$ and $\text{ops}_2 \cdot \text{push}$, and so on, so we have

$$\text{Sim}_{\text{Stack}_I} \text{ops}_1 \text{ops}_2 = R_1 \text{ops}_1 \cdot \text{empty} \land \text{ops}_2 \cdot \text{empty} \land$$

$$\vdots$$

$$R_5 \text{ops}_1 \cdot \text{isEmpty} \land \text{ops}_2 \cdot \text{isEmpty},$$

where, for example,

$$R_1 y z = (y \simeq z),$$

and

$$R_4 y z = (\forall l_1 : \text{Rep}_1. \forall l_2 : \text{Rep}_2. l_1 \simeq l_2 \implies y l_1 = \text{Nat} z l_2).$$

We expect $\text{Sim}_{\text{Stack}_I}$ to be expressed in terms of $\text{Sim}_{\text{Stack}_I_1}$ through $\text{Sim}_{\text{Stack}_I_5}$. Since each $R_i$ has the same type as $\text{Sim}_{\text{Stack}_I_1}$ (both expect one argument of type $\text{Stack}_I$, $\text{Rep}_1$ and one of type $\text{Stack}_I$, $\text{Rep}_2$) we assume $R_i = \text{Sim}_{\text{Stack}_I_1}$. This gives us a guideline for the definition of $\text{Sim}$ for the other three cases.

Using this equality, we have

$$\text{Sim}_{\text{Stack}_I} \text{ops}_1 \text{ops}_2 = \text{Sim}_{\text{Stack}_I_1} \text{ops}_1 \cdot \text{empty} \land \text{ops}_2 \cdot \text{empty} \land$$

$$\vdots$$

$$\text{Sim}_{\text{Stack}_I_5} \text{ops}_1 \cdot \text{isEmpty} \land \text{ops}_2 \cdot \text{isEmpty},$$

Generalizing this from $\text{ops}_1$ and $\text{ops}_2$ to arbitrary $y$ and $z$, and unfolding the definition of $\text{Stack}_I$ gives

$$\text{Sim}_{\lambda X : *, \emptyset : \text{Stack}_I_1, \ldots, \text{isEmpty} : \text{Stack}_I_5} y z = \text{Sim}_{\text{Stack}_I_1} y \cdot \text{empty} z \cdot \text{empty} \land$$

$$\vdots$$

$$\text{Sim}_{\text{Stack}_I_5} y \cdot \text{isEmpty} z \cdot \text{isEmpty},$$

Now we generalize from $\text{Stack}_I_j$ to arbitrary $I_j$, from the specific labels $\emptyset \ldots \text{isEmpty}$ to arbitrary labels $l_1 \ldots l_n$, giving

$$\text{Sim}_{\lambda X : *, \emptyset : I_1 X, \ldots, \text{isEmpty} : I_n X} y z = \text{Sim}_{I_1} y \cdot l_1 z \cdot l_1 \land \ldots \land \text{Sim}_{I_n} y \cdot l_n z \cdot l_n.$$
6.4. PARAMETRICITY

Case $I = \lambda X.:*, X$

The interface $\text{StackI}_1$ (for the empty field) is equal to this $I$. The corresponding relation $R_1$ as introduced above is defined as $R_1 \ y \ z = (y \simeq z)$, and we assumed $R_1 = Sim_{\text{StackI}_1}$. This immediately determines the definition of $Sim$ for this case, as follows.

$$Sim_{(\lambda X.:*, X)} \ y \ z = (y \simeq z)$$

Case $I = \lambda X.:*, I_1 \ X \to I_2 \ X$

We have several choices of $\text{StackI}_j$ that have this form, so as to guide us to a definition of $Sim_I$. We take the interface $\text{StackI}_4$ (for the top field), which is equal to $\lambda X.:*, \text{StackI}_{41} \ X \to \text{StackI}_{42} \ X$ with $\text{StackI}_{41} = \lambda X.:*, X$ and $\text{StackI}_{42} = \lambda X.:*, \text{Nat}$. We assumed above that

$$Sim_{\text{StackI}_1} \ y \ z = \forall 11: \text{Rep1}. \forall 12: \text{Rep2}. 11 \simeq 12 \implies y \ 11 = \text{Nat} \ z \ 12.$$ 

We expect $Sim_{\text{StackI}_4}$ to be expressed in terms of $Sim_{\text{StackI}_{41}}$ and $Sim_{\text{StackI}_{42}}$. As we have seen in the case above, $Sim_{\text{StackI}_{41}} u v = u \simeq v$, which gives

$$Sim_{\text{StackI}_4} \ y \ z = \forall 11: \text{Rep1}. \forall 12: \text{Rep2}. Sim_{\text{StackI}_{41}} 11 \ 12 \implies y \ 11 = \text{Nat} \ z \ 12 .$$

This suggests that $Sim_{\text{StackI}_{42}}$ should be used in the conclusion, and since $Sim_{\text{StackI}_{42}}$ expects two naturals as arguments, we assume $Sim_{\text{StackI}_{42}} u v = (u = \text{Nat} \ v)$, which gives us a guideline for the last case. Using this equality and unfolding $\text{StackI}_1$ gives

$$Sim_{\lambda X.:*, \text{StackI}_{41} \to \text{StackI}_{42} \ X} \ y \ z = \forall 11: \text{Rep1}. \forall 12: \text{Rep2}. Sim_{\text{StackI}_{41}} 11 \ 12 \implies Sim_{\text{StackI}_{42}} (y \ 11) (z \ 12) .$$

$Sim_{\text{StackI}_4}$ expects an argument of type $\text{StackI}_{41} \ \text{Rep1}$ and one of type $\text{StackI}_{41} \ \text{Rep2}$, and its arguments here are 11 and 12, so let us reformulate the quantifications as

$$\forall 11: \text{StackI}_{41} \ \text{Rep1}. \forall 12: \text{StackI}_{41} \ \text{Rep2} . . . .$$

By generalizing from $\text{StackI}_{41}$ to $I_1$ and $\text{StackI}_{42}$ to $I_2$ and renaming this gives us:

$$Sim_{(\lambda X.:*, I_1 \ X \to I_2 \ X)} \ f \ g = \forall u: I_1 \ \text{Rep1}. \forall v: I_2 \ \text{Rep2}. Sim_{I_1} u v \implies Sim_{I_2} (f \ u) (g \ v) .$$

Case $I = \lambda X.:*, T$

The interface $\text{StackI}_{42}$ has this form, with $T = \text{Nat}$. The corresponding assumption we made is

$$Sim_{\text{StackI}_{42}} u v = (u = \text{Nat} \ v) .$$

Generalizing from $\text{Nat}$ to an arbitrary $T$ gives

$$Sim_{(\lambda X.:*, T)} \ y \ z = (y = T \ z) .$$

End cases
We considered only certain parts of the 5 properties for determining $Sim_I$, so we should check that unfolding $Sim_{Stack1}$ $ops1$ $ops2$ using this definition indeed gives the conjunction of the five properties. It turns out there is only a syntactical difference for the push operation:

\[
Sim_{Stack1} \; ops1\text{-}push \; ops2\text{-}push = \\
\forall m: \text{Nat}. \forall n: \text{Nat}. \; m = n \implies \\
\forall i1: \text{Rep1}. \forall i2: \text{Rep2}. \; i1 \simeq i2 \implies \\
\; ops1\text{-}push \; m \; 11 \simeq ops2\text{-}push \; n \; 12,
\]

but since this is logically equivalent to property 2 above, we have equivalence between $Sim_{Stack1}$ $ops1$ $ops2$ and the 5 properties and this is sufficient.

The formal definition of $Sim_I$ is just a slight reformulation of the results found above, and we consider arbitrary $Y$, $Z$ and $R$ instead of $Rep1$, $Rep2$ and $\simeq$. This definition is equal to the one in [PA93], except that we also consider records. Furthermore, the notion of simulation is quite similar to the notion of "logical relation" [Mit90].

**Definition 6.4.2.2 (Simulation)**
Given a simply-typed type-constructor $I : \ast \rightarrow \ast$, and two datatypes $Y, Z : \ast$, then $Sim_I : (Y \rightarrow Z \rightarrow \ast) \rightarrow (I \; Y \rightarrow I \; Z \rightarrow \ast)$ is defined as follows (where we assume $X \notin \text{FV}(I_t)$ for each $i$):

\[
\begin{align*}
Sim_{\lambda X: \ast \rightarrow \ast, \; X} \; \; Y \; z & \; = \; Y \; z \\
Sim_{\lambda X: \ast \rightarrow \; T, \; X} \; \; Y \; z & \; = \; T \; z , \; \text{if} \; X \notin \text{FV}(T) \\
Sim_{\lambda X: \ast \rightarrow I_1 \; X \rightarrow I_2 \; X} \; \; Y \; z & \; = \; \forall a: I_1 \; Y. \forall b: I_2 \; Z. \; Sim_{I_1} \; a \; b \implies Sim_{I_2} \; R \; (f \; a) \; (g \; b) \\
Sim_{\lambda X: \ast \rightarrow \prod I_n \; X \rightarrow \prod I_n \; X} \; \; Y \; z & \; = \; \prod Sim_{I_n} \; R \; y \; i_1 \; z \; i_1 \; \land \ldots \land \prod Sim_{I_n} \; R \; y \; i_n \; z \; i_n
\end{align*}
\]

The second clause overlaps with both the third and the fourth clause, but it is easy to see that this does not give conflicts, so $Sim$ is well-defined up to logical equivalence.

For the rest of this thesis, we consider $Y$ and $Z$ to be parameters of $Sim_I$, so

\[
Sim_I : \forall Y, Z: \ast. (Y \rightarrow Z \rightarrow \ast) \rightarrow (I \; Y \rightarrow I \; Z \rightarrow \ast).
\]

\[
\square
\]

The type of $Sim$ shows that $Sim_I Y Z$ lifts a relation between $Y$ and $Z$ to a relation between $I \; Y$ and $I \; Z$. This notion of simulation has a clear categorical interpretation: if datatypes are objects, and relations are morphisms, then the pair $(I, Sim_I)$ forms a functor for each $I$.

In contrast with notions of simulation occurring in some other areas of computer science, our notion is symmetric, i.e.

\[
Sim_I Y Z \; R \; y \; z \leftrightarrow Sim_I Z Y \; R^{-1} \; z \; y.
\]

The five properties listed in Section 6.4.1 that say that $\simeq$ is a simulation between $ops1$ and $ops2$ are now formally written as follows:

\[
Sim_{Stack1} \; \text{Rep1} \; \text{Rep2} (\simeq) \; \text{ops1} \; \text{ops2},
\]

where $11 \simeq 12$ expresses that $12$ is the reversal of $11$. (Here the relation $\simeq$ is enclosed in brackets, because $\simeq$ is here not used as infix symbol, see Section 3.2.3.)
Later on, we will use the notion of bisimulation, which is defined in terms of simulation. It expresses that a set of operations respects some binary relation on a datatype. (In other areas of theoretical computer science, e.g. those dealing with non-determinism, also the notions of simulation and bisimulation are used. Our notions do not precisely correspond with the use in other areas, because of the difference in context.)

**Definition 6.4.2.3 (Bisimulation)** The notion of bisimulation \(\text{Bisim}_I\) is defined as follows:

\[
\text{Bisim}_I \ Y \ R \ y = \text{Sim}_I \ Y \ Y \ R \ y \ y .
\]

So, given an interface \(I\), a datatype \(Y\), a relation \(R : Y \rightarrow Y \rightarrow *_p\), and a program \(y : I \ Y\), the expression \(\text{Bisim}_I \ Y \ R \ y\) expresses that \(R\) is a bisimulation for \(y\). This means that the program \(y\) respects the relation \(R\). \(\square\)

### 6.4.3 Parametricity for Existential Types

As argued in Section 6.4.1, implementations of an ADT with the same behaviour should be considered equal, and having the same behaviour is expressed by the existence of a simulation between the respective operations. Using the formal definition of simulation given in Section 6.4.2 we formalize that implementations with the same behaviour are equal with the following axiom:

\[
\text{parSigma}_I : \forall Y, Z : *_p, \forall R : Y \rightarrow Z \rightarrow *_p, \forall y : I \ Y, \forall z : I \ Z .
\]

\[
\text{Sim}_I \ Y \ Z \ R \ y \ z \Rightarrow \text{pack} \langle Y, y \rangle = \Sigma \ x \in A, I \ x \ \text{pack} \langle Z, z \rangle
\]

for all simply-typed interfaces \(I\). Since we have a simulation for the operations \(\text{ops}_1\) and \(\text{ops}_2\), we can now formally prove

\[
\text{impl} = \text{stack} \ \text{impl}.
\]

We call \(\text{parSigma}\) parametricity for existential types, since it is an instance of the more general parametricity scheme (see Corollary 6.4.4.4). Parametricity for existential types is the logical counterpart of the programming principle of hiding.

### 6.4.4 General Parametricity

The general theory of parametricity [Rey83] expresses that all polymorphic functions behave uniformly across the type parameter. In other words, we cannot make a \(\lambda \omega\), function of type \(\Pi X : *_p, T\) (\(X\) may occur free in \(T\)) that makes a case distinction to the type variable \(X\). For example, the only function of type \(\Pi X : *_p, X \rightarrow X\) is the polymorphic identity; we cannot make a function of this type that adds 1 to its second argument if \(X\) is \texttt{Nat}, and is the identity otherwise.

Parametricity for system \(F\) (\(\lambda 2\)) is described in [Wad89, PA93, Tak97]. The first article introduces parametricity to derive theorems about polymorphic functions of which only the type is known. The other two articles generalize the results and express them in a formal logic. Fortunately, their logic is a subsystem of \(\lambda \omega_L\), so we can easily express their results in \(\lambda \omega_L\).

All results in these articles concern \(\lambda 2\), whereas we work with the programming language \(\lambda \omega\). This raises two questions. First, do we need to generalize their results? The answer is
no, since we do not consider higher-order quantifications such as \((ΠX:*_s \rightarrow *_s, \cdots)\) in our
ADTs. Second, are their results still valid for \(\lambda\omega_3\)? We believe so. Parametricity for \(\lambda 2\) can be
(technically) justified by giving a PER model and showing parametricity holds in that
model [BFSS90]; such models can be easily extended to \(\lambda\omega_3\) (see for example [Pol94]), which
suggests the same notion of parametricity is valid for \(\lambda\omega_3\). Furthermore, we use only a part
of the theory described in the literature as we explain below Definition 6.4.4.1.

We first give the axiom scheme for polymorphic functions of a certain form, and then apply
this scheme for existential types, which gives the principle \(par\Sigma\) for proving equality of
implementations of abstract datatypes we used in Section 6.4.3. This is one application of
general parametricity; other applications of general parametricity occur in Section 6.7.5.

**Definition 6.4.4.1 (Parametricity)** \(par\Pi I\) is LOLPLUS extended with the set of axioms containing

\[
parPi I : \forall f : (ΠX:*_s, I X). \forall Y,Z:*_s, \forall R : Y \rightarrow Z \rightarrow *_p, \text{Sim}_I Y Z R \ f Y \ (f Z)
\]

for every simply-typed \(I : *_s \rightarrow *_s\).

So, given a polymorphic function \(f\), every relation \(R\) between \(Y\) and \(Z\) is a simulation between
\(f Y\) and \(f Z\). So \(f Y\) and \(f Z\) are related if \(Y\) and \(Z\) are related.

The axiom given in the literature [PA93] is more general in that it quantifies over all
functions of type

\[
ΠX:\!*_s, \ldots, ΠX_n:*_s T
\]

We refrain from this generality, since the corresponding definition of \(\text{Sim}\) is more complex
and much harder to formalize in Yarrow. These are the main reasons for introducing the
limitation to simply-typed type-constructors in Definition 6.4.2.1.

**Example 6.4.4.2** Taking \(I \equiv \lambda X:*_s, X \rightarrow \text{Nat}\), we obtain

\[
\forall f : (ΠX:*_s, X \rightarrow \text{Nat}). \forall Y,Z:*_s, \forall R : Y \rightarrow Z \rightarrow *_p, \forall y : Y, \forall z : Z, R y z \implies f Y \ y = \text{Nat} \ f Z \ z.
\]

Taking \(R \equiv \lambda y : Y, \lambda z : Z. \text{True}\) and simplifying, gives

\[
\forall f : (ΠX:*_s, X \rightarrow \text{Nat}). \forall Y,Z:*_s, \forall y : Y, \forall z : Z, f Y \ y = \text{Nat} \ f Z \ z.
\]

In other words, a polymorphic function of type \((ΠX:*_s, X \rightarrow \text{Nat})\) always delivers the same
number; it cannot make any distinction based on its arguments.

**Example 6.4.4.3** Taking \(I \equiv \lambda X:*_s, X \rightarrow X\), we obtain

\[
\forall f : (ΠX:*_s, X \rightarrow X). \forall Y,Z:*_s, \forall R : Y \rightarrow Z \rightarrow *_p, \forall y : Y, \forall z : Z, R y z \implies R (f Y y) (f Z z).
\]

Now take any \(f : \Pi X:*_s, X \rightarrow X\), any \(B : *_s\) and any \(b : B\). Substitute \(Y \equiv B\), \(Z \equiv B\),
\(R \equiv \lambda u,v : B. (u =_B b), y \equiv b\) and \(z \equiv b\) and simplify. This gives

\[
f B b =_B b.
\]

In other words, every function of type \(ΠX:*_s, X \rightarrow X\) is (extensionally) equal to the polymorphic
identity.

Now we consider the impact of parametricity for existential types, simply by considering
the \text{pack} (_)\) construction as a polymorphic function.
Corollary 6.4.4.4 Consider a simply-typed $I : * \rightarrow *$. We define

$$\text{pack}_I := \lambda X:*_I, \lambda \text{ops} : I X. \text{pack} \langle X, \text{ops} \rangle \text{ in } \Sigma X:*_I, I X$$

$$\text{ Sim}_I Y Z R y z \implies \text{ pack } \langle Y, y \rangle =_{\Sigma X:*_I, I X} \text{ pack } \langle Z, z \rangle$$

Since $\text{pack}_I$ is a polymorphic function, and $(\lambda X:*_I, I X \rightarrow (\Sigma X:*_I, I X))$ is simply-typed, we can use the corresponding $\text{parPi}$ axiom on $\text{pack}_I$. This gives, after unfolding the definition of $\text{pack}_I$:

$$\text{parSigma}_I := \vdots : \forall Y, Z:*_I. \forall R : Y \rightarrow Z \rightarrow *_I. \forall y : I Y. \forall z : I Z. \text{ Sim}_I Y Z R y z \implies \text{ pack } \langle Y, y \rangle =_{\Sigma X:*_I, I X} \text{ pack } \langle Z, z \rangle$$

This is exactly the axiom given in Section 6.4.3. In other words, $\text{parSigma}$ does not need to be introduced as an axiom; it is a corollary of the general scheme given by $\text{parPi}$.

Parametricity for existential types as derived in [PA93] is stronger than the principle we have derived here, but this principle suffices for our purposes.

Remark 6.4.4.5 (No liberal $\Sigma$ elimination rule)

Now we explain why parametricity is incompatible with the ($\Sigma$-liberal-elim) rule as indicated in the discussion in Section 6.1.2. This is the liberal elimination rule:

$$(\Sigma\text{-liberal-elim}) \quad (\Gamma, X : K, x : B \vdash C : C) \quad \Gamma \vdash a : \Sigma X : K. B \quad \Gamma \vdash C : s \quad \frac{}{\Gamma \vdash \text{unpack } a \text{ as } \langle X, x \rangle \text{ in } c \vdash C}$$

First we give a counterexample, i.e. we show that the combination of ($\Sigma$-liberal-elim) and parametricity leads to inconsistency. Then we give a more informal, and hopefully intuitive, explanation of why ($\Sigma$-liberal-elim) is incompatible with parametricity.

For the counterexample, we take in the elimination rule $s = *_I$, and consider the existential type $\Sigma X:*_I, I X$, where the interface $I$ is defined as follows.

$$I := \lambda X:*_I, \{ l : X, r : X \}$$

This existential type, seen as interface of an ADT, is rather peculiar, since there is no observable behaviour; there is no program that can distinguish between implementations of this ADT. Indeed, one can easily prove using the parametricity scheme that

$$\text{pack } \langle R_1, \text{ops1} \rangle =_{\Sigma X:*_I, I X} \text{ pack } \langle R_2, \text{ops2} \rangle \quad \text{(i)}$$

for any $R_1, R_2 : *_I$ and any $\text{ops1} : I R_1$ and $\text{ops2} : I R_2$ (take for the simulation relation $\lambda r_1 : R_1, \lambda r_2 : R_2. \text{True}$). For example, consider the following two implementations.

$$\text{imp1} := \text{pack } \langle \text{Bool}, \{ l = \text{false}, r = \text{false} \} \rangle : \Sigma X:*_I, I X$$

$$\text{imp2} := \text{pack } \langle \text{Bool}, \{ l = \text{true}, r = \text{false} \} \rangle : \Sigma X:*_I, I X$$

By instantiating (i) we have

$$\text{imp1} =_{\Sigma X:*_I, I X} \text{imp2} \quad \text{(ii)}$$
Using \((\Sigma\text{-liberal-elim})\) we can make a predicate \(\text{Spec}\) that distinguishes two implementations of this existential type:

\[
\text{Spec} := \lambda \text{imp}:(\Sigma X:*_p, I X). \text{unpack} \text{ imp as } \langle \text{Rep}, \text{ops} \rangle \text{ in ops}.1 =_{\text{Rep}} \text{ops}.r
\]

Note that the predicate \(\text{Spec}\) is only well-typed using \((\Sigma\text{-liberal-elim})\), but not with \((\Sigma\text{-elim})\).

Now, we reduce the expressions \(\text{Spec imp}_i\) for \(i = 1, 2\).

\[
\begin{align*}
\text{Spec imp}_1 & \Rightarrow_{\beta_\delta} \text{unpack} \left(\text{pack} \langle \text{Bool}, \{1 = \text{false}, r = \text{false}\} \rangle\right) \text{ as } \langle \text{Rep}, \text{ops} \rangle \text{ in ops}.1 =_{\text{Rep}} \text{ops}.r \\
& \Rightarrow_{\beta} \text{false} =_{\text{Bool}} \text{false}
\end{align*}
\]

\[
\begin{align*}
\text{Spec imp}_2 & \Rightarrow_{\beta_\delta} \text{true} =_{\text{Bool}} \text{false}
\end{align*}
\]

So we have \(\text{Spec imp}_1\) and \(\neg(\text{Spec imp}_2)\). But with (ii) we may replace \(\text{imp}_1\) by \(\text{imp}_2\) in any proposition, so we have also \(\text{Spec imp}_2\), and we have a contradiction!

This counterexample to show inconsistency is contrived, but Remark 6.5.1.1 shows this inconsistency with a quite sensible counterexample.

Now we come to a more informal explanation. Parametricity for existential types says: two implementations that are indistinguishable by programs, i.e. that have the same behaviour, i.e. between which there exists a simulation relation, are Leibniz’ equal. In the example above, \(\text{imp}_1\) and \(\text{imp}_2\) are indistinguishable by programs, which leads by parametricity to equation (ii). Equality of \(\text{imp}_1\) and \(\text{imp}_2\) means that every predicate that holds for \(\text{imp}_1\) holds also for \(\text{imp}_2\) and vice versa, and therefore \(\text{imp}_1\) and \(\text{imp}_2\) should be indistinguishable by propositions, too.

In \(\lambda \omega^*_L\) propositions have a far greater power of discernment than programs. For example, it is possible to make a predicate of type \(\Pi X:*_p, \text{Bool}\) that expresses that its parameter \(X\) has two elements, whereas it is impossible to make a program \(\Pi X:*_p, \text{Bool}\) that only returns \text{true} if \(X\) has two elements. Another example is the presence of a polymorphic equality of type \(\Pi X:*_p, X \rightarrow X \rightarrow *_p\) in the programming logic (viz. Leibniz’ equality), whereas it is impossible to define a program \(\Pi X:*_p, X \rightarrow X \rightarrow \text{Bool}\) that delivers only \text{true} if applied to two values which are the same.

Now, suppose we have the liberal elimination rule. This allows a proposition \(P\) to inspect implementations directly, because

\[
\text{unpack} \left(\text{pack} \langle A,a \rangle\right) \text{ as } \langle X,x \rangle \text{ in } P \Rightarrow_{\beta} P[X := A][x := a]
\]

Since propositions can now inspect implementations directly, they can use their power of discernment to distinguish between implementations of ADTs that cannot be distinguished by programs. In our example above, \(\text{Spec}\) distinguishes \(\text{imp}_1\) and \(\text{imp}_2\), although \(\text{imp}_1\) and \(\text{imp}_2\) have the same behaviour. But we stated that implementations with the same behaviour should be indistinguishable by propositions, so we have a contradiction.

So, the liberal rule allows propositions to inspect implementations directly. This makes it possible to distinguish between implementations that have exactly the same behaviour, and are equal by parametricity. If we can distinguish between equal values the system is inconsistent. Without the liberal rule, no such direct inspection is possible, and we keep the system consistent. \(\Box\)
6.5 Example: Bags

In this section we consider another example of an ADT, namely finite bags of natural numbers. Just as in the introduction of this chapter and in the example of stacks, the ADT is treated in 5 steps.

1. The user gives the interface.

2. The user gives the specification.

3. The implementor gives an implementation.

4. The implementor gives a correctness proof of the implementation.

5. The user uses the implementation and its correctness proof.

In fact, we will consider three possible implementations of bags. Each implementation serves to explain a problem occurring in step 4, namely that it is difficult or impossible for the implementor to prove his implementation correct. The origin of these difficulties is that there is no one-to-one correspondence between concrete values and abstract values (for our implementation of stacks, we have this correspondence, see Figure 6.1).

There are two reasons why there may not be a one-to-one correspondence. First, several concrete values may correspond with one abstract value. This problem occurs in our first implementation (Section 6.5.1), where bags are implemented as (unsorted) lists; all permutations of a list correspond to the same bag. Here different notions of equality on lists come into play, namely Leibniz’ equality, and the equivalence relation generated by permutation.

Second, some concrete values may not correspond with an abstract value at all. This problem appears in attempts of the correctness proof of the second implementation (Section 6.5.2), where bags are implemented as sorted lists; unsorted lists do not correspond to a bag. Here different notions of quantification come into play, namely quantification over all lists, and quantification over all sorted lists.

The two reasons of having no one-to-one correspondence, and the problems they cause, can occur together; this is shown in Section 6.5.3 with a third implementation.

The structure of the problems and solutions for each of the implementations is very similar. Therefore we refine our presentation of step 4 as follows.

4a. The description of the problem the implementor has in proving the specification.

4b. Presenting one solution to this problem: finding an alternative implementation that does satisfy the specification, and show that the original and the alternative implementation have the same behaviour, so that they can be considered equal (using parametricity), and hence we prove the original implementation also satisfies the specification.

The disadvantage with this solution is that it is unacceptable and sometimes even impossible to give another implementation.

4c. Presenting another solution to this problem: use a weaker specification, such that the implementation given in step 3 does satisfy the specification.

The problem with this approach is that it may be very awkward for the user to reason with this weaker specification.
4d. Show that there is a canonical way to give an alternative implementation, which can be used as in step 4b. This alternative implementation is generated with an additional axiom. This axiom postulates the existence of some new datatypes, namely quotient types (Section 6.5.1) and subset types (Section 6.5.2). We show that this solution combines the best of both worlds (solution 4b and solution 4c), since the implementor only has to show correctness with respect to the weaker specification (as in 4c), but the user has a strong specification available (as in 4b).

Since the interface for the second implementation will be a slightly extended version of the interface for the first implementation, we will present all 5 steps for each implementation.

6.5.1 Implementation 1

The development of this implementation and its correctness proof is joint work with Erik Poll, and appeared as article in [PZ99].

1. Interface

Again, we start with the interface. We consider the simplest case, where we have only an operation of adding an element to a bag, an operation to inspect how often an element occurs in a bag, and an empty bag. So the interface of bags, and the corresponding existential type, are defined as follows.

\[ \text{BagI} \quad := \quad \lambda X: \star. \{ \text{empty : } X, \text{add : } \text{Nat} \rightarrow X \rightarrow X, \text{card : } \text{Nat} \rightarrow X \rightarrow \text{Nat} \} \]

\[ \text{BagImp} \quad := \quad \Sigma X: \star, \text{BagI} \ X \]

\[ \quad : \quad \star \rightarrow \star \]

For future reference, we give here the principle of simulation for BagI.

\[ \text{Sim}_{\text{BagI}} \text{ Rep1 Rep2 (\sim) ops1 ops2} \]

\[ \iff \]

\[ \text{ops1.empty} \sim \text{ops2.empty} \land \]

\[ (\forall x : \text{Rep1}. \forall y : \text{Rep2}. \forall m : \text{Nat}. x \sim y \implies \text{ops1.add} \ m \ x \sim \text{ops2.add} \ m \ y) \land \]

\[ (\forall x : \text{Rep1}. \forall y : \text{Rep2}. \forall m : \text{Nat}. x \sim y \implies \text{ops1.card} \ m \ x =_{\text{Nat}} \text{ops2.card} \ m \ y) \]

2. Specification

We use the following specification, which we call naive, for reasons that will become clear later.

\[ \text{NaiveSpec} \quad := \quad \lambda \text{Bag} : \star. \lambda \text{ops} : \text{BagI} \text{ Bag.} \]

\[ \forall m, n : \text{Nat}. \forall \text{v} : \text{Bag.} \]

\[ \text{ops.card} \ m \ \text{ops.empty} =_{\text{Nat}} 0 \land \]

\[ \text{ops.card} \ m \ (\text{ops.add} \ m \ b) =_{\text{Nat}} S \ (\text{ops.card} \ m \ b) \land \]

\[ (\sim (m =_{\text{Nat}} n) \implies \text{ops.card} \ m \ (\text{ops.add} \ n \ b) =_{\text{Nat}} \text{ops.card} \ m \ b) \land \]

\[ \text{ops.add} \ m \ (\text{ops.add} \ n \ b) =_{\text{Bag}} \text{ops.add} \ n \ (\text{ops.add} \ m \ b) \]

\[ : \quad \Pi \text{Bag} : \star. \text{BagI} \text{ Bag} \rightarrow \star \]
The first three clauses specify the empty and add operations in terms of \textit{card}. We discuss the last clause, and the reasons for including it, more extensively.

Suppose \( b_1 \equiv \text{ops} \cdot \text{add} \ n \ (\text{ops} \cdot \text{add} \ b) \) and \( b_2 \equiv \text{ops} \cdot \text{add} \ n \ (\text{ops} \cdot \text{add} \ m \ b) \). By the last clause

\[
b_1 =_{\text{bag}} b_2,
\]

i.e. \( b_1 \) and \( b_2 \) are Leibniz’ equal. From the other clauses it follows only that

\[
\forall p: \text{Nat}. \ \text{ops} \cdot \text{card} \ p \ b_1 =_{\text{Nat}} \text{ops} \cdot \text{card} \ p \ b_2. \quad (i)
\]

One might argue that this is sufficient, since the programmer can only observe bags through the \textit{card} operation, and that therefore the last clause of the specification is superfluous. But this argumentation ignores the advantage of reasoning with direct Leibniz’ equality above reasoning about bags using the \textit{card} operation. Only if we have Leibniz’ equality of bags \( b_1 \) and \( b_2 \), we can replace \( b_1 \) by \( b_2 \) in any proposition. This makes it possible to apply the technique of program transformation.

For example, if a large program

\[
q_1 := \text{unpack impBag as (Bag, ops) in prog}
\]

contains a term \( b_1: \text{Bag} \) and we have \( b_1 =_{\text{Bag}} b_2 \), we can prove directly that \( q_1 \) and \( q_2 \) are equal if \( q_2 \) is \( q_1 \) with \( b_1 \) replaced by \( b_2 \). On the other hand, if we have only \( (i) \), then we can prove \( q_1 \) and \( q_2 \) equal only by traversing the whole program. Every time we encounter in \( q_1 \) a subterm \( \text{ops} \cdot \text{card} \ n \ b_1 \) we use \( (i) \) to prove that this is equal to the corresponding subterm \( \text{ops} \cdot \text{card} \ n \ b_2 \) in \( q_2 \). Every time we encounter in \( q_1 \) a subterm \( \text{ops} \cdot \text{add} \ n \ b_1 \) we use \( (i) \) to relate this subterm to the corresponding subterm in \( q_2 \), i.e. we show

\[
\forall n: \text{Nat}. \ \text{ops} \cdot \text{card} \ n \ b_1 =_{\text{Nat}} \text{ops} \cdot \text{card} \ n \ b_2, \quad (ii)
\]

so that we can use \( (ii) \) in the same way as \( (i) \). We use this step repeatedly until we encounter a \textit{card} operation, only then do we have an ordinary Leibniz’ equality of the two subterms. This traversal is very cumbersome, therefore we rather have the direct equality \( b_1 =_{\text{Bag}} b_2 \) in the specification than the “observational equality” \( (i) \).

We have gone through the effort of explaining the relevance of this clause, because it is exactly this clause that gives a problem when the implementor tries to prove correctness of his implementation (step 4a below), and that therefore forces us to develop a considerable theory about ADTs to solve this problem.

\textit{NaiveSpec} can be turned into a predicate on \textit{BagImp} as follows:

\[
\text{UserSpec} \triangleq \lambda \text{imp}: \text{BagImp}. \exists \text{Rep}: \text{*}. \exists \text{ops}: \text{BagI Rep}. \quad \text{imp} =_{\text{BagImp}} \text{pack} \ (\text{Rep}, \text{ops}) \land \text{NaiveSpec Rep ops}
\]

Clearly

\[
\text{NaiveSpec Rep ops} \implies \text{UserSpec (pack (Rep, ops))}.
\]

(But beware that the reverse implication does not necessarily hold.)
3. Implementation

The most straightforward implementation of Bag is one using (unsorted) list of numbers as representation.

$$\text{Rep1} := \text{List Nat}$$

$$\text{ops1} := \{\text{empty} = \text{nil Nat}, \text{add} = (\cdot)\text{Nat}, \text{card} = \text{count}\}$$

$$\text{imp1} := \text{pack (Rep1, ops1)}$$

$$\text{BagImp}$$

where \(\text{count} : \text{Nat} \to \text{List Nat} \to \text{Nat}\) counts the number of occurrences of a given number in a list.

An essential property of this representation is that several concrete representations (e.g. the lists \([1, 2, 3]\) and \([2, 1, 3]\)) correspond with one abstract value (e.g. the bag \([1, 2, 3]\)). In this case, all concrete representations which are permutations correspond to the same abstract value. So some information is hidden: the order of the elements in the representation type (list) is not visible to the user. Note the contrast with the implementation of stacks, where no information was hidden.

4a. The problem with the naive specification

The specification UserSpec might be what the user of the ADT wants, but it may be a problem for the implementor of the ADT to meet this specification.

Can we prove UserSpec \text{imp1}?

We could prove this by proving

$$\text{NaiveSpec Rep1 ops1}.$$ 

This means we have to prove, amongst others, that

$$\forall m, n : \text{Nat}. \forall b : \text{Rep1}. m; n; b = \text{Rep1}. n; m; b.$$ 

Unfortunately, this is not true, the lists \(m; n; b\) and \(n; m; b\) are only equal if \(n\) equals \(m\). So we have \(\neg (\text{NaiveSpec Rep1 ops1})\). The problem with \text{NaiveSpec} is that it uses Leibniz' equality. The equality above make sense for Leibniz' equality of bags, but not for Leibniz' equality of lists. The equality above only makes sense for lists if we consider a weaker notion of equality on lists, namely \(\text{PermNat}\), the permutation relation as defined in Section 4.7. Leibniz' equality of concrete values (lists) does not correspond to Leibniz' equality of abstract values (bags), since several concrete values (lists) correspond to one abstract value (bag) (see step 3 above).

We now discuss two ways to solve (or avoid) the problem above. Neither of these is really acceptable, which is why we then propose to extend the logic to solve the problem in a satisfactory way.
4b. One solution: Finding another implementation

Recall that by definition of UserSpec

\[ UserSpec \text{ imp1 } \iff \exists \text{Rep}. * \_ \exists \text{ops}. \text{BagI Rep.} \]
\[ \text{imp1} =_{\text{BagImp}} \text{pack} (\text{Rep}, \text{ops}) \land \]
\[ \text{NaiveSpec Rep ops} \]

So we can prove UserSpec by finding another implementation pack (Rep, ops) of the ADT such that \text{imp1} =_{\text{BagImp}} \text{pack} (\text{Rep}, \text{ops}) for which we can prove NaiveSpec Rep ops.

It turns out that such an implementation exists, for example the implementation which represent bags as sorted lists. So

\[
\text{RepSort} := \text{List Nat} \\
\text{opsSort} := \{ \text{empty} = \text{nil Nat}, \text{add} = \text{insert}, \text{card} = \text{count} \} \\
\text{impSort} := \text{pack} (\text{RepSort}, \text{opsSort}) \\
\text{BagImp} 
\]

where insert inserts a natural number in a sorted list keeping it sorted (see ...). Since we have

\[
\forall m, n : \text{Nat. } \forall b : \text{List Nat. } (\text{insert } m (\text{insert } n b)) =_{\text{List Nat}} (\text{insert } n (\text{insert } m b))
\]

we can prove

\[
\text{NaiveSpec.opsSort} := \ldots : \text{NaiveSpec RepSort opsSort} \\
\text{UserSpec.impSort} := \ldots : \text{UserSpec impSort}
\]

The reason we can prove NaiveSpec for this implementation is due to the fact that for this particular implementation Leibniz’ equality of concrete (sorted) lists coincides with Leibniz’ equality of the abstract bags they represent.

Using parametricity we can prove

\[
\text{imp1.is.impSort} := \ldots : \text{imp1} =_{\text{BagImp}} \text{impSort}
\]

namely by showing that PermNat is a simulation between the two representations. Now UserSpec imp1 follows from UserSpec impSort and imp1 = impSort:

\[
\text{clumsy.UserSpec.imp1} := \ldots : \text{UserSpec imp1}
\]

There are obvious drawbacks to this way of proving UserSpec imp1. First, it is not acceptable that to prove correctness of our original implementation imp1 we should have to come up with a second implementation impSort. Moreover, it may not always be possible to find a second implementation that does meet the specification, i.e. for which concrete and abstract equality coincide! For example, for a generic datatype Bag X of bags over an arbitrary type X we would have a problem; there is no way to extend the representation using sorted lists to arbitrary types, since there is no generic sorting algorithm for arbitrary types.
Remark 6.5.1.1 We can also use impSort to show inconsistency of the liberal \( \Sigma \) elimination rule (see Remark 6.4.4.5). If we have this liberal rule, the following propositions can be formed:

\[
\begin{align*}
\text{unpack impSort as } (\text{Rep, ops}) & \text{ in NaiveSpec Rep ops, and} \\
\text{unpack imp1 as } (\text{Rep, ops}) & \text{ in NaiveSpec Rep ops.}
\end{align*}
\]

These propositions reduce to NaiveSpec RepSort opsSort and NaiveSpec Rep1 ops1 respectively. The former proposition is true, as shown above, since insert is "commutative", but the latter is false, since \( \text{insertNat} \) is not "commutative". By parametricity \( \text{imp1} =_{\text{StackImp}} \text{impSort} \), so both propositions are equivalent, and we have a contradiction. \( \square \)

4c. Another solution: Using a weaker specification

The best one could prove for imp1 is that

\[
\forall m, n : \text{Nat}. \forall l : \text{List Nat}. \text{PermNat} (m; n; l) (n; m; l).
\]

Note that \( \text{PermNat} \) is a bisimulation for ops1, i.e.

\[
\text{Bisim}_{\text{Perm-ops1}} : \cdots : \text{Bisim}_{\text{Bag1 Rep1 PermNat ops1}}
\]

since

\[
\begin{align*}
\text{PermNat} (\text{nil Nat}) (\text{nil Nat}) & \land \\
\forall m : \text{Nat}. \forall l, l' : \text{List Nat}. \text{PermNat} (l l') \Rightarrow \text{PermNat} (m l) (m; l') & \land \\
\forall m : \text{Nat}. \forall l, l' : \text{List Nat}. \text{PermNat} (l l') & \Rightarrow \text{count} m l =_{\text{Nat}} \text{count} m l'
\end{align*}
\]

With this in mind, one could propose a weaker specification for bags (that we call ImplemSpec as it is the specification for the implementor):

\[
\text{ImplemSpec} := \lambda \text{Rep:*}. \lambda \text{ops:Bag1 Rep}. \exists (\approx) : \text{Rep} \rightarrow \text{Rep} \rightarrow *_p' . \\
\text{Spec Rep} (\approx) \text{ ops} \land \text{Bisim}_{\text{Bag1 Rep}} (\approx) \text{ ops} \land \text{IsER}_{\text{Bag}} (\approx)
\]

\[
\begin{align*}
\text{ImplemSpec} & := \lambda \text{Rep:*}. \lambda (\approx) : \text{Rep} \rightarrow \text{Rep} \rightarrow *_p . \lambda \text{ops:Bag1 Rep}. \\
& \forall m, n : \text{Nat}. \forall r : \text{Rep}. \\
& \text{ops-card} m \text{ ops-empty} =_{\text{Nat}} 0 \land \\
& \text{ops-card} m (\text{ops-add} m r) =_{\text{Nat}} S (\text{ops-card} m r) \land \\
& (\langle m =_{\text{Nat}} n \rangle \Rightarrow \text{ops-card} m (\text{ops-add} n r) =_{\text{Nat}} \text{ops-card} m r) \land \\
& \text{ops-add} m (\text{ops-add} n r) \approx \text{ops-add} n (\text{ops-add} m r) \\
& : \text{ImplemSpec:*}, (\text{Rep} \rightarrow \text{Rep} \rightarrow *_p) \rightarrow \text{Bag1 Rep} \rightarrow *_p
\end{align*}
\]

and \( \text{IsER}_{\text{Bag}} (\approx) \) formalizes that \( \approx \) is an equivalence relation on \( \text{Rep} \) (see Section 6.7.1 for a formal definition).

Note that \( \text{NaiveSpec Rep} =_{\beta} (\text{Spec Rep} (\approx_{\text{Rep}})) \) for every \( \text{Rep} \); \( \text{NaiveSpec} \) is called naive since the equality used is fixed, viz. Leibniz' equality, whereas \( \text{Spec} \) is called abstract since it abstracts over the equality used.
Turning $ImplSpec$ into a predicate $WeakUserSpec$ on $BagImp$,

$$WeakUserSpec := \lambda bagImp: BagImp. \\
\exists Rep:*$, $\exists ops: BagI Rep. \exists (\simeq): Rep \rightarrow Rep \rightarrow *_p. \\
\text{bagImp} = BagImp \text{ pack } (\text{Rep}, \text{ops}) \land \\
\text{Spec Rep} (\simeq) \text{ ops } \land \\
\text{Bisim}^{BagI} \text{ Rep} (\simeq) \text{ ops } \land \\
\text{IsER}_{\text{Bag}} (\simeq)$$

\[ : BagImp \rightarrow *_p \]

The implementor of the ADT will be happy with this weaker specification, as it is possible to prove

$$\text{SpecPermOps1} := \ldots : \text{Spec Rep1 PermNat ops1}$$
$$\text{ImplSpecOps1} := \ldots : \text{ImplSpec Rep1 ops1}$$

and hence

$$WeakUserSpec_{\text{imp1}} := \ldots : WeakUserSpec \text{ imp1}$$

The user of the ADT on the other hand will be less happy with $WeakUserSpec$. Rather than using the standard Leibniz' equality of bags, the user has to reason about bags using some bisimulation $\simeq$ as notion of equality for bags. It seems an unnecessary complication: there is no reason why the user should not use Leibniz' equality instead of $\simeq$. Indeed, this is precisely the abstraction that the abstract datatype is supposed to provide.

4d. Quotient algebras

Given that the two solutions discussed above are not satisfactory, we will now present an extension of the logic $PAR$ that provides a satisfactory solution of the problem.

What we really want is a way to relate the two specifications, $ImplSpec$ and $UserSpec$, by proving

$$\forall Rep:*$, $\forall ops: BagI Rep. \text{ImplSpec Rep ops } \implies UserSpec (\text{pack } (\text{Rep}, \text{ops})).$$

Then the implementor of the ADT would only have to establish $ImplSpec$, and the user of the ADT could assume $UserSpec$. So we have to prove

$$\text{Spec Rep} (\simeq) \text{ ops } \land \text{Bisim}^{BagI} \text{ Rep} (\simeq) \text{ ops } \land \text{IsER}_{\text{Bag}} (\simeq)$$

$$\implies UserSpec (\text{pack } (\text{Rep}, \text{ops}))$$

for arbitrary $\text{Rep: *}$, and $\text{ops: BagI Rep}$ and $(\simeq): \text{Rep} \rightarrow \text{Rep} \rightarrow *_p$. A way to do this is to consider quotients of types, as we will now explain.

Suppose $\text{Spec Rep} (\simeq) \text{ ops}$ and $\text{Bisim}^{BagI} \text{ Rep} (\simeq) \text{ ops}$ and $\text{IsER}_{\text{Bag}} (\simeq)$. Assume there exists some quotient type $\text{Rep}/\simeq$, i.e. a type with as elements $\simeq$-equivalence classes of $\text{Rep}$. Note that $\text{Bisim}^{BagI} \text{ Rep} (\simeq) \text{ ops}$ says that the functions $\text{ops-empty}$, $\text{ops-add}$, and $\text{ops-card}$ respect $\simeq$-equivalence classes. This means that these functions on $\text{Rep}$ induce associated
functions on $\text{Rep}/\simeq$. We will refer to the record of associated functions as $[\text{ops}]_\simeq$, so $[\text{ops}]_\simeq : \text{BagI} \ (\text{Rep}/\simeq)$. By the principle of simulation it follows that

$$\text{pack} \ (\text{Rep}, \text{ops}) =_{\text{BagI}} \text{pack} \ (\text{Rep}/\simeq, [\text{ops}]_\simeq), \quad (i)$$

taking the obvious simulation relation between $\text{Rep}$ and $\text{Rep}/\simeq$. The interesting thing about the quotient is that it follows from $\text{Spec Rep} (\simeq) \text{ops}$ that

$$\text{Spec Rep}/\simeq \ (\simeq_{\text{Rep}}) [\text{ops}]_\simeq. \quad (ii)$$

Hence $\text{NaiveSpec Rep}/\simeq [\text{ops}]_\simeq$, so $\text{UserSpec} (\text{pack} \ (\text{Rep}/\simeq, [\text{ops}]_\simeq))$, and so by (i) it follows that $\text{UserSpec} (\text{pack} \ (\text{Rep}, \text{ops}))$. Note that this proof goes along the lines as indicated in Solution 1; $\text{Rep}/\simeq$ and $[\text{ops}]_\simeq$ provide a canonical alternative implementation.

We could consider adding quotient types to the syntax of $\lambda \omega^+_T$. Quotient types have been considered as extensions for type theories. For example, quotient types are available in Nuprl [C+86] and HOL [GM93], and have been proposed as extensions of other type theories, e.g. in [Hof95, Bar93]. The disadvantage of this approach is that it requires a lot of work: apart from the new syntax also new reduction and typing rules have to be given, and the meta-theory should be adapted accordingly.

Instead, we simply add an axiom to the logic stating that quotients exist. More precise: an axiom stating that for every datatype $R$, equivalence relation $\simeq$ and operations $\text{ops}R$ that respect $\simeq$, there is a quotient consisting of a datatype $Q$ (i.e. $R/\simeq$) and operations $\text{ops}Q$ (i.e. $[\text{ops}]_\simeq$):

$$\forall \ast : \text{Q}. \forall \ast : R \rightarrow R \rightarrow \ast. \forall \text{ops}R: \text{BagI} R. \text{IsER}R (\simeq) \Rightarrow \text{Bisim}_{\text{BagI}} R (\simeq) \text{ops}R \Rightarrow \exists Q : \text{Q}. \exists \text{ops}Q: \text{BagI} Q. \text{IsQuotAlg}_{\text{BagI}} Q \text{ops}R (\simeq) \text{ops}Q$$

Here $\text{IsQuotAlg}_{\text{BagI}} Q \text{ops}R (\simeq) \text{ops}Q$ expresses that $Q$ is the quotient type $R/\simeq$, and that $[\text{ops}]_\simeq$ is the set of the operations corresponding to $[\text{ops}]_\simeq$. (In algebraic terms, $Q, \text{ops}Q$ equals the quotient algebra $(R/\simeq, [\text{ops}]_\simeq)$; hence the name $\text{IsQuotAlg}$.)

In $\lambda \omega^+_T$, we do not have syntax for quotients. In order to define $\text{IsQuotAlg}$, we first consider the relation between $R$ and $R/\simeq$. There is the representation function that maps every element $r$ of $R$ to the equivalence class $[r]_\simeq$ in $R/\simeq$. Furthermore, this function is surjective (the only elements of $R/\simeq$ are equivalence classes), and for all elements $r, r' : R$ we have $r \simeq r' \iff [r]_\simeq =_R [r']_\simeq$. So the datatype $Q$ is the quotient $R/\simeq$ if we have a function $\text{surj}: R \rightarrow Q$ with the following two properties.

1. $\text{surj}$ is a surjection: $\text{IsSurjection}_{\text{Q}, \text{Q}} \text{surj}.$

$(\text{IsSurjection})$ is formally defined in Section 6.7.1).

2. $\text{surj}$ is injective with respect to $\simeq$:

$$\forall r, r' : R. r \simeq r' \iff \text{surj} r =_\text{Q} \text{surj} r'.$$

Now we come to the operations on the quotient $\text{ops}Q$. We have assumed the operations $\text{ops}R$ respect $\simeq$, i.e. $\simeq$ is a bisimulation for $\text{ops}R$, so:

$$\text{Sim}_{\text{BagI}} R R (\simeq) \text{ops}R \text{ops}R.$$
Hence we have that the representation function is a so-called morphism between \( \text{opsR} \) and \([\text{opsR}])\_\sim\), i.e.

\[
\text{Sim}_{\text{BagI}} \ (R/\sim) (\lambda r: R. \lambda q: R/\sim. \ [r]_\sim =_{R/\sim} q) \ \text{opsR} [\text{opsR}]_\sim.
\]

Replacing \( R/\sim \) by \( Q, [\text{opsR}]_\sim \) by \( \text{opsQ} \) and the representation function by \( \text{surj} \), we get:

3. \( \text{surj} \) is a morphism:

\[
\text{Sim}_{\text{BagI}} \ R Q (\lambda r: R. \lambda q: Q. \ \text{surj} r =_Q q) \ \text{opsR} \ \text{opsQ}.
\]

So we define \( \text{IsQuotAlg} \ R \ \text{opsR} (\sim) Q \ \text{opsQ} \) as the existence of a function \( \text{surj} \) with properties 1, 2 and 3. This leads to the following formal definition.

**Definition 6.5.1.2** \( \text{QUOT}_{\text{BagI}} \) is \( \text{PAR} \) extended with the following definition and axiom

\[
\text{IsQuotAlg}_{\text{BagI}} \ := \ \lambda R: * \cdot \lambda \text{opsR}: \text{BagI} R. \ \lambda (\sim): R \to R \to * \cdot \lambda Q: * \cdot \lambda \text{opsQ}: \text{BagI} Q.
\]

\[
\exists \text{surj}: R \to Q.
\]

\[
(\forall r, r^\prime: R. r \sim r^\prime \iff \text{surj} r =_Q \text{surj} r^\prime) \land
\]

\[
\text{IsSurjections}_{R, Q, \text{surj}} \land
\]

\[
\text{Sim}_{\text{BagI}} R Q (\lambda r: R. \lambda q: Q. q =_Q \text{surj} r) \ \text{opsR} \ \text{opsQ}.
\]

\[
\ \exists \text{II}: R, \ \text{BagI} R \to (R \to R \to *) \to (\text{IIQ}: {*} \cdot \text{BagI} Q \to *)
\]

\[
\text{exis\_QuotAlg}_{\text{BagI}} \ := \ \forall R: * \cdot \forall \text{opsR}: \text{BagI} R. \ \forall (\sim): R \to R \to * \cdot
\]

\[
\text{Bisim}_{\text{BagI}} R (\sim) \ \text{opsR} \implies \text{IsER}_{\text{BagI}} \ (\sim) \implies
\]

\[
\exists Q: * \cdot \exists \text{opsQ}: \text{BagI} Q. \ \text{IsQuotAlg}_{\text{BagI}} R \ \text{opsR} (\sim) Q \ \text{opsQ}.
\]

This axiom can be justified by a PER model. In fact, all types in PER models are quotient types! This axiom is specific for interface \( \text{BagI} \); in Section 6.6 we formulate a general axiom scheme.

Now we formalize the proofs given above, leading to \( \text{UserSpec} \ \text{imp1} \). First (i) and (ii) of page 142:

\[
\text{pack\_Quot} \ := \ \ldots \ := \ \forall R: * \cdot \forall \text{opsR}: \text{BagI} R. \ \forall (\sim): R \to R \to * \cdot \forall Q: * \cdot \forall \text{opsQ}: \text{BagI} Q.
\]

\[
\text{IsQuotAlg}_{\text{BagI}} R \ \text{opsR} (\sim) Q \ \text{opsQ} \implies
\]

\[
\text{pack} (R, \text{opsR}) =_{\text{BagI}} \text{pack} (Q, \text{opsQ}).
\]

\[
\text{Spec\_Sens} \ := \ \ldots \ := \ \forall R: * \cdot \forall \text{opsR}: \text{BagI} R. \ \forall (\sim): R \to R \to * \cdot \forall Q: * \cdot \forall \text{opsQ}: \text{BagI} Q.
\]

\[
\text{IsQuotAlg}_{\text{BagI}} R \ \text{opsR} (\sim) Q \ \text{opsQ} \implies
\]

\[
\text{Spec} R (\sim) \ \text{opsR} \implies \text{Spec} Q (=_Q \ \text{opsQ}.
\]

The first lemma is proved by parametricity and the definition of \( \text{IsQuotAlg} \). The proof of the second lemma depends on the actual form of \( \text{Spec} \), and does not generally hold for all specifications; in Section 6.6.2 we will give a set of specifications, called "sensible", for which this proposition holds. Hence the name \( \text{Spec\_Sens} \).

Using \( \text{exis\_QuotAlg}, \text{pack\_Quot} \) and \( \text{Spec\_Sens} \) we prove

\[
\text{Implen\_UserSpec} \ := \ \ldots \ := \ \forall \text{Rep}: {*} \cdot \forall \text{ops}: \text{BagI} \ \text{Rep}.
\]

\[
\text{ImplenSpec Rep ops} \implies \text{UserSpec} \ (\text{pack} (\text{Rep}, \text{ops})).
\]

and using this proposition we conclude that \( \text{imp1} \) satisfies the user's specification.

\[
\text{UserSpec\_imp1} \ := \ \ldots \ := \ \text{UserSpec} \ \text{imp1}.
\]
Overview of the 5 steps

Let us recapitulate how the ADT of bags is specified, implemented, proved correct and used. This overview also serves as a template for other ADTs; it is not specific for bags, as will be shown in Section 6.6.

1. The user gives the interface of the ADT.

\[ \text{BagI} \ := \ \ldots \ : \ *_{\mathcal{S}} \to *_{\mathcal{S}} \]

This determines the corresponding existential type.

\[ \text{BagImp} \ := \ \Sigma \text{Rep} : *_{\mathcal{S}}, \text{BagI Rep} \]

\[ : *_{\mathcal{S}} \]

2. The user gives the abstract specification of the ADT, which is abstracted over the notion of equality used, and proves that this specification is "sensible" (we elaborate on this in Section 6.6.2).

\[ \text{Spec} \ := \ \ldots \ : \ \Pi \text{Rep} : *_{\mathcal{S}}, (\text{Rep} \to \text{Rep} \to *_{\mathcal{P}}) \to \text{BagI Rep} \to *_{\mathcal{P}} \]

\[ \text{Spec_Sens} \ := \ \ldots \ : \ \forall \text{R}, \exists \text{Rep : BagI R, V(=) : R} \to \text{R} \to *_{\mathcal{P}}, \forall \text{Q : *_{\mathcal{S}}, Vops : BagI Q, IsQuo}_{\text{BagI}} \text{R opsR (=} \text{Q opsQ} \implies \text{Spec R (=} \text{opsR} \implies \text{Spec Q (=} \text{opsQ}} \]

\[ \text{Spec} \]

determines definitions of the user's and the implementor's specification and proof of the relation between the two.

\[ \text{UserSpec} \ := \ \lambda \text{bagImp : BagImp}. \ \exists \text{Rep : *_{\mathcal{S}}, Vops : BagI Rep.} \]

\[ \text{bagImp} =_{\text{bagImp}} \text{pack (Rep, ops)} \land \text{Spec Rep (=} \text{Ops} \land \text{ops} \]

\[ : \ \text{BagImp} \to *_{\mathcal{P}} \]

\[ \text{ImplemSpec} \ := \ \lambda \text{Rep : *_{\mathcal{S}}, Vops : BagI Rep.} \exists (=) : \text{Rep} \to \text{Rep} \to *_{\mathcal{P}}. \]

\[ \text{Spec Rep (=} \land \text{Bissim}_{\text{BagI}} \text{Rep (=} \land \text{IsER}_{\text{Rep}} (=) \]

\[ : \ \Pi \text{Rep : *_{\mathcal{S}}, BagI Rep} \to *_{\mathcal{P}} \]

\[ \text{Implem_UserSpec} \ := \ [\text{generic}] \]

\[ : \ \forall \text{Rep : *_{\mathcal{S}}, Vops : BagI Rep.} \]

\[ \text{ImplemSpec Rep ops} \implies \text{UserSpec (pack (Rep, ops))} \]

The existence of quotients is essential in our proof of \text{Implem_UserSpec}.

3. The implementor gives a representation type and the operations.

\[ \text{Rep1} \ := \ \ldots \ : \ *_{\mathcal{S}} \]

\[ \text{ops1} \ := \ \ldots \ : \ \text{BagI Rep1} \]

These determine the implementation value.

\[ \text{imp1} \ := \ \text{pack (Rep1, ops1)} \]

\[ : \ \text{BagImp} \]
4. The implementor gives an equivalence relation $SimI$ which is a bisimulation, and proves $opsI$ satisfies $Spec$ with $SimI$ as notion of equality.

$$SimI := \ldots \quad Rep1 \rightarrow Rep1 \rightarrow *_p$$
$$IsER_{Rep1SimI} := \ldots \quad IsER_{Rep1SimI}$$
$$ Bisim_{Rep1SimI} := \ldots \quad Bisim_{BagI} Rep1 SimI opsI$$
$$ Spec_{Rep1SimIopsI} := \ldots \quad Spec Rep1 SimI opsI$$

These determine a proof of $ImplemSpec$ and hence, using $Implem\_UserSpec$, of $UserSpec$.

$$ImplemSpec\_opsI := [\text{generic}]$$
$$\quad ImplemSpec Rep1 opsI$$

$$UserSpec\_impI := [\text{generic}]$$
$$\quad UserSpec impI$$

5. The user can prove correctness of programs that use an implementation by the following proof principle. This shows clearly that the user can reason Leibniz' equality of bags, and is not bothered with some equivalence relation.

$$principle := [\text{generic}]$$
$$\forall \text{imp:BagImp. UserSpec imp} \Rightarrow$$
$$\forall A:*_p, \forall Q:A \rightarrow *_p. \forall \text{body:}(\Pi X:*_p. \text{BagI} X \rightarrow A).$$
$$\quad (\forall X:*_p, \forall \text{ops:BagI} X. \text{Spec} X =_\text{X} \text{ops} \Rightarrow$$
$$\quad \quad Q (\text{body} X \text{ops}) \Rightarrow$$
$$\quad \quad Q (\text{unpack imp as} (X, \text{ops}) \text{ in body} X \text{ops})$$

**Summary**

Let us review the representation of bags as unsorted lists. An essential property of this representation is that several concrete representations (e.g. the lists $[1, 2, 3]$ and $[2, 1, 3]$) correspond with one abstract value (e.g. the bag $\{1, 2, 3\}$). This situation is sketched in Figure 6.3. Here we see that some information is hidden: the order of the elements in the representation type (list) is not visible to the user. Note the contrast with the implementation of stacks, where no information was hidden.
The user of the ADT does not want to be bothered by this, and wants to use Leibniz' equality of abstract values, even if their concrete representations differ (e.g. see the last conjunct of $Spec$). The implementor of the ADT must therefore define an equivalence relation on the representation type that puts concrete values representing the same abstract value in the same equivalence class. Of course he has to show that this equivalence relation is respected by all operations, i.e. the equivalence relation is a bisimulation for these operations. He uses a specification where Leibniz' equality is replaced by this equivalence relation.

In order to connect the implementor's specification and the user's specification we use quotients (or, rather, an axiom stating the existence of quotients). The equality on the quotient type corresponds to the equality on the abstract type, and a concrete implementation is proved equal to the implementation with quotients using parametricity.
6.5.2 Implementation 2

In this section we consider an other implementation: we represent bags by sorted lists. This different implementation has important consequences for the correctness proof. Now each abstract value (bag) corresponds with one concrete value (list), just like our implementation for stacks and unlike the implementation of bags as unsorted lists, but now there are concrete values that do not correspond with an abstract value, viz. the unsorted lists. We will see that instead of quotient algebras, we now need subset algebras.

The structure of this section is very similar to that of the first implementation. The only differences lie in step 4, when we consider the correctness of the implementation. For the purpose of the example, we consider a slightly extended version of bags, with an operation that delivers an upper bound of the elements of the bag.

1. Interface

The interface of bags and the corresponding type of implementations are now:

\[
\begin{align*}
\text{BagI} & := \lambda X:\ast, \emptyset \text{ empty} : X, \\
& \quad \text{add} : \text{Nat} \to X \to X, \\
& \quad \text{card} : \text{Nat} \to X \to \text{Nat}, \\
& \quad \text{bound} : X \to \text{Nat} [] \\
& \quad \ast \to \ast,
\end{align*}
\]

\[
\begin{align*}
\text{BagImp} & := \Sigma X:\ast, \text{BagI X}
\end{align*}
\]

2. Specification

The naive specification has an additional clause for the bound operation.

\[
\begin{align*}
\text{NaiveSpec} & := \lambda \text{Bag}:\ast, \lambda \text{ops}:\text{BagI Bag}, \\
& \quad \forall m, n : \text{Nat}. \forall b : \text{Bag}. \\
& \quad \text{ops} \cdot \text{card} m \text{ ops} \cdot \text{empty} = \text{Nat} 0 \land \\
& \quad \text{ops} \cdot \text{card} m (\text{ops} \cdot \text{add} m b) = \text{Nat} S (\text{ops} \cdot \text{card} m b) \land \\
& \quad (\neg (m = \text{Nat} n) \Rightarrow \text{ops} \cdot \text{card} m (\text{ops} \cdot \text{add} n b) = \text{Nat} \text{ ops} \cdot \text{card} m b) \land \\
& \quad \text{ops} \cdot \text{add} m (\text{ops} \cdot \text{add} n b) = \text{Nat} \text{ ops} \cdot \text{add} n (\text{ops} \cdot \text{add} m b) \land \\
& \quad (1 \leq \text{ops} \cdot \text{card} m b \Rightarrow m \leq \text{ops} \cdot \text{bound} b) \\
& \quad \Pi \text{Bag}:\ast, \text{BagI Bag} \to \ast_p
\end{align*}
\]

The last clause specifies the bound operation: the bound is greater or equal than any element occurring at least once in the bag. We will show that now this clause is the trouble-maker. 

\[\text{NaiveSpec}\] is again turned into a predicate on \(\text{BagImp}\) as follows:

\[
\begin{align*}
\text{UserSpec} & := \lambda \text{imp}:\text{BagImp}, \exists \text{Rep}:\ast, \exists \text{ops}:\text{BagI Rep}, \\
& \quad \text{imp} = \text{BagImp} \text{ pack} (\text{Rep}, \text{ops}) \land \\
& \quad \text{NaiveSpec} \text{ Rep ops} \\
& \quad \text{BagImp} \to \ast_p
\end{align*}
\]
CHAPTER 6. \(\lambda \omega^+_L\) AND MODELLING ABSTRACT DATATYPES

3. Implementation

The representation of Bag by sorted lists is defined as follows.

\[
\text{Rep2} := \text{List Nat} \\
\text{ops2} := \{ \text{empty = nil Nat,} \\
\text{add = insert,} \\
\text{card = count,} \\
\text{bound = last Nat 0} \} \\
\text{imp2} := \text{pack (Rep2, ops2)} \\
\text{BagImp}
\]

where \text{last a} delivers the last element of the list, or \text{a} if the list is empty.

An essential property of this representation is that not all concrete representations correspond with an abstract value, but every abstract value is represented by exactly one concrete value. So some information is hidden: all concrete representations that do not correspond with an abstract value are not visible to the user. Note the contrast with the implementation of stacks and the first implementation of bags, where all concrete values represent an abstract value.

4a. The problem with the naive specification

The specification \text{UserSpec} might be what the user of the ADT wants, but it may be a problem for the implementor of the ADT to meet this specification.

\textit{Can we prove UserSpec imp2 ?}

We could prove this by proving

\textit{NaiveSpec Rep2 ops2}.

This means we have to prove, amongst others, that

\[
\forall m: \text{Nat.} \forall b: \text{Rep2.1} \leq \text{count m b} \implies m \leq \text{last Nat 0 b}.
\]

This proposition states that in all lists, the value of every element is at most the value of the last element. This is clearly not true. But this statement is valid for the \textit{sorted} lists. So quantification over the concrete type does not correspond to quantification over the abstract type, since not all concrete values (lists) correspond to an abstract value (bag) (see point 3 above).

Note the difference between this problem and the problem in Section 6.5.1: here the problem lies in the quantification over all elements of the representation type, whereas in the previous section the problem lies in the Leibniz' equality.

Again, there are two ways to solve the problem above. Neither of these is really acceptable, which is why we then propose to extend the logic again to solve the problem in a satisfactory way.
4b. One solution: Finding another implementation

Again, we can prove \( UserSpec \) by finding another implementation \( \text{pack} \langle \text{Rep}, \text{ops} \rangle \) of the ADT such that \( \text{imp2} =_{\text{bagg}} \text{pack} \langle \text{Rep}, \text{ops} \rangle \) for which we can prove \( \text{NaiveSpec Rep ops} \). We will not pursue this solution, for the same reasons as in Section 6.5.1. First, it is not acceptable that to prove correctness of one implementation we have to come up with a second implementation. Second, it may not always be possible to find a second implementation that matches the specification.

4c. Another solution: Using a weaker specification

The best we could prove for \( \text{imp2} \) is that

\[
\forall m: \text{Nat}. \forall b: \text{Rep}2. \text{Ordered } b \implies S \ 0 \leq \text{ops2-card m b} \implies m \leq \text{ops2-bound b}.
\]

Note that \( \text{Ordered} \) is an \textit{invariant} for \( \text{ops2} \), i.e.

\[
\text{Ordered (ops2-empty)} \land \\
\forall m: \text{Nat}. \forall b: \text{Rep}2. \text{Ordered } b \implies \text{Ordered (ops2-add m b)}.
\]

Abstracting from the particular representation and invariant, we define:

\[
\text{IsInvar}_{\text{Bag}2} := \lambda \text{Rep}^*: \lambda \text{Inv: Rep} \to \text{*}, \lambda \text{ops: BagI Rep}.
\]

\[
\text{Inv ops-empty} \land \\
(\forall m: \text{Nat}. \forall r: \text{Rep}. \text{Inv r} \implies \text{Inv (ops-add m r)})
\]

\[
\Pi \text{Rep}^*: \lambda (\text{Rep} \to \text{*}) \to \text{BagI Rep} \to \text{*}
\]

With the notion of invariant in mind, one could propose a weaker specification for bags, namely

\[
\text{ImplSpec} := \lambda \text{Rep}^*: \lambda \text{ops: BagI Rep}. \exists \text{Inv: Rep} \to \text{*},
\]

\[
\text{Spec Rep Inv ops IsInvar}_{\text{Bag}2} \text{Rep Inv ops}
\]

\[
\Pi \text{Rep}^*: \lambda (\text{Rep} \to \text{*}) \to \text{BagI Rep} \to \text{*}
\]

where the abstract specification \( \text{Spec} \) is the specification \( \text{NaiveSpec} \) with each quantification over type \( \text{Rep} \) strengthened by the invariant:

\[
\text{Spec} := \lambda \text{Rep}^*: \lambda \text{Inv: Rep} \to \text{*}, \lambda \text{ops: BagI Rep}.
\]

\[
\forall m, n: \text{Nat}. \forall r: \text{Rep}. \text{Inv r} \implies
\]

\[
\text{ops-card m ops-empty} =_{\text{Nat}} 0 \land \\
\text{ops-card m (ops-add m r)} =_{\text{Nat}} S (\text{ops-card m r}) \land \\
(\neg (m =_{\text{Nat}} n) \implies \text{ops-card m (ops-add n r)} =_{\text{Nat}} \text{ops-card m r}) \land \\
\text{ops-add m (ops-add n r)} =_{\text{Rep}} \text{ops-add n (ops-add m r)} \land \\
(1 \leq \text{ops-card m r} \implies m \leq \text{ops-bound r})
\]

\[
\Pi \text{Rep}^*: \lambda (\text{Rep} \to \text{*}) \to \text{BagI Rep} \to \text{*}
\]

Note that

\[
\text{NaiveSpec Rep ops} \iff \text{Spec Rep (\lambda r: Rep. True) ops}
\]

for every \( \text{Rep} \) and \( \text{ops} \). We called \( \text{NaiveSpec} \) naive since the invariant used is the trivial one.
CHAPTER 6. $\lambda_\text{AT}^+$ AND MODELLING ABSTRACT DATATYPES

Turning Spec into a predicate WeakUserSpec on BagImp,

$$WeakUserSpec := \lambda \text{bagImp} : \text{BagImp}. \exists \text{Rep} : *, \exists \text{ops} : \text{BagI Rep}. \exists \text{Inv} : \text{Rep} \to *_p, \text{bagImp} = \text{BagImp}_{\text{bagImp}} \text{pack} \langle \text{Rep}, \text{ops} \rangle \land \text{Spec Rep Inv ops} \land \text{IsInv}_{\text{BagI Rep Inv ops}}$$

The implementor of the ADT will be happy with this weaker specification, as it is possible to prove

$$\text{Spec Ordered ops2} := \ldots : \text{Spec Rep2 Ordered ops2}$$
$$\text{ImplemSpec ops2} := \ldots : \text{ImplemSpec Rep2 ops2}$$

and hence

$$WeakUserSpec \text{ imp2} := \ldots : WeakUserSpec \text{ imp2}$$

The user of the ADT on the other hand will be less happy with WeakUserSpec. Every time he wants to use a part of the specification for some bag b, he has to prove that b satisfies the invariant. This seems an unnecessary burden: by the abstraction mechanism the invariant must hold for every bag b; the user can never construct a bag that does not satisfy the invariant.

4d. Subset algebras

Given that the two solutions discussed above are not satisfactory, we will now present an extension of the logic PAR that provides a satisfactory solution of the problem.

What we really want is a way to relate the two specifications, ImplemSpec and UserSpec, by proving

$$\forall \text{Rep} : *, \forall \text{ops} : \text{BagI Rep. ImplemSpec Rep ops} \Longrightarrow UserSpec (\text{pack} \langle \text{Rep}, \text{ops} \rangle).$$

We achieve this by proving

$$\text{Spec Rep Inv ops} \land \text{IsInv}_{\text{BagI Rep Inv ops}} \Longrightarrow UserSpec (\text{pack} \langle \text{Rep}, \text{ops} \rangle)$$

for arbitrary Rep : *, and ops : BagI Rep and Inv : Rep $\to *_p$. A way to do this is to consider subset types, as we will now explain.

Suppose Spec Rep Inv ops and IsInv_{BagI Rep Inv ops}. Assume the subset-type Rep|_Inv exists, which is informally defined as the set \{r : Rep | Inv r\}. Note that since Inv is an invariant, Rep|_Inv is closed under the operations opsempty and opssub. This means that these functions induce associated functions on Rep|_Inv. Of course, the other operations, opscard and ops-bound also have associated functions on Rep|_Inv (by just restricting the domain). We will refer to the record of all associated operations as ops|_Inv, so ops|_Inv : BagI (Rep|_Inv). By the principle of simulation it follows that

$$\text{pack} \langle \text{Rep}, \text{ops} \rangle = \text{BagImp}_{\text{bagImp}} \text{pack} \langle \text{Rep}|_{\text{Inv}}, \text{ops}|_{\text{Inv}} \rangle, \quad (i)$$
taking the obvious simulation relation between $\text{Rep}$ and $\text{Rep}_{|_{\text{Ins}}}$. The interesting thing about the subset is that it follows from $\text{Spec} \ \text{Rep} \ \text{Inv} \ \text{ops}$ that

$$\text{Spec} \ \text{Rep}_{|_{\text{Ins}}} \left( \lambda r : \text{Rep}_{|_{\text{Ins}}}. \ \text{True} \right) \ \text{ops}_{|_{\text{Ins}}} .$$

(ii)

Hence $\text{NaiveSpec} \ \text{Rep}_{|_{\text{Ins}}} \ \text{ops}_{|_{\text{Ins}}}$, and so, by (i), it follows that $\text{UserSpec} \ (\text{pack} \ (\text{Rep}, \ \text{ops}))$.

We could consider adding subset types to the syntax of the second-order lambda calculus. Subset types have been considered as extensions for type theories. For example, subset types are available in Nuprl [C+86], and have been proposed as extensions of PTSs in [Bar95]. The disadvantage of this approach is that it requires a lot of work: apart from the new syntax also new reduction and typing rules have to be given, and the meta-theory should be adapted accordingly.

Instead, we simply add an axiom to the logic stating that subsets exist. More precise: for every datatype $\mathcal{R}$, operations $\text{ops}_\mathcal{R}$ and invariant $\text{Inv}$ for $\text{ops}_\mathcal{R}$, there is a subset consisting of a datatype $\mathcal{S}$ (i.e. $\mathcal{R}_{|_{\text{Ins}}}$) and operations $\text{ops}_\mathcal{S}$ (i.e. $\text{ops}_\mathcal{R}_{|_{\text{Ins}}}$):

$$\forall \mathcal{R} : \mathcal{S} , \ \forall \text{Inv}_\mathcal{R} : \mathcal{R} \ \forall \text{ops}_\mathcal{R} : \text{BagI} \ \mathcal{R} . \ \text{IsInvarBag}_\mathcal{R} : \mathcal{R} \ \text{Inv} \ \text{ops}_\mathcal{R} \Rightarrow \exists \mathcal{S} : \mathcal{S} . \ \forall \text{ops}_\mathcal{S} : \text{BagI} \ \mathcal{S} . \ \text{IsSubsetAlg}_\mathcal{S} : \mathcal{R} \ \text{ops}_\mathcal{S} \ \text{Inv} \ \text{ops}_\mathcal{S} \ \text{ops}_\mathcal{S}

Here $\text{IsSubsetAlg}_\mathcal{R} : \mathcal{R} \ \text{ops}_\mathcal{R} \ \text{Inv} \ \text{ops}_\mathcal{S}$ expresses that $\mathcal{S}$ is the subset type $\mathcal{R}_{|_{\text{Ins}}}$, and that $\text{ops}_\mathcal{S}$ is the set of operations corresponding to $\text{ops}_\mathcal{R}_{|_{\text{Ins}}}$. (In algebraic terms, $(\mathcal{S}, \text{ops}_\mathcal{S})$ is the subset algebra $(\mathcal{R}_{|_{\text{Ins}}}, \text{ops}_\mathcal{R}_{|_{\text{Ins}}})$; hence the name $\text{IsSubsetAlg}$.)

In $\lambda \omega^+_L$ we do not have syntax for subsets. In order to define $\text{IsSubsetAlg}$, we first consider the relation between $\mathcal{R}$ and $\mathcal{R}_{|_{\text{Ins}}}$. There is an injection function that maps every element of $\mathcal{R}_{|_{\text{Ins}}}$ to an element of $\mathcal{R}$. Furthermore, the image of this function is $\text{Inv}$. So the datatype $\mathcal{S}$ is the subset $\mathcal{R}_{|_{\text{Ins}}}$ if we have a function $\text{inj} : \mathcal{S} \rightarrow \mathcal{R}$ with the following two properties.

1. $\text{inj}$ is a injection.

2. The image of $\text{inj}$ is $\mathcal{R}_{|_{\text{Inv}}}$:

$$\forall \mathcal{R} : \mathcal{R} \ \text{Inv} \ \mathcal{r} \iff \exists \mathcal{S} : \mathcal{S} . \ \text{inj} \ \mathcal{r} =_{\mathcal{R}} \ \mathcal{r} .$$

Now we come to the operations on the subset $\text{ops}_\mathcal{S}$. We have assumed that $\text{Inv}$ is an invariant for $\text{ops}_\mathcal{R}$, i.e.

$$\text{IsInvarBag}_\mathcal{R} : \mathcal{R} \ \text{Inv} \ \text{ops}_\mathcal{R} .$$

Hence we have that

$$\text{Sim}_{\text{BagI}} : \mathcal{R} (\mathcal{R}_{|_{\text{Ins}}}) \left( \lambda r : \mathcal{R} . \ \lambda s : \mathcal{R}_{|_{\text{Ins}}}. \ \mathcal{r} =_{\mathcal{R}} \mathcal{s} \right) \ \text{ops}_\mathcal{R} \ \text{ops}_{|_{\text{Ins}}} .$$

Replacing $\mathcal{R}_{|_{\text{Ins}}}$ by $\mathcal{S}$, $\text{ops}_\mathcal{R}_{|_{\text{Ins}}}$ by $\text{ops}_\mathcal{S}$, and inserting the injection, we get:

3. $\text{inj}$ is a morphism:

$$\text{Sim}_{\text{BagI}} : \mathcal{R} \ \mathcal{S} \left( \lambda r : \mathcal{R} . \ \lambda s : \mathcal{S}. \ \mathcal{r} =_{\mathcal{R}} \mathcal{inj} \mathcal{s} \right) \ \text{ops}_\mathcal{R} \ \text{ops}_\mathcal{S} .$$

So we define $\text{IsSubsetAlg}_\mathcal{R} \ \text{ops}_\mathcal{R} \ \text{Inv} \ \mathcal{S} \ \text{ops}_\mathcal{S}$ as the existence of a function $\text{inj}$ which properties 1, 2 and 3. This leads to the following formal definition.
Definition 6.5.2.1 \texttt{SUBSET_{BagI}} is \texttt{PAR} extended with the following definition and axiom:

\[
\begin{align*}
\text{IsSubsetAlg}_{\text{BagI}} & := \lambda R : \ast, \lambda \text{opsR:BagI} R. \lambda \text{Inv:R} \rightarrow \ast_p. \lambda S : \ast, \lambda \text{opsS:BagI} S. \\
& \quad \exists \text{inj:S} \rightarrow R. \\
& \quad \text{IsInjections}_{S,R} \ \text{inj} \land \\
& \quad (\text{Inv} \iff \text{Images}_{S,R} \ \text{inj}) \land \\
& \quad \text{Sim}_{\text{BagI}} R S (\lambda r : R. \lambda s : S. r =_S \ \text{inj} s) \ \text{opsR} \ \text{opsS} \\
& \quad \Pi R : \ast_p. \text{BagI} R \rightarrow (R \rightarrow \ast_p) \rightarrow (\Pi S : \ast. \text{BagI} S \rightarrow \ast_p)
\end{align*}
\]

\[
\begin{align*}
\text{exis\_SubsetAlg}_{\text{BagI}} & := \forall R : \ast, \forall \text{opsR:BagI} R. \forall \text{Inv:R} \rightarrow \ast_p. \\
& \quad \text{IsInvar}_{\text{BagI}} R \ \text{Inv} \ \text{opsR} \implies \\
& \quad \exists S : \ast, \exists \text{opsS:BagI} S. \ \text{IsSubsetAlg}_{\text{BagI}} R \ \text{opsR} \ \text{Inv} S \ \text{opsS}
\end{align*}
\]

The predicate \text{IsInjections}_{S,R} \ f \ expresses \ that \ f \ is \ a \ injective \ function, \ and \ \text{Images}_{S,R} \ f \ gives \ the \ image \ of \ a \ function \ as \ predicate. \ Both \ are \ formally \ defined \ in \ Section \ 6.7.1.

This axiom can be justified by a PER model. Intuitively, subsets exist in PER models because in these models types are interpreted as partial equivalence relations. This axioms is specific for interface \text{BagI}; in Section 6.6 we formulate a general axiom scheme.

Now we formalize the proofs given above, leading to \texttt{UserSpec imp2}. First (i) and (ii) on page 151:

\[
\begin{align*}
\text{pack\_Subset} & := \ldots : \forall R : \ast, \forall \text{opsR:BagI} R. \forall \text{Inv:R} \rightarrow \ast_p. \forall S : \ast, \forall \text{opsS:BagI} S. \\
& \quad \text{IsSubsetAlg}_{\text{BagI}} R \ \text{opsR} \ \text{Inv} S \ \text{opsS} \implies \\
& \quad \text{pack} (R, \text{opsR}) =_{\text{BagI}} \text{pack} (S, \text{opsS})
\end{align*}
\]

\[
\begin{align*}
\text{Spec\_Sens} & := \ldots : \forall R : \ast, \forall \text{opsR:BagI} R. \forall \text{Inv:R} \rightarrow \ast_p. \forall S : \ast, \forall \text{opsS:BagI} S. \\
& \quad \text{IsSubsetAlg}_{\text{BagI}} R \ \text{opsR} \ \text{Inv} S \ \text{opsS} \implies \\
& \quad \text{Spec} R \ \text{Inv} \ \text{opsR} \implies \text{Spec} S (\lambda S : S. \text{True}) \ \text{opsS}
\end{align*}
\]

The first lemma is proved by parametricity and definition of \text{IsSubsetAlg}. The proof of the second lemma depends on the actual form of \text{Spec}, and does not generally hold for all specifications; in Section 6.6.2 we will give a set of specifications, called "sensible", for which this proposition holds.

Using \text{exis\_SubsetAlg}, \text{pack\_Subset} and \text{Spec\_Sens} we prove

\[
\begin{align*}
\text{Implem\_UserSpec} & := \ldots : \forall \text{Rep : \ast}, \forall \text{ops:BagI} \ \text{Rep}. \\
& \quad \text{ImplemSpec} \ \text{Rep} \ \text{ops} \implies \text{UserSpec} (\text{pack} (\text{Rep}, \text{ops}))
\end{align*}
\]

and using this proposition we conclude that \text{imp2} satisfies the user’s specification.

\[
\text{UserSpec\_imp2} := \ldots : \text{UserSpec imp2}
\]

Overview of the 5 steps

Let us recapitulate how the ADT of bags is specified, implemented and proved correct, and used. This overview also serves as a template for other ADTs; it is not specific for bags.
6.5. EXAMPLE: BAGS

1. The user gives the interface of the ADT.

\[ \text{BagI} := \ldots : * \rightarrow * \]

This determines the corresponding existential type.

\[ \text{BagImp} := \Sigma \text{Rep} : * . \text{BagI Rep} \]

: \[*\]

2. The user gives the abstract specification of the ADT, which is abstracted over the invariant used, and proves that this specification is "sensible" (we elaborate on this in Section 6.6.2).

\[ \text{Spec} := \ldots : \Pi \text{Rep} : * . (\text{Rep} \rightarrow *^p) \rightarrow \text{BagI Rep} \rightarrow *^p \]

\[ \text{Spec.Sens} := \ldots : \forall \text{R} : * . \forall \text{opsR:BagI R. Inv:R} \rightarrow *^p . \forall \text{S} : * . \forall \text{opsS:BagI S.}
\]

\[ \text{IsSubsets}_{\text{bagI}} \text{ R opsR Inv S opsS} \Rightarrow \]

\[ \text{Spec R Inv opsR} \Rightarrow \]

\[ \text{Spec S (\lambda x:S. True) opsS} \]

These terms determine definitions of the user’s and the implementor’s specification and a proof of the relation between both.

\[ \text{UserSpec} := \lambda \text{bagImp:BagImp. } \exists \text{Rep} : * . \exists \text{ops:BagI Rep.}
\]

\[ \text{bagImp = bagImp.pack (Rep, ops)} \land \]

\[ \text{Spec Rep (\lambda r:Rep. True) ops} \]

: \[ \text{BagImp} \rightarrow *^p \]

\[ \text{ImplSpec} := \lambda \text{Rep} : * . \lambda \text{ops:BagI Rep. } \exists \text{Inv:Rep} \rightarrow *^p . \]

\[ \text{Spec Rep Inv ops} \land \]

\[ \text{IsInvar}_{\text{bagI}} \text{ Rep Inv ops} \land \]

\[ \exists r:Rep. \text{Inv } r \]

: \[ \Pi \text{Rep} : * . \text{BagI Rep} \rightarrow *^p \]

\[ \text{Impl/UserSpec} := [\text{generic}] \]

: \[ \forall \text{Rep} : * . \forall \text{ops:BagI Rep.}
\]

\[ \text{ImplSpec Rep ops} \Rightarrow \text{UserSpec (pack (Rep, ops))} \]

Subsets are essential in our proof of Impl/UserSpec.

3. The implementor gives a representation type and the operations.

\[ \text{Rep1} := \ldots : * \]

\[ \text{ops1} := \ldots : \text{BagI Rep1} \]

These determine the implementation value.

\[ \text{imp1} := \text{pack (Rep1, ops1)} \]

: \[ \text{BagImp} \]
4. The implementor gives an invariant \( \text{Inv1} \) and proves \( \text{Ops1} \) satisfies \( \text{Spec} \) with \( \text{Inv1} \) as invariant.

\[
\text{Inv1} := \ldots : \text{Rep1} \rightarrow *_p \\
\text{IsInvVarInv1} := \ldots : \text{IsInvVarBag1 Rep1 Inv1 Ops1}
\]

These determine a proof of \( \text{ImplemSpec} \) and hence of \( \text{UserSpec} \).

\[
\text{ImplemSpec_ops1} := \quad [\text{generic}] \\
\quad : \text{ImplemSpec Rep1 Ops1} \\
\text{UserSpec_impl1} := \quad [\text{generic}] \\
\quad : \text{UserSpec Impl1}
\]

5. The user can prove correctness of programs that use an implementation by the following proof principle. This shows clearly that the user can use the specification for all inhabitants of \( \text{Rep} \), and is not bothered with some invariant.

\[
\text{principle} := \quad [\text{generic}] \\
\quad : \text{Vimp:BagImp. UserSpec Imp} \implies \\
\forall X:*_p. \forall Q:A \rightarrow *_p. \forall \text{body}:(\Pi X:*_p. \text{Bag} I X \rightarrow A). \\
(\forall X:*_p. \forall \text{ops:BagI X. Spec X (\lambda x: X. True) ops} \implies Q (\text{body X ops})) \implies \\
Q (\text{unpack Imp as (X, ops)} in \text{body X ops})
\]

Summary

Let us review the representation of bags as sorted lists. An essential property of this representation is that not all concrete representations correspond with an abstract value, but every abstract value is represented by exactly one concrete value. This situation is sketched in Figure 6.4. Here we see that some information is hidden: all concrete representations that do not correspond with an abstract value are not visible to the user. Note the contrast with the implementation of stacks and the first implementation of bags, where all concrete values represent an abstract value.

The specification typically says that some propositions hold for all abstract values. Therefore these propositions should also hold for all concrete values that represent an abstract value. But for concrete values not visible to the user these propositions do not have to hold; the user cannot access them anyway.

The implementor gives an invariant that indicates which concrete values do represent an abstract value. He only proves that all concrete values that satisfy the invariant satisfy the specification. Note that the set of values satisfying the invariant may be larger than the set of values which are reachable through the operations (which we will see with the implementation of Section 6.5.3). The user does not want to be bothered by this invariant; the user cannot violate the invariant by virtue of the hiding mechanism. This informally justifies that the user's specification (without invariant) follows from the implementor's specification (with invariant).

In order to formally connect the implementor's and the user's specification, we use subsets (or, more precisely, axioms stating the existence of subsets). We take the subset of the representation type that satisfies the invariant, and the operations on this subset type. For all values in the subset type, the invariant trivially holds, so we can show the operations of the
Figure 6.4: Relation between concrete and abstract values for the implementation of bags with sorted lists.

subset satisfy the specification without invariant. By parametricity the original implementation and the implementation with subsets are equal, so the original implementation satisfies the user's specification too.
6.5.3 Implementation 3

The two previous sections indicated two distinct problems for the implementor when trying to meet the specification, and they also indicated how to solve them. The first problem was that the specification contained equalities on the representation type, where the implementor could only show equivalence with respect to some equivalence relation ≈. This was caused by the fact that several concrete values corresponded to one abstract value. The second problem was that the specification contained universal quantifications over the representation type, where the implementor could only prove the property for all representations satisfying some invariant. This was caused by the fact that not all concrete values correspond to an abstract value.

These two problems are orthogonal, and can occur together, which will be shown here. Fortunately, the solutions to these two problems can be combined. Now the abstract specification is abstracted over both an invariant and an equivalence relation. We need a combination of quotients and subsets in order to connect the implementor’s specification and the user’s specification.

The structure of this section is very similar to that of the first two implementations for bags.

1. Interface

The interface of bags BagI and the corresponding type of implementations BagImp are the same as in our second implementation (sorted lists), on page 147.

2. Specification

The naive specification NaiveSpec, and the user’s specification UserSpec are also the same as in our second implementation, see page 147.

3. Implementation

A bag is now represented by a record with an els field containing an unsorted list of elements and a mx field which keeps track of the maximum element of the list (which is returned by the bound operation).

\[
\text{Rep3} := \{ \text{els : List Nat, mx : Nat} \}
\]

\[
\text{ops3} := \{ \text{empty = \{els = nil Nat, mx = 0\}, add = \lambda n : \text{Nat}. \lambda r : \text{Rep3}. \{\text{els = n; r.els}, \text{mx = max r.mx n}\}, card = \lambda n : \text{Nat}. \lambda r : \text{Rep3}. \text{count n r.els}, \text{bound} = \lambda r : \text{Rep3}. \text{r.mx} \}
\]

\[
\text{imp3} := \text{pack} \langle \text{Rep3, ops3} \rangle : \text{BagI Rep3}
\]

Here max delivers the maximum of its two arguments.

Essential properties of this representation are that not all concrete values correspond with an abstract value, and that several concrete values may represent the same abstract value. Two kinds of information are hidden. First, representations that do not correspond with an
abstract value (e.g. \{els = [2,3], mx = 1\}) are not visible to the user. Second, the precise representation of an abstract value (e.g. the order of elements in the els field) is hidden. For the implementation with unsorted lists (Section 6.5.1) only the second kind of information was hidden. For the implementation with sorted lists (Section 6.5.2) only the first kind was hidden. For the implementation of stacks, no information was hidden at all.

4a. The problems with the naive specification

The specification UserSpec might be what the user of the ADT wants, but it may be a problem for the implementor of the ADT to meet this specification. Proving NaiveSpec Rep3 ops3 (with the intention of showing UserSpec imp3) means we have to prove, amongst others, that

\[
\forall m, n: \text{Nat}. \forall b: \text{Rep}3. \text{ops3}\cdot\text{add} \, m (\text{ops3}\cdot\text{add} \, n \, b) =_{\text{Rep3}} \text{ops3}\cdot\text{add} \, n (\text{ops3}\cdot\text{add} \, m \, b) \quad \text{(i)}
\]

\[
\forall m: \text{Nat}. \forall b: \text{Rep}3. 1 \leq \text{ops3}\cdot\text{card} \, m \, b \implies m \leq \text{ops3}\cdot\text{bound} \quad \text{(ii)}
\]

But (i) and (ii) are not true: swapping two add operations does not give rise to equal representations (just as in Section 6.5.1), and the bound operation does not deliver a proper bound for all elements of the representation type (as in Section 6.5.2). The problems have two causes. First, several concrete values of Rep3 correspond to the same abstract value (bag). Second, not all concrete values of Rep3 correspond to an abstract value.

Again, there are two unsatisfactory ways to solve the problems above. We will not discuss the first way, i.e. finding an alternative implementation, again. We will discuss the second way, i.e. using a weaker specification, and then we come up with a satisfactory solution, which uses a combination of quotients and subsets.

4c. Another solution: Using a weaker specification

Since this implementation imp3 suffers from both the problems of imp1 and imp2, the most obvious solution for imp3 is the combination of the solutions for imp1 and imp2.

So we define the abstract specification Spec as a modification of NaiveSpec: we weaken each quantification over Rep by an abstract invariant Inv, and we replace each Leibniz' equality by an abstract notion of equality.

\[
\text{Spec} := \lambda \text{Rep} \cdot \ast_p. \lambda (\approx). \text{Rep} \rightarrow \text{Rep} \rightarrow \ast_p. \lambda \text{Inv} \cdot \text{Rep} \rightarrow \ast_p. \lambda \text{ops} \cdot \text{BagI} \cdot \text{Rep}.
\]

\[
\forall m, n: \text{Nat}. \forall r: \text{Rep}. \text{Inv} \Rightarrow
\]

\[
\text{ops}\cdot\text{card} \, m \, \text{ops}\cdot\text{empty} =_{\text{Nat}} 0 \land
\text{ops}\cdot\text{card} \, m (\text{ops}\cdot\text{add} \, m \, r) =_{\text{Nat}} S (\text{ops}\cdot\text{card} \, m \, r) \land
(\neg (m =_{\text{Nat}} n) \Rightarrow \text{ops}\cdot\text{card} \, m (\text{ops}\cdot\text{add} \, n \, r) =_{\text{Nat}} \text{ops}\cdot\text{card} \, m \, r) \land
\text{ops}\cdot\text{add} \, m (\text{ops}\cdot\text{add} \, n \, r) \equiv \text{ops}\cdot\text{add} \, n (\text{ops}\cdot\text{add} \, m \, r) \land
(1 \leq \text{ops}\cdot\text{card} \, m \, r \Rightarrow m \leq \text{ops}\cdot\text{bound} \, r)
\]

\[
: \Pi \text{Rep} \cdot \ast_p. (\text{Rep} \rightarrow \ast_p) \rightarrow (\text{Rep} \rightarrow \ast_p) \rightarrow \text{BagI} \cdot \text{Rep} \rightarrow \ast_p
\]

Note that

\[
\text{NaiveSpec} \, \text{Rep} \, \text{ops} \iff \text{Spec} \, \text{Rep} \, (=_{\text{Rep}}) \, (\lambda r: \text{Rep}. \text{True}) \, \text{ops}
\]

for every Rep and ops.

Next, we define the implementor's specification for bags as the existence of a suitable notion of equality and invariant for which Spec holds:
\[ \text{ImplemSpec} := \lambda \text{Rep}^+ \cdot \lambda \text{ops}: \text{BagI Rep}. \]
\[ \exists (\simeq): \text{Rep} \rightarrow \text{Rep} \rightarrow *_p. \exists \text{Inv}: \text{Rep} \rightarrow *_p. \]
\[ \text{Spec Rep} (\simeq) \text{ Inv ops} \land \]
\[ \text{Bisim}^{\text{BagI}} \text{ Rep} (\simeq) \text{ ops} \land \]
\[ \text{IsInvar}^{\text{BagI}} \text{ Rep Inv ops} \land \]
\[ \text{IsERon}_{\text{BagI}} (\simeq) \text{ Inv} \]
\[ : \Pi \text{Rep}^+ \cdot \lambda \text{BagI Rep} \rightarrow *_p \]

Here \text{IsERon}_{\text{BagI}} (\simeq) \text{ Inv} expresses that \simeq is an equivalence relation on \text{Rep} restricted to predicate \text{Inv}, since we are only interested in values satisfying the invariant, the relation \simeq should be an equivalence only on those values. \text{IsERon} is formally defined in Section 6.7.1.

Again, we turn \text{Spec} into a predicate on \text{WeakUserSpec} over \text{BagImp}:

\[ \text{WeakUserSpec} := \lambda \text{bagImp}: \text{BagImp}. \]
\[ \exists \text{Rep}^+ \cdot \exists \text{ops}: \text{BagI Rep}. \exists (\simeq): \text{Rep} \rightarrow \text{Rep} \rightarrow *_p. \exists \text{Inv}: \text{Rep} \rightarrow *_p. \]
\[ \text{bagImp} = \text{BagImp pack} (\text{Rep}, \text{ops}) \land \]
\[ \text{Spec Rep} (\simeq) \text{ Inv ops} \land \]
\[ \text{Bisim}^{\text{BagI}} \text{ Rep} (\simeq) \text{ ops} \land \]
\[ \text{IsERon}_{\text{BagI}} (\simeq) \text{ Inv} \land \]
\[ \text{IsInvar}^{\text{BagI}} \text{ Rep Inv ops} \]
\[ : \text{BagImp} \rightarrow *_p \]

The implementor of the ADT will be happy with this weaker specification, as he can easily show \text{imp3} satisfies it, which we show next.

It is straightforward to give an invariant \text{Inv3} for his implementation: the invariant holds for those elements \( r \) for which \( r \cdot \text{mx} \) is larger than all elements of \( r \cdot \text{els} \).

\[ \text{Inv3} := \lambda r: \text{Rep3}. \forall n: \text{Nat}. \text{ElemNat} n r \cdot \text{els} \Rightarrow n \leq r \cdot \text{mx} \]
\[ : \text{Rep3} \rightarrow *_p \]
\[ \text{IsInvar}_{\text{Inv3}} := \ldots : \text{IsInvar}^{\text{BagI}} \text{ Rep3 Inv3 ops3} \]

Note that we could have supplied a stronger invariant (namely that \( r \cdot \text{mx} \) is the maximum of the elements, or 0 for no elements), but this invariant suffices for our specification.

The notion of equality \text{Sim3} is also rather obvious, given that it must be a bisimulation for the operations. Two representations \( r \) and \( r' \) are equivalent if \( r \cdot \text{els} \) and \( r' \cdot \text{els} \) are permutations and the maximums \( r \cdot \text{mx} \) and \( r' \cdot \text{mx} \) are equal.

\[ \text{Sim3} := \lambda r, r': \text{Rep3}. \text{PermNat} r \cdot \text{els} r' \cdot \text{els} \land r \cdot \text{mx} =_{\text{Nat}} r' \cdot \text{mx} \]
\[ : \text{Rep3} \rightarrow \text{Rep3} \rightarrow *_p \]
\[ \text{Bisim}_{\text{Sim3}} := \ldots : \text{Bisim}^{\text{BagI}} \text{ Rep3 Sim3 ops3} \]
\[ \text{IsERon}_{\text{Sim3}} := \ldots : \text{IsERon}_{\text{Rep3}} \text{ Sim3 Inv3} \]

With \text{Inv3} and \text{Sim3} the implementor can prove

\[ \text{Spec.ops3} := \ldots : \text{Spec Rep3 Sim3 Inv3 ops3} \]
\[ \text{ImplemSpec.ops3} := \ldots : \text{ImplemSpec Rep3 ops3} \]

and hence

\[ \text{WeakUserSpec.imp3} := \ldots : \text{WeakUserSpec imp3} \]
The user of the ADT on the other hand will be not happy with \textit{WeakUserSpec}. Every time he wants to use a part of the specification for some bag \( b \), he has to prove that \( b \) satisfies the invariant. Furthermore, he cannot reason with Leibniz’ equality, but is forced to use some equivalence relation. Both are unnecessary burdens.

4d. Combining quotients and subsets

Again, we want to relate \textit{ImplemSpec} and \textit{UserSpec}. The most obvious approach, given Sections 6.5.1 and 6.5.2, is to do this by combining quotients and subsets. We first take the subset, keeping only the elements satisfying the invariant, and then quotient this out over the equivalence relation.

It is useful to introduce an abbreviation for this combination of quotients and subsets: given \( R, Q, \text{opsR}, \text{opsQ}, \simeq \) and \( \text{Inv} \), the proposition \( \text{IsQuotSubsetAlg}_{\text{BagI}} \) \( R \text{opsR} (\simeq) \text{Inv} Q \text{opsQ} \) will express that \( Q \) is \( (R|_{\text{Inv}})/\simeq \), and that \( \text{opsQ} \) is \( \text{[opsR|}_{\text{Inv}}]_{\simeq} \). We define \( \text{IsQuotSubsetAlg} \) directly, instead of defining it in terms of \( \text{IsQuotAlg} \) and \( \text{IsSubsetAlg} \), because the direct formulation is much easier to reason about.

However, such a direct formulation is a bit more complicated than the definitions of \( \text{IsSubsetAlg} \) and \( \text{IsQuotAlg} \). In the definition of \( \text{IsSubsetAlg} \) we used an injection \( \text{inj} : R|_{\text{Inv}} \rightarrow R \) to relate \( R \) and its subset. Similarly, in the definition of \( \text{IsQuotAlg} \), we also used a function, called \( \text{surj} \), which maps each value to its equivalence class. However, we cannot relate elements of \( R \) and \( (R|_{\text{Inv}})/\simeq \) by a function, so we use a relation \( \text{Rel} \) to relate elements of \( R \) and \( (R|_{\text{Inv}})/\simeq \); informally, \( \text{Rel} r q \) corresponds to

\[
\exists s : R|_{\text{Inv}}. \text{inj } s = r \land \text{surj } s = q.
\]

This correspondence is useful for understanding the formal definition of \( \text{IsQuotSubsetAlg}_{\text{BagI}} \).

\[
\text{IsQuotSubsetAlg}_{\text{BagI}} := \lambda R:*, \lambda \text{opsR}:\text{BagI} R. \lambda (\simeq):R \rightarrow R \rightarrow *, \lambda \text{Inv}:R \rightarrow *,
\]

\[
\exists \text{Rel}:R \rightarrow Q \rightarrow *,
\]

\[
(\text{Inv} \iff \text{Dom}_{\text{Q}} \text{Rel}) \land
\]

\[
(\forall q:Q. \exists r : R. \text{Rel } r q) \land
\]

\[
(\forall r, r' : R. \forall q, q' : Q. \text{Rel } r q \implies \text{Rel } r' q' \implies r \simeq r' \iff q = q') \land
\]

\[
\text{Sim}_{\text{BagI}} R Q \text{Rel} \text{opsR} \text{opsQ}
\]

\[
: \text{IIQ} : *, \text{BagI} Q \rightarrow (R \rightarrow R \rightarrow *) \rightarrow (R \rightarrow *) \rightarrow
\]

\[
\text{IIQ} : *, \text{BagI} Q \rightarrow *
\]

The first demand says the domain of \( \text{Rel} \) is equal to (the set determined by) \( \text{Inv} \); so exactly those elements of \( R \) that satisfy \( \text{Inv} \) have a related element in \( Q \). The second demand says every element in \( Q \) is related to some element in \( R \). The third demand expresses that two elements in \( R \) are related by \( \simeq \) if and only if the corresponding elements in \( Q \) are equal. The last demand says \( \text{Rel} \) is a simulation between \( \text{opsR} \) and \( \text{opsQ} \), i.e., \( \text{opsQ} \) corresponds to \( \text{[opsR|}_{\text{Inv}}]_{\simeq} \).

We can prove the existence of \( \text{QuotSubset} \) algebras by combining the appropriate axioms for quotients and subsets:
exis_QuotSubsetAlgBagI := ... : \forall R:*, \forall opsR:BagI R. \forall (\simeq): R \rightarrow R \rightarrow *_p. \forall Inv: R \rightarrow *_p.

IsERonR (\simeq) Inv \implies 
BisimBagI R \simeq opsR \implies 
IsInvarBagI R Inv opsR \implies 
\exists Q:*, \exists opsQ:BagI Q.
IsQuotSubsetAlgBagI R opsR (\simeq) Inv Q opsQ

Note that the demand IsERonR (\simeq) Inv means that \simeq is an equivalence relation on the subset of R determined by Inv (it is not necessary that \simeq relates only elements for which Inv holds, but for these elements, \simeq must be an equivalence relation).

Remark 6.5.3.1 We could slightly simplify our formalization of the combination of quotients and subsets by combining \simeq and Inv into one partial equivalence relation \sim equivalent to the restriction of \simeq to Inv, and introducing a predicate IsPerAlg. Going even further, we could postulate an axioms exis_PermAlg, instead of the axioms exis_QuotAlg and exis_SubsetAlg (see also Section 6.8.2). It is interesting that the notion of partial equivalence relations comes up here, as in best known models of \(\lambda_\omega\) datatypes are interpreted as PERs! An advantage of an exis_PermAlg could be that it is easier to justify on the basis of PER models.

We refrained from this approach, because Inv and \simeq have clear interpretations for the implementor of an ADT, viz. invariable and equivalence relation. There is not such a simple interpretation for \sim. Furthermore, Inv and \simeq have quite distinct effects on the specification.

Now we use the existence of these QuotSubset algebras to prove the user's specification from the implementor's specification in the same way as before. First we observe that our specification Spec of bags is "sensible", i.e.

Spec_Sens := ... : \forall R:*, \forall opsR:BagI R. \forall (\simeq): R \rightarrow R \rightarrow *_p. \forall Inv: R \rightarrow *_p.

\forall Q:*, \forall opsQ:BagI Q.
IsQuotSubsetAlgBagI R opsR (\simeq) Inv Q opsQ \implies 
Spec R (\simeq) Inv opsR \implies Spec Q (\=\emptyset) (\lambda q: Q. True) opsQ

Now, given some Rep: *, and ops : BagI Rep with ImplemSpec Rep ops, we have to show UserSpec (pack (Rep, ops)), i.e.

\exists Rep':*, \exists ops':I Rep.
pack (Rep, ops) =_{BagImp} pack (Rep', ops') \land 
Spec Rep' (\=Rep') (\lambda r: Rep'. True) ops'

By ImplemSpec and exis_QuotSubsetAlgBagI, we have \simeq and Inv with Spec Rep (\simeq) Inv ops and Q and opsQ with IsQuotSubsetAlgBagI Rep ops (\simeq) Inv Q opsQ. Because of Spec_Sens we also have Spec (\=\emptyset) (\lambda q: Q. True) opsQ. With parametricity, it is easy to show that pack (Rep, ops) =_{BagImp} pack (Q, opsQ). Now we are done by taking Rep := Q and ops' := opsQ.

In particular, since we ImplemSpec Rep3 ops3, we have

UserSpec_imp3 := ... : UserSpec imp3
Overview of the 5 steps

We again give the 5 steps of developing and using an ADT.

1. The user gives the interface of the ADT.

   \[
   \text{BagI} := \ldots \Rightarrow * \rightarrow *
   \]

   This determines the definition of the corresponding \( \Sigma \)-type.

   \[
   \text{BagImp} := \Sigma \text{Rep} : * \Rightarrow \text{BagI Rep} \\
   : * 
   \]

2. The user gives the abstract specification of the ADT, which is both abstracted over the invariant and the equality being used. And again, the user has to prove this specification is "sensible" (see Section 6.6.2).

   \[
   \text{Spec} \Rightarrow \ldots \Rightarrow \Pi \text{Rep} : * \Rightarrow (\text{Rep} \rightarrow \text{Rep} \rightarrow *) \rightarrow (\text{Rep} \rightarrow *) \rightarrow \text{BagI Rep} \rightarrow * \\
   \text{Spec. Sens} \Rightarrow \ldots \Rightarrow \forall \text{R} : * \Rightarrow \forall \text{VopsR} : \text{BagI R. R. } \forall \text{(=} : } R \Rightarrow R \rightarrow * \Rightarrow \forall \text{Inv: R} \rightarrow * \Rightarrow \\
   \forall \text{Q} : * \Rightarrow \forall \text{VopsQ} : \text{BagI Q.} \\
   \text{IsQuotSubsetBagI R} \Rightarrow \text{opsR} (\sim) \text{ Inv Q} \Rightarrow \text{opsQ} \Rightarrow \\
   \text{Spec R (=} \Rightarrow \text{Inv opsR} \Rightarrow \text{Spec Q (=} \Rightarrow (\lambda \text{q} : Q. \text{ True}) \text{ opsQ)} \\
   \]

   These terms determine definitions of the user's and the implementor's specification and a proof of the relation between both.

   \[
   \text{UserSpec} := \lambda \text{bagImp} : \text{BagImp. } \exists \text{Rep} : *, \exists \text{Vops} : \text{BagI Rep.} \\
   \text{bagImp = bagImp pack } (\text{Rep, ops}) \land \\
   \text{Spec Rep (=} \Rightarrow (\lambda r : \text{Rep. True}) \text{ ops)} \\
   \\
   : \text{BagImp} \rightarrow * \\
   \text{ImplemSpec} := \lambda \text{Rep} : *, \lambda \text{ops} : \text{BagI Rep.} \\
   \exists (\text{=} : \text{Rep} \rightarrow \text{Rep} \rightarrow * \Rightarrow \exists \text{Inv: Rep} \rightarrow * \\
   \text{Spec Rep (=} \Rightarrow \text{Inv ops} \land \\
   \text{BisimBagI Rep (=} \Rightarrow \text{ops} \land \\
   \text{IsInvariantBagI Rep Inv ops} \land \\
   \text{IsEronRep (=} \Rightarrow \text{Inv } \\
   \\
   : \Pi \text{Rep} : *, \text{BagI Rep} \rightarrow * \\
   \text{Implem-UserSpec} := [\text{generic}] \\
   : \forall \text{Rep} : *, \forall \text{Vops} : \text{BagI Rep.} \\
   \text{ImplemSpec Rep ops} \Rightarrow \text{UserSpec (pack } (\text{Rep, ops})) \\
   \]

   The existence of quotients and subsets and the principle of parametricity are essential in the proof of \text{Implem-UserSpec}.

3. The implementor gives a representation type and the operations, which determine the implementation value.

   \[
   \text{Rep1} := \ldots \Rightarrow * \\
   \text{ops1} := \ldots \Rightarrow \text{BagI Rep1} \\
   \text{im1} := \text{pack } (\text{Rep1, ops1}) \\
   : \text{BagImp} \\
   \]
4. The implementor gives \( \text{Inv1} \) and \( \text{Sim1} \). This \( \text{Sim1} \) must be an equivalence relation on \( \text{Rep1} \) restricted to \( \text{Inv1} \), and \( \text{Sim1} \) must be respected by \( \text{ops1} \). The predicate \( \text{Inv1} \) must be an invariant for \( \text{ops1} \), and, most importantly, the implementation must satisfy the specification with \( \text{Sim1} \) as notion of equality, and \( \text{Inv1} \) as invariant.

\[
\begin{align*}
\text{Inv1} & := \ldots : \text{Rep1} \rightarrow *_p \\
\text{Sim1} & := \ldots : \text{Rep1} \rightarrow \text{Rep1} \rightarrow *_p \\
\text{IsERon_Sim1} & := \ldots : \text{IsERon}_{\text{Rep1}} \text{Sim1 Inv1} \\
\text{Bsim_Sim1} & := \ldots : \text{Bsim}_{\text{Bag1}} \text{Sim1 ops1} \\
\text{IsInvar_Inv1} & := \ldots : \text{IsInvar}_{\text{Bag1}} \text{Inv1 ops1} \\
\text{Spec_ops1} & := \ldots : \text{Spec Rep1 Sim1 Inv1 ops1}
\end{align*}
\]

Together with 2 and 3 these determine a proof of \( \text{ImplemSpec} \) and hence of \( \text{UserSpec} \).

\[
\begin{align*}
\text{ImplemSpec_ops1} & := [\text{generic}] \\
& \quad : \text{ImplemSpec Rep1 ops1} \\
\text{UserSpec_imp1} & := [\text{generic}] \\
& \quad : \text{UserSpec imp1}
\end{align*}
\]

5. The user can prove correctness of programs that use an implementation by the following proof principle. This shows clearly that the user is not bothered with some invariant and that he can use Leibniz' equality.

\[
\text{principle} := [\text{generic}] \\
\quad : \forall \text{imp}:\text{BagImp}. \text{UserSpec imp} \implies \\
\quad \forall A:*_p. \forall Q:A \rightarrow *_p. \forall \text{body}:(\forall X:*_p, \text{BagI X} \rightarrow A). \\
\quad (\forall X:*_p, \forall \text{ops}:\text{BagI X}. \text{Spec X ( =_x ) ( } \lambda r:X. \text{True} \text{ ops} ) \implies \\
\quad Q (\text{body X ops})) \implies \\
\quad Q (\text{unpack imp as } (X, \text{ops}) \text{ in body X ops})
\]

**Summary**

Let us give a review of the third implementation of bags and the method for proving it correct. Essential properties of this representation are that not all concrete values correspond with an abstract value and that several concrete values may represent the same abstract value. This situation is sketched in Figure 6.5. We see that two kinds of information are hidden. First, representations that do not correspond with an abstract value (e.g. \( \text{els} = [2,3], \text{mx} = 1 \)) are not visible to the user. Second, the precise representation of an abstract value (e.g. the order of elements in the \( \text{els} \) field) is hidden. For the implementation with unsorted lists (Section 6.5.1) only the second kind of information was hidden. For the implementation with sorted lists (Section 6.5.2) only the first kind was hidden. For the implementation of stacks, no information was hidden at all.

The user does not want to be bothered by the hidden information. In his specification he wants to use Leibniz' equality of abstract values, even if their concrete representations differ. Furthermore, he wants to be able to quantify over all abstract values even though this quantification may not be valid for all concrete values.

Therefore the implementor must give an invariant \( \text{Inv} \) and an equivalence relation \( \simeq \) (which must be an equivalence on the set determined by \( \text{Inv} \)). The invariant says which concrete
values represent an abstract value, and $\simeq$ indicates which concrete values represent the same abstract value. He has to show that $\simeq$ is respected by all operations, i.e. it is a bisimulation. The implementor uses a specification where each quantification $\forall r : \text{Rep.} \ldots$ is replaced by $\forall r : \text{Rep.} \ Inv \ r \implies \ldots$, and each Leibniz equality $=_{\text{rep}}$ is replaced by $\simeq$.

As noted in the summary of Section 6.5.2, the set of values satisfying the invariant may be larger than the set of values reachable through the operations. The implementation given here has this property: the value $\{els = [2, 3], mx = 5\}$ is not reachable, but it satisfies the invariant. The important thing is that the specification is valid with these values, so they are not harmful. Would we have the stronger specification, that says the bound operation delivers the maximum element of the bag, we would have to exclude these values by strengthening the invariant.

In order to formally connect the implementor's and the user's specification, we use quotients of subsets. We first take the subset of the representation type $\text{Rep}$ determined by $\text{Inv}$, and then take the quotient over $\simeq$. By taking the subset we remove concrete values not representing an abstract value, so we can prove a quantification over all values in the subset if we can prove it for all values satisfying the invariant. By taking the quotient we equate concrete values representing the same abstract value, so Leibniz' equality on this quotient corresponds with $\simeq$ equivalence on $\text{Rep}$. Hence the quotient of the subset satisfies the user's specification (if the specification is sensible). By parametricity the original implementation and the implementation using the quotient of the subset are equal, so the original implementation satisfies the user's specification too.

A very important property of our development of ADTs in this section is that it is a generalization of the two previous sections. This allows us to use the method given in this section not only for implementations which need a combination of equivalence relation and invariant (e.g. the example in this section), but also when we need only an equivalence relation (e.g. the implementation with unsorted lists of Section 6.5.1) or when we need only an invariant (e.g. the implementation with sorted lists of Section 6.5.2). When we need only an equivalence relation, we take for the invariant just $\lambda r : \text{Rep.} \ True$, and when we need only an invariant we
take for the equivalence relation Leibniz' equality $\equiv_{lep}$. It is easy to check that the proof obligations for the implementor (step 4) reduce to the ones given in Section 6.5.1 and Section 6.5.2 respectively. Of course, the method works also when we need no equivalence relation and no invariant (e.g. the implementation of stacks in Section 6.3.2). In this case the only proof obligations that remain are that the implementation satisfies $Spec$ with Leibniz' equality and the trivial invariant (the "naive" specification). For each combination of absence/presence of equivalence and absence/presence of invariant, we have seen one implementation.

Because this approach generalizes the ones in Sections 6.5.1 and 6.5.2, this is the approach that always should be followed. So the specification should be abstracted over both an invariant and a notion of equality, to give the implementor the freedom to use either, or both, or none.

We give two rules that help the implementor in deciding whether he uses an invariant or an equivalence relation, if he already has an implementation in mind.

- If every concrete value corresponds with an abstract value, the implementor does not need an invariant. Typically, if some concrete value does not correspond to an abstract value, the implementor does need the invariant in order to fulfil his proof obligations. For example, in the implementation with sorted lists in Section 6.5.2, not all concrete values (the unsorted lists) correspond to a bag, and indeed the invariant is necessary.

- If each concrete value (that satisfies the invariant) uniquely determines one abstract value, the implementor does not need an equivalence relation. Typically, if there are several concrete values for one abstract value, the implementor does need an equivalence relation. For example, in the implementation with unsorted lists in Section 6.5.1, the implementor has $Perm_{Nat}$ as equivalence relation.

So far we have considered a particular interface and a particular specification, namely those for bags. In the next section we will generalize everything to arbitrary interfaces and arbitrary specifications for those interfaces.
6.6 General Interfaces and Specifications

So far we have considered proof rules for ADTs only for a few specific interfaces and specifications. In this section we generalize our results to a class of interfaces and specifications.

In Section 6.6.1 we generalize our notion of invariant to arbitrary interfaces, introduce a restriction to first-order interfaces, and postulate general axioms for quotients and subsets. Section 6.6.2 shows not all specifications can be admitted, so we introduce in Section 6.6.3 a restriction to well-behaved specifications. Our ultimate goal is Theorem 6.6.4.1, which generalizes the proof principle for bags to arbitrary first-order interfaces and arbitrary well-behaved specifications.

6.6.1 General Interfaces

The proof rules developed for our examples are justified by additional axioms stating the existence of quotient and subset algebras for interface BagI. In order to make the proof rules for arbitrary interfaces, we also need to generalize these axioms.

The axiom stating the existence of quotient algebras for BagI (Definition 6.5.1.2) is formulated in terms of SimBagI; since we have defined SimI for arbitrary simply-typed interfaces I (Definition 6.4.2.2), we can easily generalize this axiom to all simply-typed interfaces I. However, the axiom stating the existence of subset algebras for BagI (Definition 6.5.2.1) is formulated in terms of IsInvarBagI, but no general notion IsInvarI was defined for simply-typed interfaces I. Therefore we do this here.

Definition 6.6.1.1 (Invariant) Given a simply-typed type-constructor $I : *_p \rightarrow *_p$ and a datatype $Y : *_p$, then $IsInvarI : (Y \rightarrow *_p) \rightarrow (I \ Y \rightarrow *_p)$ is defined as follows (where we assume $X \not\in \text{FV}(I_i)$ for each $i$):

$$
\begin{align*}
IsInvarI_{(AX:*_p,X)} P y & = P y \\
IsInvarI_{(AX:*_p,T)} P y & = \text{True}, \text{ if } X \not\in \text{FV}(T) \\
IsInvarI_{(AX:*_p, I_1 X \leftarrow I_2 X)} P y & = \forall a : I_1 Y. IsInvarI_{I_1} P a \Longrightarrow IsInvarI_{I_2} P (y a) \\
IsInvarI_{(AX:*_p, \{i_1 : I_1 \text{ X}_1 \ldots \text{X}_n \mid i_n \}} P y & = IsInvarI_{I_1} P y \cdot i_1 \wedge \ldots \wedge IsInvarI_{I_n} P y \cdot i_n
\end{align*}
$$

The second clause overlaps with both the third and the fourth clause, but it is easy to see that this does not give conflicts, so IsInvar is well-defined up to logical equivalence. For the rest of this thesis, we consider $Y$ to be a parameter of IsInvarI, so

$$
IsInvarI : \Pi Y : *_p. (Y \rightarrow *_p) \rightarrow (I \ Y \rightarrow *_p).
$$

The type of IsInvar shows that IsInvarI $Y$ lifts a predicate on $Y$ to a predicate on $I \ Y$. The notions of IsInvar and Sim appear to be quite similar. In fact, IsInvar and Sim are the unary and binary versions of a general scheme, see [PA93, Tak97].

The specific definition of IsInvarBagI (Section 6.5.2, page 149) is equivalent to the general definition given here instantiated to $I = \text{BagI}$, so there is no danger of confusion when we write IsInvarBagI.

Now we can give the generalizations of IsQuotAlgI, IsSubsetAlgI and IsQuotSubsetAlgI to arbitrary simply-typed interfaces.
Definition 6.6.1.2 We extend the context with
\[
\begin{align*}
\text{IsQuotAlg} & := \lambda \Gamma : \ast, \lambda \text{opsR} : I \ R, \lambda (\equiv) : R \rightarrow \ast_p, \lambda Q : \ast_q, \lambda \text{opsQ} : I \ Q. \\
& \exists \text{surj} : R \rightarrow Q. \\
& (\forall r, r' : R. r \equiv r' \iff \text{surj} r =_q \text{surj} r') \wedge \\
& \text{IsSurjection}_{R,Q} \text{ surj} \wedge \\
& \text{Sim}_I R Q (\lambda r : R. \lambda q : Q. q =_q \text{surj} r) \text{ opsR opsQ} \\
& : \Pi \Gamma : \ast_p, I R \rightarrow (R \rightarrow R \rightarrow \ast_p) \rightarrow (\Pi Q : \ast_q, I Q \rightarrow \ast_p) \\
\text{IsSubsetAlg} & := \lambda \Gamma : \ast, \lambda \text{opsR} : I \ R, \lambda \text{Inv} : R \rightarrow \ast_p, \lambda S : \ast_q, \lambda \text{opsS} : I \ S. \\
& \exists \text{inj} : S \rightarrow R. \\
& \text{IsInjection}_{S,R} \text{ inj} \wedge \\
& (\text{Inv} \iff \text{Image}_{S,R} \text{ inj}) \wedge \\
& \text{Sim}_I R S (\lambda r : R. \lambda s : S. r =_s \text{inj} s) \text{ opsR opsS} \\
& : \Pi \Gamma : \ast_q, I R \rightarrow (R \rightarrow \ast_p) \rightarrow (\Pi S : \ast_q, I S \rightarrow \ast_p) \\
\text{IsQuotSubsetAlg} & := \lambda \Gamma : \ast, \lambda \text{opsR} : I \ R, \lambda (\equiv) : R \rightarrow \ast_p, \lambda \text{Inv} : R \rightarrow \ast_p. \\
& \lambda Q : \ast_q, \lambda \text{opsQ} : I \ Q. \\
& \exists \text{Rel} : R \rightarrow Q \rightarrow \ast_p. \\
& (\text{Inv} \iff \text{Dom}_{Q,R} \text{ Rel}) \wedge \\
& (\forall q : Q. \exists r : R. \text{Rel} r q) \wedge \\
& (\forall r, r' : R. \forall q : Q. \text{Rel} r q \implies \text{Rel} r' q' \implies \\
& r \equiv r' \iff q =_q q') \wedge \\
& \text{Sim}_I R Q \text{ Rel} \text{ opsR opsQ} \\
& : \Pi \Gamma : \ast_q, I R \rightarrow (R \rightarrow \ast_p) \rightarrow (R \rightarrow \ast_p) \rightarrow (\Pi Q : \ast_q, I Q \rightarrow \ast_p)
\end{align*}
\]
for every simply-typed \(I\).

Now we could give axioms stating the existence of quotients and subset algebras for arbitrary simply-typed \(I\). For example, we could postulate

for all simply-typed \(I\).

\[
\begin{align*}
\forall \Gamma : \ast, \forall \text{opsR} : I \ R. \forall \text{Inv} : R \rightarrow \ast_p, \text{IsInv}_{R,R} \text{ Inv} \text{ opsR} & \implies (i) \\
\exists S : \ast_q, \exists \text{opsS} : I \ S. \text{IsSubsetAlg} R \text{ opsR Inv S opsS}
\end{align*}
\]

However, adding axioms is always dangerous, and the more powerful the axioms are, the greater the chance of introducing inconsistency. In fact, it turns out that we would introduce inconsistency if we would assume (i)! We have a counter-example for interface \(I = \lambda X : \ast : X \rightarrow X\). (\(X \rightarrow X\) \( \rightarrow X\)) (see Example 6.8.2.4). Note that this interface contains the type of a higher-order function \((X \rightarrow X) \rightarrow X\): a function of this type accepts another function (of type \(X \rightarrow X\)) as argument. Therefore we choose to restrict ourselves to the so-called first-order interfaces. (In Section 6.8.2 we will elaborate on this choice, and give an alternative.) Intuitively, the first-order interfaces correspond to "traditional" interfaces: no function types may occur as domain of other function types. (Beware that the notion "order" as used here is different from "order" as in "second-order \(\lambda\)-calculus." First-order interfaces are defined formally as follows; an interface is higher-order if it is simply-typed but not first-order.

Definition 6.6.1.3 (First-order interfaces) The first-order interfaces are interfaces of the form \(\lambda X : \ast : \text{FirstO where FirstO}\) is defined as follows.

\[
\begin{align*}
\text{Basic} & := X \mid T \text{ with } X \not\in \text{FV}(T) \mid || l_1 : \text{Basic}, \ldots, l_n : \text{Basic} || \\
\text{FirstO} & := \text{Basic} \mid \text{Basic} \rightarrow \text{FirstO} \mid || l_1 : \text{FirstO}, \ldots, l_n : \text{FirstO} ||
\end{align*}
\]
Similarly, the basic interfaces are interfaces of the form $\lambda \mathbf{X} : *_{p}$. Basic. We consider interfaces up to $\beta\delta$-conversion.

We postulate the existence of quotient and subset algebras for the first-order interfaces.

**Definition 6.6.1.4** QUOTSUBSET is the extension of PAR with the following axioms.

$$
\begin{align*}
\text{exis}_\text{QuotAlg} & : \forall R : *_{p}. \forall \text{ops}R : I R. \forall (\simeq) : R \rightarrow R \rightarrow *_{p}.
\quad \text{Bisim}_{I} R (\simeq) \text{ops}R \Rightarrow \text{IsER}_{R} (\simeq) \Rightarrow
\quad \exists Q : *_{p}. \exists \text{ops}Q : I Q. \text{IsQuotAlg}_{I} R \text{ops}R (\simeq) Q \text{ops}Q
\end{align*}
$$

$$
\begin{align*}
\text{exis}_\text{SubsetAlg} & : \forall R : *_{p}. \forall \text{ops}R : I R. \forall \text{Inv}R : R \rightarrow *_{p}. \text{IsInvAlg}_{I} R \text{Inv} \text{ops}R \Rightarrow
\quad \exists S : *_{p}. \exists \text{ops}S : I S. \text{IsSubsetAlg}_{I} R \text{ops}R \text{Inv} S \text{ops}S
\end{align*}
$$

for all first-order interfaces $I$.

We claim that QUOTSUBSET is consistent. This follows from the validity of QUOTSUBSET in the PER model of [Pol94]. Another way of showing consistency of QUOTSUBSET is by proving these axioms in $\lambda \omega_{p}$ extended with quotients and subsets, e.g. as in [Bar95].

In QUOTSUBSET we can prove the existence of combined quotient and subset algebras for all first-order $I$:

$$
\begin{align*}
\text{exis}_\text{QuotSubsetAlg} & := \ldots : \forall R : *_{p}. \forall \text{ops}R : I R. \forall (\simeq) : R \rightarrow R \rightarrow *_{p}. \forall \text{Inv}R : R \rightarrow *_{p}.
\quad \text{IsER}_{R} (\simeq) \text{Inv} \Rightarrow
\quad \text{Bisim}_{I} R (\simeq) \text{ops}R \Rightarrow
\quad \text{IsInvAlg}_{I} R \text{Inv} \text{ops}R \Rightarrow
\quad \exists Q : *_{p}. \exists \text{ops}Q : I Q.
\quad \text{IsQuotSubsetAlg}_{I} R \text{ops}R (\simeq) \text{Inv} Q \text{ops}Q
\end{align*}
$$

Now we have generalized the prerequisites of our theory for arbitrary first-order interfaces. In the next section we will consider general specifications for these interfaces. Then we give our general proof rules for ADTs.

**6.6.2 Sensible Specifications**

In each example implementation one of the proof obligations was to show that the abstract specification was “sensible”. In this section we formally define this notion for specifications for arbitrary simply-typed interfaces, and show that we cannot avoid such a proof obligation. In the next section (6.6.3) we will consider a simple syntactic criterion that guarantees that a specification is sensible.

We define the notion of sensible in the general setting of Section 6.5.3, so specifications are abstracted over both a notion of equality and an invariant, so they have type

$$
\Pi R : *_{p}. (R \rightarrow R \rightarrow *_{p}) \rightarrow (R \rightarrow *_{p}) \rightarrow I R \rightarrow *_{p}.
$$

The predicate $\text{Sensible}$ on specifications of this type is defined as follows.
Definition 6.6.2.1 (Sensible)

\[
\text{Sensible} := \lambda \text{Spec}: \Pi \text{R}:*_{\ldots}, (R \rightarrow R \rightarrow *_{\ldots}) \rightarrow (R \rightarrow *_{\ldots}) \rightarrow I R \rightarrow *_{\ldots},
\forall \text{R}:*_{\ldots}, \forall \text{opsR}: I \text{R}, \forall (\ldots): R \rightarrow R \rightarrow *_{\ldots}, \forall \text{Inv}: R \rightarrow *_{\ldots}, \forall \text{Q}: I \text{R}, \forall \text{Q}: I \text{Q}, \text{IsQuotSubsetAlg} \text{R opsR} (\ldots) \text{Inv Q opsQ} \Rightarrow \text{Spec R Q opsR} \Rightarrow \text{Spec Q Q opsQ} \Rightarrow \text{Spec Q Q opsQ}
\]

for all simply-typed interfaces \( I \).

This is a general formulation of one of the proof obligations for the user in Section 6.5.3: the proposition named Spec.Sens in the overview on page 161 is summarized by Sensible_{\text{bagI}} Spec. Similar demands were made in Sections 6.5.1 and 6.5.2.

We are not so happy with this proof obligation, for two reasons. First, this obligation is formulated in terms of quotients and subsets, which we rather not expose to the programmers. Second, it is awkward to prove that a specification fulfils this obligation.

In this section we will show that not all specifications are sensible, and that we cannot avoid such a notion. In the next section we will define the notion of well-behaved specifications, which form a subset of the sensible specifications, and we indicate a large collection of specifications that are well-behaved, so that it becomes easy to show for most specifications that they are sensible.

Roughly, an abstract specification of the form

\[
\lambda \text{R}:*_{\ldots}, \lambda (\ldots): R \rightarrow R \rightarrow *_{\ldots}, \lambda \text{Inv}: R \rightarrow *_{\ldots}, \lambda \text{ops}: I \text{R}, R, P
\]

is well-behaved (and hence also sensible) if \( P \) does not contain

- Leibniz' equalities on type \( R \) (instead \( \simeq \) should be used), and
- quantifications over all elements of \( R \), except of the form "\( \forall x: R. \text{Inv} x \Rightarrow \ldots \)", i.e. restricted to those elements in \( R \) that satisfy \( \text{Inv} \).

Not all specifications are sensible

Typically, specifications for ADTs will be sensible, e.g. the one given in Section 6.5.3. But there are also contrived examples of specifications which are not sensible. For example, take

\[
\text{NonsenseSpec} := \lambda \text{Rep}:*_{\ldots}, \lambda (\ldots): \text{Rep} \rightarrow \text{Rep} \rightarrow *_{\ldots}, \lambda \text{Inv}: \text{Rep} \rightarrow *_{\ldots}, \lambda \text{ops}: \text{BagI} \text{Rep}.
\]

\[
\exists x, x' : \text{Rep}. x \simeq x' \land \neg (x \approx_{\text{bagI}} x')
\]

\[
\Pi \text{Rep}:*_{\ldots}, (\text{Rep} \rightarrow \text{Rep} \rightarrow *_{\ldots}) \rightarrow (\text{Rep} \rightarrow *_{\ldots}) \rightarrow \text{BagI} \text{Rep} \rightarrow *_{\ldots}
\]

This NonsenseSpec says that its datatype parameter has at least two distinct elements, which are equivalent by \( \simeq \). Note that we also use Leibniz' equality on \text{Rep} in NonsenseSpec: the occurrence of \( \approx_{\text{bagI}} \) is the reason that this specification is not sensible, as we will show.

Take as representation type \text{Bool} (or any other type with at least two elements), take as set of operations \text{opsBool}: \text{BagI} \text{Bool} for which the card and bound functions always delivers 0 and take as bisimulation the relation \( \simeq \) that always returns \text{True}, and take the trivial invariant \( \text{Inv} = \lambda r: \text{Bool}. \text{True} \). It is easy to show that \( (\simeq) \) is an equivalence relation and a bisimulation for \text{opsBool}. Since \text{Bool} has two elements, NonsenseSpec \text{Bool} (\ldots) Inv opsBool holds.
Propositional connectives

Second, the (pointwise) implication between two well-behaved specifications is again well-behaved.

\[
WB_{\impl} := \ldots : \forall P, Q : (\Pi R : * \cdot (R \to R \to *) \to (R \to *) \to I R \to *). \\
WB_{\impl} P \impl Q \\
WB_{\impl} (\lambda R : * \cdot (\lambda (\sim) : R \to R \to * \cdot \lambda Inv : R \to * \cdot \lambda ops : I R. \\
\text{P R (\sim) Inv ops } \impl Q \text{ R (\sim) Inv ops})
\]

So if the specifications \( P \) and \( Q \) are well-behaved and have the form

\[
P = \lambda R : * \cdot (\lambda (\sim) : R \to R \to * \cdot \lambda Inv : R \to * \cdot \lambda ops : I R. \text{ "bodyP"} \\
Q = \lambda R : * \cdot (\lambda (\sim) : R \to R \to * \cdot \lambda Inv : R \to * \cdot \lambda ops : I R. \text{ "bodyQ"}
\]

where \( R, (\sim), \text{Inv} \) and \( \text{ops} \) may occur in "bodyP" and "bodyQ", then the specification

\[
\lambda R : * \cdot (\lambda (\sim) : R \to R \to * \cdot \lambda Inv : R \to * \cdot \lambda ops : I R. \text{ "bodyP" } \impl \text{ "bodyQ"}
\]

is also well-behaved. (We use the quotes to emphasize that \text{bodyP} \text{ and } \text{bodyQ} \text{ are metavariables, standing for arbitrary terms, in which } R, (\sim), \text{Inv} \text{ and } \text{ops} \text{ may occur.)}

In combination with the fact that the specification

\[
\lambda R : * \cdot (\lambda (\sim) : R \to R \to * \cdot \lambda Inv : R \to * \cdot \lambda ops : I R. \text{False}
\]

6.6. GENERAL INTERFACES AND SPECIFICATIONS

By the existence of QuotSubset algebras there is a type \( Q \) and operations \( \text{opsQ} \) such that \((Q, \text{opsQ})\) is the quotient of \((\text{Bool}, \text{opsBool})\) over \( \sim \) (we have the trivial invariant, so taking the subset amounts to nothing). Note that \( Q \) has only one element. So

\[
\text{NonsenseSpec} \ Q \ (=_{\tilde{Q}}) \ (\lambda q : Q. \ True) \ \text{opsQ}
\]

does not hold. So \text{NonsenseSpec} is a counterexample to the proposition that all specifications are sensible.

Furthermore this specification \text{NonsenseSpec} \text{ and operations } \text{opsBool} \text{ form also a counterexample to the proposition that}

\[
\forall \text{Rep} : *, \forall \text{ops} : \text{BagI Rep}. \text{ImplemSpec Rep ops } \impl \text{UserSpec (pack (Rep, ops))}
\]

holds for \text{ImplemSpec} and \text{UserSpec} constructed from any \text{Spec}. The \text{ImplemSpec} constructed from \text{NonsenseSpec} holds for \text{Bool} and \text{opsBool}, i.e.

\[
\exists (\sim) : \text{Bool } \to \text{Bool } \to * \cdot \exists \text{Inv} : \text{Bool } \to * \cdot \exists \text{NonsenseSpec} \text{ Bool } (\sim) \ \text{Inv opsBool} \land \\
\text{Bisim_{BagI} Bool } (\sim) \ \text{opsBool} \land \\
\text{IsInv_{BagI} Bool Inv opsBool} \land \\
\text{IsERon_{Bool} } (\sim) \ \text{Inv}
\]

by taking the same notion of equality and invariant as above.

But the \text{UserSpec} constructed from \text{NonsenseSpec} for \((\text{pack (Bool, opsBool)})\), i.e.

\[
\exists \text{Rep} : *, \exists \text{ops} : \text{BagI Rep}. \\
\text{pack (Bool, opsBool)} ^{\text{BagImp}} \text{ pack (Rep, ops)} \land
\]
Quantifications over datatypes not containing the representation type \( R \)

Now we consider specifications consisting of a universal quantification. First we handle the “simple” case where the datatype \( A \) over which we quantify does not contain \( R \). Such a specification has the form

\[
\lambda R: *_a, \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. \\
\forall a: A \text{ "some body in terms of } a, R, (z), Inv \text{ and ops"}
\]

We express this neatly in type theory by abstracting over \( a, R, (z), Inv \) and \( \text{ops} \). So we write the specification as

\[
\lambda R: *_a, \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. \\
\forall a: A. P a R (z) Inv \text{ ops}
\]

where

\[
P : A \rightarrow \Pi R: *_a. (R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p.
\]

It turns out that this specification is well-behaved, if \( P \) is well-behaved for all \( a: A \). Note that \( P \) has the correct type for a specification.

\[
WB_{\text{Univ}} := \ldots : \forall A: *_a. \forall P: A \rightarrow (\Pi R: *_a. (R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p). \\
(\forall a: A. WB_I (P a)) \implies \\
WB_I (\lambda R: *_a. \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. \\
\forall a: A. P a R (z) Inv \text{ ops})
\]

and where predicate \( C \) may not contain \( R, z, Inv, \text{ops} \), and datatype \( A \) may not contain \( R \) (but \( a \) and \( b \) can contain \( R \) and \( \text{ops} \)).

This characterization is formalized within \( \lambda \omega^+_L \) by giving a list of properties \( WB \) satisfies, as we will do next. We will see this formalization is awkward.

Constant specifications

First, every specification of which the body \( P \) does not contain \( R, z, Inv \) or \( \text{ops} \) is well-behaved.

\[
WB_{\text{Prop}} := \ldots : \forall P: *_p. \\
WB_I (\lambda R: *_a. \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. P)
\]

For example, the specification

\[
\lambda R: *_a, \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. m =_{\text{nat}} n
\]

is well-behaved (if \( m, n: \text{Nat} \)). Note that the \( \lambda \)-abstraction over variables \( R, (z), Inv \) cannot bind these variables in \( P \). So this lemma \( WB_{\text{Const}} \) cannot be used to derive that the specification

\[
\lambda R: *_a, \lambda (z): R \rightarrow R \rightarrow *_p, \lambda Inv: R \rightarrow *_p, \lambda \text{ops}: I R. \forall x: R. x \equiv_b x
\]

is well-behaved, since \( R \) occurs in the body.
Since we work in a classical logic, and negation preserves well-behavedness, this results also holds for existential quantification:

\[ WB_{\text{Exists}} = \ldots : \forall A : \ast, \forall P : A \rightarrow (\Pi R : \ast, (R \rightarrow R \rightarrow \ast_p)) \rightarrow (R \rightarrow \ast_p) \rightarrow I \rightarrow \ast_p. \]

\[ (\forall a : A. \; WB_I (P a)) \Rightarrow \]

\[ WB_I (\lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : I \rightarrow R. \; \exists a : A. \; P a (\sim) \; Inv \; ops) \]

Quantifications over the representation type \( R \)

Now we turn towards quantifications over \( R : R \). These should be restricted to the invariant, so a specification has the form

\[ \lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : I \rightarrow R. \]

\[ \forall R : R. \; \text{Inv} \; r \Rightarrow \text{"some body in terms of} \; R, \; (\sim), \; \text{Inv} \; \text{and ops"} \]

Again, we express this neatly by abstracting over \( R, \; (\sim), \; \text{Inv} \; \text{and ops} \). But we have to pay attention, because the type of \( R \) is \( R \), so we have to abstract over \( R \) before we abstract over \( R \). We choose to make the abstraction over \( R \) the last one. So the specification can be written as

\[ \lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : I \rightarrow R. \]

\[ \forall R : R. \; \text{Inv} \; r \Rightarrow P \; R (\sim) \; \text{Inv} \; \text{ops} \; r \]

where

\[ P : \Pi R : \ast, (R \rightarrow \ast_p) \rightarrow (R \rightarrow \ast_p) \rightarrow I \rightarrow R \rightarrow \ast_p. \]

Now, we consider the demand that should be put on \( P \) to make the specification well-behaved. Roughly, it turns out that it is sufficient that \( P \) is well-behaved. But this is no formal proposition, since \( P \) has the wrong type: instead of a value of type \( I \rightarrow R \) as argument \( P \; R (\sim) \; \text{Inv} \) expects arguments of type \( I \rightarrow R \) and \( R \). We solve this by pairing these two arguments into one argument of type \( \{ \{ I : R, r : R \} \} \), and using \( WB \) on the corresponding interface \( \lambda X : \ast, \{ \{ I : X, r : X \} \} \). So we formalize the proposition that \( P \) is well-behaved by

\[ WB (\lambda X : \ast, \{ \{ I : X, r : X \} \}) (\lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : \{ \{ I : I, r : R \} \}) \]

\[ P \; (\sim) \; \text{Inv} \; \text{ops} \cdot 1 \; \text{ops} \cdot r \]

So \( \text{ops} \cdot 1 \) corresponds here to the real operations of the ADT, and \( \text{ops} \cdot r \) corresponds to the \( r : R \) over which was quantified. This results in the following formal theorem:

\[ WB_{\text{UnivRep}} = \ldots : \forall P : \Pi R : \ast, (R \rightarrow R \rightarrow \ast_p) \rightarrow (R \rightarrow \ast_p) \rightarrow I \rightarrow R \rightarrow \ast_p. \]

\[ WB (\lambda X : \ast, \{ \{ I : X, r : X \} \}) (\lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : \{ \{ I : I, r : R \} \}) \]

\[ P \; (\sim) \; \text{Inv} \; \text{ops} \cdot 1 \; \text{ops} \cdot r \Rightarrow \]

\[ WB_I (\lambda R : \ast, \lambda (\sim) : R \rightarrow \ast_p. \; \lambda Inv : R \rightarrow \ast_p. \; \lambda ops : I \rightarrow R. \]

\[ (\forall R : R. \; \text{Inv} \; r \Rightarrow P \; (\sim) \; \text{Inv} \; \text{ops} \; r) \]

For existential quantifications, we have by classical logic a similar result, where the body of the specification is \( \exists r : R. \; \text{Inv} \; r \land P \; (\sim) \; \text{Inv} \; \text{ops} \; r \).
**Example 6.6.3.1** We show that the specification $Spec$ for bags used in Section 6.5.3 is well-behaved.

$$
Spec := \lambda \text{Rep}:* \cdot \lambda (\vdash) : \text{Rep} \rightarrow \text{Rep} \rightarrow *_p \cdot \lambda \text{Inv}: \text{Rep} \rightarrow *_p \cdot \lambda \text{ops}: \text{BagI Rep}.
\forall m, n : \text{Nat}. \forall r : \text{Rep}. \text{Inv} r \implies
\begin{align*}
&\text{ops.card} m \quad \text{ops.empty} = \text{Nat} 0 \land \\
&\text{ops.card} m (\text{ops.add} m r) = \text{Nat} S (\text{ops.card} m r) \land \\
&\neg (m = \text{Nat} n) \implies \text{ops.card} m (\text{ops.add} n r) = \text{Nat} \text{ops.card} m r \land \\
&\text{ops.add} m (\text{ops.add} n r) \preceq \text{ops.add} n (\text{ops.add} m r) \land \\
&(1 \leq \text{ops.card} m r \implies m \leq \text{ops.bound} r) \\
&: \Pi \text{Rep}:* \cdot (\text{Rep} \rightarrow \text{Rep} \rightarrow *_p) \rightarrow (\text{Rep} \rightarrow *_p) \rightarrow \text{BagI Rep} \rightarrow *_p
\end{align*}
$$

First we apply $WB.Univ$ twice, and then apply $WB.UnivRep$. The remaining specification is a conjunction of 5 clauses. Since the conjunction of well-behaved specifications is also well-behaved, we have to show each of the five clauses is well-behaved. We focus here on the third, so we have to prove

$$
WB_{\lambda \text{Rep}:* \cdot \lambda (\vdash) : \text{Rep} \rightarrow \text{Rep} \rightarrow *_p \cdot \lambda \text{Inv}: \text{Rep} \rightarrow *_p \cdot \lambda \text{ops}: \text{BagI Rep}, r : \text{Rep}}.
\lambda m, n : \text{Nat}. \neg (m = \text{Nat} n) \implies
\text{ops.1.card} m (\text{ops.1.add} n \text{ops.r}) = \text{Nat} \text{ops.1.card} m \text{ops.r}
$$

in the context extended by $m, n : \text{Nat}$. Note that the record $\text{ops}$ in this formula contains both the proper operations of the ADT (in the 1 field) and the $r$ value which occurred in the universal quantification (in the $r$ field).

This is an implication, so it is sufficient if both sides of the implication are well-behaved. For the hypothesis we use $WB.Const$ (R and $\text{ops}$ do not occur in the hypothesis), and for the conclusion we use $WB.Rel$ and we are done. So we can prove $Spec$ is well-behaved, by using just the theorems above indicated by the form of $Spec$. Since $Spec$ is well-behaved, it is also sensible.

**Summary**

The theorems above span up a large inductive space of well-behaved abstract specifications. As "atoms" we have

- propositions not involving R or $\text{ops}$,
- propositions where R and $\text{ops}$ occur only in subterms which are programs (whose types do not contain R), and
- equivalences of programs of type R.

This space is closed under

- the propositional connectives,
- quantifications over a datatype (that does not contain R), and
- quantifications over R restricted to the invariant.
Not included in this space are equality over types containing \( \mathbb{R} \) and quantifications over a kind (polymorphic propositions) and over propositions. The counterexample \textit{NonsenseSpec} we have given in the beginning of Section 6.6.2 falls in the first category, since it contains \( =_\mathbb{R} \), and is therefore not well-behaved by the theorems above.

Although this space is large, it does not cover all well-behaved specifications; some specifications are well-behaved but do not fall inside this space. This means that this space is a subset of the well-behaved specifications, which is again a subset of the sensible specifications. The important thing is that we can easily prove many common specifications to be sensible, by showing they belong to this inductive space.

With this theory of well-behaved abstract specifications, we can make step 2 — giving an abstract specification — more precise. First the user gives a naive specification \textit{NaiveSpec}, abstracted over only \( \mathbb{R} \) and \( \text{ops} \). Then he abstracts over \( \simeq \) and \( \text{Inv} \), and performs the following replacements.

- Replace \( " =_\mathbb{R} " \) by \( " \simeq " \).
- Replace \( " \forall r: \mathbb{R}. " \) by \( " \forall r: \mathbb{R}. \text{Inv} r \Rightarrow " \).
- Replace \( " \exists r: \mathbb{R}. " \) by \( " \exists r: \mathbb{R}. \text{Inv} r \wedge " \).

In this way the user obtains the abstract specification \( \text{Spec} \), for which

\[
\text{Spec} \ ( \mathbb{R} \ ( =_\mathbb{R} ) \ ( \lambda r: \mathbb{R}. \text{True} ) \ \text{ops} \ ) \iff \text{NaiveSpec} \ \mathbb{R} \ \text{ops}
\]

for all \( \mathbb{R} \) and \( \text{ops} \). Finally, the user tries to prove \( WB \text{Spec} \). Because of the theorems given above, this will succeed for many specifications, in any case for all abstract specifications where \textit{NaiveSpec} can be written in first-order predicate logic. This includes all traditional specifications. Naive specifications for which this process will fail are quantifications over a polymorphic type or over a kind (both are not useful, since the interface must be simply-typed), or quantifications over propositions.

### 6.6.4 General Proof Rules for ADTs

The main result of our study of the implementations of bags was that we could prove

\[
\forall \text{Rep} : * \, . \, \forall \text{ops} : \text{BagI Rep}. \text{Implemspec Rep ops} \Rightarrow \text{UseSpec (pack (Rep, ops))}
\]

where \( \text{Implemspec} \) and \( \text{UseSpec} \) are defined in terms of a sensible specification \( \text{Spec} \). We proved this using the existence of quotients and subsets and using parametricity. In order to generalize this result we have already generalized quotients and subsets, and the notion of sensible specifications. What remains to be generalized are the notions of \( \text{Implemspec} \) and \( \text{UseSpec} \). This is straightforward. We define \( \text{MkImplemspec} \) ("make the implementor's specification") and \( \text{MkUseSpec} \) as follows: they turn an arbitrary abstract specification into
the corresponding implementor’s and user’s specification respectively.

\[
MkImplenSpec_I := \lambda Spec:\Pi R:*_p.(R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p. \\
\lambda R:*_p.\lambda ops: I R. \\
\exists (\subseteq): R \rightarrow R \rightarrow *_p. \exists Inv: R \rightarrow *_p. \\
Spec R (\subseteq) Inv ops \land \\
Bisim_I R (\subseteq) ops \land \\
IsInvair_I R Inv ops \land \\
IsERosr_R (\subseteq) Inv \\
: (\Pi R:*_p.(R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p) \rightarrow \\
(\Pi R:*_p. I R \rightarrow *_p) \\
MkUserSpec_I := \lambda Spec:\Pi R:*_p.(R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p. \\
\lambda imp:\Sigma X:*_p.I X. \\
\exists R:*_p.\exists ops: I R. \\
imp = \Sigma X:*_p.I X \text{ pack } \langle \text{Rep}, \text{ops} \rangle \land \\
Spec R (=) (\lambda x: R. True) ops \\
: (\Pi R:*_p.(R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p) \rightarrow \\
(\Sigma X:*_p.I X) \rightarrow *_p
\]

Using these notions we can formulate our general proof rule for implementing ADTs.

**Theorem 6.6.4.1 (Proof rule for implementing an ADT)**

In *QUOTSUBSET* it is provable that

\[
\forall Spec:\Pi R:*_p.(R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p. \\
WB_I Spec \implies \\
\forall \text{Rep}:*_p.\forall \text{ops}: I \text{Rep}. \\
MkImplenSpec_I Spec \text{ Rep ops} \implies \\
MkUserSpec_I Spec \langle \text{pack } \langle \text{Rep}, \text{ops} \rangle \rangle
\]

for all first-order interfaces \( I \).

**Proof:** Similar to the corresponding proofs in Sections 6.5.1, 6.5.2 and 6.5.3. Given a first-order interface \( I \), a specification \( Spec \) with \( WB_I Spec \), a datatype \( \text{Rep} \), operations \( \text{ops} \) and \( MkImplenSpec_I Spec \text{ Rep ops} \), we have to prove \( MkUserSpec_I Spec \langle \text{pack } \langle \text{Rep}, \text{ops} \rangle \rangle \), i.e.

\[\exists \text{Rep}:*_p.\exists \text{ops}:*_p.I \text{Rep}. \\
\text{pack } \langle \text{Rep}, \text{ops} \rangle = \Sigma X:*_p.I X \text{ pack } \langle \text{Rep}', \text{ops}' \rangle \land \\
\text{Spec Rep'} (=_{\text{Rep}'}) (\lambda x: \text{Rep}'. True) \text{ ops}'\]

By the axioms *exis_QuotAlgI* and *exis_SubsetAlgI* we have the existence of *QuotSubset* algebras with interface \( I \). Using with \( MkImplenSpec_I Spec \text{ Rep ops} \), we have \( \subseteq \) and \( Inv \) with \( Spec \text{ Rep } (\subseteq) Inv \text{ ops and Q and opsQ with IsQuotSubsetAlgI Rep ops } (\subseteq) Inv \text{ Q opsQ} \). From \( WB_I Spec \) follows *SensibleI Spec*, which gives with \( Spec \text{ Rep } (\subseteq) Inv \text{ ops that Spec } (=) (\lambda q: Q. True) \text{ opsQ} \). With parametricity, it is easy to show that

\[\text{pack } \langle \text{Rep}, \text{ops} \rangle = \Sigma X:*_p.I X \text{ pack } \langle Q, \text{opsQ} \rangle .\]

Now we are done by taking \( \text{Rep'}:= Q \) and \( \text{ops'}:= \text{opsQ} \). \(\Box\)
This theorem shows that steps 1 through 4 of our overview in Section 6.5.3 are valid in general, i.e. for arbitrary first-order interfaces and arbitrary specifications.

Step 5 of this overview — the use of an ADT — is easily generalized.

**Theorem 6.6.4.2 (Proof rule for using an ADT)**
In QUOTSUBSET it is provable that

\[
\forall \text{Spec} : \Pi R : *_p, (R \rightarrow R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow \text{Bag} I R \rightarrow *_p, \\
\forall \text{imp} : \Sigma X : *_p, I X. \text{MkUserSpec}1 \text{Spec} \text{imp} \implies \\
\forall A : *_p, \forall Q : A \rightarrow *_p, \forall \text{body} : (\Pi X : *_p, I X \rightarrow A). \\
(\forall X : *_p. \forall \text{ops} : I X. \text{Spec} X (=_X) (\lambda x : X. \text{True}) \text{ops} \implies \\
Q (\text{body} X \text{ops})) \implies \\
Q (\text{unpack} \text{imp} \text{as} (X, \text{ops}) \text{in} \text{body} X \text{ops})
\]

for all first-order interfaces \(I\).

**Proof:** Easy. \(\Box\)

For sake of completeness, we give the 5 steps of implementing and using an ADT which are justified by these two theorems.

1. The *user* gives the first-order interface of the ADT.

\[
I := \ldots : *_p \rightarrow *_p
\]

This determines the definition of the corresponding \(\Sigma\)-type.

\[
\text{Imp} := \Sigma \text{Rep} : *_p, I \text{Rep} \\
\quad : *_p
\]

2. The *user* gives the abstract specification, which is both abstracted over the invariant and the equality being used. With the lemmas of Section 6.6.3 the user can easily show for an "ordinary" specification that it is well-behaved.

\[
\text{Spec} := \ldots : \Pi \text{Rep} : *_p, (\text{Rep} \rightarrow \text{Rep} \rightarrow *_p) \rightarrow (\text{Rep} \rightarrow *_p) \rightarrow I \text{Rep} \rightarrow *_p \\
W_B \text{Spec} := \ldots : W_B I \text{Spec}
\]

These terms determine definitions of the user’s and the implementor’s specification.

\[
\text{UserSpec} := \text{MkUserSpec}1 \text{Spec} \\
\quad : \text{Imp} \rightarrow *_p \\
\text{ImplemSpec} := \text{MkImplemSpec}1 \text{Spec} \\
\quad : \Pi \text{Rep} : *_p, I \text{Rep} \rightarrow *_p
\]

3. The *implementor* gives a representation type and the operations, which determine the implementation value.

\[
\text{Rep1} := \ldots : *_p \\
\text{ops1} := \ldots : I \text{Rep1} \\
\text{imp1} := \text{pack} (\text{Rep1}, \text{ops1}) \\
\quad : \text{Imp}
\]


4. The implementor shows his implementation satisfies the implementor’s specification, by giving Inv1 and Sim1. This Sim1 must be an equivalence relation on Rep1 restricted to Inv1, and Sim1 must be respected by ops1. The predicate Inv1 must be an invariant for ops1, and, most importantly, the implementation must satisfy the specification with Sim1 as notion of equality, and Inv1 as invariant.

\[
\begin{align*}
\text{Inv1} & : \ldots : \text{Rep1} \rightarrow ^*_p \\
\text{Sim1} & : \ldots : \text{Rep1} \rightarrow \text{Rep1} \rightarrow ^*_p \\
\text{IsERon_Sim1} & : \ldots : \text{IsERon}_{\text{Rep1}} \text{ Sim1 Inv1} \\
\text{Bisim_Sim1} & : \ldots : \text{Bisim}_2 \text{ Sim1 ops1} \\
\text{IsInvar_Inv1} & : \ldots : \text{IsInvar}_1 \text{ Inv1 ops1} \\
\text{Spec_ops1} & : \ldots : \text{Spec Rep1 Sim1 Inv1 ops1}
\end{align*}
\]

With this Sim1 and Inv1 it is easy to show his implementation satisfies ImplemSpec and hence, with Theorem 6.6.4.1, UserSpec.

\[
\begin{align*}
\text{ImplemSpec_ops1} & : [\text{generic}] \\
\text{UserSpec_imp1} & : [\text{generic}]
\end{align*}
\]

5. The user can prove correctness of programs that use an implementation by the following proof principle, which is an instantiation of Theorem 6.6.4.2. This shows clearly that the user is not bothered with some invariant and that he can use Leibniz’ equality.

\[
\begin{align*}
\text{principle} & : \text{generic} \\
& : \forall \text{imp}: \text{Imp}. \text{UserSpec imp} \implies \\
& \forall A:*_p, \forall Q:A \rightarrow ^*_p, \forall \text{body}:(\Pi X:*_p. I X \rightarrow A). \\
& \quad (\forall X:*_p, \forall \text{ops}: I X. \text{Spec X ( =_X ) ( } \lambda r:X. \text{True } \text{ ops } \implies \\
& \quad \quad Q(\text{body X ops})) \implies \\
& \quad Q(\text{unpack imp as (X,ops) in body X ops})
\end{align*}
\]
6.7 Formalization in Yarrow

In the previous sections we have developed general proof rules for ADTs. However, this development is not totally formal, in the sense that some notions (e.g. Sim) were defined at the meta-level instead of within \( \lambda \omega^+_L \), and that well-known other notions were not defined at all. Furthermore, we ignored most of the proofs.

In this section we make the development totally formal, resulting in a \( \lambda \omega^+_L \) library for ADTs. So this library serves three purposes:

- To list all formal definitions in proper order, including some definitions we have not given before.

- To show how notions like SimplyT (simply-typed) and Sim are formalized inside \( \lambda \omega^+_L \), so they can be used in Yarrow.

- To give formal proofs we have ignored before. In this sense this library serves as an appendix to the preceding theory.

Section 6.7.1 gives some preliminary definitions, Section 6.7.2 formally develops the theory of parametricity, Section 6.7.3 shows some results for first-order interfaces, Section 6.7.4 treats quotients and subsets formally, Section 6.7.5 handles sensible and well-behaved specifications, and Section 6.7.6 gives the generalized form of the implementor’s and the user’s specification and the final result: the general proof rule for implementing an ADT (Theorem 6.6.4.1).

We want to stress that all theory has been checked with Yarrow.

6.7.1 Preliminaries

Here we give several more or less standard definitions concerning functions and relations that we need later on. First we give the definitions of the predicates IsInjection, IsSurjection and Image on functions.

\[
\begin{align*}
\text{IsInjection} & := \lambda A, B: *_{s}, \lambda f: A \rightarrow B. \forall a1, a2: A. f a1 =_{B} f a2 \implies a1 =_{A} a2 \\
& \quad : \Pi A, B: *_{s}. (A \rightarrow B) \rightarrow *_{p} \\
\text{IsSurjection} & := \lambda A, B: *_{s}, \lambda f: A \rightarrow B. \forall b: B. \exists a: A. f a =_{B} b \\
& \quad : \Pi A, B: *_{s}. (A \rightarrow B) \rightarrow *_{p} \\
\text{Image} & := \lambda A, B: *_{s}, \lambda f: A \rightarrow B. \lambda b: B. \exists a: A. f a =_{B} b \\
& \quad : \Pi A, B: *_{s}. (A \rightarrow B) \rightarrow B \rightarrow *_{p}
\end{align*}
\]

Now we turn towards relations. The prop-constructor Inverse gives the inverse of a relation, \( \text{Dom} \) gives the domain of a relation, and \( \text{IsER}_{r} \) \( R \) states \( R \) is an equivalence relation on \( Y \).

\[
\begin{align*}
\text{Inverse} & := \lambda Y, Z: *_{s}, \lambda R: Y \rightarrow Z \rightarrow *_{p}, \lambda z: Z. \lambda y: Y. R y z \\
& \quad : \Pi Y, Z: *_{s}. (Y \rightarrow Z \rightarrow *_{p}) \rightarrow Z \rightarrow Y \rightarrow *_{p} \\
\text{Dom} & := \lambda A, B: *_{s}, \lambda P: A \rightarrow B \rightarrow *_{p}, \lambda a: A. \exists b: B. P a b \\
& \quad : \Pi A, B: *_{s}. (A \rightarrow B \rightarrow *_{p}) \rightarrow A \rightarrow *_{p} \\
\text{IsER} & := \lambda Y: *_{s}, \lambda R: Y \rightarrow Y \rightarrow *_{p}, \\
& \quad (\forall y: Y. R y y) \land \\
& \quad \left( (\forall y1, y2: Y. R y1 y2 \implies R y2 y1) \land \\
& \quad (\forall y1, y2, y3: Y. R y1 y2 \implies R y2 y3 \implies R y1 y3) \right) \\
& \quad : \Pi Y: *_{s}. (Y \rightarrow Y \rightarrow *_{p}) \rightarrow *_{p}
\end{align*}
\]
We also used the expression \( \text{IsERon}_T \ R \ P \), which states that \( R \) is an equivalence on the set of all elements that satisfy \( P \). The predicate \( \text{IsERon} \) is defined as follows.

\[
\text{IsERon} \quad := \quad \lambda Y:*_*, \lambda R:Y \to Y \to *_p, \lambda P:Y \to *_p, \\
(\forall y:Y. P y \iff R y y) \land \\
(\forall y_1,y_2:Y. P y_1 \iff P y_2 \iff R y_1 y_2 \iff R y_2 y_1) \land \\
(\forall y_1,y_2,y_3:Y. P y_1 \iff P y_2 \iff P y_3 \iff R y_1 y_2 \iff R y_2 y_3 \iff R y_1 y_3) \\
: \quad \Pi Y:*_*(Y \to Y \to *_p) \to (Y \to *_p) \to *_p
\]

The composition \( \text{Comp} \) of two relations is defined as usual.

\[
\text{Comp} \quad := \quad \lambda A,B,C:*_*, \lambda (\sim):A \to B \to *_p, \lambda (\equiv):B \to C \to *_p, \\
\lambda a:A. \lambda c:C. \exists b:B. a \sim b \land b \equiv c \\
: \quad \Pi A,B,C:*_*. (A \to B \to *_p) \to (B \to C \to *_p) \to A \to C \to *_p
\]

The prop-constructor \( \text{LeftC} \) (left composition) takes a relation between two datatypes \( Y \) and \( Z \) and turns it into a relation on \( Y \) as follows. It is equivalent to taking the composition of a relation with its inverse. \( \text{RightC} \) is defined similarly.

\[
\text{LeftC} \quad := \quad \lambda Y,Z:*_*. \lambda (\sim):Y \to Z \to *_p, \lambda y_1,y_2:Y. \exists z:Z. y_1 \sim z \land y_2 \sim z \\
: \quad \Pi Y,Z:*_*. (Y \to Z \to *_p) \to Y \to Y \to *_p
\]

\[
\text{RightC} \quad := \quad \lambda Y,Z:*_*. \lambda (\sim):Y \to Z \to *_p, \lambda z_1,z_2:Z. \exists y:Y. y \sim z_1 \land y \sim z_2 \\
: \quad \Pi Y,Z:*_*. (Y \to Z \to *_p) \to Z \to Z \to *_p
\]

The restriction of a relation on a datatype to a predicate on that datatype is a new relation on that datatype.

\[
\text{Restr} \quad := \quad \lambda A:*_. \lambda R:A \to A \to *_p, \lambda P:A \to *_p, \lambda a,a':A. P a \land R a a' \land P a' \\
: \quad \Pi A:*_. (A \to A \to *_p) \to (A \to *_p) \to A \to A \to *_p
\]

Finally, the predicate \( \text{IsZclosed} \) on a relation \( R \) between two datatypes \( Y \) and \( Z \) expresses that \( R \) has the "\( \sim \)" closure property as sketched in Figure 6.6.

\[
\text{IsZclosed} \quad := \quad \lambda Y,Z:*_*. \lambda R:Y \to Z \to *_p, \\
\forall y_1,y_2:Y. \forall z_1,z_2:Z. R y_1 z_1 \iff R y_1 z_2 \iff R y_2 z_2 \iff R y_2 z_1 \\
: \quad \Pi Y,Z:*_*. (Y \to Z \to *_p) \to *_p
\]

We always write datatype arguments of all these defined variables subscript, e.g. we write \( \text{IsER}_T (\equiv) \) for \( \text{IsER}_T (\equiv) \).
6.7. FORMALIZATION IN YARROW

6.7.2 Parametricity

In this section we show how we formalize the theory of parametricity in Yarrow. The first problem we encountered is that Sim cannot be defined in \( \lambda \omega \) as a term of type

\[
\text{Sim} : \Pi I : * \rightarrow * . \Pi Y, Z : * . (Y \rightarrow Z \rightarrow *) \rightarrow (I Y \rightarrow I Z \rightarrow *) .
\]

The problem is that Sim is defined by induction on the type-constructor I, but we cannot make such definition inside \( \lambda \omega \). So we need an other way of fitting Sim in Yarrow. One solution is to build Sim in as primitive, i.e. Sim\(_I\) is an abbreviation for every simply-typed I. The disadvantage of this method is that all definitions that use Sim then also have to be built in as primitives. For example, we cannot just define

\[
\text{Bisim} := \lambda I : * . \lambda Y, Z : * . \lambda (\sim) : Y \rightarrow Y \rightarrow * . \lambda y : I Y . \text{Sim} \_I Y Y (\sim) y y
\]

because the I in Sim\(_I\) is a formal variable and hence is not simply-typed. There are a lot of definitions that use Sim (e.g. \text{IsQuotSubAlg} and \text{exis. QuotSubAlg}), including some fairly large proof terms, therefore we find it unacceptable to have all these terms built in as primitive in Yarrow. (If Yarrow were to be used for verifying realistic programs, it is useful to have such notions built in, but we use Yarrow mainly to find proof rules.)

We adopt another solution, namely to postulate a variable Sim with its properties.

\[
\text{Sim} : \Pi I : * . \Pi Y, Z : * . (Y \rightarrow Z \rightarrow *) \rightarrow (I Y \rightarrow I Z \rightarrow *)
\]

We will continue to write the interface as subscript, but now it is a formal argument of Sim; Sim\(_I\) is just a fancy notation for the application Sim I. We will write the interface as subscript throughout Section 6.7. In the properties of Sim we use Leibniz’ equality of relations, denoted by \( \iff \) (just as Leibniz’ equality on propositions and predicates, see Definition 4.3.9).

\[
\begin{align*}
\text{Sim\_id} & : \forall Y, Z : * . \forall (\sim) : Y \rightarrow Z \rightarrow * . \text{Sim} \_I X * Y Z (\sim) \iff (\sim) \\
\text{Sim\_const} & : \forall I, Y, Z : * . \forall (\sim) : Y \rightarrow Z \rightarrow * . \text{Sim} \_I X * Y Z (\sim) \iff (\equiv) \\
\text{Sim\_arrow} & : \forall I, I' : * . \forall Y, Z : * . \forall (\sim) : Y \rightarrow Z \rightarrow * . \\
& \quad \text{Sim} \_I X * Y Z (\sim) \iff \\
& \quad (\lambda ops : \text{IY} \rightarrow \text{IZ} . \text{Ops} \_I : \text{IY} \rightarrow \text{IZ}) \iff \text{Sim} \_I' Y Z (\sim) (\text{ops} Y) (\text{ops} Z z)
\end{align*}
\]

Unfortunately, we cannot quantify over all type-constructors of the form

\[
\lambda X : X. \{ l_1 : I X, \ldots, l_n : \text{In} X \}
\]

(the same problem occurred in Section 6.2). Concerning record types, we only postulate the property of Sim for interfaces \( \lambda X : X. \{ 1 : I X, r : IZ X \} \) (1 and r stand for left and right).

\[
\text{Sim\_rec} : \forall I, I' : * . \forall Y, Z : * . \forall (\sim) : Y \rightarrow Z \rightarrow * . \\
& \quad \text{Sim} \_I X * Y Z (\sim) \iff \\
& \quad (\lambda ops : \{ l : I Y, r : IIZ \} . \text{Ops} \_I : \{ l : I Y, r : IIZ \}) \iff \\
& \quad \text{Sim} \_I' Y Z (\sim) \text{ops} Y \_1 \text{ops} Z \_1 \land \text{Sim} \_I'' Y Z (\sim) \text{ops} Y \_r \text{ops} Z \_r
\]

Now we can formally quantify over all interfaces, but we have only the properties of Sim for the subset of simply-typed interfaces, where we use only record types with two fields, 1 and r.
In order to prove properties about Sim, we need to restrict quantifications over interfaces to this subset. It is characterized by a formal predicate \( \text{Simply}T: (\ast_s \rightarrow \ast_s) \rightarrow \ast_p \); it is the smallest predicate with

- \( \text{Simply}T (\lambda \mathbf{X}: \ast_s, \mathbf{X}) \).
- \( \forall \mathbf{T}: \ast_s. \text{Simply}T (\lambda \mathbf{X}: \ast_s, \mathbf{T}) \)
- \( \forall \mathbf{I}_1, \mathbf{I}_2: \ast_s \rightarrow \ast_s. \text{Simply}T \mathbf{I}_1 \Rightarrow \text{Simply}T \mathbf{I}_2 \Rightarrow \text{Simply}T (\lambda \mathbf{X}: \ast_s, \mathbf{I}_1 \mathbf{X} \rightarrow \mathbf{I}_2 \mathbf{X}) \).
- \( \forall \mathbf{I}_1, \mathbf{I}_2: \ast_s \rightarrow \ast_s. \text{Simply}T \mathbf{I}_1 \Rightarrow \text{Simply}T \mathbf{I}_2 \Rightarrow \text{Simply}T (\lambda \mathbf{X}: \ast_s, \{1 : \mathbf{I}_1 \mathbf{X}, r : \mathbf{I}_2 \mathbf{X}\}) \).

We can define this predicate in \( \lambda \omega^+_I \), in a similar fashion as we defined inductive predicates on programs in Chapter 4 (e.g. \( \leq, \text{Elem} \) and \( \text{Ordered} \)), as follows.

\[
\begin{align*}
\text{Simply}T & \quad := \quad \lambda \mathbf{I}: \ast_s \rightarrow \ast_s. \forall \mathbf{P}: (\ast_s \rightarrow \ast_s) \rightarrow \ast_p. \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
At last, we can postulate the general parametricity scheme \( \text{parPi} \), which we use to prove its corollary on existential types \( \text{parSigma} \).

\[
\text{parPi} \quad : \quad \forall \mathbf{I} : \ast \rightarrow \ast. \text{SimplyT I} \implies \\
\forall \mathbf{I} \cdot (\Pi \mathbf{X} : \ast \rightarrow \ast_\mathbf{X} . \mathbf{I} \mathbf{X}) . \forall \mathbf{Y}, \mathbf{Z} : \ast \rightarrow \ast_\mathbf{Y}. \forall (\sim) : \mathbf{Y} \rightarrow \mathbf{Z} \rightarrow \ast_\mathbf{Z} . \\
\text{Sim}_\mathbf{X} \mathbf{Y} \mathbf{Z} (\sim) (\mathbf{f} \mathbf{Y}) (\mathbf{f} \mathbf{Z})
\]

\[
\text{parSigma} \quad := \quad \ldots \quad \forall \mathbf{Y}, \mathbf{Z} : \ast \rightarrow \ast. \text{SimplyT I} \implies \\
\forall \mathbf{Y}, \mathbf{Z} : \ast \rightarrow \ast. \forall (\sim) : \mathbf{Y} \rightarrow \mathbf{Z} \rightarrow \ast_\mathbf{Z}. \forall \mathbf{y} : \mathbf{I} \mathbf{Y}. \forall \mathbf{z} : \mathbf{I} \mathbf{Z} . \\
\text{Sim}_\mathbf{Y} \mathbf{Z} (\sim) (\mathbf{y} \mathbf{z}) \implies \text{pack} \langle \mathbf{y}, \mathbf{y} \rangle = \text{pack} \langle \mathbf{y}, \mathbf{z} \rangle
\]

Next is the definition of \( \text{Bisim} \) and some properties of \( \text{Sim} \), which we need later on.

\[
\text{Bisim} \quad := \quad \lambda \mathbf{I} : \ast \rightarrow \ast. \lambda \mathbf{R} : \ast. \lambda (\sim) : \mathbf{R} \rightarrow \mathbf{R} \rightarrow \ast_\mathbf{R}. \lambda \mathbf{opsR} : \mathbf{R} . \\
\text{Sim}_\mathbf{R} \mathbf{R} (\sim) \mathbf{opsR} \mathbf{opsR} \\
\quad := \quad \Pi \mathbf{I} : \ast \rightarrow \ast. \Pi \mathbf{R} : \ast. (\mathbf{R} \rightarrow \mathbf{R} \rightarrow \ast_\mathbf{R}) \implies \mathbf{I} \mathbf{R} \rightarrow \ast_\mathbf{R}
\]

\[
\text{Sim_sym} \quad := \quad \ldots \quad \forall \mathbf{I} : \ast \rightarrow \ast. \text{SimplyT I} \implies \\
\forall \mathbf{Y}, \mathbf{Z} : \ast \rightarrow \ast. \forall (\sim) : \mathbf{Y} \rightarrow \mathbf{Z} \rightarrow \ast_\mathbf{Z}. \forall \mathbf{opsY} : \mathbf{I} \mathbf{Y}. \forall \mathbf{opsZ} : \mathbf{I} \mathbf{Z} . \\
\text{Sim}_\mathbf{R} \mathbf{Z} (\sim) \mathbf{opsY} \mathbf{opsZ} \implies \text{Sim}_\mathbf{R} \mathbf{Z} (\sim) \mathbf{opsY} \mathbf{opsZ}
\]

\[
\text{Sim_is} \quad := \quad \ldots \quad \forall \mathbf{I} : \ast \rightarrow \ast. \text{SimplyT I} \implies \forall \mathbf{R}, \mathbf{Q} : \ast. \forall (\sim) : \mathbf{R} \rightarrow \mathbf{Q} \rightarrow \ast_\mathbf{Q} . \\
\text{IsZclosed}_{\mathbf{R}, \mathbf{Q}} (\sim) \implies \text{IsZclosed}_{\mathbf{R}, \mathbf{Q}} (\sim)
\]

These properties are proved straightforwardly by induction over the simply-typed interface.

Because of the similarity of \( \text{IsInvar} \) to \( \text{Sim} \) (see Definition 6.6.1.1), we also define \( \text{IsInvar} \) here. The formalization of this lifting of predicates is similar to \( \text{Sim} \).

\[
\text{IsInvar} \quad := \quad \Pi \mathbf{I} : \ast \rightarrow \ast. \Pi \mathbf{Rep} : \ast. \ (\mathbf{Rep} \rightarrow \ast_\mathbf{Rep}) \rightarrow \mathbf{I} \mathbf{Rep} \rightarrow \ast_\mathbf{I}
\]

\[
\text{IsInvar_id} \quad := \quad \forall \mathbf{Rep} : \ast. \forall \mathbf{Inv} : \mathbf{Rep} \rightarrow \ast_\mathbf{Rep} . \text{IsInvar}_{\lambda \mathbf{R} \ast_\mathbf{Rep}} . \mathbf{Rep} \ \text{Inv} \iff \text{Inv}
\]

\[
\text{IsInvar_const} \quad := \quad \forall \mathbf{R}, \mathbf{Rep} : \ast. \forall \mathbf{Inv} : \mathbf{Rep} \rightarrow \ast_\mathbf{Rep} . \text{IsInvar}_{\lambda \mathbf{R} \ast_\mathbf{Inv}} . \mathbf{Rep} \ \text{Inv} \iff (\lambda \mathbf{t} : \mathbf{T} . \mathbf{True})
\]

\[
\text{IsInvar_arrow} \quad := \quad \forall \mathbf{I}, \mathbf{I}_1 : \ast \rightarrow \ast. \forall \mathbf{Rep} : \ast. \forall \mathbf{Inv} : \mathbf{Rep} \rightarrow \ast_\mathbf{Rep} . \\
\text{IsInvar}_{\lambda \mathbf{R} \ast_\mathbf{Rep}} . \mathbf{I}_1 \mathbf{I} \mathbf{Rep} \mathbf{Inv} \iff \\
(\lambda \mathbf{I} : \mathbf{I}_1 . \mathbf{I} \mathbf{Rep} \rightarrow \mathbf{I}_2 . \mathbf{Rep} . \\
\forall \mathbf{X} : \mathbf{I}_1 . \text{IsInvar}_{\mathbf{I}_1} . \mathbf{Rep} \mathbf{Inv} \iff \text{IsInvar}_{\mathbf{I}_2} . \mathbf{Rep} \mathbf{Inv} (\mathbf{f} \mathbf{X}))
\]

\[
\text{IsInvar_rec} \quad := \quad \forall \mathbf{I}, \mathbf{I}_1 : \ast \rightarrow \ast. \forall \mathbf{Rep} : \ast. \forall \mathbf{Inv} : \mathbf{Rep} \rightarrow \ast_\mathbf{Rep} . \\
\text{IsInvar}_{\lambda \mathbf{R} \ast_\mathbf{Rep}} . \mathbf{I}_1 \mathbf{I} \mathbf{Rep} \mathbf{Inv} \iff \\
(\lambda \mathbf{I} : [1 : \mathbf{I}_1 . \mathbf{Rep} . \mathbf{I}_2 \mathbf{Rep} ] . \\
\text{IsInvar}_{\mathbf{I}_1} . \mathbf{Rep} \mathbf{Inv} (\mathbf{f} . \mathbf{1}) \land \text{IsInvar}_{\mathbf{I}_2} . \mathbf{Rep} \mathbf{Inv} (\mathbf{f} \cdot \mathbf{I} \mathbf{Rep})
\]

### 6.7.3 First-Order Interfaces

The predicates \( \text{Basic} \) and \( \text{FirstO} \), which express that an interface is basic and first-order, respectively, are formalized in \( \lambda w_\mathbb{L}^+ \) similarly to \( \text{SimplyT} \).
\textbf{Basic} ::= \lambda I : * \to * . \forall P:(* \to *) \to *_p .
\begin{align*}
P(\lambda X : *_s . X) & \Rightarrow \\
(\forall T : *_s . P(\lambda X : *_s . T)) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . P I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . \{ I : I1 X, r : I2 X \}) & \Rightarrow \\
P I &
\end{align*}

\textbf{Basic}._id ::= \ldots : Basic(\lambda X : *_s . X)

\textbf{Basic}._const ::= \ldots : \forall T : *_s . Basic(\lambda X : *_s . T)

\textbf{Basic}._rec ::= \ldots : \forall I, I2 : *_s \to *_s . Basic I \Rightarrow Basic I2 \Rightarrow 
Basic(\lambda X : *_s . \{ I : I1 X, r : I2 X \})

\textbf{Basic}._elim ::= \ldots : \forall I : *_s \to *_s . Basic I \Rightarrow 
\forall P : (*_s \to *_s ) \to *_p . 
\begin{align*}
P(\lambda X : *_s . X) & \Rightarrow \\
(\forall T : *_s . P(\lambda X : *_s . T)) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . Basic I \Rightarrow Basic I2 \Rightarrow \\
P I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . \{ I : I1 X, r : I2 X \}) & \Rightarrow \\
P I &
\end{align*}

\textbf{FirstO} ::= \lambda I : *_s \to *_s . \forall P : (*_s \to *_s ) \to *_p .
\begin{align*}
(\forall I : *_s \to *_s . Basic I \Rightarrow P I) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . Basic I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . I1 X \to I2 X)) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . P I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . \{ I : I1 X, r : I2 X \}) & \Rightarrow \\
P I &
\end{align*}

\textbf{FirstO}._Basic ::= \ldots : \forall I : *_s \to *_s . Basic I \Rightarrow FirstO I

\textbf{FirstO}._arrow ::= \ldots : \forall I, I2 : *_s \to *_s . Basic I \Rightarrow FirstO I2 \Rightarrow 
FirstO(\lambda X : *_s . I1 X \to I2 X)

\textbf{FirstO}._rec ::= \ldots : \forall I, I2 : *_s \to *_s . FirstO I \Rightarrow FirstO I2 \Rightarrow 
FirstO(\lambda X : *_s . \{ I : I1 X, r : I2 X \})

\textbf{FirstO}._elim ::= \ldots : \forall I : *_s \to *_s . FirstO I \Rightarrow 
\forall P : (*_s \to *_s ) \to *_p .
\begin{align*}
(\forall I : *_s \to *_s . Basic I \Rightarrow P I) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . FirstO I2 \Rightarrow Basic I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . I1 X \to I2 X)) & \Rightarrow \\
(\forall I, I2 : *_s \to *_s . FirstO I \Rightarrow FirstO I2 \Rightarrow \\
P I \Rightarrow P I2 \Rightarrow \\
P(\lambda X : *_s . \{ I : I1 X, r : I2 X \}) & \Rightarrow \\
P I &
\end{align*}

In order to prove the existence of QuotSubset algebras, we need some properties of Sim for first-order interfaces. First, Sim is "transitive", i.e.
for first-order I, and for all X, Y, Z, ~, _, opsX, opsY and opsZ. In order to prove this, we need a stronger auxiliary result for basic interfaces.

Sim Basic Comp := ... : \forall I : \ast \rightarrow \ast. Basic I \implies \\
\forall X, Y, Z : \ast. \forall (\sim) : X \rightarrow Y \rightarrow \ast, \forall (\sim) : Y \rightarrow Z \rightarrow \ast.
\implies Comp_{X,Y,Z} (Sim_{X} X (\sim)) (Sim_{Y} Y (\sim)) \iff \\
Sim_{X,Z} (Comp_{X,Y,Z} (\sim) (\sim))

Sim FirstO Trans := ... : \forall I : \ast \rightarrow \ast. FirstO I \implies \\
\forall X, Y, Z : \ast. \forall (\sim) : X \rightarrow Y \rightarrow \ast, \forall (\sim) : Y \rightarrow Z \rightarrow \ast.
\implies \forall opsX : I X. \forall opsY : I Y. \forall opsZ : I Z.
\implies Sim_{X,Y} (\sim) opsX opsY \implies Sim_{Y,Z} (\sim) opsY opsZ \implies \\
Sim_{X,Z} (Comp_{X,Y,Z} (\sim) (\sim)) opsX opsZ

Second, we need that the restriction of a lifted relation to a lifted predicate is included in the lifted restriction of the relation to the predicate. Again, we need a stronger auxiliary result for basic interfaces.

Sim Basic Restr := ... : \forall I : \ast \rightarrow \ast. Basic I \implies \\
\forall Y : \ast. \forall (\sim) : Y \rightarrow \ast, \forall P : Y \rightarrow \ast.
\implies Restr_{Y} (Sim_{Y} Y (\sim)) (IsInvar_{Y} Y P) \iff \\
Sim_{Y} Y (Restr_{Y} (\sim) P)

Sim FirstO Restr := ... : \forall I : \ast \rightarrow \ast. FirstO I \implies \\
\forall Y : \ast. \forall (\sim) : Y \rightarrow \ast, \forall P : Y \rightarrow \ast.
\implies \forall opsY : opsY'. I Y.
\implies Restr_{Y} (Sim_{Y} Y (\sim)) (IsInvar_{Y} Y P) opsY opsY' \implies \\
Sim_{Y} Y (Restr_{Y} (\sim) P) opsY opsY'

Lemmas Sim FirstO trans and Sim FirstO Restr cannot be generalized to arbitrary simply-typed interfaces; the restriction to first-order interfaces is really necessary.

6.7.4 Quotients and Subsets

We first give the definitions of what it means to be a quotient algebra, a subset algebra and QuotSubset algebra.

IsQuotAlg := \lambda I : \ast \rightarrow \ast, \lambda R : \ast, \lambda opsR : I R. \lambda (\sim) : R \rightarrow \ast, \lambda Q : \ast, \lambda opsQ : I Q.
\exists \text{surj} : R \rightarrow Q.
(\forall r, r' : R. r \equiv r' \iff \text{surj} r =_{Q} \text{surj} r') \land
IsSurjection_{R,Q} \text{surj} \land
Sim_{R} Q (\lambda r : R. \lambda q : Q. q =_{Q} \text{surj} r) opsR opsQ
: \Pi I : \ast \rightarrow \ast, \Pi R : \ast, I R \rightarrow (R \rightarrow \ast) \rightarrow (IIQ : \ast, I Q \rightarrow \ast)
CHAPTER 6. $\lambda \omega^+_D$ AND MODELLING ABSTRACT DATATYPES

\begin{align*}
\text{IsSubsetAlg} & \coloneqq \lambda I: * \to *, \lambda R: * \to *, \lambda \text{opsR}: I \times R. \lambda \text{Inv}: R \to *; \lambda S: * \to S. \\
& \quad \exists \text{inj}: S \to R. \\
& \quad \text{IsInjection}_{R,S} \text{ inj} \land \\
& \quad (\text{Inv} \iff \text{Images}_{R,S} \text{ inj}) \land \\
& \quad \text{Sim}_1 R S (\lambda x: R. \lambda s: S. x =_S \text{ inj} s) \text{ opsR opsS} \\
& \quad \Pi I: * \to *, \Pi R: * \to *, \Pi I R \to (R \to *) \to (\Pi S: *, I S \to *) \\
\text{IsQuotSubsetAlg} & \coloneqq \lambda I: * \to *, \lambda R: * \to *, \lambda \text{opsR}: I \times R. \lambda \text{Inv}: R \to *; \lambda \text{Inv}: R \to *; \\
& \quad \exists \text{Q}: * \to \text{Q}. \lambda \text{opsQ}: I \times \text{Q}. \\
& \quad \exists \text{Rel}: R \to \text{Q} \to * \\
& \quad (\text{Inv} \iff \text{Dom}_{R,\text{Q}} \text{ Rel}) \land \\
& \quad (\forall q: \text{Q}; \exists r: R. \text{ Rel} r q) \land \\
& \quad (\forall r, r': R. \forall q, q': \text{Q}. \text{ Rel} r q \implies \text{ Rel} r' q' \implies \\
& \quad \quad r \simeq r' \iff q = q') \land \\
& \quad \text{Sim}_1 R \text{ Q} \text{ Rel} \text{ opsR opsQ} \\
& \quad \Pi I: * \to *, \Pi R: * \to *, \Pi I R \to (R \to R \to *) \to (R \to *) \\
& \quad (\Pi \Pi \text{Q}: *, I \text{ Q} \to *) \\
\end{align*}

Next, we postulate the two axioms stating the existence of quotients and subset algebras for first-order interfaces.

\begin{align*}
\text{exis. QuotAlg} & \coloneqq \forall I: * \to *, \text{ FirstO I} \implies \\
& \quad \forall R: * \to *, \forall \text{opsR}: I \times R. \forall (\simeq): R \to *; \\
& \quad \text{Bisim}_1 R (\simeq) \text{ opsR} \implies \text{IsEq}_R (\simeq) \implies \\
& \quad \exists \text{Q}: *. \exists \text{opsQ}: I \times \text{Q}. \text{IsQuotAlg}_1 R \text{ opsR} (\simeq) \text{ Q opsQ} \\
\text{exis. SubsetAlg} & \coloneqq \forall I: * \to *, \text{ FirstO I} \implies \\
& \quad \forall R: * \to *, \forall \text{opsR}: I \times R. \forall (\simeq): R \to *; \forall \text{Inv}: R \to *; \\
& \quad \text{IsEq}_R (\simeq) \text{ Inv} \implies \\
& \quad \text{Bisim}_1 R (\simeq) \text{ opsR} \implies \\
& \quad \text{IsInvary} R \text{ Inv} \text{ opsR} \implies \\
& \quad \exists \text{Q}: * \to \text{Q}. \exists \text{opsQ}: I \times \text{Q}. \text{IsSubsetAlg}_1 R \text{ opsR} (\simeq) \text{ Inv Q opsQ} \\
\end{align*}

From these two axioms we can prove the existence of QuotSubset algebras using the properties \text{Sim}_1 \text{ FirstO.trans} and \text{Sim}_1 \text{ FirstO.Restr.}

\begin{align*}
\text{exis. QuotSubsetAlg} & \coloneqq \ldots \Rightarrow \forall I: * \to *, \text{ SimplyT I} \implies \\
& \quad \forall R: * \to *, \forall \text{opsR}: I \times R. \forall (\simeq): R \to *; \forall \text{Inv}: R \to *; \\
& \quad \text{IsEq}_R (\simeq) \text{ Inv} \implies \\
& \quad \text{Bisim}_1 R (\simeq) \text{ opsR} \implies \\
& \quad \text{IsInvary} R \text{ Inv} \text{ opsR} \implies \\
& \quad \exists \text{Q}: * \to \text{Q}. \exists \text{opsQ}: I \times \text{Q}. \text{IsQuotSubsetAlg}_1 R \text{ opsR} (\simeq) \text{ Inv Q opsQ} \\
\text{exis. SubsetAlg} & \coloneqq \ldots \Rightarrow \\
& \quad \forall I: * \to *, \text{ SimplyT I} \implies \\
& \quad \forall R: * \to *, \forall \text{opsR}: I \times R. \forall (\simeq): R \to *; \forall \text{Inv}: R \to *; \\
& \quad \forall Q: * \to \text{Q}. \forall \text{opsQ}: I \times \text{Q}. \forall \text{R opsR} (\simeq) \text{ Inv Q opsQ} \implies \\
& \quad \text{pack} (R, \text{opsR}) = \Sigma I: * \to \text{I} \times \text{pack} (Q, \text{opsQ}) \\
\end{align*}
6.7. FORMALIZATION IN YARROW

6.7.5 Sensible and Well-Behaved Specifications

In this section we formally define Sensible and WB, and give a number of their properties, together with the proofs. We do not repeat properties that can be easily derived from the properties below (e.g. the conjunction of well-behaved specifications is well-behaved again).

First we repeat the definition of Sensible.

\[
\begin{align*}
\text{Sensible} &:= \lambda I:*_p \rightarrow *_p, \lambda \text{Spec:} \text{IR:*}_p, (R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p, \\
& \forall R:*_p, \forall \text{opsR:} I R, \forall \text{qR:*}_p, \forall \text{Inv:} R \rightarrow *_p, \forall \text{qQ:} *_p, \forall \text{opsQ:} I Q, \\
& \text{IsQuotSubsetAlg}_1 R \text{ opsR (\sim) Inv Q opsQ} \Rightarrow \\
& \text{Spec R (\sim) Inv opsR} \Rightarrow \text{Spec Q (\sim)} (\lambda q: Q, \text{True}) \text{ opsQ} \\
& : \Pi I:*_p \rightarrow *_p, (\Pi R:*_p, (R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p) \rightarrow *_p
\end{align*}
\]

Our definition of well-behaved (WB) is complicated, and found by a lot of experimenting, where we had two properties for WB in mind. First, that the collection of well-behaved specifications is a subset of the sensible specifications (theorem WB..Sensible below). Second, that the collection of well-behaved specifications has nice syntactical properties, which we have given in Section 6.6.3, and which will be repeated below. Remember that programmers will never have to use the definition of WB, but use only its properties.

\[
\begin{align*}
\text{WB} &:= \lambda I:*_p \rightarrow *_p, \lambda \text{Spec:} \text{IR:*}_p, (R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p, \\
& \forall Y, Z:*_p, \forall \text{RelY:} Y \rightarrow *_p, \forall \text{RelZ:} Z \rightarrow *_p, \\
& \forall \text{InvY:} Y \rightarrow *_p, \forall \text{InvZ:} Z \rightarrow *_p, \\
& \text{IsERen}_Y \text{ RelY InvY} \Rightarrow \text{IsERen}_Z \text{ RelZ InvZ} \Rightarrow \\
& \forall \text{opsY:} I Y, \forall \text{opsZ:} I Z, \\
& (\exists (\sim): Y \rightarrow Z \rightarrow *_p). \\
& \text{Sim}_Y Z (\sim) \text{ opsY opsZ} \land \\
& \text{Restry} \text{ RelY InvY} \leftrightarrow \text{LeftC}_{Y,Z} (\sim) \land \\
& \text{Restry} \text{ RelZ InvZ} \leftrightarrow \text{RightC}_{Y,Z} (\sim) \land \\
& \text{IsZclosed}_{Y,Z} (\sim) \Rightarrow \\
& \text{Spec Y RelY InvY opsY} \Rightarrow \text{Spec Z RelZ InvZ opsZ} \\
& : \Pi I:*_p \rightarrow *_p, (\Pi R:*_p, (R \rightarrow *_p) \rightarrow (R \rightarrow *_p) \rightarrow I R \rightarrow *_p) \rightarrow *_p
\end{align*}
\]

This definition says a specification Spec is well-behaved if Spec Y RelY InvY opsY implies Spec Z RelZ InvZ opsZ for all Y, RelY, InvY, opsY, Z, RelZ, InvZ, opsZ that satisfy some conditions. The relations RelY and RelZ play the role of the equivalence in the specification, and InvY and InvZ play the role of the invariant. The conditions are:

- RelY and RelZ are equivalence relations on InvY and InvZ respectively (this condition is also a premise for the existence of QuotSubset algebras), and

- there is a simulation \(\sim\) between opsY and opsZ (so pack \(\langle Y, \text{opsY}\rangle\) and pack \(\langle Z, \text{opsZ}\rangle\) are equal), with LeftC\(_{Y,Z}\) (\(\sim\)) equivalent to Restry RelY InvY and RightC\(_{Y,Z}\) (\(\sim\)) equivalent to Restry RelZ InvZ, and which has the Z closure property.

Note that the conditions in WB are symmetric under swapping Y, RelY, InvY and opsY with Z, RelZ, InvZ and opsZ. Because of this symmetry, WB has nice syntactical properties. This stands in contrast to Sensible, which is not symmetric in this sense, and has not such nice syntactical properties.
With this definition of $WB$, it is easy to show that all well-behaved specifications are sensible.

$$WB_{\text{Sensible}} := \ldots \vdash \forall I: \ast \rightarrow \ast . Simply \ T \ I \rightarrow \forall Spec: \Pi R: \ast , (R \rightarrow R \rightarrow \ast _p) \rightarrow (R \rightarrow \ast _p) \rightarrow I \ R \rightarrow \ast _p.$$  

$$WB_{\text{Spec}} \rightarrow \forall WB_{\text{Spec}} \rightarrow \forall Sensible_{\text{Spec}}$$

Proof: Assume $WB_{\text{Spec}}$. Unfold the definition of $Sensible_{\text{Spec}}$. Assume $R: \ast$, $opsR: I \ Q$, $(\sim): R \rightarrow R \rightarrow \ast _p$, $Inv: R \rightarrow \ast _p$, $Q: \ast$, $opsQ: I \ Q$ and $IsQuot SubsetAlg_R \ opsR (\sim) Inv \ Q \ opsQ$.

We have left to prove

$$Spec \ R (\sim) Inv \ opsR \rightarrow Spec \ Q (\equiv) (\lambda q: Q. True) \ opsQ.$$  

Just apply $WB_{\text{Spec}}$, taking $Y := R$, $RelY := (\sim)$, $InvY := Inv$, $opsY := ops$, $Z := Q$, $RelZ := (\equiv)$, $InvZ := \lambda q: Q. True$, and $opsZ := opsQ$. This leaves us with 3 conditions: $IsERon_R (\sim) Inv$ and $IsERon_Q (\equiv) (\lambda q: Q. True)$ and the existence of a $\sim$ with some properties. All these conditions follow from $IsQuot SubsetAlg_{\text{Spec}} \ opsR (\sim) Inv \ Q \ opsQ$. □

Now we move on to the theorems that span up a large space of well-behaved specifications.

$$WB_{\text{Prop}} := \ldots \vdash \forall I: \ast \rightarrow \ast , \forall P: \ast. \ WB_{\text{Prop}} (\lambda R: \ast, \lambda (\sim): R \rightarrow R \rightarrow \ast _p, \lambda Inv: R \rightarrow \ast _p, \lambda ops: I \ R \ P)$$

The proof of this property is trivial.

$$WB_{\text{Impl}} := \ldots \vdash \forall I: \ast \rightarrow \ast . Simply \ T \ I \rightarrow \forall P, Q: \Pi R: \ast , (R \rightarrow R \rightarrow \ast _p) \rightarrow (R \rightarrow \ast _p) \rightarrow I \ R \rightarrow \ast _p.$$ 

$$WB_{\text{Impl}} (\lambda R: \ast, \lambda (\sim): R \rightarrow R \rightarrow \ast _p, \lambda Inv: R \rightarrow \ast _p, \lambda ops: I \ R \ P) \rightarrow \forall P R, Q \ R (\sim) Inv \ ops \rightarrow Q R (\sim) Inv \ ops$$

Proof: Assume $WB_{\text{Impl}} P$ and $WB_{\text{Impl}} Q$. We have to show that given $Y, Z, \ldots, \ opsZ$ as in the definition of $WB$ that

$$(P Y RelY InvY opsY \rightarrow Q Y RelY InvY opsY) \rightarrow (P Z RelZ InvZ opsZ \rightarrow Q Z RelZ InvZ opsZ).$$

By $WB_{\text{Impl}} Q$, taking the same $Y, Z$ etc., we have

$$Q Y RelY InvY opsY \rightarrow Q Z RelZ InvZ opsZ .$$

By $WB_{\text{Impl}} P$, where we interchange $Y$ and $Z$ etc., we have

$$P Z RelZ InvZ opsZ \rightarrow P Y RelY InvY opsY ,$$

if we can prove the 3 conditions with $Y$ and $Z$ interchanged. For the first two conditions this is trivial. For the last condition, the existence of a simulation, this is easy, using $Sim_{\text{sym}}$. Then we are done by simple propositional logic. □
Predicates over a datatype not containing the representation type R are well-behaved:

\[ WB_{Pred} := \ldots : \forall I : \ast_i \rightarrow \ast_i. Simply T I \rightarrow \forall A : \ast_i. \forall f : \Pi R : \ast_i. I R \rightarrow A. \forall P : A \rightarrow \ast_p. \]
\[ WB_1 (\lambda R : \ast_i. \lambda (\sim) : R \rightarrow \ast_p. \lambda Inv : R \rightarrow \ast_p. \lambda \text{ops} : I R. P (f R \text{ops})) \]

**Proof:** Assume \( A : \ast_i, f : \Pi \text{Rep} : \ast_i, I \text{Rep} \rightarrow A \) and \( P : A \rightarrow \ast_p \). Given \( Y, Z, \ldots, \text{opsZ} \) as in the definition of \( WB \), we have to show that

\[ P (f Y \text{opsY}) \rightarrow P (f Z \text{opsZ}) . \]

Because of the existence of a simulation \( \sim \) between \( \text{opsY} \) and \( \text{opsZ} \), we have by the parametricity axiom \( \text{parPi} \) that \( f Y \text{opsY} =_A f Z \text{opsZ} \). So we are done.

So here we use the general version of the parametricity axiom; before we used only the special instantiation \( \text{parSigma} \).

\[ WB_{Univ} := \ldots : \forall I : \ast_i \rightarrow \ast_i. \forall A : \ast_i. \forall P : A \rightarrow (\Pi R : \ast_i. (R \rightarrow R \rightarrow \ast_p) \rightarrow (R \rightarrow \ast_p) \rightarrow I R \rightarrow \ast_p). \]
\[ (\forall a : A. WB_1 (P a)) \rightarrow \]
\[ WB_1 (\lambda R : \ast_i. \lambda (\sim) : R \rightarrow \ast_p. \lambda Inv : R \rightarrow \ast_p. \lambda \text{ops} : I R. \]
\[ \forall a : A. P \text{ a R} (\sim) \text{Inv} \text{ops} ) \]

The proof of this property is straightforward.

\[ WB_{Tw} := \ldots : \forall I : \ast_i \rightarrow \ast_i. Simply T I \rightarrow \forall f, g : \Pi R : \ast_i. I R \rightarrow R. \]
\[ WB_1 (\lambda R : \ast_i. \lambda (\sim) : R \rightarrow \ast_p. \lambda Inv : R \rightarrow \ast_p. \lambda \text{ops} : I R. \]
\[ f \text{ R ops} \sim g R \text{ ops} ) \]

The proof of this property uses \( \text{parPi} \).

\[ WB_{UnivRep} := \ldots : \forall I : \ast_i \rightarrow \ast_i. Simply T I \rightarrow \forall P : \Pi R : \ast_i. (R \rightarrow R \rightarrow \ast_p) \rightarrow (R \rightarrow \ast_p) \rightarrow I R \rightarrow R \rightarrow \ast_p. \]
\[ WB_1 (\lambda R : \ast_i. \lambda (\sim) : R \rightarrow R \rightarrow \ast_p. \lambda \text{Inv} : R \rightarrow \ast_p. \lambda \text{ops} : \{} \text{l} : I R, r : R \{. \]
\[ P R (\sim) \text{Inv} \text{ops} \text{l} \text{ops} \text{r} ) \rightarrow \]
\[ WB_1 (\lambda R : \ast_i. \lambda (\sim) : R \rightarrow \ast_p. \lambda Inv : R \rightarrow \ast_p. \lambda \text{ops} : I R. \]
\[ \forall r : R. Inv r \rightarrow P R (\sim) \text{Inv} \text{ops} r ) \]

**Proof:** Relatively simple: just some predicate logic, using the definitions of \( WB, Sim, LeftC, RightC \) and \( \text{Restr} \).  \(\square\)
6.7.6 The Implementor’s and the User’s Specification

The main results of this chapter, given in Section 6.6 are now easy to formalize.

\begin{align*}
\text{MkUserSpec} & \quad := \quad \lambda I: \star_s \rightarrow \star_s \cdot \\
& \quad \lambda Spec:\Pi R: \star_s, (R \rightarrow R \rightarrow \star_p) \rightarrow (R \rightarrow \star_p) \rightarrow I R \rightarrow \star_p \cdot \\
& \quad \lambda \text{imp}: \Sigma X: \star_s, I X \cdot \\
& \quad \exists R: \star_s, \exists \text{ops}: I R \cdot \\
& \quad \quad \text{imp} = \Sigma X: \star_s, I X \ \text{pack} (R, \text{ops}) \land \\
& \quad \quad \text{Spec} R (\equiv) (\lambda x: R. \ True) \ \text{ops} \\
& \quad : \quad \Pi I: \star_s \rightarrow \star_s, (\Pi R: \star_s, (R \rightarrow R \rightarrow \star_p) \rightarrow (R \rightarrow \star_p) \rightarrow I R \rightarrow \star_p) \rightarrow \\
& \quad \quad (\Sigma X: \star_s, I X) \rightarrow \star_p \\
\text{MkImplemSpec} & \quad := \quad \lambda I: \star_s \rightarrow \star_s \cdot \\
& \quad \lambda Spec:\Pi R: \star_s, (R \rightarrow R \rightarrow \star_p) \rightarrow (R \rightarrow \star_p) \rightarrow I R \rightarrow \star_p \cdot \\
& \quad \lambda R: \star_s, \lambda \text{ops}: I R \cdot \\
& \quad \exists (\equiv): R \rightarrow R \rightarrow \star_p, \exists \text{Inv}: R \rightarrow \star_p, \\
& \quad \text{Spec} R (\equiv) \ \text{Inv} \ \text{ops} \land \\
& \quad \text{Bisim}_R (\equiv) \ \text{ops} \land \\
& \quad \text{IsInvVar}_R \ \text{Inv} \ \text{ops} \land \\
& \quad \text{IsErone} (\equiv) \ \text{Inv} \\
& \quad : \quad \Pi I: \star_s \rightarrow \star_s, (\Pi R: \star_s, (R \rightarrow R \rightarrow \star_p) \rightarrow (R \rightarrow \star_p) \rightarrow I R \rightarrow \star_p) \rightarrow \\
& \quad \quad (\Sigma X: \star_s, I X \rightarrow \star_p)
\end{align*}

Theorem 6.6.4.1 — our proof rule for implementing an ADT — is formalized as follows.

\[ \text{ImplemUserSpec} \quad := \quad \ldots : \quad \forall I: \star_s \rightarrow \star_s, \text{FirstO} I \implies \\
& \quad \forall Spec:\Pi R: \star_s, (R \rightarrow R \rightarrow \star_p) \rightarrow (R \rightarrow \star_p) \rightarrow I R \rightarrow \star_p, \\
& \quad \forall \text{Rep}: \star_s, \forall \text{ops}: I \text{Rep} \cdot \\
& \quad \text{MkImplemSpec} \ \text{Spec} \ \text{Rep} \ \text{ops} \implies \\
& \quad \text{MkUserSpec} \ \text{Spec} \ \text{pack} (\text{Rep}, \text{ops}) \]

Finally, we have Theorem 6.6.4.2.

\[ \text{principle} \quad := \quad \ldots : \quad \forall I: \star_s \rightarrow \star_s, \\
& \quad \forall Spec:\Pi \text{Rep}: \star_s, (\text{Rep} \rightarrow \text{Rep} \rightarrow \star_p) \rightarrow (\text{Rep} \rightarrow \star_p) \rightarrow I \text{Rep} \rightarrow \star_p, \\
& \quad \forall \text{imp}: \Sigma X: \star_s, I X, \text{MkUserSpec} \ \text{Spec} \ \text{imp} \implies \\
& \quad \forall X: \star_s, \forall Q: A \rightarrow \star_p, \forall \text{body}: \Sigma X: \star_s, I X \rightarrow A. \\
& \quad (\forall X: \star_s, \forall \text{ops}: I X, \text{Spec} X (\equiv) (\lambda x: X. \ True) \ \text{ops} \implies \\
& \quad \quad Q (\text{body} X \ \text{ops}) \implies \\
& \quad \quad Q (\text{unpack} \ \text{imp} \ \text{as} (X, \text{ops}) \ \text{in} \ \text{body} X \ \text{ops}) \]

Let us make a remark about how we formalized this theory in Yarrow. We were forced to limit the definition of Sim (and everything that relies on it) to interfaces with only record types with only an 1 and r field (Section 6.7.2). So, we cannot formalize the examples we have given using this formal Sim, since they contain records with other labels (e.g. empty, add, card and count). For the examples, we have chosen the solution to define a new variable, e.g.
SimBagI, by hand. Furthermore, we made a specific copy of all general axioms and definitions for BagI, e.g.

\[
\begin{align*}
\text{BisimBagI} & \quad := \quad \lambda y : * \cdot \lambda (\cdot) : Y \to Y \to *_p \cdot \lambda y : \text{BagI} \ Y. \ \text{SimBagI} \ Y \ (\cdot) \ y \ y \\
& \quad \quad : \quad \Pi Y : * \cdot (Y \to Y \to *_p) \to \text{BagI} \ Y \to *_p
\end{align*}
\]

This is not entirely satisfactory, since we can make errors in defining SimBagI or in the process of copying. Furthermore, it is remotely possible that the results for two field records do not generalize to arbitrary records. We think this is not a reasonable objection, since we can encode arbitrary records with two field records. The totally safe alternative would be to use only those two field records (or cartesian products, for that matter), forcing the encoding upon the examples. From a pragmatic and didactic point of view, this is very awkward.

So, for the formalization of general proof rules, the present development with only two field records is entirely satisfactory, but for the use of proof rules for (practical) examples, it is not.
6.8 Axiomatizing Quotients and Subsets

This section studies the axioms that postulate the existence of quotient and subset algebras. This section is quite technical, and is not directly concerned with proof rules for ADTs. In Section 6.8.1 we discuss an alternative for the axioms we have introduced in Section 6.6, retaining the restriction to first-order interfaces. In Section 6.8.2 we work this alternative out further, yielding quotients and subsets for arbitrary simply-typed interfaces, including higher-order ones. We also discuss the proof rules resulting from this alternative. In Section 6.8.3 we give a brief summary concerning our axiomatizations of quotients and subsets.

6.8.1 First-Order Algebras

In Definition 6.6.1.4 we defined QUOTSUBSET, with two axioms that postulate the existence of quotient and subset algebras for first-order interfaces. These are rather complicated axioms, and it would be better if we could replace them by simpler, more basic axioms. In this section we discuss an attempt to do so. This discussion focuses on quotients, but a similar story holds for subsets.

Let us repeat the axiom that postulates quotient algebras:

\[
\text{exis\textunderscore QuotAlg} \quad : \quad \forall I:*, \rightarrow *, \text{FirstO I} \implies \\
\quad \forall R:*, \forall \text{opsR}: I R. \forall (\equiv): R \rightarrow R \rightarrow *, \exists \text{Ops}_R: I R \equiv (\equiv) \rightarrow I \text{QuotAlg}_R R \text{opsR} (\equiv) \implies Q \text{opsQ}
\]

where

\[
\text{IsQuotAlg} \quad := \quad \lambda I:*, \rightarrow *, \lambda R:*, \lambda \text{opsR}: I R. \lambda (\equiv): R \rightarrow R \rightarrow *, \lambda Q:*, \lambda \text{opsQ}: I Q. \\
\quad \exists \text{surj}: R \rightarrow Q. \\
\quad (\forall r, r': R. r \equiv r' \iff \text{surj} r =_Q \text{surj} r') \land \\
\quad \text{IsSurjection}_R Q \text{surj} \land \\
\quad \text{Sim}_R Q \rightarrow Q. \lambda r R. \lambda q Q. q =_Q \text{surj} r) \text{opsR} \text{opsQ} \\
\quad : \quad \Pi I:*, \rightarrow *, \Pi R:*, I R \rightarrow (R \rightarrow R \rightarrow *) \rightarrow ((Q:*, I Q \rightarrow *)
\]

We consider \text{exis\textunderscore QuotAlg} to be rather complicated, because it depends on the interface I and the notion of simulation \text{Sim}_R. We need I and \text{Sim}_R because we consider not only the datatype R, but also the operations \text{opsR}: I R; the pair (R, \text{opsR}) forms an algebra with interface (signature) I. If we ignore the operations we do not need to consider I or \text{Sim}_R, and we just postulate the existence of quotient types:

\[
\text{exis\textunderscore QuotType} \quad : \quad \forall R:*, \forall (\equiv): R \rightarrow R \rightarrow *, \text{IsER}_R (\equiv) \implies \exists Q:*, \text{IsQuotType} R (\equiv) Q
\]

where

\[
\text{IsQuotType} \quad := \quad \lambda R:*, \lambda (\equiv): R \rightarrow R \rightarrow *, \lambda Q:*, \exists \text{surj}: R \rightarrow Q. \\
\quad (\forall r, r': R. r \equiv r' \iff \text{surj} r =_Q \text{surj} r') \land \\
\quad \text{IsSurjection}_R Q \text{surj} \land \\
\quad \Pi R:*, (R \rightarrow R \rightarrow *) \rightarrow (*_*, \rightarrow _*)
\]
6.8. AXIOMATIZING QUOTIENTS AND SUBSETS

This axiom is considerably simpler, because it does not depend on I or Sim₁. These quotient types are a bit weaker than the quotient types occurring in the type-theoretic literature [Bar95, Hof95].

Now we would like to infer \( \text{exis}_\cdot \text{QuotAlg} \) from \( \text{exis}_\cdot \text{QuotType} \). So let us suppose we have a first-order \( I, R, \text{ops} R, \sim \) with

\[
\begin{align*}
\text{Bisim}_1 R (\sim) \text{ops} R & \quad (i) \\
\text{IsER}_R (\sim) & \quad (ii)
\end{align*}
\]

Now we have to prove

\[
\exists Q: \ast, \exists \text{ops} Q: I Q, \text{IsQuotAlg}_R R \text{ops} R (\sim) Q \text{ops} Q.
\]

Using \( \text{exis}_\cdot \text{QuotType} \) with (ii) we have \( Q: \ast \) and \( \text{surj}: R \to Q \) with

\[
\forall r, r': R. r \sim r' \iff \text{surj} r =_Q \text{surj} r' \quad (iii)
\]

\[
\land \text{IsSurjection}_R Q \text{surj}
\]

Unfolding the definition of \( \text{IsQuotAlg} \), we see we have left to prove

\[
\exists \text{ops} Q: I Q, \text{Sim}_1 R Q (\lambda r: R. \lambda q: Q. q =_Q \text{surj} r) \text{ops} R \text{ops} Q. 
\] (iv)

However, this is impossible to prove for arbitrary first-order \( I \). First we consider a small example where we manage to make operations \( \text{ops} Q \) with (iv).

**Example 6.8.1.1** Take \( I \equiv \lambda X: \ast, X \to \text{Nat}, \) so \( \text{ops} R: \text{Nat} \to R \). For \( \text{ops} Q \) we can just take \( \lambda n: \text{Nat}. \text{surj} (\text{ops} R n) \). It is trivial to show that with this \( \text{ops} Q \) we have (iv). \( \square \)

**Example 6.8.1.2** Now consider the first-order interface \( I \equiv \lambda X: \ast, X \to \text{Nat}, \) so \( \text{ops} R: R \to \text{Nat}, \) and by (i) we have

\[
\forall r, r': R. r \sim r' \implies \text{ops} R r =_{\text{Nat}} \text{ops} R r'. 
\] (v)

We cannot define \( \text{ops} Q \) as we did in Example 6.8.1.1. We would like to define \( \text{ops} Q \equiv \lambda q: Q. \text{ops} R (\text{surj}^{-1} q) \), where \( \text{surj}^{-1} q \) is the set of all \( r \) with \( \text{surj} r =_q q \); intuitively this is a correct function definition since all elements in \( \text{surj}^{-1} q \) are related (because of (iii)), and \( \text{ops} R \) gives the same result for all related elements (property (v)). However, we cannot make such an inverse \( \text{surj}^{-1} \) in \( \omega^2 \).

The most obvious extension of our logic to make such an inverse is to postulate the axiom of choice. The axiom of choice (defined below) allows us to infer

\[
\exists \text{inverse}: Q \to R, \forall q: Q. \text{surj} (\text{inverse} q) =_q q,
\]

i.e. the existence of an inverse of \( \text{surj} \), from

\[
\forall q: Q. \exists r: R. \text{surj} r =_q q,
\]

i.e. from the fact that \( \text{surj} \) is a surjection. (Alternatively, we could infer the existence of a proper \( \text{ops} Q \) directly with the axiom of choice, without using an inverse of the surjection.)

We see no way of proving (iv) without postulating the axiom of choice. \( \square \)
So we cannot prove the existence of quotient algebras from the existence of quotient types (for all first-order interfaces) without the axiom of choice. But the axiom of choice is the only additional axiom we need: from the existence of quotient types and the axiom of choice we can infer the existence of quotient algebras. A similar result holds for subset algebras. This is formally captured as follows.

**Definition 6.8.1.3** The context \( AC \) is the context \( PAR \) (Definition 6.4.4.1) plus the following two definitions and three axioms.

\[
\begin{align*}
\text{IsQuotType} & : = \lambda R : * \cdot, \lambda (\equiv) : R \rightarrow R \rightarrow *_p, \lambda Q : *_s, \\
& \quad \exists \text{surj} : R \rightarrow Q, \\
& \quad (\forall r, r' : R. r \equiv r' \iff \text{surj} r =_Q \text{surj} r') \land \\
& \quad \text{IsSurjection}_{R, Q} \text{surj} \\
\text{IsSubsetType} & : = \lambda R : * \cdot, \lambda \text{Inv} : R \rightarrow *_p, \lambda S : *_s, \\
& \quad \exists \text{inj} : S \rightarrow R, \\
& \quad \text{IsInjection}_{S, R} \text{inj} \land \\
& \quad (\text{Inv} \iff \text{Image}_{S, R} \text{inj}) \\
& \quad \exists \text{is}_*, \exists \text{is}_s. (R \rightarrow *_p) \rightarrow *_s \rightarrow *_p \\
\text{exis. QuotType} & : = \forall R : * \cdot, \forall (\equiv) : R \rightarrow R \rightarrow *_p. \text{IsER}_{R} (\equiv) \Rightarrow \exists Q : *_s. \text{IsQuotType} R (\equiv) Q \\
\text{exis. SubsetType} & : = \forall R : * \cdot, \forall \text{Inv} : R \rightarrow *_p. \exists S : *_s. \text{IsSubsetType} R \text{ Inv} S \\
\text{axiom. choice} & : = \forall A, B : * \cdot, \forall P : A \rightarrow B \rightarrow *_p, \\
& \quad (\forall x : A. \exists y : B. P x y) \Rightarrow (\exists f : A \rightarrow B. \forall x : A. P x (f x))
\end{align*}
\]

\( \square \)

**Theorem 6.8.1.4** In \( AC \) the axioms \( \text{exis. QuotAlg} \), \( \text{exis. SubsetAlg} \) and \( \text{exis. QuotSubsetAlg} \) of \( QUOTSUBSET \) (Definition 6.6.1.4) are provable for all first-order \( I \).

So we could replace the axioms for quotient and subset algebras (in context \( QUOTSUBSET \)) by the simpler axioms for quotient and subset types and the simple axiom of choice; every proposition that is derivable in \( QUOTSUBSET \) is derivable in \( AC \).

Unfortunately, this axiom of choice does not hold in PER models! Therefore we have postulated the existence of quotient and subset algebras rather than the existence of quotient and subset types in combination with the axiom of choice.

### 6.8.2 Higher-Order Algebras

We conjecture the validity of axioms for quotient and subset algebras for arbitrary simply-typed interfaces, including higher-order ones. First we consider quotients, then subsets, and then quotients of subsets. We will see that we have to adapt the formulation of some axioms.

**Quotients**

First we consider quotient algebras for arbitrary simply-typed interfaces.
6.8. AXIOMATIZING QUOTIENTS AND SUBSETS

Conjecture 6.8.2.1 In the PER model of [Pol94] the following axiom is valid.

\[
\exists \text{is}_\text{QuotAlgHO} : \forall i : \star, \text{Sim}_T i \implies \\
\forall R : \star, \forall \text{ops}_R : I R. \forall \langle \sim \rangle : R \to R \to \star,
\]
\[
\text{Bisim}_1 R \langle \sim \rangle \text{ops}_R \implies \text{IsER}_R \langle \sim \rangle \implies \\
\exists q : \star. \exists \text{ops}_Q : I Q. \text{IsQuotAlg}_R R \text{ops}_R \langle \sim \rangle Q \text{ops}_Q
\]

This conjecture is exactly the same as \(\exists \text{is}_\text{QuotAlg}\), except that it quantifies over all simply-typed interfaces. Note that the simply-typed interfaces include higher-order interfaces like \(\lambda X : \star. ( X \to X ) \to X \); hence the suffix \(HO\) in the name of this lemma. We have two reasons for this conjecture. First, \(\exists \text{is}_\text{QuotAlgHO}\) is a generalization of \(\exists \text{is}_\text{QuotAlg}\), which is valid (see Definition 6.6.1.4). Second, and more importantly, we can prove \(\exists \text{is}_\text{QuotAlgHO}\) in \(AC\), i.e. using the axiom of choice and the existence of quotient types. The key lemma to this proof is the following lemma:

\[
\text{Sim}_L C : \ldots : \forall i : \star. \text{Sim}_T i \implies \\
\forall Y, Z : \star. ( \exists r : Y. \text{True} ) \implies ( \exists s : Z. \text{True} ) \implies \\
\forall \langle \sim \rangle : Y \to Z \to \star. \text{IsZclosed}_Y Z \langle \sim \rangle \implies \\
\text{Sim}_1 Y Y ( \text{Left}_C Y Z \langle \sim \rangle ) \iff \text{Left}_C Y Y ( \text{Sim}_1 Y Z \langle \sim \rangle )
\]

This might seem obscure, but replacing the equivalence of relations by equivalence of propositions, and taking only the implication \(\implies\) gives (we ignore the quantifications over \(I, Y, Z, \) and \(\sim\)):

\[
\forall \text{ops}_Y, \text{ops}_Y' : I Y. \\
\text{Sim}_1 Y Y ( \text{Left}_C Y Z \langle \sim \rangle ) \text{ops}_Y \text{ops}_Y' \implies \\
\text{Left}_C Y Y ( \text{Sim}_1 Y Z \langle \sim \rangle ) \text{ops}_Y \text{ops}_Y'
\]

Now taking \(\text{ops}_Y\) and \(\text{ops}_Y'\) equal, using the definition of \(\text{Bisim}\), and unfolding \(\text{Left}_C\) yields:

\[
\forall \text{ops}_Y : I Y. \\
\text{Bisim}_1 Y Y ( \text{Left}_C Y Z \langle \sim \rangle ) \text{ops}_Y \implies \\
\exists \text{ops}_Z : I Z. \text{Sim}_1 Y Z \langle \sim \rangle \text{ops}_Z \text{ops}_Z
\]

Now the use of \(\text{Sim}_L C\) in proving \(\exists \text{is}_\text{QuotAlgHO}\) should not be so obscure any more: we take \(Y \equiv R, Z \equiv \mathcal{Q}, \text{ops}_Y \equiv \text{ops}_R\) and \(\langle \sim \rangle \equiv ( \lambda r : R. \lambda q : \mathcal{Q}. q =_\mathcal{Q} \text{surj} r )\), so that \(\text{Left}_C \mathcal{Q} \langle \sim \rangle\) corresponds to \(\sim\).

The lemma \(\text{Sim}_L C\) plays a similar role when proving results for subset and \(\text{QuotSubset}\) algebras.

Subsets

Now let us consider subset algebras.

Conjecture 6.8.2.2 In the PER model of [Pol94] the following axiom is valid.

\[
\exists \text{is}_\text{SubsetAlgHO} : \forall i : \star, \text{Sim}_T i \implies \\
R : \star, \forall \text{ops}_R : I R. \forall \text{Inv}_R : R \to \star,
\]
\[
( \exists r : R. \text{Inv}_R r ) \implies \\
\text{Bisim}_1 R \langle r_1, r_2 : R. \text{Inv}_R r_1 \land r_1 =_\mathcal{Q} r_2 \rangle \text{ops}_R \implies \\
\exists q : \star. \exists \text{ops}_Q : I Q. \text{IsSubsetAlg}_R R \text{ops}_R \text{Inv}_R Q \text{ops}_Q
\]
Again, we have proved this proposition in AC, i.e. using the axiom of choice and the existence of subset types.

There are three differences between this conjecture and \textit{exis\_SubsetAlg} (Definition 6.6.1.4).

\begin{itemize}
\item \textit{exis\_SubsetAlgHO} concerns all simply-typed interfaces, \textit{exis\_SubsetAlg} only first-order interfaces.
\item Here we have the demand
\[
\text{Bisim}_\uparrow \mathcal{R} (\lambda r_1, r_2 : \mathcal{R}. \text{Inv} r_1 \land r_1 =_\mathcal{R} r_2) \text{ opsR} \quad (i)
\]
on the operations \text{opsR}; in \textit{exis\_SubsetAlg} we have the demand
\[
\text{IsInvar}_\uparrow \mathcal{R} \text{ Inv} \text{ opsR} \quad (ii)
\]
The first question is whether (i) and (ii), despite their different appearances, are equivalent. The answer is that they are equivalent for all first-order interfaces, but not for arbitrary simply-typed interfaces. In fact, Example 6.8.2.3 shows that (i) and (ii) are not equivalent in general (using a higher-order interface). The second question is whether \textit{exis\_SubsetAlg} is correct for higher-order interfaces. The answer is no; we show this in Example 6.8.2.4.

\item In proposition \textit{exis\_SubsetAlgHO} the condition ($\exists r : \mathcal{R}. \text{Inv} r$) appears. This condition is really necessary: assuming \textit{exis\_SubsetAlgHO} without this condition leads to inconsistency (Example 6.8.2.5 below). It is peculiar that the condition is not necessary for first-order interfaces (see Theorem 6.8.1.4).
\end{itemize}

**Example 6.8.2.3** This example shows that in context QUOTSUBSET the equivalence

\[
\text{IsInvar}_\uparrow \mathcal{R} \text{ Inv} \text{ opsR}
\iff
\text{Bisim}_\uparrow \mathcal{R} (\lambda r_1, r_2 : \mathcal{R}. \text{Inv} r_1 \land r_1 =_\mathcal{R} r_2) \text{ opsR}
\]
does \textit{not} generally hold for all simply-typed \( I \). For this example we assume a datatype \( \mathcal{R} \) with exactly three elements \( a, b, \) and \( c \), and an equality function \( \text{eq} : \mathcal{R} \to \mathcal{R} \to \text{Bool} \) (so \( \text{eq} x y =_{\text{Bool}} \text{true} \iff x =_\mathcal{R} y \)). It is easy to make such a datatype. We take

\[
\begin{align*}
I & \equiv \lambda X : \ast . (X \to X) \to X \\
\text{Inv} & \equiv \lambda r : \mathcal{R}. \neg (r =_\mathcal{R} c) \\
\text{opsR} & \equiv \lambda h : \mathcal{R} \to \mathcal{R}. \text{if } (\text{eq} (h \ c) \ c) \ a \ (h \ c)
\end{align*}
\]

So \text{Inv} holds only for \( a \) and \( b \), and \text{opsR} \( h \) is \( h \ c \) if \( h \ c \) is \( a \) or \( b \), and is \( a \) if \( h \ c \) is \( c \). It is easy to see that

\[
\text{IsInvar}_\uparrow \mathcal{R} \text{ Inv} \text{ opsR}
\]
holds, since \text{opsR} \( h \) returns \( a \) or \( b \) for all \( h \). Furthermore, we have

\[
-(\text{Bisim}_\uparrow \mathcal{R} (\lambda r_1, r_2 : \mathcal{R}. \text{Inv} r_1 \land r_1 =_\mathcal{R} r_2) \text{ opsR}), \quad (i)
\]
so we have no equivalence. In order to show (i), assume

\[
\text{Bisim}_\uparrow \mathcal{R} (\lambda r_1, r_2 : \mathcal{R}. \text{Inv} r_1 \land r_1 =_\mathcal{R} r_2) \text{ opsR},
\]
6.8. AXIOMATIZING QUOTIENTS AND SUBSETS

\[ \forall f,g: R \rightarrow R. \ Sim_{\lambda x: * \rightarrow \ast} R R (\lambda r1, r2: R. \ Inv \ r1 \land r1 \equiv \ r2) f g \implies \]
\[ Inv (\text{opsR} f) \land \text{opsR} f =\_ \text{opsR} g \]  \hspace{1cm} \text{(ii)}

Now take \( f \equiv \lambda x: R. \text{if} (\text{eq} \ x \ c) \ a \ x \) and \( g \equiv \lambda x: R. \text{if} (\text{eq} \ x \ c) \ b \ x \). These \( f \) and \( g \) satisfy the condition in (ii) (\( f \) and \( g \) respect \( \text{Inv} \) and behave the same on \( R \) restricted to \( \text{Inv} \)). So with (ii) have

\[ \text{Inv} (\text{opsR} f) \land \text{opsR} f =\_ \text{opsR} g . \]

However, \( \text{opsR} f =\_ \text{f} \ c =\_ \text{a} \neq \text{b} \text{g} \ c =\_ \text{a} \text{opsR} g \), hence we have a contradiction with our assumption, so we have (i), and we are done. \( \Box \)

Example 6.8.2.4 This example shows that \textit{exis\_SubsetAlg} (Definition 6.6.1.4) generalized to arbitrary simply-typed \( I \) leads to inconsistency. We show this by assuming \textit{exis\_SubsetAlg} generalized to all simply-typed \( I \), and take \( I, R, \text{opsR} \) and \( \text{Inv} \) as in Example 6.8.2.3. Since \( \text{opsR} \) respects \( \text{Inv} \), the assumption gives us \( S: * \) and \( \text{opsS}: I S \) and \( \text{inj}: S \rightarrow R \) with

\[ \text{IsInjection} \ \text{inj} \land \]
\[ \text{Inv} \iff \text{Image} \ \text{inj} \land \]
\[ \Sim (r) R S (\lambda x: R. \lambda s: S. r =\_ s) \text{opsR opsS} \]

it requires a bit of work to infer from these clauses that

\[ \text{Bisim}_R (\lambda r, r': R. \text{Inv} r \land r =\_ r') \text{opsR} . \]

However, this is in contradiction with (i) of Example 6.8.2.3. So \textit{exis\_SubsetAlg} cannot be generalized to arbitrary simply-typed interfaces. \( \Box \)

Example 6.8.2.5 In this example we show that the demand \((\exists r: R. \text{Inv} r)\) is really necessary in proposition \textit{exis\_SubsetAlgHO}. So we assume

\[ \forall I: * \rightarrow * . \ \text{SimplyT} I \implies \]
\[ \forall R: * \rightarrow * , \text{opsR}: I R. \forall \text{Inv}: R \rightarrow *' , \]
\[ \text{Bisim}_R (\lambda r1, r2: R. \text{Inv} r1 \land r1 =\_ r2) \text{opsR} \implies \]
\[ \exists S: *' , \exists \text{opsS}: I S. \text{InSubSetAlg}_R \text{opsR Inv S opsS} , \]

and derive a contradiction.

We need an empty datatype \( E \) (e.g. the type \((\Pi X: * . X)\) is empty), and a datatype \( R \) with an element \( r: R \) (e.g. \text{true}:\text{Bool}). Take

\[ I \equiv \lambda X: * . (X \rightarrow E) \rightarrow E \]
\[ \text{opsR} \equiv \lambda f: R \rightarrow E. f \ r \]
\[ \text{Inv} \equiv \lambda r: R. \text{False} \]

It is easy to verify that

\[ \text{Bisim}_R (\lambda r1, r2: R. \text{Inv} r1 \land r1 =\_ r2) \text{opsR} \]

holds, so with (i) we have \( S: * \) and \( \text{opsS}: I S \), i.e. \( \text{opsS}(S \rightarrow E) \rightarrow E \). Because of our definition of \( \text{Inv} \), the subset \( S \) is also empty. Since \( S \) and \( E \) are both empty, there is an "empty" function \( f \) of type \( S \rightarrow E \) (to show this formally requires the axiom of choice). Applying \( \text{opsS} \) to \( f \) yields an element of \( E \). This is in contradiction with the fact that \( E \) is empty. So (i) leads to a contradiction, hence the demand that \( \text{Inv} \) is not empty is really necessary in proposition \textit{exis\_SubsetAlgHO} above. \( \Box \)
CHAPTER 6. \( \lambda \nu_T^+ \) AND MODELLING ABSTRACT DATATYPES

The second example shows \texttt{exis\_SubsetAlg} (the "old" formulation) is no good for higher-order interfaces. The third example indicates that \texttt{exis\_SubsetAlgHO} (the "new" formulation) cannot be strengthened any further. Therefore we conjecture that \texttt{exis\_SubsetAlgHO} is the proper formulation of the existence of subset algebras, which is valid in PER models.

Quotients of subsets

Now let us consider \texttt{QuotSubset} algebras, i.e. quotients of subsets.

Conjecture 6.8.2.6 In the PER model of [Pol94] the following axiom is valid.

\[ \texttt{exis\_QuotSubsetAlgHO} : \forall V : * \rightarrow *_p. \text{SimplyTI} \implies \]
\[ \forall R : *_p. \forall \text{opsR}: I \rightarrow R. \forall (\simeq) : R \rightarrow *_p. \forall \text{Inv} : R \rightarrow *_p. \]
\[ \text{IsERo} \_R (\simeq) \text{Inv} \implies \]
\[ \exists r : R. \text{Inv} r \implies \]
\[ \text{Bisim}_1 R (\text{Restr}_R (\simeq) \text{Inv}) \text{opsR} \implies \]
\[ \exists Q : *_p. \exists \text{opsQ}: I \rightarrow Q. \text{IsQuotSubsetAlg}_R R \text{opsR} (\simeq) \text{Inv} Q \text{opsQ} \]

Yet again, this axiom can be proved in \( AC \), i.e. using the axiom of choice and the existence of quotient and subset types.

There are three differences between this proposition and lemma \texttt{exis\_QuotSubsetAlg} (below Definition 6.6.1.4).

- \texttt{exis\_QuotSubsetAlgHO} concerns all simply-typed interfaces, \texttt{exis\_QuotSubsetAlg} only first-order interfaces.
- In the general formulation we have the condition that \texttt{Inv} may not be empty. This is only to be expected, since we also have this condition for subset algebras.
- In \texttt{exis\_QuotSubsetAlg} we have two conditions on \texttt{opsR}:

\[ \text{IsInv} \_R \text{Inv} \text{opsR} \wedge \]
\[ \text{Bisim}_1 R (\simeq) \text{opsR} \quad \text{(i)} \]

In \texttt{exis\_QuotSubsetAlgHO} we have the following condition instead:

\[ \text{Bisim}_1 R (\text{Restr}_R (\simeq) \text{Inv}) \text{opsR} \quad \text{(ii)} \]

(Recall that \texttt{Restr}_R (\simeq) \text{Inv} is defined as \( \lambda r_1, r_2 : R. \text{Inv} r_1 \wedge r_1 \simeq r_2 \wedge \text{Inv} r_2 \).) Without giving further evidence, we say that (i) and (ii) are generally not equivalent (but (i) implies (ii) for first-order interfaces). This is not so surprising: in the light of our study of subset algebras, we might have expected the following two conditions:

\[ \text{Bisim}_1 R (\lambda r_1, r_2 : R. \text{Inv} r_1 \wedge r_1 \simeq r_2) \text{opsR} \wedge \quad \text{(i')} \]
\[ \text{Bisim}_1 R (\simeq) \text{opsR} \]

Also (i') and (ii) are generally not equivalent. We state here that both (i) and (i') are wrong conditions for the existence of \texttt{QuotSubset} algebras in general.
6.8. AXIOMATIZING QUOTIENTS AND SUBSETS

An interesting point of the formulation in Conjecture 6.8.2.6 is that the relation

\[ \text{Rest} \approx (\approx) \text{Inv}, \]

which plays a central role there, is a partial equivalence relation (PER). This provides supporting evidence that in order to reason about quotients and subsets PER models are very suitable. Furthermore, the formulation suggests a slightly more elegant formulation in terms of PERs (see also Remark 6.5.3.1).

Proof rules for higher-order ADTs

Let us assume Conjecture 6.8.2.6. Since we could derive proof rules for first-order ADTs (i.e. ADTs with a first-order interface) from the existence of first-order \textit{QuotSubset} algebras, an obvious question is what proof rules can be derived from Conjecture 6.8.2.6, for arbitrary simply-typed algebras (including higher-order ones). The answer is that we can use the same proof rules, with the following two modifications, which correspond with the conditions in \textit{exs. QuotSubsetHO}. First, we have an additional proof obligation that \textit{Inv} may not be empty. Second, in Section 6.6 we have the conditions

\[ \text{IsInv}_{\mathcal{R}} \text{ R Inv op} \in \mathcal{R}, \]
\[ \text{Bisim}_{\mathcal{R}} (\approx) \text{ op} \in \mathcal{R}. \]

These should be replaced by

\[ \text{Bisim}_{\mathcal{R}} (\text{Rest} \approx (\approx) \text{Inv}) \text{ op} \in \mathcal{R}. \]

There is another point we have to attend to. A naive specification for such a higher-order ADT typically contains higher-order quantifications (e.g. "\text{\forall f:Rep \rightarrow Rep. ...}") and higher-order equalities (e.g. "f =_{\text{rep}} g"). The user of the ADT has to show that the abstract specification is well-behaved. So we have to show how these higher-order quantifications and equalities should appear in the abstract specification, such that it is well-behaved. We have found the following results (compare to Section 6.6.3).

- Equalities "=_{\text{rep}}" in the naive specification should be replaced by

  "Sim_{\mathcal{J}} \text{ Rep Rep (Rest}_{\text{rep}} (\approx) \text{Inv})"

in the abstract specification; this will result in a well-behaved abstract specification. (Here \text{\mathcal{J}} indicates which higher-order equality we have, e.g. \text{\mathcal{J}} \equiv \lambda x : * \mapsto, x \rightarrow x \text{ for } =_{\text{rep}} =_{\text{rep}}.)

- Quantifications "\text{\forall f:J Rep.}" in the naive specification should be replaced by

  "\text{\forall f:J Rep. Bisim}_{\mathcal{J}} \text{ Rep (Rest}_{\text{rep}} (\approx) \text{Inv}) f \rightarrow \rightarrow}"

in the abstract specification; this will result in a well-behaved abstract specification. A similar replacement should be made for higher-order existential quantifications.

In order to prove these results, lemma \textit{Sim LeftC} plays an important role again.
6.8.3 Summary of Axiomatizing

Let us summarize our results concerning axiomatizing quotient and subset algebras.

We have used QUOTOSUBSET (Definition 6.6.1.4) in the bulk of this chapter (up to Section 6.7 inclusive). The axioms in QUOTOSUBSET state the existence of quotient and subset algebras, and can be justified semantically by a PER model, or syntactically by considering a type theory with additional syntax for quotients and subsets (e.g. [Bar95]). However, these axioms concern only first-order interfaces. Consequently, we can only consider ADTs with first-order interfaces.

We have defined $AC$ (Definition 6.8.1.3) containing axioms for quotient and subset types, and the axiom of choice. These axioms are simpler, and lead to quotient and subset algebras for arbitrary simply-typed interfaces, including higher-order ones. These algebras yield proof rules for ADTs with simply-typed interfaces. However, $AC$ cannot be justified by well-known semantic models, or by existing type systems with quotients and subsets. Furthermore, the theory concerning higher-order interfaces is strewn with pitfalls, e.g. concerning subsets which may not be empty.

Therefore we have chosen the axiomatization in QUOTOSUBSET with only first-order interfaces for the bulk of this chapter, and leave a well-justified theory for higher-order interfaces as future work.
6.9 Conclusions

6.9.1 Related Work

One of the first articles that considered correctness of abstract datatypes was [Hoa72]. Its approach is the following. A specification is stated in terms of an abstraction function \( \text{abs} \), that relates a representation type to some fixed abstract domain. For each operation of the ADT, there is one clause that specifies what effect the operations has on the abstract domain, using the abstraction function. A correctness proof of an implementation consists of

- an abstraction function, that maps elements of the concrete representation type to elements of the abstract domain,
- a proof that each operation satisfies the corresponding clause in the specification, and
- an invariant.

For the example of bags, the abstract domain would be the functions of type \( \text{Nat} \to \text{Nat} \). The (naive) specification in our framework would then be

\[
\begin{align*}
\lambda \text{Rep} &; +, \lambda \text{ops}: \text{BagI Rep}. \\
\exists \text{abs}: \text{Rep} &\to (\text{Nat} \to \text{Nat}). \\
\text{abs ops empty} &\equiv \text{Nat} \to \text{Nat} \{ \} \land \\
(\forall m: \text{Nat}. \forall r: \text{Rep}. \text{abs (ops add m r)} &\equiv \text{Nat} \to \text{Nat} \{ \text{abs r} \cup \{m\} \} \land \\
(\forall m: \text{Nat}. \forall r: \text{Rep}. \text{ops card m r} &\equiv \text{Nat} \#(\text{abs r}, m))
\end{align*}
\]

where

\[
\begin{align*}
\{ \} &\equiv \lambda n: \text{Nat}. 0 \\
b \cup \{m\} &\equiv \lambda n: \text{Nat}. \text{if eq m n} (S (b n)) (b n) \\
\#(b, m) &\equiv b m
\end{align*}
\]

We do not need Leibniz’ equality on \( \text{Rep} \) in such specifications, since we always reason about abstract values in this approach. Therefore we do not need the quotients as in Section 6.5.1. However, we do need the subsets of Section 6.5.2 to allow invariants.

Unfortunately, we cannot fit this specification easily in our framework, since it contains a quantification over a function type \( \exists \text{abs}: \text{Rep} \to (\text{Nat} \to \text{Nat}). \ldots \), so the specification is not easily shown to be well-behaved. In Section 6.8.2 we have shown that with more powerful axioms we can deal with such specifications. But these powerful axioms are not really necessary; with a few additional lemmas we can also handle this specific form of specifications.

The approach of [Hoa72] has always remained important in the folklore for proving ADTs correct. We were led by another approach in the folklore [GHM78], where specifications are more direct, and usually consist of a set of equalities in terms of the operations. We briefly discuss the extensive literature on the algebraic theory which was inspired by this kind of specifications below, and first relate to [GHM78].

In this approach, the implementor of the ADT may use an invariant and an equivalence relation instead of equality. So from a pragmatic point of view, our results are not surprising. But we have given a formal justification of this common practice, using parametricity, quotients and subsets. This formal justification yields two new insights. First, we have found that not all specifications of \( \lambda \omega^+_L \) are sensible for ADTs (Section 6.6.2). Although all specifications
written in first-order predicate logic (which is the logic used in the folklore) are sensible, in the more powerful logic $\lambda^+_\omega$ it has to be proved that the specification is well-behaved. Second, we have obtained insight in proving correctness of ADTs with arbitrary higher-order interfaces (where the operations of the ADT may have functions as arguments); in the folklore only first-order interfaces are considered (no functions as argument to the operations). Section 6.8.2 shows this generalization is not straightforward. We should note that the proof rules indicated in Section 6.8.2 rely on some strong axioms. We leave a semantic justification of these axioms and an elaboration of an ADT with a higher-order interface as future work.

There is extensive literature on the algebraic theory of ADTs. In this algebraic approach specifications consist solely of equalities, e.g. our specification of stacks (Section 6.3) and the first specification of bags (Section 6.5.1) are algebraic. The theory proceeds by considering the class of all algebras that satisfy the equalities. This class determines one algebra — the initial algebra — which is minimal in a certain sense, i.e. the initial algebra is the smallest algebra that equates all terms as indicated by the equalities, but nothing more.

The notion of (correct) implementation varies in the several algebraic accounts of ADTs (for an overview, see [NOS89]). We will consider just the approach taken in [GTW78]; other approaches are even less related to our approach. In [GTW78] an algebra $A$ with equivalence relation $\simeq$ is a correct implementation of a specification if a subalgebra of $A/\simeq$ is isomorphic to the initial algebra. This is roughly similar to our result, in the sense that we also use quotients and subsets. An important difference is that our result for correctness of an implementation is much more direct, since we do not use initial algebras. Indeed, initial algebras for our kind of specifications do not necessarily exist, since they are written in a rich logic, and are not restricted to equalities. This constitutes another important difference. The third, and most important difference is the following. In our type-theoretical framework bindings are explicit. This makes it possible to formally model encapsulation, i.e. data hiding. With the theory of parametricity this modelling yields an interesting notion of equality on implementations (using the principle of simulation). From this interesting notion of equality, and the existence of quotients and subsets we can derive proof rules for correct implementations. The algebraic framework, however, works with an (implicit) set-theoretic framework, in which encapsulation cannot be formally modelled. Hence the notion of correct implementation is defined using quotients and subsets, and this definition is only justified by appealing to the intuition. So we have proof rules derived from encapsulation, and in the algebraic theory proof rules are defined without formal justification.

The only literature (that we are aware of) on correctness of ADTs in a type-theoretic framework is [Und94], [Luo94] and [Han99]. [Und94] treats the example of bags, just as we, but her bags are more general: bags parametrized over the datatype of the elements, and over equality on that datatype. Furthermore, the bags have a recursion operation. However her specification does not include clauses like the “commutativity” of the add operation, and hence, quotients are not needed. Similarly, she does not consider implementations which need subsets for the correctness proof. From a methodological point of view, the most important difference is that in [Und94] just one example is given, whereas we have given general proof rules for ADTs, which resulted, for example, in the notion of well-behaved specifications.

In chapter 8 of [Luo94] the example of stacks is treated, with as implementation pairs (array,pointer) using the natural equivalence relation (two values are equivalent if their pointers are equal and the arrays are equal up to the index indicated by the pointer). Since Luo uses strong $\Sigma$-types (see Section 6.1.2), which do not provide encapsulation, he is forced to
express the specification as an existential quantification over the equivalence relation used. This is essentially the same as our rejected solution "Using a weaker specification" (step 4c in Section 6.5.1), and has the same disadvantages.

In [Han99] a formal connection is made between algebraic specification of ADTs and the type-theoretic approach. [Han99] essentially uses our logic QUOTSUBSET, including parametricity and axioms for quotient and subset types.

6.9.2 Future Work

Syntactic sugar

Although the notation for ADTs in $\lambda \omega^+_t$ is very precise, it rather emphasizes the technical aspects (e.g. variable bindings), and it is not very readable for the ordinary programmer. Some syntactic sugar improves the readability, by making some variable bindings and some term constructions implicit. Figure 6.7 shows the first example of bags (Section 6.5.1) as sugared input to (an extended version of) Yarrow where keywords are underlined. We go briefly through the example and explain some details of this notation, which can be translated to pure $\lambda \omega^+_t$ without difficulties.

In the definition of the interface (step 1), the abstract construct implicitly abstracts over $Bag:*_t$, so this construct corresponds with the definition of the interface $BagI:*_s \rightarrow *_s$.

In the specification (step 2) we implicitly abstract both over the representation type $Bag:*_t$ and the operations $ops$, and allow $ops::l$ to be abbreviated to just $l$ (similar to the with construct found in the language Pascal [JW85]). A salient point of this notation for the specification is that the programmer does not have to mention the invariant or the equivalence relation; by syntactic manipulation, we can automatically change the specification, so that it is abstracted over the invariant and the equivalence, and so that they are used in the proper places. If the specification satisfies some syntactic criteria, we can then automatically prove it is well-behaved (see Section 6.6.3). So the specification construct introduces two definitions in the context, namely that of the abstract specification, and the automatically generated proof that it is well-behaved.

The implementation construct introduces three definitions, namely for the representation type, the operations, and the packed combination.

The prove correctness construct introduces the proof of UserSpec $bagImpl$ in the context. We have slightly simplified the third proof obligation here, using the fact that the invariant is trivial.

In step 5 we have an ordinary definition (def), but the unpack construct is a sugared version of the basic unpack, so that we can refer directly to the operations (just as in the specification). In the correctness proof of $prog1$ we use a sugared tactic use correctness.

A final remark about the syntactic sugar used in Figure 6.7. It is conceivable to allow a shorter notation, in which more parts of the development are combined. We refrained from this in order to retain the advantages of the $\lambda \omega^+_t$ encoding of ADTs, namely that multiple implementations and specifications for one interface are allowed.

So, the standard notation in $\lambda \omega^+_t$ is sometimes awkward, because we are limited to a small number of primitives. Only extending $\lambda \omega^+_t$ with syntactic sugar makes it usable as a practical programming logic; the translation to the underlying, very explicit, primitives can then be considered as a formal semantics for the more practical constructions.
1. abstract Bag is
   empty : Bag,
   add : Nat → Bag → Bag,
   card : Nat → Bag → Nat
end abstract

2. specification SpecBag for Bag is
   ∀m:Nat. card m empty =_Nat 0 ∧
   ∀m:Nat. ∀b:Bag. card m (add m b) =_Nat S (card m b) ∧
   ∀m,n:Nat. ∀b:Bag. ¬(m =_Nat n) → card m (add n b) =_Nat card m b ∧
   ∀m,n:Nat. ∀b:Bag. add m (add n b) =_Bag add n (add m b)
end specification

3. implementation bagImpl for Bag as List Nat is
   empty = nil Nat,
   add = (λn. ),
   card = count
end implementation

4. prove bagImpl.correct : correctness of bagImpl for specification SpecBag
   where equality is PermNat
   and invariant is λx:List Nat. True
   "tactics for proving the following proof obligations"
   • IsREflNat PermNat (λx:List Nat. True)
   • PermNat (nil Nat) (nil Nat) ∧
     ∀m:Nat. ∀r,r':List Nat. PermNat r r' → PermNat (m;r) (m;r') ∧
   • ∀m:Nat. ∀r':List Nat. PermNat r r' → count m r =_Nat count m r' ∧
    ∀m,n:Nat. ∀r:List Nat. ¬(m =_Nat n) → count m (n;r) =_Nat count m r ∧
   • ∀m,n:Nat. ∀r:List Nat. PermNat (m;n;r) (n;m;r)
   exit

5. def prog1 := unpack bagImpl in card 5 (add 5 empty)

   prove prog1_correct : prog1 =_Nat 1
   unfold prog1
   use correctness bagImpl_correct
   "tactics for proving that under the assumptions
   Bag:∗,
   empty:Bag
   add:Nat → Bag → Bag
   card:Nat → Bag → Nat
   ∀m:Nat. count m empty =_Nat 0
   ∀m:Nat. ∀b:Bag. card m (add m b) =_Nat S (card m b)
   ∀m,n:Nat. ∀b:Bag. ¬(m =_Nat n) → card m (add n b) =_Nat card m b
   ∀m,n:Nat. ∀b:Bag. add m (add n b) =_Bag add n (add m b)
   we have
   card 5 (add 5 empty) =_Nat 1"
   exit

Figure 6.7: Development and use of bags using syntactic sugar.
6.9. CONCLUSIONS

Generalization of our results

Although our approach is a generalization of the principles found in the folklore, it is not as general as the type system of $\lambda \omega F^+$ admits. First, we could consider so-called multi-sorted ADTs, i.e. where several abstract datatypes are introduced at once. We expect that these can be integrated straightforwardly into our theory. Second, we have the restriction to ADTs with first-order interfaces. In Section 6.8.2 we studied higher-order interfaces, but our formalization requires additional axioms which lack semantic justification. Third, we do not admit polymorphic operations. We exclude those because we wanted to keep the presentation as simple as possible, and to facilitate formalization in Yarrow. But the theory of parametricity (including simulation) given in the literature [PA93] is general enough to cope with polymorphic operations, so it remains to discover how to formalize the general theory in $\lambda \omega F^+$ to allow polymorphic operations. Fourth, we do not admit abstract datatpe-constructors such as bags over arbitrary elements. The reason for this is deeper: we would have to extend the theory of parametricity from $\lambda 2$ to $\lambda \omega$.

6.9.3 Formalizing ADTs in $\lambda \omega F^+$ and Yarrow

The nice thing about $\lambda 2$ (and hence $\lambda \omega$) from the viewpoint of programming languages is that it captures data abstraction (ADTs) in an elegant way [MP84]. Therefore a logic for $\lambda 2$ is very suitable for reasoning about ADTs. The line of research was started by [Rey83], and carried on by others [Wad89, PA93], introducing parametricity. PER models give the semantic justification of the theory of parametricity. We have shown that parametricity is not enough for practical proof rules for ADTs (we also need the existence of quotients and subsets), but PER models are enough (since they contain quotients and subsets). By including parametricity and axioms for the existence of quotients and subsets in our logic, we can derive the usual proof rules for ADTs. So the rules for proving correctness of ADTs are formally derived from the definition of ADTs, using a few additional axioms.

When we first considered correctness of ADTs, we still thought parametricity would be enough to obtain the usual proof rules. When we started to perform the formalization in Yarrow we were forced to distinguish between equality on concrete representation and abstract representation. Only then we realized quotients and subsets were needed to complete the proofs. This is one of the biggest advantages of using a proof assistant: one is forced to consider every detail of the formalization, so one cannot do steps which are "intuitively clear" but not derivable in the theory. Sometimes these steps are not valid at all, and sometimes they are valid but need an additional axiom. Adding axioms may be dangerous, but at least it is clear which assumptions are made. This advantage is particularly strong if the proofs concerned are long and complicated.

A disadvantage of $\lambda \omega F^+$ is that no proper induction over the structure of interfaces is possible within the logic (see the "definition of Sim in Section 6.7.2); it is possible to work around this shortcoming by postulating additional axioms, but that is not very elegant. Furthermore, some properties are awkward to express within $\lambda \omega F^+$, e.g. those of $WB$ in Section 6.6.3.

This brings us to the axioms we introduced. The proper way to justify them is to give a model in which these axioms are valid. However, this is beyond the scope of this thesis. Therefore we confine ourselves to an informal justification, often referring to PER models in the literature. We treat the axioms in four groups. First, the basic axioms about records and existential types in Section 6.2 are easily justified by a PER model [Pol94]. Second, the parametricity axiom can also be justified by such a PER model [BFSS90], at least for the
programming language \( \lambda 2 \), so we assume it is also valid for \( \lambda \omega \) (see Section 6.4.4). Third, we introduced axioms stating the existence of quotient and subset algebras, which are easy to justify in PER models (Section 6.6), or using type systems with quotients [Bar95, Hof95]. Fourth, we speculated in Section 6.8 about an alternative way to obtain quotient and subset algebras, using the axiom of choice. This axiom cannot be justified by the usual PER models, although it is a very common mathematical principle.

A formal justification of the axioms we use is the weakest point in this chapter, but such a justification — a semantic matter — is beyond the scope of this thesis, which focuses on syntactical and pragmatic aspects.
Chapter 7

PTSs with Subtyping

This chapter extends Pure Type Systems with a certain form of subtyping, suitable for modelling OOP. Just as the PTSs form a family of typed λ-calculi, the PTSs with subtyping (PTS≤s) form a family of typed λ-calculi with subtyping. We present the formal definition of these PTS≤s, prove they possess a number of important theoretical properties, and are practically implementable on a computer.

The rest of this introduction is divided into two parts. First, we sketch what form of subtyping, and constructs related to subtyping, are used in the existential model, which we use for encoding objects (elaborated in Chapter 9).

Although we define general PTS≤s, we are mainly interested in one particular PTS≤, namely \(\lambda_\omega^≤\) with subtyping \((\lambda_\omega^≤)^≤\). We give a definition of the framework of PTS≤s instead of an ad-hoc definition of \(\lambda_\omega^≤\), because the former is much shorter. In the second part of this introduction we explain why our definition of the framework is such that we can develop a general meta-theory, instead of just considering the instantiation \(\lambda_\omega^≤\).

Subtyping in the existential model

The existential model of [PT94] is based on \(F^≤\), the higher-order \(\lambda\)-calculus with subtyping. In this model, the types of the methods of a class are characterized by a datatype-constructor of type \(* \rightarrow *\), which is called the interface of the class (similar to the interfaces of ADTs as in Chapter 6). Objects belonging to a class C with interface CI, have as type Object CI, where Object is a particular type-constructor of type \((* \rightarrow *) \rightarrow *\). As example we consider a class of vehicles, with interface VehicleI, and a class of cars, which is a subclass of vehicles (and hence has more methods), with interface CarI. So objects belonging to the class of cars have type Object CarI, e.g. beetle : Object CarI. Here, we are not interested in the actual definitions of Object, VehicleI, CarI and beetle according to the existential model, but concentrate on typing.

Suppose we have a function getSpeed : Object VehicleI \rightarrow \text{Real}, which uses some methods present in the object given as argument to calculate the speed. By the type of getSpeed, we may apply this function to any vehicle object. By the idea of subtyping in OOP, getSpeed should also be applicable to car objects, since the class of cars is a subclass of the vehicles, and hence all methods for vehicles are also available for cars. In the existential model we may apply getSpeed to car objects because of the following. First, we have a subtyping relation \(\leq\) in \(F^≤\), and by the rules for \(\leq\) and the definition of Object, CarI and
VehicleI, we have that Object CarI is a subtype of Object VehicleI:

\[ \text{Object CarI} \leq \text{Object VehicleI} \]

This statement can be interpreted as "the cars form a subset of the vehicles". Second, \( F_{\leq} \) has the following so-called subsumption rule, which captures the idea of this interpretation:

\[
\text{(subsum)} \quad \frac{a : A \quad A \leq B}{a : B}
\]

If \( a \) is an element of \( A \), and \( A \) is a subset of \( B \), then \( a \) is also an element of \( B \). Using this rule and the subtyping statement, we can give beetle type Object VehicleI, so that we may apply method getSpeed to beetle. However, as we will see, subtyping and the subsumption rule are not sufficient.

Consider the function setSpeed which updates the speed of a vehicle. First, we consider it as having the following type.

\[
\text{setSpeed} : \text{Object VehicleI} \rightarrow \text{Real} \rightarrow \text{Object VehicleI}
\]

Note that we work in a functional framework, so setSpeed cannot update the state of an object, but must deliver a new object with an updated state. In an imperative setting, the same situation occurs for functions that copy or clone an object.

By virtue of subtyping between Object CarI and Object VehicleI, and the subsumption rule, we may apply setSpeed to beetle, so

\[
\text{setSpeed beetle 120.0} : \text{Object Vehicle}
\]

Now we have lost information, namely that our fast-moving beetle is still a car; we want to have that setSpeed beetle 120.0 has type Object CarI. One attempt to achieve is to give setSpeed a polymorphic type:

\[
\text{setSpeed} : \Pi \Pi : * \rightarrow *, \text{Object I} \rightarrow \text{Real} \rightarrow \text{Object I}
\]

This is better than the previous typing, since we now have

\[
\text{setSpeed CarI beetle 120.0} : \text{Object CarI}
\]

so we have not lost the information that the fast moving beetle is still a car. However, the type for setSpeed is too general, since we may apply setSpeed to the interface of any class (and even to any type-constructor of type \( * \rightarrow * \)); it is impossible to define a sensible setSpeed in such a way that it has this type. The polymorphism in setSpeed needs to be restricted, so that it can only be applied to interfaces I for classes which are a subclass of vehicles.

This problem is solved in the existential model by the combination of two mechanisms present in \( F_{\leq} \). The first mechanism is called lifted subtyping, and lifts the subtyping relation point-wise from datatypes to datatype-constructors of any kind, i.e. for datatype-constructors \( A \) and \( B \) we have \( A \leq B \iff \) for all types \( X \) we have \( A \times X \leq B \times X \). The relevance of lifted subtyping for OOP is that the interface for cars is a subtype of the interface for vehicles, so

\[
\text{CarI} \leq \text{VehicleI}
\]
The second mechanism, called *bounded quantification*, allows the polymorphism over I of type * → *, to be restricted to those Is which are subtypes of VehicleI. That is, we can define setSpeed so that it has as type a bounded quantification, notated as follows.

\[
\text{setSpeed} : \Pi I \leq \text{VehicleI} : * \rightarrow *, \text{Object I} \rightarrow \text{Real} \rightarrow \text{Object I}
\]

Still, setSpeed may be applied to CarI, since CarI \leq VehicleI, but setSpeed may not be applied to interfaces which are not a subtype of VehicleI. Since I is a subtype of VehicleI, we know objects of type Object I have at least all the methods available for vehicles, so we can make a sensible definition of setSpeed. The name “bounded quantification” indicates that the type-variable (I in this case) is bounded by some datatype-constructor (VehicleI) in the quantification (i.e. \(\Pi\)-type).

So the existential model uses the following features of \(F^\omega\_\leq\):

- a subtyping judgment \(\Gamma \vdash A \leq B\), induced by subtyping rules, and the subsumption rule,
- lifted subtyping, and
- bounded quantifications.

(These are all features in \(F^\omega\_\leq\) related to subtyping.) So our framework of \(PTS\_\leq s\) will have these three features to allow the use of the existential model, which will be worked out in Chapter 9.

**Meta-theory of \(PTS\_\leq s\)**

Now we consider how we should define the \(PTS\_\leq\) framework. As a first (and abandoned) attempt, we did so by simply combining the rules of \(PTS\_s\) with the rules of \(F^\omega\_\leq\). However, it is very hard to show this system behaves well, i.e. that a number of meta-theoretical properties hold. The main problem is that the subtyping rules depend on the typing rules and vice versa. This mutual dependence is awkward for the meta-theory, since results about the subtyping judgment cannot be proved independently of results about the typing judgments: soon one gets circular dependencies of lemmas about subtyping and lemmas about typing. Usually, these circles are broken by considering the different layers of the system. In \(F^\omega\_\leq\), there are three layers, viz. kinds, datatypes and programs. First typing judgments (\(\Gamma \vdash a : A\)) concerning kinds and datatypes are considered, which are independent of the subtyping judgments (\(\Gamma \vdash B \leq C\)) and the typing judgment for terms. Then subtyping judgments (for datatypes) are considered, which do depend on typing of datatypes, but not on typing of programs. Finally, results about typing programs are proved. So in \(F^\omega\_\leq\) the ramification into three layers breaks the circle of dependencies.

Since \(\lambda\omega^F_{\leq}\) is also a layered system, the first attempt proceeded by restricting the meta-theory to that for \(\lambda\omega^F_{\leq}\), using the same proof strategy as for \(F^\omega\_\leq\). But since \(\lambda\omega^F_{\leq}\) has 6 layers (apart from kinds, datatypes and programs also propkinds, propositions and proofs), the meta-theory became very lengthy. We ended up proving similar lemmas for all of the 6 layers. Apart from that, the formulation of the \(PTS\_\leq s\), which is a general scheme where all layers are treated simultaneously, was incompatible with our proof strategy that considers the theory layer by layer.
This led us to consider a reformulation of the definition of the \(PTS\)'s, where we conform to the following major design decision:

\[
\text{The subtyping rules do not depend on the typing rules.}
\]

In other words, we define the subtyping relation on pseudoterms rather than only on well-typed terms. Now we do not need the layers anymore: we develop the theory for subtyping first, and then proceed to the typing judgment. Thus our meta-theory is valid for general \(PTS\)'s. We will not claim the theory is short now, but we think it is acceptable considering the generality.

In Section 7.1 we define the syntax of \(PTS\)'s, and give the typing and subtyping rules. We also relate to subtyping systems in the literature. The section closes with a short summary of the meta-theoretical results, in particular for those readers who are not interested in the scores of lemmas and proofs necessary to achieve those results. Section 7.2 gives those lemmas and proofs, that culminate in the properties of Subject Reduction and Minimal Typing. An important feature is a reformulation of the subtyping rules, so that they have better theoretical properties, although they are less natural and less readable than those of Section 7.1. Section 7.3 defines a type-checking algorithm for a class of \(PTS\)'s, including \(\lambda\omega_{\text{F}}^{\leq}\). We use this algorithm in Yarrow. We prove soundness, completeness and termination of the algorithm modulo certain provisos; we conjecture these provisos to hold for \(\lambda\omega_{\text{F}}^{\leq}\). This implies decidability of typing for \(\lambda\omega_{\text{F}}^{\leq}\). Section 7.4 gives the conclusions.
7.1 Syntax and Typing Rules

We specify the bare syntax of $PTS \leq$ s in Section 7.1.1, and the typing and subtyping rules in Section 7.1.2. Section 7.1.3 shows how many existing systems with subtyping can be considered as a $PTS \leq$, starting with very simple systems and gradually moving up to more advanced ones. These examples may help the reader in understanding some of the rules, since the general scheme is per definition not very concrete. We show in Section 7.1.4 a number of alternatives and extensions for our rules, and why we rejected those. Finally, Section 7.1.5 gives a short summary of the meta-theoretical properties which hold for $PTS \leq$ s.

7.1.1 Syntax

The following three constructs are new in $PTS \leq$ s:

- bounded abstractions, of the form $\lambda x \leq a : A. \, b$,
- bounded quantifications, of the form $\Pi x \leq a : A. \, B$, and
- bounded declarations, of the form $\Gamma, x \leq a : A$.

We use this terminology in the explanation of specifications of $PTS \leq$ s.

Definition 7.1.1.1 A specification of a $PTS \leq$ is a 5-tuple $(S, A, R, S \leq, R \leq)$, with the following properties.

1. $S$ is a set of symbols called the sorts.
2. $A \subseteq S \times S$, a set of axioms of the form $(s : s')$.
3. $R \subseteq S \times S \times S$, a set of rules of the form $(s_1, s_2, s_3)$.
4. $S \leq \subseteq S$ is a set of subtyping sorts.
5. $R \leq \subseteq S \times S \times S$, a set of bounded rules.

We write $(s_1, s_2)$ for a (bounded) rule, as abbreviation for $(s_1, s_2, s_3)$. All $PTS \leq$ s mentioned in this thesis will only have rules of the form $(s_1, s_2)$. The first three elements of the tuple serve exactly the same purpose as in $PTS$s (see Section 2.1).

The subset of sorts $S \leq$ controls on which levels we can introduce subtyping. To be more precise, we can make a bounded declaration $x \leq a : A$, which declares variable $x$ as a subtype of $a$, if $a : A$ and $A : s$ and $s \in S \leq$. Intuitively, in the system $\lambda \omega \leq$ where $S \leq = \{\square\}$, we admit

\[
\begin{align*}
\text{Nat} \leq \text{Int} : * & \quad \text{since } * : \square \text{ and } \square \in S \leq \\
\text{CarI} \leq \text{VehicleI} : * \rightarrow * & \quad \text{since } * \rightarrow * : \square \text{ and } \square \in S \leq \\
\text{but not } x \leq \text{true : Bool} & \quad \text{since } \text{Bool} : * \text{ but } * \not\in S \leq
\end{align*}
\]

Note that the sort $s \in S \leq$ (e.g. $s = \square$) is one level above $A$ (e.g. $\star$ or $\star \rightarrow \star$), two levels above both $a$ (Int or VehicleI) and $x$ (Nat or CarI), and three levels above possible inhabitants of $x$ (e.g. zero of type Nat), on which we can perform subsumption (e.g. zero : Int). So, since subtyping is mainly used for the subsumption rule, taking $s \in S \leq$ is mainly useful when there are terms three levels under $s$. We will see that the subtyping relation itself is defined for all levels, and not only for levels on which we can introduce subtyping. For example, in
\( \lambda \omega \leq \) we will have \( \Gamma \vdash \text{true} \leq \text{true} \) and \( \Gamma \vdash * \leq * \) although declarations \( x \leq \text{true} : \text{Bool} \) and \( x \leq * : \Box \) are not admitted.

Just as \( \mathcal{R} \) controls which \( \Pi \)-types (quantifications) can be formed, \( \mathcal{R} \leq \) controls which bounded \( \Pi \)-types (bounded quantifications) can be formed, and hence also which bounded abstractions can be formed. For example, in \( \lambda 2 \) the rule \((\Box, \times) \in \mathcal{R}\) makes the \( \Pi \)-type \( \Box \times \times \to \times \) possible, and similarly, in \( \lambda 2 \leq \) the bounded rule \((\Box, \times) \in \mathcal{R} \leq \) permits the bounded quantification \( \Pi X \leq \text{Int} : * \times \to \times \). The restriction of the first element of a bounded rule to \( \mathcal{S} \leq \) means that bounded quantifications are only permitted when there is subtyping on the appropriate sort. Typically, \( \mathcal{R} \leq \) is a subset of \( \mathcal{R} \).

**Definition 7.1.1.2 (Pseudoterms)**

The set of *pseudoterms* \( T \) of a \( \mathcal{PTS} \leq \lambda (S, A, R, S \leq, R \leq) \) is defined by

\[
T ::= V \mid S \mid (T \ T) \mid (\lambda V : T. \ T) \mid (\Pi V : T. \ T) \mid (\lambda V \leq T : T. \ T) \mid (\Pi V \leq T : T. \ T)
\]

where \( V \) is the set of variables.

In a pseudoterm \( \lambda x \leq a : A \) the \( \lambda \) binds occurrences of \( x \) in \( b \), and similarly for \( \Pi \). The notions of free and bound variables are defined accordingly.

As usual, we write \( A \rightarrow B \) for \( \Pi x : A \) \( B \) when \( x \notin \text{FV}(B) \).

**Definition 7.1.1.3 (Pseudocontexts)**

The set of *pseudocontexts* \( C \) of a \( \mathcal{PTS} \leq \lambda (S, A, R, S \leq, R \leq) \) is defined by

- \( \epsilon \in \mathcal{C} \),
- \( \Gamma, x : A \in \mathcal{C} \) if \( \Gamma \in \mathcal{C}, A \in T, x \in V \) and \( x \notin \text{FV}(\Gamma) \cup \text{FV}(A) \),
- \( \Gamma, x \leq a : A \in \mathcal{C} \) if \( \Gamma \in \mathcal{C}, a, A \in T, x \in V \) and \( x \notin \text{FV}(\Gamma) \cup \text{FV}(a) \cup \text{FV}(A) \).

Here \( \epsilon \) denotes the empty context, and \( \text{FV}(\epsilon) = \emptyset \), \( \text{FV}(\Gamma, x : A) = \text{FV}(\Gamma) \cup \{x\} \cup \text{FV}(A) \), and \( \text{FV}(\Gamma, x \leq a : A) = \text{FV}(\Gamma) \cup \{x\} \cup \text{FV}(a) \cup \text{FV}(A) \). The domain of a pseudocontext \( \Gamma \), written as \( \text{dom}(\Gamma) \) is the set of all variables declared in \( \Gamma \), so \( \text{dom}(\epsilon) = \emptyset, \text{dom}(\Gamma, x : A) = \text{dom}(\Gamma) \cup \{x\} \) and \( \text{dom}(\Gamma, x \leq a : A) = \text{dom}(\Gamma) \cup \{x\} \).

The usual definition of pseudocontext [Bar92] is more liberal, and imposes no restrictions on the variables. However, the typing rules of [Bar92] enforce our restrictions for well-formed contexts.

We have chosen this definition of pseudocontext over the more usual definition, for the following reason. The subtyping rules may not depend on the typing judgment. Therefore, subtyping may neither depend on the well-formedness judgment \( (\Gamma \vdash ok) \), since this judgment depends on typing. So the contexts in subtyping judgments can be any pseudocontext; we cannot enforce these contexts to be well-formed. But we need a certain hygiene of these contexts. Therefore we enforce this hygiene by restricting the set of pseudocontexts. A counter-example below Lemma 7.2.3.3 shows this hygiene is necessary.

**Definition 7.1.1.4 (Reduction)** The \( \beta \)-reduction relation \( \triangleright_\beta \leq \) \( T \times T \) is defined by

\[
(\lambda x : A. \ b) \ a \triangleright_\beta b[x := a]
\]

\[
(\lambda x \leq a' : A. \ b) \ a \triangleright_\beta b[z := a]
\]

and all the compatibility rules. The relation \( \triangleright_\beta \) is the reflexive and transitive closure of \( \triangleright_\beta \).
7.1. SYNTAX AND TYPING RULES

7.1.2 Typing Rules

We introduce a new kind of judgment:

\[ \Gamma \vdash A \leq B \]

This judgment says that \( A \) is a subtype of \( B \) in pseudocontext \( \Gamma \). We retain the old forms of judgments: \( \Gamma \vdash ok \) and \( \Gamma \vdash a : A \).

Definition 7.1.2.1 (Well-formedness of contexts)

\[
\begin{align*}
(C\text{-empty}) & \quad \varepsilon \vdash ok \\
(C\text{-var}) & \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash ok} \\
(C\text{-Bvar}) & \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash A : s}{\Gamma, x \leq a : A \vdash ok} \quad s \in S^\leq
\end{align*}
\]

Note that by our definition of pseudocontext, \( x \) cannot occur in \( \Gamma \) (\( x \not\in \text{FV}(\Gamma) \)) in rules (C-var) and (C-Bvar).

Discussion Rules (C-Empty) and (C-var) are the same as for ordinary PTSs. The (C-Bvar) rule says we can introduce a variable \( x \) with bound \( a \) of type \( A \) in the context, if indeed \( a \) has type \( A \) and \( A \) belongs to a sort \( s \in S^\leq \). So the set \( S^\leq \) controls on which levels we may introduce subtyping. e.g. if \( S = \{*, \square\} \), \( \mathcal{A} = \{(*:\square)\} \) and \( S^\leq = \{\square\} \), the context

\[
\text{Int:}*\square, \text{Nat} \leq \text{Int:}*\]

is well-formed.

Definition 7.1.2.2 (Unbounded typing rules) These rules are exactly the same as for ordinary PTSs (see Definition 2.2.1), except for the absence of the (conv) rule.

\[
\begin{align*}
\text{(axiom)} & \quad \frac{\Gamma \vdash ok}{\Gamma \vdash s_1 : s_2} \quad (s_1:s_2) \in \mathcal{A} \\
\text{(var)} & \quad \frac{\Gamma \vdash ok}{\Gamma \vdash x : A} \quad x : A \in \Gamma \\
\text{(II-form)} & \quad \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \frac{\Gamma \vdash (\Pi x : A. B) : s_3}{(s_1,s_2,s_3) \in \mathcal{R}} \\
\text{(II-intro)} & \quad \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \quad \frac{\Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B)}{\Gamma \vdash B[b/x := a]} \\
\text{(II-elim)} & \quad \Gamma \vdash b : (\Pi x : A. B) \quad \Gamma \vdash a : A \quad \frac{\Gamma \vdash B a : B[x := a]}{
}
\]

\[\Box\]
Definition 7.1.2.3 (Bounded typing rules)

(subsum) \[ \Gamma \vdash b : B \quad \Gamma \vdash B' : a \quad \Gamma \vdash B \leq B' \]
\[ \Gamma \vdash b : B' \]

(Bvar) \[ \Gamma \vdash \text{ok} \]
\[ \Gamma \vdash x : A \]
\[ x \leq a : A \in \Gamma \]

(BII-form) \[ \Gamma \vdash A : s_1 \quad \Gamma, x \leq a : A \vdash B : s_2 \]
\[ \Gamma \vdash (\Pi x \leq a : A. B) : s_3 \quad (s_1, s_2, s_3) \in \mathcal{R}^\leq \]

(BII-intro) \[ \Gamma, x \leq a : A \vdash b : B \quad \Gamma \vdash (\Pi x \leq a : A. B) : s \]
\[ \Gamma \vdash (\lambda x \leq a : A. b) : (\Pi x \leq a : A. B) \]

(BII-elim) \[ \Gamma \vdash b : (\Pi x \leq a : A. B) \quad \Gamma \vdash a' : A \quad \Gamma \vdash a' \leq a \]
\[ \Gamma \vdash b[a' := a'] \]

Discussion

- The (subsum) rule is the essential rule for subtyping. It says that if \( b \) has type \( B \) and \( B \) is a subtype of \( B' \), then \( b \) also has type \( B' \). This property is called subsumption, hence the name of the rule. This rule can easily be understood by interpreting \( b : B \) as \( "b \) is an element of \( B" \) and \( B \leq B' \) as \( "B \) is a subset of \( B'\)".

The demand \( \Gamma \vdash B' : s \) is necessary to ensure \( B' \) is not an ill-behaved pseudoterm (recall that subtyping is possible on all pseudoterms). This rule is similar to the (conv) rule in ordinary PTSs (see Definition 2.2.1). Indeed, the (subsum) rule is a generalization of the (conv) rule. Instead of demanding \( B =_\beta B' \) we have \( \Gamma \vdash B \leq B' \). By looking ahead to the (\( \leq \)-conv) rule in Definition 7.1.2.4, we see that (subsum) indeed generalizes the (conv) rule.

Note that we do not put any restrictions on \( s \). We discuss a more constrained version of this rule in Section 7.1.4.

- The rules (Bvar) through (BII-elim) are the bounded analogies of rules (var) through (II-elim). The formation rule (BII-form) is parametrized by \( \mathcal{R}^\leq \) instead of \( \mathcal{R} \). Rule (BII-elim) expresses what a bounded quantification means; if \( b \) has type \( \Pi x \leq a : A. B \), then it may only be applied to terms \( a' \) which are a subtype of \( a \).

Definition 7.1.2.4 (Subtyping rules)

(\( \leq \)-conv) \[ \frac{a =_\beta b}{\Gamma \vdash a \leq b} \]

(\( \leq \)-trans) \[ \frac{\Gamma \vdash a \leq b \quad \Gamma \vdash b \leq c}{\Gamma \vdash a \leq c} \]

(\( \leq \)-var) \[ \frac{\Gamma \vdash x \leq a}{x \leq a : A \in \Gamma} \]

(\( \leq \)-II) \[ \frac{\Gamma \vdash A' \leq A \quad \Gamma, x : A' \vdash B \leq B'}{\Gamma \vdash (\Pi x : A. B) \leq (\Pi x : A'. B')} \]
7.1. SYNTAX AND TYPING RULES

\[
\begin{align*}
(\leq\text{-BII}) & \quad \frac{\Gamma, x \leq a : A \vdash B \leq B'}{\Gamma \vdash (\Pi x \leq a : A. B) \leq (\Pi x \leq a : A. B')} \\
(\leq\text{-}\lambda) & \quad \frac{\Gamma, x : A \vdash b \leq b'}{\Gamma \vdash (\lambda x : A. b) \leq (\lambda x : A. b')} \\
(\leq\text{-app}) & \quad \frac{\Gamma \vdash b \leq b'}{\Gamma \vdash b \ a \leq b' \ a}
\end{align*}
\]

Discussion First note that no subtyping rule depends on a typing judgment, so we comply to our design decision. As a consequence, \(\leq\) is a relation on pseudoterm.

- \((\leq\text{-conv})\) says \(\leq\) is a reflexive relation with respect to \(\beta\)-conversion.
- Rule \((\leq\text{-trans})\) expresses the transitivity of subtyping.
- Rule \((\leq\text{-var})\) is the source of subtyping: a variable declared as a subtype of some term \(a\) is indeed a subtype of \(a\). Subtyping between base types, a common source of subtyping in type systems, can be achieved by some initial context, e.g.

\[
\Gamma_{init} = \text{Int} : *, \text{Nat} \leq \text{Int} : *
\]

and we have \(\Gamma_{init} \vdash \text{Nat} \leq \text{Int}\) by \((\leq\text{-var})\).

- Rule \((\leq\text{\text{-}\Pi})\) is a more general formulation of the usual subtyping rule for \(\to\)-types:

\[
\begin{align*}
(\leq\text{-}\to) & \quad \frac{\Gamma \vdash A' \leq A \quad \Gamma \vdash B \leq B'}{\Gamma \vdash A \to B \leq A' \to B'}
\end{align*}
\]

(Recall that \(A \to B\) is short for \(\Pi x : A. B\) if \(x \notin \text{FV}(B)\).) Note that since we have subtyping on all terms, we can use this rule even if there is no interesting subtyping on \(B\) (or \(A\)). E.g. take \(\Gamma \equiv \text{Int} : *, \text{Nat} \leq \text{Int} : *\), then

\[
\frac{\text{Nat} \leq \text{Int} : * \in \Gamma}{\Gamma \vdash \text{Nat} \leq \text{Int}}
\]

\[
\frac{(\leq\text{-}\text{var}) \quad \Gamma \vdash * \leq *}{(\leq\text{-}\to) \quad \Gamma \vdash \text{Int} \to * \leq \text{Nat} \to *}
\]

- Rule \((\leq\text{-BII})\) is called the kernel-Fun rule, since it appears in Cardelli and Wegner's original Fun calculus [CW85]. There are alternatives for this rule which are more general (see Section 7.1.3.2), but \((\leq\text{-BII})\) has the best meta-theoretical properties and seems adequate for our purposes.
- Rules \((\leq\text{-}\lambda)\) and \((\leq\text{-app})\) are the usual rules for lifting subtyping pointwise to higher levels in systems with type constructors, e.g. in \(F^\omega\) [PS97] and in \(\lambda P^\omega\) [AC96b]. Pointwise lifting of \(\leq\) means that function \(f\) is a subtype of function \(g\) iff for all \(x\) we have \(f \ x \leq g \ x\).

**Definition 7.1.2.5** The notions of well-typed and typable, as defined in Definition 2.2.1.1 for PTSs, are also used for PTSs.

1. A term \(A\) is well-typed in context \(\Gamma\) if \(\Gamma \vdash A : B\) or \(\Gamma \vdash B : A\) for some \(B\).
2. A term \(a\) is typable in context \(\Gamma\) if \(\Gamma \vdash a : A\) for some \(A\). \(\square\)
CHAPTER 7. PTSs WITH SUBTYPING

7.1.3 Examples of PTSs

Now we will give examples of some well-known type systems with subtyping, and show how they fit in this framework. Most of these examples are systems of the λ-cube [Bar92] extended with subtyping. The systems in the λ-cube all have

\[ \mathcal{S} = \{*, \square\}, \mathcal{A} = \{(*: \square)\} \]

and \( \mathcal{R} \) consists only of pairs. The systems are extended with subtyping by choosing \( \mathcal{S}^\leq = \{\square\} \), and taking for \( \mathcal{R}^\leq \) a subset of rules \( \square, s_2 \) from \( \mathcal{R} \). By Definition 7.1.1.1, the first element of a bounded rule \( (s_1, s_2) \) must be in \( \mathcal{S}^\leq \), so we have \( s_1 \equiv \square \). At the very end of this section, we discuss why \( * \not\in \mathcal{S}^\leq \).

In order to discuss the examples, we introduce the following terminology (suggested by the interpretation of the systems as programming languages):

- If \( \Gamma \vdash K : \square \) then \( K \) is a kind. Examples of kinds are \( *, * \rightarrow * \) and \( \text{Nat} \rightarrow * \) (the last two kinds are only present in some systems).
- If \( \Gamma \vdash A : * \) then \( A \) is a datatype.
- If \( \Gamma \vdash A : K : \square \) then \( A \) is a datatype-constructor. This includes all datatypes, namely \( K \equiv * \). If \( K \not= * \) then \( A \) is a proper datatype-constructor.
- If \( \Gamma \vdash a : A : * \) then \( a \) is a program.

Now we turn to the individual examples. Each example consists of the definition of the PTS, some examples of what is derivable in this system, a discussion how the general derivation rules are used in this particular system, and comparable systems from the literature.

7.1.3.1 The Simply Typed λ-Calculus with Subtyping

The PTS \( \lambda \rightarrow \leq \) is specified by:

\[ \mathcal{S} = \{*, \square\}, \mathcal{A} = \{(*: \square)\}, \mathcal{R} = \{(*, *)\}, \mathcal{S}^\leq = \{\square\}, \mathcal{R}^\leq = \emptyset \]

Since \( \square \in \mathcal{S}^\leq \) we can make and use subtyping declarations. For example, let \( \Gamma_{\text{Nat}} \) declare the datatype Int and Nat as a subtype of Int, and the natural one, formally:

\[ \Gamma_{\text{Nat}} \equiv \text{Int}: *, \text{Nat} \leq \text{Int} : *, \text{one}: \text{Nat} , \]

then an example of a derivable judgment in \( \lambda \rightarrow \leq \) is:

\[ \Gamma_{\text{Nat}}: \text{negate: Int} \rightarrow \text{Int} \vdash \text{negate one : Int} \]

This is easy to derive by using the (subsum) rule to give one type Int, or by using (\( \leq \)-II) and (subsum) to give negate type Nat \( \rightarrow \) Int.

Many of the rules for PTS \( \leq \) are not applicable in this simple type system.

- Rules (BII-form) through (BII-elim) and (\( \leq \)-BII) are not applicable, since \( \mathcal{R}^\leq \) is empty.
- There is no \( \beta \)-conversion on datatypes, so (\( \leq \)-conv) can only be used to derive \( \Gamma \vdash A \leq A \) if \( A \) is a datatype.
7.1. SYNTAX AND TYPING RULES

- There are no proper datatype-constructors, so subtyping cannot be lifted to them. Hence (≤-λ) and (≤-app) are not used in λ→≤.

The system λ→≤ is the standard extension of λ→ with subtyping, e.g. defined in [Com95].
λ→≤ is the pure fragment of [Car88], where pure means that practical constructs like some primitive datatypes, records, variants and recursive types have been left out.

7.1.3.2 The Polymorphic λ-Calculus with Subtyping

The system λ2≤ is specified by:

\[
S = \{*, \Box\}, \quad A = \{(*:\Box)\}, \quad \mathcal{R} = \{(*, *), (\Box, *)\}, \quad \mathcal{S}^{\leq} = \{\Box\}, \quad \mathcal{R}^{\leq} = \{(*, *)\}
\]

Since (\Box, *) ∈ \mathcal{R}^{\leq}, we can make bounded quantifications. As an example, consider the extensions of \(\Gamma_{\mathbb{Nat}}\) with the declaration of the polymorphic max function, which works for any subtype of Int. We have

\[
\Gamma_{\mathbb{Nat}}, \text{max} : \Pi X \leq \text{Int} : \ast, \text{X} \rightarrow \text{X} \rightarrow \text{X} \vdash \text{max} \text{ Nat one} \text{ one} : \text{Nat}
\]

This example clarifies the reason for having bounded quantifications. Without them, \text{max} could only be declared as \text{max} : \text{Int} → \text{Int} → \text{Int}, and the term \text{max one one} would only have type Int, losing the information that the term is a Nat.

When \(\Gamma_{\mathbb{Nat}}\) is extended with declarations for some ordinary primitives (like \text{less} : \text{Int} → \text{Int} → \text{Bool}), we can define \text{max} as follows:

\[
\lambda X \leq \text{Int} : \ast. \lambda x, y : X. \text{ if (less x y) y x}
\]

The restriction of X to subtypes of Int is necessary to type the subterm less x y.

Just as for λ→≤, the rules (≤-λ) and (≤-app) are not used in λ2≤, since there are no proper datatype-constructors. But the rules for bounded quantifications are used. There has been quite some debate about the subtyping rule for these bounded quantifications. The main concern is the meta-theory. Our rule (≤-BII) is the Fun rule [CW85] for bounded quantification which yields a decidable system. Instead of (≤-BII), other rules have been proposed, that allow the bounds in the quantifications to be different. The first, and most natural, proposal was the rule of system \(F_{\leq}\) [CG92]:

\[
(≤-\text{BII-contr}) \quad \Gamma \vdash a' \leq a \quad \Gamma, x \leq a' : A \vdash B \leq B' \quad \Gamma \vdash (\Pi x \leq a : A. B) \leq (\Pi x \leq a' : A. B')
\]

However, this rule renders subtyping undecidable in \(F_{\leq}\) [Pie94]. Another alternative for (≤-BII), described in [CP94], turned out to destroy the minimal typing property [CP95]. Minimal typing (see Theorem 7.2.5.19) is necessary to keep type-checking tractable.

The system λ2≤ is equal to kernel-Fun (the pure fragment of Fun [CW85]), except for their Top type. The rules for the Top type in λ2≤ would be:

\[
\begin{align*}
(\text{Top-form}) \quad & \Gamma \vdash \text{ok} \\
\hline
& \Gamma \vdash \text{Top} : \ast \\
(≤-\text{Top}) \quad & \Gamma \vdash A : \ast \\
\hline
& \Gamma \vdash A \leq \text{Top}
\end{align*}
\]
The second rule says that any datatype is a subtype of $\text{Top}$. We did not include $\text{Top}$, since the subtyping rule ($\leq\text{-Top}$) essentially depends on a typing judgment. This is incompatible with our approach, where subtyping does not depend on typing.

The absence of $\text{Top}$ types in $\text{PTS}^\leq$ is not as bad as it seems. The original reason for having $\text{Top}$ is that an ordinary quantification over types, i.e. $\Pi X : \text{Top}$. $A$ can be considered as a bounded quantification with bound $\text{Top}$, i.e. $\Pi X \leq \text{Top} : \times$. $A$, so ordinary quantifications over types are redundant, decreasing the number of rules. In our approach, we need ordinary quantifications anyway, e.g. for programs of type $\Pi x : \text{Nat}, \text{Nat}$, conventionally written as $\text{Nat} \to \text{Nat}$. So $\text{Top}$ cannot eliminate the need for ordinary quantifications, hence the original reason for having $\text{Top}$ is not valid for $\text{PTS}^\leq$.

An advantage of having $\text{Top}$ is that records with subtyping can be encoded using $\text{Top}$ [Car92]. We will have to do without such an encoding, and introduce records as primitives.

### 7.1.3.3 The Higher-Order Polymorphic $\lambda$-Calculus with Subtyping, $\lambda\omega^\leq$

The system $\lambda\omega^\leq$ is specified by:

$$
\mathcal{S} = \{*, \square\}, \quad \mathcal{A} = \{(*,\square)\}, \quad \mathcal{R} = \{(*,\ast), (\square,\ast), (\square,\square)\},
\mathcal{S}^\leq = \{\square\}, \quad \mathcal{R}^\leq = \{(\square,\ast)\}
$$

The difference with $\lambda\omega^\leq$ is that now $(\square,\square) \in \mathcal{R}$. As a consequence, we have proper datatype-constructors. This has two effects in combination with subtyping.

1. We have bounded quantifications were the bound is a proper datatype-constructor.
2. We have lifted subtyping on proper datatype-constructors.

For example, take the following context:

$$
\Gamma \equiv \begin{array}{lcl}
\text{VehicleI}:\ast & \rightarrow & \ast, \\
\text{Object}:(* \rightarrow \ast) & \rightarrow & \ast, \\
\text{increaseSpeed}: (\Pi \leq \text{VehicleI} : \ast \rightarrow \ast, \text{Object I} \rightarrow \text{Object I}), \\
\text{CarI} \leq \text{VehicleI} : \ast \rightarrow \ast, \\
\text{beetle}:\text{Object CarI}
\end{array}
$$

It declares a (proper) datatype-constructor $\text{VehicleI}$ (intention: interface of the class of vehicles), a datatype-constructor $\text{Object}$ (intention: given an interface, it delivers the type of objects with that interface), $\text{increaseSpeed}$ which given an $I$ which is a subtype of $\text{VehicleI}$ transforms an object with interface $I$, $\text{CarI}$ (intention: interface of the class of cars, which is a subclass of the class of vehicles), and finally $\text{beetle}$ (intention: an object in the class of cars). The type of $\text{increaseSpeed}$ is an example of effect 1 indicated above, and the declaration of $\text{CarI}$ is an example of effect 2.

Now we have

$$
\Gamma \vdash \text{increaseSpeed CarI beetle} : \text{Object CarI}
$$

so $\text{increaseSpeed}$ working on $\text{beetle}$ delivers a new car object.

All derivation rules for $\text{PTS}^\leq$ are used in $\lambda\omega^\leq$, including ($\leq\ast$) and ($\leq\text{app}$) for lifting subtyping to datatype-constructors.
7.1. SYNTAX AND TYPING RULES

\( \lambda \omega \leq \) is equal to \( F^\omega \leq \) [PS97], except that \( F^\omega \leq \) has a family of \( \text{Top} \)-types, for every kind one, e.g. \( \text{Top}(\ast) \), which corresponds to \( \text{Top} \) in the previous section, and \( \text{Top}(\ast \rightarrow \ast) \). The rules for these \( \text{Top} \) types in \( \lambda \omega \leq \) would be:

\[
\begin{align*}
(\text{Top-form}) & \quad \Gamma \vdash K : \Box \\
& \quad \Gamma \vdash \text{Top}(K) : K \\
(\leq \text{-Top}) & \quad \Gamma \vdash A : K \\
& \quad \Gamma \vdash K : \Box \\
& \quad \Gamma \vdash A \leq \text{Top}(K)
\end{align*}
\]

These rules could be easily generalized to arbitrary \( PTS \leq s \), by replacing \( \Box \) with an \( s \in S \leq \). However, as discussed for \( \lambda \Omega \leq \), the rule \( (\leq \text{-Top}) \) is incompatible with our approach.

The system \( F^\omega \leq_\lambda \) [Com95] can be considered as an extension of \( \text{PS97} \) with intersection types (although both systems were developed independently). The intersection of types \( A \) and \( B \) is the greatest lower bound of \( A \) and \( B \) with respect to the subtyping ordering, so \( A \land B \) is both a subtype of \( A \) and \( B \).

7.1.3.4 \( \lambda \omega \leq \) with Bounded Constructor Abstraction

The system \( \lambda \omega \leq^{+} \) is specified by:

\[
S = \{\ast, \Box\}, \quad A = \{(\ast : \Box)\}, \quad R = \{(\ast ,\ast), (\Box, \ast), (\Box, \Box)\}, \quad S \leq = \{\Box\}, \quad R \leq = \{(\Box, \ast), (\Box, \Box)\}
\]

The only difference with \( \lambda \omega \leq \) is that we now have \( (\Box, \Box) \in R \leq \). In this system, we can derive

\[
Y : \ast \vdash (\lambda X \leq Y : \ast. \ X \rightarrow X) \ : (\Pi X \leq Y : \ast. \ast)
\]

using \( (\Box, \Box) \in R \leq \). This \( \lambda \)-abstraction is called a bounded constructor abstraction, since it is a datatype-constructor and a bounded abstraction. In \( \lambda \omega \leq^{+} \) itself, these bounded constructor abstractions have not much use, since there is no way we can use the fact that a variable (in our example, \( X \)) is bounded: only in programs subsumption can be used, and datatypes cannot contain programs, so subsumption cannot be used in datatypes. As a consequence, we may consistently replace these bounded constructor abstractions with unbounded abstractions and the resulting term would still be typable. However, in extensions of \( \lambda \omega \leq^{+} \) these bounded abstractions can be useful, for example if some primitive operations on datatypes put certain subtyping demands on their arguments.

The system \( \lambda \omega \leq^{+} \) corresponds with the system \( F^\omega \leq_\lambda \) defined in [CG97]. There are two differences. First, we have no \( \text{Top} \)-types (see Sections 7.1.3.2 and 7.1.3.3). Second, we do not have subtyping on these bounded abstractions (they do):

\[
(\leq \text{-B} \lambda) \quad \Gamma, x \leq a : A \vdash b \leq b' \quad \Gamma \vdash (\lambda x \leq a : A. \ b) \leq (\lambda x \leq a : A. \ b')
\]

We do not have this rule because it destroys an important meta-theoretic property, formulated in Lemma 7.2.3.8, see Remark 7.2.3.10. (The system \( F^\omega \leq_\lambda \) does have this property, but only for typable terms, whereas we need it for all pseudoterms.)
7.1.3.5 The Simply Dependent \( \lambda \)-Calculus with Subtyping

The system \( \lambda P^\leq \) is specified as follows:

\[
S = \{*, \square\}, \quad A = \{(*: \square)\}, \quad R = \{(*, *), (*, \square), (\square, \square), (\square, \triangle)\},
\]

\[
S^\leq = \{\square\}, \quad R^\leq = \emptyset
\]

The rule \((*, \square) \in R\) allows the following datatypes depending on programs, giving proper datatype-constructors, for which lifted subtyping is possible.

For example, take the following well-formed context. It declares a datatype of collections, a datatype-constructor \texttt{Bag}, which given a number \( n \) represents the bags with size \( n \), and a datatype-constructor \texttt{List}, which given a number \( n \) represents the lists with size \( n \).

\[
\Gamma \equiv \quad \text{Nat} : *,
\]

\[
\quad \text{Collection} : *
\]

\[
\quad \text{Bag} : (\lambda n : \text{Nat}. \text{Collection}) : \text{Nat} \rightarrow *
\]

\[
\quad \text{List} \leq \text{Bag} : \text{Nat} \rightarrow *
\]

In this context, for every \( n \) we have \( \text{Bag} n \leq \text{Collection} \), so an element of \( \text{Bag} n \) is a Collection, and \( \text{List} n \leq \text{Bag} n \), so an element of \( \text{List} n \) is also a \( \text{Bag} n \), and hence also a Collection.

Rules \((\leq\rightarrow\lambda)\) and \((\leq\rightarrow\Pi)\) give subtyping on dependent types.

The system \( \lambda P^\leq \) as described in [AC96b] is roughly the same as this \( PTS^\leq \). To be more precise, typing on programs in both systems is equivalent. Subtyping, and typing on datatype-constructors are not exactly the same (but this has no effect on programs, surprisingly).

Consider the context \( \Gamma \equiv \text{Int} : *, \text{Nat} \leq \text{Int} : *, \text{Bool} : * \). In our version of \( \lambda P^\leq \) we have \( \Gamma \vdash \text{Int} \rightarrow * \leq \text{Nat} \rightarrow * \) and also \( \Gamma, \text{even} : \text{Int} \rightarrow * \vdash \text{even} : \text{Nat} \rightarrow * \), but in [AC96b] neither judgment is derivable.

7.1.3.6 The Calculus of Constructions, \( \lambda C^\leq \)

The \( PTS^\leq \lambda C^\leq \) is specified as follows:

\[
S = \{*, \square\}, \quad A = \{(*: \square)\}, \quad R = \{(*, *), (*, \square), (\square, *), (\square, \square)\},
\]

\[
S^\leq = \{\square\}, \quad R^\leq = \{(\square, *), (\square, \square)\}
\]

This is the Calculus of Constructions [CH88], the most powerful system in the \( \lambda \)-cube, extended with subtyping and bounded quantifications. It includes all systems given above. In contrast to \( \lambda \omega^\leq \), the bounded rule \((\square, \square) \in R^\leq \) is quite useful here. For example, take

\[
\Gamma \equiv \quad A : *,
\]

\[
f : (\Pi X : A : *. X \rightarrow X),
\]

\[
equals : (\Pi X : *. X \rightarrow X \rightarrow *)
\]

This context declares a datatype \( A \), a program \( f \) which transforms elements of subtypes of \( A \), and a polymorphic function \( \text{equals} \) (intention: equality of programs). Now we have

\[
\Gamma \vdash (\lambda X : A : *. \lambda x : X. \text{equals } X (f X x) x) : (\Pi X : A : *. X \rightarrow *)
\]

This bounded abstraction can be interpreted as polymorphic predicate, and its formation depends on \((\square, \square) \in R^\leq \).

For more examples we refer to the next chapter, where \( \lambda \omega^\leq \), a refinement of this system, will be used.

The system \( \lambda C^\leq \) has not come up in the literature.
7.1. SYNTAX AND TYPING RULES

7.1.3.7 \( \lambda C \leq \) without Bounded Quantifications

The \( PTS \leq \lambda C \leq \) is specified as follows:

\[
\begin{align*}
S &= \{ *, \square \}, \\
A &= \{ \{ *, \square \} \}, \\
R &= \{ \{ *, \}, \{ *, \square \}, \{ \square, \} \}, \\
S^\leq &= \{ \square \}, \\
R^\leq &= \emptyset
\end{align*}
\]

This is the subsystem of \( \lambda C \leq \) where bounded quantifications have been left out. The \( PTS \leq \lambda C \leq \) is exactly the same as the system defined in [Che97] (although Chen calls his system \( \lambda C \leq \), which we find rather unfortunate). An interesting observation in [Che97] is that the rule \((\leq, \lambda)\) is redundant in this system.

7.1.3.8 The Programming Logic \( \lambda \omega F \)

\( \lambda \omega F \) is specified as follows:

\[
\begin{align*}
S &= \{ *, \square, _s, _p, \square_p \} \\
A &= \{ \{ *, \square \}, \{ *, \square_p \} \} \\
R &= \{ \{ \square, \square \}, \\
& \quad \{ \square, _s \}, \{ _s, _s \}, \\
& \quad \{ _s, _p \}, \{ _p, \square_p \}, \{ \square_p, \square_p \}, \\
& \quad \{ \square, _p \}, \{ _s, _p \}, \{ _p, _p \}, \{ _p, _p \} \} \\
S^\leq &= \{ \square \} \\
R^\leq &= \{ \{ \square, _s \}, \{ \square, _p \}, \{ \square, \square_p \} \}
\end{align*}
\]

This system is a refinement of \( \lambda C \leq \), by splitting \( * \) and \( \square \) into \( _s \), \( _p \) and \( _s \), \( _p \) respectively. Note that we do not have \( \{ \square, _s \} \) in \( R^\leq \). We have refrained from this bounded rule for exactly the same reason as that having \( \{ \square, \square \} \in R^\leq \) in \( \lambda \omega F \) does not make much sense (see Section 7.1.3.4). We allow bounded variables of sort \( _s \), which will be interpreted as datatype-constructors, but we do not allow bounded variables of sort \( \square_p \), which will be interpreted as propositions and proposition-constructors.

Examples of the use of this system will be given in the next chapter.

7.1.3.9 Why no Subtyping on Programs

In the examples above we had \( S^\leq = \{ \square \} \). In principle it is also possible to take \( S^\leq = \{ * \} \), or even \( S^\leq = \{ *, \square \} \). In this section we discuss why taking \( * \in S^\leq \) is not useful.

First we give a very short and intuitive argument. Having \( * \in S^\leq \) results in subtyping on programs, e.g. a declaration \( x : \text{true} : \text{bool} \) is now admitted, and this is nonsense.

A longer and more technical argument is the following. Suppose \( * \in S^\leq \) and \( a \) and \( b \) are programs (i.e. \( a : A : * \) and \( b : B : * \)). Now it is possible to have \( a \leq b \) without \( a \) and \( b \) being equal, e.g. when \( a \) is a variable \( x \) declared as \( x \leq b : B \). We will show this makes no sense. Take the (subsum) rule:

\[
\frac{\Gamma \vdash c : a \quad \Gamma \vdash b : s \quad \Gamma \vdash a \leq b}{\Gamma \vdash c : b}
\]

This rule cannot be used for two reasons. First, \( b \) is not typable with a sort. Second, and more essentially, \( a \) does not have any inhabitant (element) \( c \). More intuitively, "\( a \leq b \)" means
any element of \( a \) is also an element of \( b \), so "\( a \leq b \)" makes only sense when \( a \) and \( b \) have elements, i.e. \( a \) and \( b \) are types. This is reflected in the phrase "\( a \) is a subtype of \( b \)" for \( a \leq b \). (Proper datatype-constructors also have no elements, but if applied to enough types, they do. Programs never have elements, and remain programs when applied to terms.)

Ignoring subsumption, we attempted to use bounded quantifications to model partial functions, giving a quite different interpretation to subtyping. These experiments were not convincing. However, chapter 2 of [Fei90] shows that there is a sensible interpretation for \( \leq \), namely the so-called "implementation relation". The \( \lambda \)-calculus defined by Feijs bears a strong resemblance to the \( PTS \leq \) with the following specification:

\[
S = \{*, \Box \}, \quad A = \{(*:\Box)\}, \quad R = \emptyset, \quad S^\leq = \{\Box\}, \quad R^\leq = \{(*,*)\}
\]

There are two important differences. First, in [Fei90] the type of a term \( (\lambda x \leq a : A. \ b) \) is \( A \to B \), and not the bounded quantification (\( \Pi x \leq a : A. \ B \)) as in the \( PTS \leq \) above. Second, in [Fei90] \( \beta \)-reduction depends on the \( \leq \) ordering: \( (\lambda x \leq a : A. \ b) \ c \to^\beta \ b[x := c] \) only if \( c \leq a \), whereas in all \( PTS \leq \)s \( \beta \)-reduction is independent of \( \leq \) (we ensure by the typing rule (\( \Pi \)-elim) that a bounded abstraction \( \lambda x \leq a : A. \ b \) is only applied to arguments \( \leq a \)). Because of these differences, and the fact that subsumption is not present in [Fei90], we feel that \( PTS \leq \)s are not very appropriate to describe this \( \lambda \)-calculus of Feijs.

Summarizing, having \( * \in S^\leq \) allows us to have \( a \leq b \) if \( a \) and \( b \) are different programs. Interpreting \( \leq \) in the usual way does not make sense since \( a \) and \( b \) are no types. Giving \( a \leq b \) another interpretation has very little practical use or is not appropriate. Therefore we take \( * \notin S^\leq \).

7.1.4 Alternatives for Rules

In this section we discuss some alternatives for the rules we have given, and why we have rejected these alternatives. Keep in mind that our goal is modelling OOP, and the means by which we hope to achieve this is the system \( \lambda \omega^T \). So we strive for a system that is as simple as possible, and reject some alternatives which do not bring additional expressiveness to \( \lambda \omega^T \). We will see in Section 7.2 that the theory for the present system is difficult enough without the following alternatives.

The alternatives spring from two sources. First, the wish to describe systems from the literature as precisely as possible. Second, the generality of the \( PTS \leq \) scheme: the generality makes many alternatives possible that have not come up in existing systems, which are particular instantiations of the scheme.

- The extension with \( Top \) types, and why we rejected them, is discussed in Sections 7.1.3.2 and 7.1.3.3. As explained there, this follows from our design choice about subtyping judgments not depending on typing judgments.

- Some alternatives for the \( (\leq \text{-BII}) \) rule are described in Section 7.1.3.2. Another alternative is the following rule:

\[
(\leq \text{-BII-liberal}) \quad \Gamma \vdash A' \leq A, \quad \Gamma, x \leq a : A' \vdash B \leq B' \quad \frac{}{\Gamma \vdash (\Pi x \leq a : A. \ B) \leq (\Pi x \leq a : A'. \ B')}
\]

This rule has not come up in the literature. It is only applicable if there are bounded quantifications and there is subtyping on two levels, viz. on the level of \( a \), and one level
higher on the level of $A$ and $A'$. No system from the literature discussed in Section 7.1.3 has both, and neither has $\lambda\omega^<_L$.

- We could introduce subtyping on bounded abstractions, as discussed in Section 7.1.3.4. This has no effect in $\lambda\omega^<_L$.

- We could consider the following rule, that says that an ordinary quantification is a subtype of a bounded quantification:

$$\frac{\Gamma, x \leq a : A \vdash B \leq B'}{\Gamma \vdash (\Pi x : A. B) \leq (\Pi x \leq a : A. B')}$$

This rule seems to behave well, but we did not include it to minimize the number of subtyping rules.

- It is possible to generalize the $(\leq_\lambda)$ rule. In a more liberal version, the domains of the functions could be different, as follows:

$$\frac{\Gamma \vdash A' \leq A \quad \Gamma, x : A' \vdash b \leq b'}{\Gamma \vdash (\lambda x : A. b) \leq (\lambda x : A'. b')}$$

This rule does not have any effect on typing programs in $\lambda\omega^<_L$. Therefore we stick to the rule $(\leq_\lambda)$, where the domains of the functions are the same.

- We first considered a more constrained version of the (subsum) rule:

$$\frac{\Gamma \vdash b : B \quad \Gamma \vdash B' : s \quad \Gamma \vdash s : s' \quad \Gamma \vdash B \leq B'}{\Gamma \vdash b : B'}$$

We did so, because we believed the meta-theory would be easier because of the additional constraints on $s$. It turned out, however, that the meta-theory was more difficult, so we rejected this rule.

The rule is really more restrictive, e.g. consider in $\lambda P \leq$ the context

$$\Gamma \equiv \text{Int} : *, \text{Nat} \leq \text{Int} : *, P : \text{Nat} \rightarrow *$$

and a derivation ending in

$$\frac{\Gamma \vdash P : \text{Nat} \rightarrow * \quad \Gamma \vdash \text{Int} \rightarrow * : \Box \quad \Gamma \vdash \text{Nat} \rightarrow * \leq \text{Int} \rightarrow *}{\Gamma \vdash P : \text{Int} \rightarrow *}$$

The last step would not be correct by (subsum'), because $\Box$ cannot be typed.

- Unbounded quantification and bounded quantification are much alike. The presentation of the system would be more elegant (fewer rules) if we could consider both quantifications as instances of one more general quantification. One way of doing this is by introducing Top-types. We already discussed why we have not done so. Another way is to introduce power types [Car91]. Intuitively, the power type $\text{Sub}_A(a)$ is the subset of $A$ with only Subtypes of $a$ as elements. The bounded quantification $\Pi x \leq a : A. B$ would now be written as $\Pi x : \text{Sub}_A(a). B$. 
But besides providing a way to consider a bounded quantification as a special case of an ordinary quantification, many new terms can be made which do not have a counterpart in our syntax, e.g. $Sub^*_b(\text{Int}) \rightarrow Sub^*_b(\text{Int})$. Thus by introducing power types, the correspondence with existing systems, as discussed in Section 7.1.3, becomes weaker. This one reason for rejecting this approach.

Another reason is that with power types it is awkward to finetune the subtyping rules, which is necessary to keep type-checking decidable (see Section 7.1.3.2).

- In $PTS_{\leq s}$, these is no subtyping on sorts, i.e. $s_1 \leq s_2 \implies s_1 \equiv s_2$. We could allow subtyping on sorts, by introducing a set $A_{\leq}$ of subtyping axioms for sorts, with rule:

$$\frac{}{s_1 \leq s_2 \in A_{\leq}}$$

Adding $A_{\leq}$ allows us to consider the extended calculus of constructions [Luo89], with its infinity hierarchy of sorts and sort inclusion, as a $PTS_{\leq}$ (ignoring the strong sum types).

We have rejected this option to keep matters simple.

### 7.1.5 Summary of Meta-Theoretical Results

The main results of this chapter are meta-theoretical in nature, and are given in Sections 7.2 and 7.3. Since these sections are quite long, we give here a short survey of the results.

**Subject Reduction** for all $PTS_{\leq s}$.

**Minimal Typing** for all functional $PTS_{\leq s}$. All $PTS_{\leq s}$ that have come up in practice are functional. Minimal typing is the $PTS_{\leq s}$ counterpart of Uniqueness of Typing for $PTS$s.

**A type-checking algorithm** for all bijective $PTS_{\leq s}$ that are $\beta$-Strongly Normalizing. The bijective $PTS_{\leq s}$ are a subclass of the functional $PTS_{\leq s}$, and include all examples given in Section 7.1.3. We do not prove SN$_\beta$ for any $PTS_{\leq s}$, since this falls outside the scope of this thesis.

**Decidability of Typing** as a consequence of termination of this algorithm, for $PTS_{\leq s}$ that are Strongly Normalizing under a more general form of reduction, called $\beta\sigma$-reduction. Again, we do not prove this property for $PTS_{\leq s}$.

It is not necessary to read Sections 7.2 and 7.3 to understand the rest of this thesis.
7.2 Meta-Theory

In this section we develop the meta-theory for PTS-s. We list here briefly the problems we encountered when developing the meta-theory and show the relation to the theory for ordinary PTS-s and existing systems with subtyping. The problems are explained in more detail later on, and a comparison with individual subtyping systems is given at the end of this chapter (Section 7.4.1).

Mutual dependency between typing and subtyping. The first problem is already discussed in the introduction of this chapter, but is briefly summarized here. The usual subtyping rules in the literature are dependent on the typing rules and vice versa. This mutual dependency makes it very hard to generalize the meta-theory for these rules to PTS-s. For specific systems this problem can be circumvented (e.g. [PS97, AC96b, Che97]). For PTS-s it seems impossible to work around this problem, which led us to define subtyping rules independent of typing, and consider subtyping for all pseudoterms (including those that are not typable).

Tractable subtyping rules. In most systems with subtyping (including ours) the subtyping rules are intractable, by which we mean that it is hard to prove properties of the subtyping judgment (e.g. show it is decidable). Important causes of this problem are the transitivity rule (\leq\text{-trans}) and the application rule (\leq\text{-app}). The usual approach (e.g. [PS97, AC96b, Che97]), which we will follow, is to define alternative subtyping rules, which should be equivalent to the original ones, but with a much better behaviour. In particular, the alternative rules do not contain the original transitivity rule. Hence the subproblem of proving equivalence between the original and the alternative rules is also called “transitivity elimination”. Although the alternative subtyping rules are roughly the same in the various systems (including ours), the actual proofs of transitivity elimination differ.

Since we define subtyping for all pseudoterms, instead of only typable terms, we have two new, additional subproblems. First, the context occurring in subtyping judgments need not be well-formed (since checking well-formedness is essentially a typing problem). We already solved this problem by enforcing a certain hygiene on pseudocontexts (Definition 7.1.1.3). In addition, some lemmas need certain assumptions to provide this hygiene. Second, we need to introduce an unusual alternative subtyping rule, in order to have equivalence between original and alternative rules.

Minimal Typing. In functional PTS-s (without subtyping), we have that every term has a unique type (modulo β-conversion). Subtyping destroys this property, but we have the property of Minimal Typing, which says that every term has a so-called minimal type. We encountered two problems when proving this property. First, we need a way to compute minimal types. We do this by introducing a reduction relation, which can also be found in several other systems with subtyping [PS97, Che97]. Second, Minimal Typing cannot be proved by a single induction, because subtyping can occur on several levels. Therefore we first prove a property called Weak Minimal Typing. This second problem is new to PTS-s. In ordinary PTS-s such a problem does not occur for Uniqueness of Types, which is closely related to Minimal Typing. In existing systems with subtyping the problem does not occur because there is subtyping on only one level.
A further difference is that in the literature Minimal Typing is usually proved in conjunction with a typing algorithm, whereas we prove the property separately. We do this because our typing algorithm works only for a class of $PTS\leq$ with certain normalization properties, and we wanted to prove Minimal Typing for all functional $PTS\leq$.

Usually the subject of type-checking is also considered part of the meta-theory. Because of its length, we devote a separate section on type-checking. There we will discuss the problems related to this subject.

Section 7.2.1 explains in more detail why alternative subtyping rules are necessary and gives alternative, more restricted rules. In Section 7.2.2 we prove some basic properties about the subtyping rules. We are able to do so, because subtyping does not depend on typing. These basic properties are necessary for Section 7.2.3, which shows equivalence of the alternative and the original rules. Then we prove that the Subject Reduction property holds for all $PTS\leq$ (Section 7.2.4). Section 7.2.5 shows Minimal Typing for functional $PTS\leq$. Finally, Section 7.2.6 shows the relation between $PTS$s and $PTS\leq$: every $PTS$ can be considered as a $PTS\leq$.

We do not give the usual properties of substitution and reduction, because their proofs are all standard. We only mention the Church-Rosser property for $\beta$-reduction.

**Theorem 7.2.1 (Church-Rosser)** If $a =_\beta b$ then there is a $c$ with $a \Rightarrow_\beta c$ and $b \Rightarrow_\beta c$.

**Proof:** Since there is no essential difference in reduction compared to ordinary $PTS$s, this property can be proved by a straightforward adaptation of the usual proofs for $PTS$s. Alternatively, this property can be proved by considering the terms and reduction of $PTS\leq$ as an orthogonal Combinatory Reduction System [KOR93].

### 7.2.1 Restricted Subtyping Rules

Unfortunately, the subtyping rules given in Definition 7.1.2.4 are quite intractable; it is hard to prove properties about them. They are so intractable, because there is some redundancy in the subtyping rules; there can be several quite different derivations of the same subtyping judgment. This is caused by the fact that the rules are not at all syntax-directed. A set of rules is syntax-directed if the following two conditions hold:

1. The shapes (syntax) of the terms $a$ and $b$ in a judgment $\Gamma \vdash a \leq b$ determine (at most) one derivation rule which can be applied to derive this judgment. For example, both $(\leq\text{-app})$ and $(\leq\text{-conv})$ can be applied when $a \equiv d e$ and $b \equiv d' e$, so the subtyping rules are not syntax-directed.

2. Each individual rule is syntax-directed, i.e. the terms in the conclusion uniquely determine the terms in the premises. For example, $(\leq\text{-trans})$ is the only rule that is not syntax-directed.

This definition of the concept “syntax-directed” is led by the desire to reason about derivation trees backwards, from conclusions to premises. Given that some judgment is derivable, we know by the first condition what the last rule in a derivation tree is, and by the second condition how this rule is used, so what the premises of the rule are. Proceeding upwards through the tree, the whole tree is determined by the judgment it derives. Hence, it is much easier to prove properties of a set of rules if this set is syntax-directed.
Therefore we introduce a set of more restricted rules, equivalent to the set in Definition 7.1.2.4, but much more syntax-directed. This set of restricted rules behaves much better, and in particular has the following crucial property: a subtype derivation using the restricted rules does not introduce untypable terms. To be more precise, if the terms in the conclusion of such a subtyping judgment are typable, then all terms occurring in the derivation of this judgment are typable. We will only be able to show this at the end of Section 7.2.4, in Lemma 7.2.4.13. The original rules do not have this property.

Most systems with subtyping suffer from intractable subtyping rules, and we follow the usual approach by giving an alternative set of restricted rules. As usual, the two rules responsible for the intractibility of the set of original rules are (≤-trans) and (≤-app). We discuss for both rules which problems they cause, which restricted rules (in Definition 7.2.1.2 below) replace them, and how these restricted rules are related to rules found in the literature.

The (≤-trans) rule

This rule is the most responsible for the intractibility of the original subtyping rules. We repeat it here:

\[
(\leq\text{-trans}) \quad \Gamma \vdash a \leq b \quad \Gamma \vdash b \leq c \quad \frac{}{\Gamma \vdash a \leq c}
\]

It can be used at any moment in a derivation, since there are no restrictions on the form of the conclusions \( a \) and \( c \). Even worse, the term \( b \) in the premises cannot be determined from \( a \) and \( c \), and even when \( a \) and \( c \) are typable terms, \( b \) can be a non-typable term. For example, take the system \( \lambda \leq \) and \( \Gamma \equiv \text{Nat} : * \) and \( a \equiv c \equiv \text{Nat} \) and \( b \equiv (\lambda x : *. \text{Nat}) (W W) \), where \( W \equiv \lambda X : *. X X \). This shows (≤-trans) is responsible for the original rules not having the crucial property mentioned above. So we want to remove this rule.

However, it is essential in two situations.

1. It is necessary to perform reductions on the terms of the subtyping judgment, using the (≤-conv) rule, as follows:

   \[
   (\leq\text{-conv}) \quad a \triangleright_{\beta} a' \quad \frac{}{\Gamma \vdash a \leq a'}
   \]

   \[
   (\leq\text{-trans}) \quad \frac{}{\Gamma \vdash a \leq b} \quad \frac{}{\Gamma \vdash b \leq b'} \quad \frac{}{\Gamma \vdash a' \leq b'}
   \]

   \[
   \frac{}{\Gamma \vdash a \leq b'}
   \]

   This use of the (≤-trans) rule is taken over by the more direct (≤-red) rule (in Definition 7.2.1.2):

   \[
   (\leq\text{-red}) \quad a \triangleright_{\beta} a' \quad b \triangleright_{\beta} b' \quad \frac{}{\Gamma \vdash a' \leq b'}
   \]

   As a side-effect, the (≤-conv) rule can be simplified to the (≤-refl) rule, which says that two syntactically equivalent terms are in the subtype relationship.

2. The (≤-trans) rule is necessary when the term \( a \) is a variable \( x \), and \( c \) is not convertible to \( a \), as follows:

   \[
   (\leq\text{-var}) \quad \frac{x \leq b : B \in \Gamma}{\Gamma \vdash x \leq b}
   \]

   \[
   (\leq\text{-trans}) \quad \frac{}{\Gamma \vdash x \leq b} \quad \frac{}{\Gamma \vdash b \leq c}
   \]

   \[
   \frac{}{\Gamma \vdash x \leq c}
   \]
This use of transitivity is taken over by the \((\leq\text{-transvar})\) rule (in Definition 7.2.1.2):

\[
\Gamma \vdash b \ d_1 \ d_2 \ \ldots \ d_n \leq c \\
\Gamma \vdots x \ d_1 \ d_2 \ \ldots \ d_n \leq c \\
x \leq b \ : \ B \in \Gamma, \ n \geq 0
\]

The \((\leq\text{-transvar})\) rule is more general; instead of just \(x\) we allow \(x\) to have a number of arguments. The rule is more general to cater for the absence of the \((\leq\text{-app})\) rule.

In all other cases, the \((\leq\text{-trans})\) rule is not essential, because it can be "pushed" upwards through the derivation, ending only in situations 1 or 2. This property, called transitivity elimination [Com95, Che97], is in fact formally proved in Lemma 7.2.3.6, and is the most difficult part of proving equivalence between the original and the restricted rules. The rule \((\leq\text{-transvar})\) is the usual solution to cater for situation 2. However, \((\leq\text{-red})\) occurs only in [Che97] and is not as important there as it is here. All other subtyping systems from the literature have some rule roughly like \((\leq\text{-red})\) to provide for reductions (or conversions) in the set of restricted rules.

**The \((\leq\text{-app})\) rule**

Another source of intractability is the \((\leq\text{-app})\) rule:

\[
\Gamma \vdash b \leq b' \\
\Gamma \vdash b \ a \leq b' \ a
\]

It is not apparent that this rule gives problems, but consider the case when \(b\) is an abstraction: instead of using \((\leq\text{-app})\), we could also reduce \(b\ \ a\) using the \((\leq\text{-trans})\) and \((\leq\text{-conv})\) rules (in the same way as above), and proceed from there. For example take the \(PTS \leq \lambda \omega \leq\) and 
\(\Gamma \equiv \Gamma \vdash \ast, \ N \leq \Gamma \vdash \ast, \ B \vdash \ast, \) and consider the judgment \(\Gamma \vdash (\lambda x : \ast. \ N) \ B \leq (\lambda x : \ast. \ I) \ B.\) One way to derive this is:

\[
\begin{align*}
(\leq\text{-conv}) \quad & (\lambda x : \ast. \ N) \ B =_\beta N \\
(\leq\text{-trans}) \quad & \Gamma \vdash \ N \leq I \\
(\leq\text{-conv}) \quad & I =_\beta (\lambda x : \ast. \ I) \ B \\
(\leq\text{-trans}) \quad & \Gamma \vdash \ N \leq (\lambda x : \ast. \ I) \ B \\
& \Gamma \vdash (\lambda x : \ast. \ N) \ B \leq (\lambda x : \ast. \ I) \ B
\end{align*}
\]

and the second way is

\[
\begin{align*}
\vdots
(\leq\text{-app}) \quad & \Gamma \vdash (\lambda x : \ast. \ N) \leq (\lambda x : \ast. \ I) \\
& \Gamma \vdash (\lambda x : \ast. \ N) \ B \leq (\lambda x : \ast. \ I) \ B
\end{align*}
\]

If a judgment of this form holds, it can always be derived using the first way. (And not always using the second way, e.g. consider \(\Gamma \vdash (\lambda x : \ast. \ X) \ B \leq (\lambda x : \ast. \ B) \ B.\) So for this kind of judgments, we do not need the \((\leq\text{-app})\) rule, and we would like to remove it, to make the rules more syntax-directed.

However, it is essential in two situations.

1. The term \(b\) (in judgment \(b \ a \leq b' \ a\)) is a variable. This is catered for by the \((\leq\text{-transvar})\) rule given in Definition 7.2.1.2, where \((\leq\text{-app})\) is combined with \((\leq\text{-trans})\) and \((\leq\text{-var})\).
2. The term $b$ is a (bounded or unbounded) II-type. For example take
\[ \Gamma \equiv B \cdot *, \text{Int} \cdot *, \text{Nat} \leq \text{Int} \cdot * \]
and consider the following derivation
\[
\frac\text{\text{(\leq-app)}} \quad \frac{\Gamma \vdash (\Pi x : B. \text{Nat}) \leq (\Pi x : B. \text{Int})}{\Gamma \vdash (\Pi x : B. \text{Nat}) B \leq (\Pi x : B. \text{Int}) B}
\]
The reader might reject this situation by saying that $b a$ is never typable if $b$ is a II-type. This is true, however subtyping is defined on pseudoterms, rather than (typable) terms, so we cannot ignore this situation here. This is a consequence of our major design decision that the subtyping rules do not depend on typing judgments.

This situation is catered for by the (\leq-IIapp) rule below. In the end, when we have shown Subject Reduction, we'll see we do not need (\leq-IIapp) after all (Lemma 7.2.4.13). This seems to be a contradiction with the statement that (\leq-app) is essential in this situation. But it is not a contradiction: for pseudoterms the (\leq-IIapp) rule is essential, and for typable terms it is redundant. In other words, the rule is needed only as a catalyst, in order to prove the meta-theory for subtyping as smooth as possible (see for example Remark 7.2.3.9). This rule (\leq-IIapp) has not appeared before in the literature.

In other situations, the (\leq-app) rule is not essential. When $b$ (in judgment $b a \leq b'$) is an abstraction, we use the (\leq-trans) and (\leq-conv) rules, as indicated above. When $b$ is a sort, $b'$ must be the same sort, so we use the (\leq-conv) rule, and when $b$ is another application, we apply our argument by induction, this results in the list of arguments in rules (\leq-transvar) and (\leq-IIapp). This is proved in Lemma 7.2.3.4.

**Definition 7.2.1.1** A term $a$ is a II-type if $a \equiv \Pi x : B. C$ or $a \equiv \Pi x \leq b : B. C$ for some $b$, $B$ and $C$. \hfill \Box

**Definition 7.2.1.2** (Restricted subtyping rules)

\[
\begin{align*}
(\leq-refl) & \quad \frac{} {\Gamma \vdash b \leq b} \\
(\leq-red) & \quad \frac{a \triangleright_{\theta} a' \quad b \triangleright_{\theta} b'} {\Gamma \vdash a \leq b} \\
(\leq-transvar) & \quad \frac{\Gamma \vdash a c_1 \ldots c_n \leq b} {\Gamma \vdash x c_1 \ldots c_n \leq b} \quad x \leq a : A \in \Gamma, n \geq 0 \\
(\leq-IIapp) & \quad \frac{\Gamma \vdash a \leq b} {\Gamma \vdash a c_1 \ldots c_n \leq b \leq a c_1 \ldots c_n} \quad a \text{ is a II-type, } b \text{ is a II-type, } n \geq 1 \\
(\leq-II) & \quad \frac{\Gamma \vdash A' \leq A \quad \Gamma, x : A' \vdash B \leq B'} {\Gamma \vdash (\Pi x : A. B) \leq (\Pi x : A'. B')} \\
(\leq-BII) & \quad \frac{\Gamma, x \leq a : A \vdash B \leq B'} {\Gamma \vdash (\Pi x \leq a : A. B) \leq (\Pi x \leq a : A. B')} \\
(\leq-\lambda) & \quad \frac{\Gamma, x : A \vdash b \leq b'} {\Gamma \vdash (\lambda x : A. b) \leq (\lambda x : A. b')} \hfill \Box
\end{align*}
\]
Note that each individual rule, except (≤-red), is syntax-directed. The rule (≤-red) is not syntax-directed and can always be applied as last step in a derivation, but is not too harmful, since the reduction conditions severely limit the terms possible in the subtyping premise.

Convention: From this point onwards, we will always use the subtyping rules of Definition 7.2.1.2. We will refer to the original, liberal rules (Definition 7.1.2.4) using $\Gamma \vdash_1 a \leq b$. Note that all typing rules are defined with the liberal subtyping rules.

We have designed the new (restricted) subtyping rules to be equivalent to the liberal ones. So we want to prove:

\[
\begin{align*}
\text{soundness:} & \quad \Gamma \vdash a \leq b \implies \Gamma \vdash_1 a \leq b \\
\text{completeness:} & \quad \Gamma \vdash_1 a \leq b \implies \Gamma \vdash a \leq b
\end{align*}
\]

It is easy to prove soundness, since the new rules are a restriction of the liberal rules. Completeness is proved below.

The following two sections develop the meta-theory for subtyping. First, we treat some preliminaries and the generation properties for subtyping. These properties say how the syntactical form of $a$ and $b$ are related if $\Gamma \vdash a \leq b$. Second, we show equivalence of the original subtyping rules and the restricted rules. We do this by proving admissibility of the liberal rules, i.e. that the addition of a liberal rule does not change the set of derivable judgments. Thus we prove completeness. Simultaneously we prove the substitution property.

### 7.2.2 Generation Properties for Subtyping

Before we can prove the generation properties, we have to treat some preliminaries. These preliminaries concern the usual replacement properties, which allow a sort of $\alpha$-conversion for derivations and judgments. The replacement properties are illustrated by the proof of a simple lemma. Readers more interested in the meta-theory specific for $PTS^S$s than the usual hassle with the names of variables can proceed to the paragraph "Generation" below.

**Preliminaries**

We need to prove the replacement property both for pseudocontexts, because of our particular definition of pseudocontext, and for the subtyping judgment. The replacement property for typing will appear later.

**Lemma 7.2.2.1 (Replacement for pseudocontexts)**

- If $\Gamma, x : A, \Gamma'$ is a pseudocontext and $y \notin \text{FV}(\Gamma, x : A, \Gamma')$ then $\Gamma, y : A, \Gamma'[x := y]$ is also a pseudocontext.

- If $\Gamma, x \leq a : A, \Gamma'$ is a pseudocontext and $y \notin \text{FV}(\Gamma, x \leq a : A, \Gamma')$ then $\Gamma, y \leq a : A, \Gamma'[x := y]$ is also a pseudocontext.

**Proof:** By induction on $\Gamma'$.

**Convention:** We use the letter $\Upsilon$ as meta-variable for derivation trees, because it resembles a willow.
Lemma 7.2.2.2 (Replacement for subtyping)

- If \( \Gamma \) derives \( \Gamma, x : A, \Gamma' \vdash B \leq B' \) and \( y \notin \text{FV}(\Gamma, x : A, \Gamma') \cup \text{FV}(B) \cup \text{FV}(B') \) then there is an \( \Gamma' \) deriving \( \Gamma, y : A, \Gamma'[x := y] \vdash B[x := y] \leq B'[x := y] \) where \( \Gamma' \) has the same structure as \( \Gamma \).

- If \( \Gamma \) derives \( \Gamma, x \leq a : A, \Gamma' \vdash B \leq B' \) and \( y \notin \text{FV}(\Gamma, x \leq a : A, \Gamma') \cup \text{FV}(B) \cup \text{FV}(B') \) then there is an \( \Gamma' \) deriving \( \Gamma, y \leq a : A, \Gamma'[x := y] \vdash B[x := y] \leq B'[x := y] \) where \( \Gamma' \) has the same structure as \( \Gamma \).

The structure of a derivation is the labelled tree that is found by removing from the derivation everything but the names of the applied rules (at every node).

Proof: By simultaneous induction to the structure of \( \Gamma \). We need Replacement for pseudo-contexts (Lemma 7.2.2.1) for the \((\leq\text{-refl})\) cases. \(\Box\)

These lemmas say that the names of the declared variables in the context do not matter (i.e. we have \(\alpha\)-conversion for judgments), and we may assume declared variables to be different from any of the bound variables. This is called the Barendregt convention [Bar84]. For example, in a statement

\[
\Gamma \vdash (\Pi x : A_1. B_1) \leq (\Pi x : A_2. B_2) \implies \Gamma, x : A_2 \vdash B_1 \leq B_2
\]

we assume \( x \notin \text{FV}(\Gamma) \); without this assumption this statement may be false, because \( \Gamma, x : A_2 \) may not be a pseudocontext.

The proof of the following lemma illustrates the use of Replacement. In the rest of this chapter we will use replacement only implicitly. The lemma says that extending a context does not destroy derivable judgments.

Lemma 7.2.2.3 If \( \Gamma \subseteq \Gamma' \) (meaning \( \Gamma' \) has all declarations of \( \Gamma \), and in the same order) and \( \Gamma \vdash a \leq b \) then \( \Gamma' \vdash a \leq b \).

Proof: By straightforward induction on the derivation. We treat case \((\leq\text{-}\lambda)\) to illustrate the use of Replacement.

Suppose the last step in the derivation was

\[
(\leq\lambda) \quad \Gamma, x : C \vdash d \leq d' \quad \Gamma \vdash (\lambda x : C. d) \leq (\lambda x : C. d')
\]

Take some \( y \) with the following properties.

1. \( y \notin \text{FV}(\Gamma') \).
2. \( y \notin \text{FV}(\Gamma, x : C) \).
3. \( y \notin \text{FV}(d) \cup \text{FV}(d') \).

By properties 2 and 3, we may use Replacement on the premise to obtain a derivation with the same structure for

\[
\Gamma, y : C \vdash d[x := y] \leq d'[x := y]
\]

Since this derivation has the same structure, we can still use the IH. In order to do this, we need to show \((\Gamma, y : C) \subseteq (\Gamma', y : C)\) (which follows trivially from \( \Gamma \subseteq \Gamma' \) and that \( \Gamma', y : C \)
is a pseudocontext. This follows from $\Gamma'$ being a pseudocontext (by convention), $y \not\in \text{FV}(\Gamma')$ (property 1), and $y \not\in \text{FV}(C)$, which follows from property 2. So we have

$$\Gamma', y : C \vdash d[x := y] \leq d'[x := y],$$

from which we can derive by $(\leq \lambda)$ that

$$\Gamma' \vdash (\lambda y : C. d[x := y]) \leq (\lambda y : C. d'[x := y]).$$

Since we identify terms up to $\alpha$-conversion, we can rename the bound $y$ to $x$ by $\alpha$-conversion (this is correct because of property 3). This yields

$$\Gamma' \vdash (\lambda x : C. d) \leq (\lambda x : C. d'),$$

which completes this completely worked out case.

In the proof of other lemmas we will not explicitly perform a replacement, but we will implicitly assume that the bound variable ($x$ in this case) can always be chosen in such a way that it does not belong to some given finite set of variables ($\text{FV}(\Gamma')$ in this case). 

\hfill \Box

### Generation

Before giving the generation properties, we show that subtyping is closed under $\beta$-conversion, i.e. $\beta$-converting a term in a subtyping judgment keeps it derivable (Theorem 7.2.2.6). Even the number of interesting steps — i.e. not $(\leq \text{-red})$ — in the derivation for this judgment stays the same. To make this formal, we first have the following definitions.

**Definition 7.2.2.4** The NR-height of a subtyping derivation $\Upsilon$, written as $\text{NR-height}(\Upsilon)$, is the height of $\Upsilon$, not counting applications of the $(\leq \text{-red})$ rule. NR stands for "Not counting Reductions". We write $\Upsilon \prec \Upsilon'$ as shorthand for $\text{NR-height}(\Upsilon) < \text{NR-height}(\Upsilon')$, and similarly for $\leq$.

**Definition 7.2.2.5** $\triangleright_\beta$ is extended to contexts as follows:

- if $A \triangleright_\beta A'$ then $(\Gamma, x : A, \Gamma') \triangleright_\beta (\Gamma, x : A', \Gamma')$,
- if $A \triangleright_\beta A'$ then $(\Gamma, x \leq a : A, \Gamma') \triangleright_\beta (\Gamma, x \leq a : A', \Gamma')$,
- if $a \triangleright_\beta a'$ then $(\Gamma, x \leq a : A, \Gamma') \triangleright_\beta (\Gamma, x \leq a' : A, \Gamma')$.

$\triangleright_\beta$ and $=_\beta$ are extended accordingly.

Now we can prove the promised property.

**Theorem 7.2.2.6** ($\leq \text{-Conversion-closed}$) Suppose $\Gamma =_\beta \Gamma'$ and $a =_\beta a'$ and $b =_\beta b'$. If $\Upsilon$ derives $\Gamma \vdash a \leq b$ then there is a $\Upsilon' \leq \Upsilon$ such that $\Upsilon'$ derives $\Gamma' \vdash a' \leq b'$.

**Proof:** By induction on $\Upsilon$. We use Church-Rosser and the rule $(\leq \text{-red})$ extensively. We treat the $(\leq \text{-II})$ case in detail.

Assume that the last step in $\Upsilon$ is:

$$\frac{(\leq \text{-II}) \quad \Gamma \vdash C \leq A \quad \Gamma, x : C \vdash B \leq D}{\Gamma \vdash (\Pi x : A. B) \leq (\Pi x : C. D)}$$
7.2. META-THEORY

So \( a \equiv \Pi x : A. B \) and \( b \equiv \Pi x : C. D \), and both derivations for the premises have NR-height < \( \Upsilon \). By Church-Rosser on \( a =_{\beta} a' \) and \( b =_{\beta} b' \), and elementary properties of \( \beta \)-reduction, there exist \( A', B', C' \) and \( D' \) such that

- \( a' \mathrel{\beta} \Pi x : A'. B' \) and \( A \mathrel{\beta} A', B \mathrel{\beta} B' \), and
- \( b' \mathrel{\beta} \Pi x : C'. D' \) and \( C \mathrel{\beta} C', D \mathrel{\beta} D' \).

By applying the IH on the derivation for \( \Gamma \vdash C \leq A \), we obtain a derivation of \( \Gamma' \vdash C' \leq A' \), with NR-height < \( \Upsilon \). Similarly, we apply the IH on the derivation for \( \Gamma, x : C \vdash B \leq D \), giving a derivation of \( \Gamma', x : C' \vdash B' \leq D' \), with also NR-height < \( \Upsilon \). By combining these derivations with the \( (\leq \Pi) \) rule we obtain a derivation of \( \Gamma' \vdash (\Pi x : A'. B') \leq (\Pi x : C'. D') \) with NR-height \( \leq \Upsilon \). Finally, we apply the \( (\leq \text{-red}) \) rule, which gives the derivation \( \Upsilon' \) for \( \Gamma' \vdash a' \leq b' \), also with NR-height \( \leq \Upsilon \).

This property is very important, since it allows us to convert terms in a subtyping judgment without increasing the NR-height. This makes the NR-height a very useful induction measure. An example of use of this lemma is in the proof of the admissibility of transitivity (Lemma 7.2.3.6).

Now we come to our generation properties. We have them in three flavours:

**Pure**: What are the forms of \( a \) and \( b \) if \( \Gamma \vdash a \leq b \)?

**Left**: What is the form of \( b \) given the form of \( a \) and \( \Gamma \vdash a \leq b \)?

**Left-right**: How are the subterms of \( a \) and \( b \) related, given the form of \( a \) and \( b \), and \( \Gamma \vdash a \leq b \)?

We will use the next lemma — the pure flavour — very often in induction proofs with NR-height as measure, since subtyping derivations in the conclusion have a smaller NR-height than the subtyping derivation in the assumption.

**Lemma 7.2.2.7 (\( \leq \)-Generation)**

If \( \Upsilon \) derives \( \Gamma \vdash a \leq b \) then at least one of the following clauses holds:

1. \( a =_{\beta} b \);

2. \( a \mathrel{\beta} x \ c_1 \ldots c_n \) and \( x \leq c : C \in \Gamma \) and \( \Upsilon' \) derives \( \Gamma \vdash c_1 \ldots c_n \leq b \) for some \( c, C, c_1 \ldots c_n \) and \( \Upsilon' < \Upsilon \);

3. \( a \mathrel{\beta} a' c_1 \ldots c_n \) and \( b \mathrel{\beta} b' c_1 \ldots c_n \) and \( n \geq 1 \) and \( \Upsilon' \) derives \( \Gamma \vdash a' \leq b' \) for some \( \Pi \)-types \( a' \) and \( b' \) and some derivation tree \( \Upsilon' < \Upsilon \);

4. \( a \mathrel{\beta} \Pi x : C_1. D_1 \) and \( b \mathrel{\beta} \Pi x : C_2. D_2 \) and \( \Upsilon' \) derives \( \Gamma \vdash C_2 \leq C_1 \) and \( \Upsilon'' \) derives \( \Gamma, x : C_2 \vdash D_1 \leq D_2 \) for some \( C_1, C_2, D_1, D_2, \Upsilon' < \Upsilon \) and \( \Upsilon'' < \Upsilon \);

5. \( a \mathrel{\beta} \Pi x \leq c : C. D_1 \) and \( b \mathrel{\beta} \Pi x \leq c : C. D_2 \) and \( \Upsilon \) derives \( \Gamma, x \leq c : C \vdash D_1 \leq D_2 \) for some \( c, C, D_1, D_2 \) and \( \Upsilon' < \Upsilon \);

6. \( a \mathrel{\beta} \lambda x : C. D_1 \) and \( b \mathrel{\beta} \lambda x : C. D_2 \) and \( \Upsilon \) derives \( \Gamma, x : C \vdash D_1 \leq D_2 \) for some \( C, D_1, D_2 \) and \( \Upsilon' < \Upsilon \).
So in clause 2–6 each subtyping judgment in the conclusion has a derivation with a smaller
NR-height than \( \Gamma \).

Note that clauses 2 through 6 exclude each other.

Proof: Consider the last step in the derivation of \( \Gamma \vdash a \leq b \) other than applications of the
(\(<\)-red) rule. The rule that is used in this last step implies one of the clauses.

Lemma 7.2.2.8 (<-Generation-left)

- If \( \Gamma \vdash x \ c_1 \ldots c_n \leq b \) then \( b =_\beta x \ c_1 \ldots c_n \) or \( \Gamma \vdash d \ c_1 \ldots c_n \leq b \) with \( x \leq d : D \in \Gamma \) for
some \( d \) and \( D \).

- If \( \Gamma \vdash a_1 c_1 \ldots c_n \leq b \) and \( a_1 \) is a II-type then \( b =_\beta a_2 c_1 \ldots c_n \) and \( \Gamma \vdash a_1 \leq a_2 \) for some
II-type \( a_2 \).

- If \( \Gamma \vdash (\Pi x : A_1. \ B_1) \leq b \) then \( b =_\beta \Pi x : A_2. \ B_2 \) and \( \Gamma \vdash A_2 \leq A_1 \) and \( \Gamma, x : A_2 \vdash B_1 \leq B_2 \)
for some \( A_2 \) and \( B_2 \).

- If \( \Gamma \vdash (\Pi x \leq a : A. \ B_1) \leq b \) then \( b =_\beta \Pi x \leq a : A. \ B_2 \) and \( \Gamma, x \leq a : A \vdash B_1 \leq B_2 \) for
some \( B_2 \).

- If \( \Gamma \vdash (\lambda x : A. \ B_1) \leq b \) then \( b =_\beta (\lambda x : A. \ B_2) \) and \( \Gamma, x : A \vdash B_1 \leq B_2 \) for some \( B_2 \).

Furthermore, for each clause, if \( \Upsilon \) derives the assumption, then each subtyping judgment
in the conclusion has a derivation \( \Upsilon' \) with \( \Upsilon' \leq \Upsilon \).

Proof: Straightforward using \(<\>-Generation (Lemma 7.2.2.7) and \(<\>-Conversion-closed (Theo-
rem 7.2.2.6). Note that in contrast to the previous lemma, we cannot prove here that \( \Upsilon' \leq \Upsilon \),
because \( \Upsilon \) might consist of just an application of \(<\>-red), so \( \Upsilon' \) cannot possibly be smaller.

The last generation property is necessary only when both sides are II-types.

Lemma 7.2.2.9 (<-Generation-left-right)

1. If \( \Gamma \vdash (\Pi x : A_1. \ B_1) \leq (\Pi x : A_2. \ B_2) \) then \( \Gamma \vdash A_2 \leq A_1 \) and \( \Gamma, x : A_2 \vdash B_1 \leq B_2 \).

2. If \( \Gamma \vdash (\Pi x \leq a_1 : A_1. \ B_1) \leq (\Pi x \leq a_2 : A_2. \ B_2) \) then
\( a_1 =_\beta a_2 \) and \( A_1 =_\beta A_2 \) and \( \Gamma, x \leq a_1 : A_1 \vdash B_1 \leq B_2 \).

Proof: Straightforward using \(<\>-Generation-left (Lemma 7.2.2.8) and \(<\>-Conversion-closed
(Theorem 7.2.2.6).

7.2.3 Equivalence of Liberal and Restricted Subtyping Rules

Showing equivalence means showing soundness and completeness of the restricted (alternative)
rules with respect to the liberal (original) ones. Soundness is easy. We prove completeness
by showing that each liberal rule is admissible. A rules is admissible if it can be added to
the subtyping rules without changing the set of derivable judgments. We also show that the
substitution properties hold for subtyping judgments.

We start with the admissibility of (\(<\>-conv) and (\(<\>-var).
Lemma 7.2.3.1 (Conv-admissible) If \( a =_\beta b \) then \( \Gamma \vdash a \leq b \).

**Proof:** By Church-Rosser, there is a \( c \) such that \( a \triangleright_{\beta} c \) and \( b \triangleright_{\beta} c \). Use \((\leq\text{-refl})\) on \( c \), and \((\leq\text{-red})\) to obtain \( \Gamma \vdash a \leq b \). \( \square \)

Lemma 7.2.3.2 (Var-admissible) If \( x \leq a : A \in \Gamma \) then \( \Gamma \vdash x \leq a \).

**Proof:** Directly by \((\leq\text{-transvar})\) and \((\leq\text{-refl})\). \( \square \)

The first Substitution property for subtyping allows us to replace an unbounded variable \( y \) with any term \( c \). Note that \( c \) does not have to be typable, since we prove meta-theory for all pseudoterms. (This is a consequence of our design decision that the subtyping rules do not depend on the typing rules.)

Lemma 7.2.3.3 \((\leq\text{-Substitution})\) If \( \Gamma, y : C, \Gamma' \vdash a \leq b \) and \( \Gamma, \Gamma'[y := c] \) is a pseudocontext then \( \Gamma, \Gamma'[y := c] \vdash a[y := c] \leq b[y := c] \).

**Proof:** By induction on the derivation. Note that the substitution does not interact with the subtyping judgments, since only a variable without bound is substituted for. \( \square \)

Note that this lemma relies on our notion of pseudocontext. We clarify this by a counterexample: Take \( \Gamma \equiv x \leq y : * \) and suppose \( \Gamma, y : * \) were a pseudocontext. Then we could derive \( \Gamma, y : * \vdash x \leq y \), and hence the lemma says we can derive \( \Gamma \vdash x \leq z \) (take \( \Gamma' \equiv \varepsilon \) and \( c \equiv z \)). But this is not so. Furthermore, the demand that \( \Gamma, \Gamma'[y := c] \) is a pseudocontext is really necessary. In other systems this follows simply from \( c \) being typable, but we cannot demand this, as explained above.

The other part of the Substitution property — replacing a bounded variable — is proved at the end of this section.

For showing admissibility of the original \((\leq\text{-app})\) rule, we use the \((\leq\text{-Substitution})\) property just given.

Lemma 7.2.3.4 (App-admissible) If \( \Gamma \vdash a \leq b \) then \( \Gamma \vdash a \ c \leq b \ c \).

**Proof:** By induction on the NR-height of the derivation. By \((\leq\text{-Generation})\) on \( \Gamma \vdash a \leq b \), we have one of the following cases:

1. \( a =_\beta b \). Then \( a \ c =_\beta b \ c \), so by Conv-admissible (Lemma 7.2.3.1) \( \Gamma \vdash a \ c \leq b \ c \).
2. \( a \triangleright_{\beta} x \ c_1 \ldots c_n \) and \( x \leq d : D \in \Gamma \) and \( \Gamma \vdash d \ c_1 \ldots c_n \ c \leq b \ c \). By the induction hypothesis we get \( \Gamma \vdash d \ c_1 \ldots c_n \ c \leq b \ c \), by \((\leq\text{-transvar})\) \( \Gamma \vdash x \ c_1 \ldots c_n \ c \leq b \ c \), and by \((\leq\text{-red})\) \( \Gamma \vdash a \ c \leq b \ c \).

3.4.5 Straightforward, using the \((\leq\text{-Iapp})\) and \((\leq\text{-red})\) rules. We do not need the III.

6. \( a \triangleright_{\beta} \lambda x : C. \ d_1 \) and \( b \triangleright_{\beta} \lambda x : C. \ d_2 \) and \( \Gamma, x : C \vdash d_1 \leq d_2 \). By \((\leq\text{-Substitution})\) (Lemma 7.2.3.3) we have \( \Gamma \vdash d_1[x := c] \leq d_2[x := c] \). Since \( a \ c \triangleright_{\beta} d_1[x := c] \) and \( b \ c \triangleright_{\beta} d_2[x := c] \) we can use the \((\leq\text{-red})\) rule to obtain \( \Gamma \vdash a \ c \leq b \ c \). \( \square \)

Note that this property depends essentially on the \((\leq\text{-Iapp})\) rule; for an example see the discussion about the \((\leq\text{-app})\) rule in Section 7.2.1.

Now we turn towards admissibility of the \((\leq\text{-trans})\) rule. First we need an auxiliary result.
Lemma 7.2.3.5 If \( \Gamma \) derives \( \Gamma, x : A, \Gamma' \vdash a \leq b \) and \( \Gamma, x : A', \Gamma'' \vdash a \leq b \), then there is a derivation \( \Gamma' \) for the judgment \( \Gamma, x : A', \Gamma'' \vdash a \leq b \), with the same NR-height as \( \Gamma \).

Proof: By easy induction on the derivation.

The admissibility of transitivity is also known as transitivity elimination, because it shows transitivity can be eliminated from the original rules. It is the most difficult part of proving equivalence, and therefore its proof is treated in some length.

Lemma 7.2.3.6 (Trans-admissible) If \( \Gamma \vdash a \leq b \) and \( \Gamma \vdash b \leq c \) then \( \Gamma \vdash a \leq c \).

We first give a sketch of the proof, which is instructive, because it does not use any lemmas and shows the importance of \( \leq \)-Conversion-closed (Theorem 7.2.2.6). Then we give a quite formal proof, which is more concise because it uses more lemmas.

Proof sketch: The idea is to make case distinction to the last rule different from \( (\leq\text{-red}) \) in both the derivations of \( \Gamma \vdash a \leq b \) and the derivation of \( \Gamma \vdash b \leq c \). For both derivations we assume \( (\leq\text{-red}) \) has been used exactly once after an application of another rule. This gives the following situation, where \( r_1 \) and \( r_2 \) are not \( (\leq\text{-red}) \).

\[
\begin{array}{c}
\vdash a \Rightarrow_{\beta} a' \quad (r_1) \quad \Gamma \vdash a' \leq b' \quad b \Rightarrow_{\beta} b' \quad b \Rightarrow_{\beta} b'' \quad (r_2) \quad \Gamma \vdash b'' \leq c'' \quad c \Rightarrow_{\beta} c''
\end{array}
\]

We first perform case distinction to rule \( r_1 \). For example, we assume \( r_1 \) is \( (\leq\lambda) \), so \( a' \equiv \lambda x : A. d \) and \( b' \equiv \lambda x : A. e \) and \( \Gamma, x : A \vdash d \leq e \) for some \( A, d \) and \( e \). Since \( b' =_{\beta} b'' \) and \( \Gamma \vdash b'' \leq c'' \) is derived with rule \( r_2 \), \( r_2 \) must also be \( (\leq\lambda) \). (E.g. if \( r_2 \) would be \( (\leq\Pi) \), then \( b'' \) would be a \( \Pi \)-type, but this is impossible since \( b'' \) is convertible to \( b' \), which is an abstraction.) Since rule \( r_2 \) is \( (\leq\lambda) \), \( b'' \) and \( c'' \) must be abstractions, say \( b'' \equiv \lambda x : A'. e' \) and \( c'' \equiv \lambda x : A'. f' \), and \( \Gamma, x : A' \vdash e' \leq f' \). Now we would like to apply the IH to \( \Gamma, x : A \vdash d \leq e \) and \( \Gamma, x : A' \vdash e' \leq f' \).

However, we do not know whether \( A \equiv A' \) and \( e \equiv e' \). Here \( \leq \)-Conversion-closed comes at hand. We know \( A =_{\beta} A' \) and \( e =_{\beta} e' \) (since \( b' =_{\beta} b'' \) and \( b' \equiv \lambda x : A. e \) and \( b'' \equiv \lambda x : A'. e' \)). Therefore we can use \( \leq \)-Conversion-closed to infer from \( \Gamma, x : A' \vdash e' \leq f' \) that \( \Gamma, x : A \vdash e \leq f' \). Now we can use the IH, and infer \( \Gamma, x : A \vdash d \leq f' \), which gives by \( (\leq\lambda) \) \( \Gamma \vdash (\lambda x : A. d) \leq (\lambda x : A'. f') \). Since \( a =_{\beta} \lambda x : A. d \) and \( e =_{\beta} \lambda x : A. f' \), we can use \( \leq \)-Conversion-closed again to derive \( \Gamma \vdash a \leq c \), q.e.d.

We have to use the IH after we converted terms in a subtyping judgment with \( \leq \)-Conversion-closed. Application of this property can increase the height of the derivation, so that this height is not a suitable induction measure. However, the NR-height is not increased, so we perform induction to the NR-height of \( \Gamma \vdash b \leq c \) in order for our induction to be well-founded.

Actually, we have to make the induction measure symmetric in both derivations, because of the \( (\leq\Pi) \) rule (see the formal proof below), so the induction measure is the sum of the NR-height of the derivation of \( \Gamma \vdash a \leq b \) and the NR-height of the derivation of \( \Gamma \vdash b \leq c \).

Proof: By induction on the sum of the NR-heights of the derivations, and by \( \leq \)-Generation (Lemma 7.2.2.7) on \( \Gamma \vdash a \leq b \) (this corresponds to the case distinction to rule \( r_1 \) above). We treat only the most complicated case, namely the case corresponding to the \( (\leq\Pi) \) rule.

Let \( m \) be the NR-height of the derivation for \( \Gamma \vdash a \leq b \), and \( n \) be the NR-height of the derivation for \( \Gamma \vdash b \leq c \). We have
7.2. META-THEORY

(1) \ a \triangleright_\beta \Pi z: A. B

(2) \ b \triangleright_\beta \Pi z: A'. B'

(3) \ \Gamma \vdash A' \leq A \text{ (NR-height < m)}

(4) \ \Gamma, x: A' \vdash B \leq B' \text{ (NR-height < m)}

By \leq\text{-}Conversion\text{-}closed (Theorem 7.2.2.6) on 2 and \Gamma \vdash b \leq c we have

(5) \ \Gamma \vdash (\Pi z: A'. B') \leq c \text{ (NR-height \leq n)}

By \leq\text{-}Generation\text{-}left (Lemma 7.2.2.8) on 5 we have

(6) \ c =_\beta \Pi z: A''. B''

(7) \ \Gamma \vdash A'' \leq A' \text{ (NR-height \leq n)}

(8) \ \Gamma, x: A'' \vdash B' \leq B'' \text{ (NR-height \leq n)}

By IH on 7 and 3 (sum of NR-heights < n + m) we have

(9) \ \Gamma \vdash A'' \leq A

By Lemma 7.2.3.5 on 4 (we know by 8 that \Gamma, x: A'' is a pseudocontext) we have

(10) \ \Gamma, x: A'' \vdash B \leq B' \text{ (NR-height < m)}

By IH on 10 and 8 (sum of NR-heights < m + n) we obtain

(11) \ \Gamma, x: A'' \vdash B''

Now rule (\leq\text{-}\Pi) on 9 and 11 gives

(12) \ \Gamma \vdash (\Pi z: A. B) \leq (\Pi z: A''. B'')

By \leq\text{-}Conversion\text{-}closed (Theorem 7.2.2.6) on 1, 6 and 12

(13) \ \Gamma \vdash a \leq c

This concludes this case. \ \Box


Now all liberal rules (Definition 7.1.2.4) have been shown to be admissible, so every subtyping judgment derivable with the liberal rules is also derivable with the restricted rules: the restricted rules are complete with respect to the liberal rules. So the liberal rules and the restricted rules are equivalent.

Theorem 7.2.3.7 (Equivalence) \ \Gamma \vdash_1 a \leq b \iff \Gamma \vdash a \leq b

Proof: Soundness (\iff) is easy to prove. Completeness (\implies) follows from the admissibility Lemmas 7.2.3.1, 7.2.3.2, 7.2.3.4 and 7.2.3.6. \ \Box

Soundness only holds because the liberal rules do not have typing judgments as premise. The equivalence allows replacing each premise of the form \Gamma \vdash a \leq b in typing rules by the premise \Gamma \vdash a \leq b without changing the set of derivable typing judgments, so we can use properties like Lemma 7.2.2.9 when proving properties about the typing judgment.

Using the admissibility of (\leq\text{-}trans) and (\leq\text{-}app), we prove the other Substitution property.

Lemma 7.2.3.8 (\leq\text{-}Bounded\text{-}Substitution) If \Gamma \vdash c \leq c' and \Gamma, y \leq c': C, \Gamma' \vdash a \leq b and \Gamma', \Gamma''[y := c] is a pseudocontext then \Gamma, \Gamma''[y := c] \vdash a[y := c] \leq b[y := c].

Proof: By induction on the derivation of \Gamma, y \leq c': C, \Gamma' \vdash a \leq b. We treat case (\leq\text{-}transvar) below. Case (\leq\text{-}\Pi\text{app}) uses App\text{-}admissible (Lemma 7.2.3.4). The other cases are straightforward.
Let $A^*$ stand for $A[y := c]$ for any term or context $A$. The last step in the derivation was

$$(\leq\text{-transvar}) \quad \frac{x \leq d : D \in (\Gamma, y \leq c' : C, \Gamma') \quad \Gamma, y \leq c' : C, \Gamma' \vdash d \ c_1 \ c_2 \ldots \ c_n \leq b}{\Gamma, y \leq c' : C, \Gamma' \vdash x \ c_1 \ c_2 \ldots \ c_n \leq b}$$

We have three cases:

$x \leq d : D \in \Gamma$. Straightforward by IH.

$x \leq d : D \in \Gamma'$. Straightforward by IH.

$x \leq d : D \equiv y \leq c' : C$. Then $x \equiv y$ and $d \equiv c'$ and $D \equiv C$. By the definition of pseudocontext, $y \not\in \text{FV}(c')$. By the induction hypothesis $\Gamma, \Gamma'' \vdash (d \ c_1 \ldots \ c_n)^* \leq b^*$. This is the same as $\Gamma, \Gamma'' \vdash c' \ c_1^* \ldots \ c_n^* \leq b^*$.

By assumption, we have $\Gamma \vdash c \leq c'$, and so $\Gamma, \Gamma'' \vdash c \leq c'$. Now we use App-admissible (Lemma 7.2.3.4), so we get $\Gamma, \Gamma'' \vdash c \ c_1^* \ldots \ c_n^* \leq c' \ c_1^* \ldots \ c_n^*$.

Combining the two subtyping judgments with Trans-admissible (Lemma 7.2.3.6), we obtain $\Gamma, \Gamma'' \vdash (x \ c_1 \ldots \ c_n)^* \leq b^*$. This is the same as $\Gamma, \Gamma'' \vdash (x \ c_1 \ldots \ c_n)^* \leq b^*$.

The following two remarks point out certain subtleties concerning this lemma.

**Remark 7.2.3.9** Without $(\leq\text{-Iapp})$ rule this $(\leq$-Bounded-Substitution lemma would not hold. Take e.g.

$$\begin{align*}
\Gamma & \equiv B : *, \text{Int} : *, \text{Nat} : \text{Int} : * \\
c & \equiv \Pi x : B. \text{Nat} \\
c' & \equiv \Pi x : B. \text{Int} \\
C & \equiv * \\
\Gamma' & \equiv \epsilon \\
a & \equiv y B \\
b & \equiv (\Pi x : B. \text{Int}) B
\end{align*}$$

All the assumptions of the lemma hold (even without $(\leq\text{-Iapp})$ rule), but the conclusion of the lemma, in this case $\Gamma \vdash (\Pi x : B. \text{Nat}) B \leq (\Pi x : B. \text{Int}) B$, holds only because of the $(\leq\text{-Iapp})$ rule.

**Remark 7.2.3.10** Introduction of the $(\leq\text{-B}) \lambda$ rule of $\mathcal{F}_c^\leq$ [CG97] destroys this property. This is the definition of the rule:

$$(\leq\text{-B}) \quad \frac{\Gamma, x \leq a : A \vdash b \leq b'}{\Gamma \vdash (\lambda x : a : A. b) \leq (\lambda x : a : A. b')}$$

We have the following counterexample:

$$\begin{align*}
\Gamma & \equiv D : *, E : * \\
c & \equiv \lambda X : E : *. X \\
c' & \equiv \lambda X : E : *. E \\
C & \equiv \Pi X : E : *. * \\
\Gamma' & \equiv \epsilon \\
a & \equiv y D \\
b & \equiv E
\end{align*}$$
All the assumptions of the lemma hold, in particular we have \( \Gamma \vdash c \leq c' \) because of rule \((\leq \text{-} \text{B}\lambda)\) and \( \Gamma, y : \leq b' : C, \Gamma' \vdash a \leq b \) by \((\leq \text{-} \text{transvar})\). But the conclusion of the lemma, viz. \( \Gamma \vdash a[y := c] \leq b[y := c] \) is not derivable, which becomes clear after reduction of the terms, giving \( \Gamma \vdash D \leq E \).

In [CG97] the lemma holds for all typable terms. But our treatment of subtyping should cover all pseudoterms, so therefore we cannot introduce \((\leq \text{-} \text{B}\lambda)\). \(\square\)

The substitution properties for subtyping are essential to prove Subject Reduction, via the substitution properties for typing (Lemma 7.2.4.6 and 7.2.4.7).

One final remark about the subtyping relation as ordering. We think that it is antisymmetric with respect to \(\beta\)-conversion, i.e. if \( \Gamma \vdash a \leq b \) and \( \Gamma \vdash b \leq a \) then \( a =_\beta b \). We have not been able to prove this property, but this is not a problem, since we do not need it.

### 7.2.4 Subject Reduction

The proof of Subject Reduction goes along the same lines as in ordinary PTSs (see Theorem 2.1.2.6), and is longer but not more complicated. We have some simple auxiliary results (Lemmas 7.2.4.1 through 7.2.4.5), the substitution properties (Lemmas 7.2.4.6 and 7.2.4.7), the Generation and Correctness of Types lemmas, and then prove Subject Reduction. We also present an important consequence of the SR property, namely Lemma 7.2.4.13, which says that in each subtyping rule the terms in the premises are typable. if the two terms in the conclusion are typable.

In contrast to subtyping, we do not need alternative typing rules; the given typing rules behave well enough to develop the meta-theory.

**Lemma 7.2.4.1** If \( \Gamma \vdash a : A \) then there is a shorter derivation for \( \Gamma \vdash ok \).

**Proof:** By straightforward induction on the derivation. \(\square\)

**Lemma 7.2.4.2** If \( \Upsilon \) derives \( \Gamma, \Gamma' \vdash ok \) then then there is a derivation \( \Upsilon' \) for \( \Gamma \vdash ok \) with height at most the height of \( \Upsilon \).

**Proof:** By straightforward induction on the length of \( \Gamma' \). \(\square\)

**Lemma 7.2.4.3** If \( \Gamma \vdash a : A \) then \( \text{FV}(a) \cup \text{FV}(A) \subseteq \text{dom}(\Gamma) \).

**Proof:** By induction on the height of the derivation. \(\square\)

**Lemma 7.2.4.4** (Replacement for typing)

- Suppose \( y \not\in \text{FV}(\Gamma, x : A, \Gamma') \).
  - If \( \Upsilon \) derives \( \Gamma, x : A, \Gamma' \vdash b : B \) then there is an \( \Upsilon' \) deriving \( \Gamma, y : A, \Gamma'[x := y] \vdash b[x := y] : B[x := y] \) where \( \Upsilon' \) has the same structure as \( \Upsilon \).
  - If \( \Upsilon \) derives \( \Gamma, x : A, \Gamma' \vdash ok \) then there is an \( \Upsilon' \) deriving \( \Gamma, y : A, \Gamma'[x := y] \vdash ok \) where \( \Upsilon' \) has the same structure as \( \Upsilon \).
• Suppose \( y \not\in \text{FV}(\Gamma, x \leq a : A, \Gamma') \).
  
  - If \( \Upsilon \) derives \( \Gamma, x \leq a : A, \Gamma' \vdash b : B \) then there is an \( \Upsilon' \) deriving
    \[ \Gamma, y \leq a : A, \Gamma'[x := y] \vdash b[x := y] : B[x := y] \] where \( \Upsilon' \) has the same structure as \( \Upsilon \).
  
  - If \( \Upsilon \) derives \( \Gamma, x \leq a : A, \Gamma' \vdash ok \) then there is an \( \Upsilon' \) deriving
    \[ \Gamma, y \leq a : A, \Gamma'[x := y] \vdash ok \] where \( \Upsilon' \) has the same structure as \( \Upsilon \).

Proof: By simultaneous induction on \( \Upsilon \). We need Lemma 7.2.4.3 and of course Replacement for
subtyping (Lemma 7.2.2.2). \( \Box \)

Just as Replacement for subtyping, we will use Replacement for typing often implicitly.

Lemma 7.2.4.5 If \( \Gamma' \vdash ok \) and \( \Gamma \subseteq \Gamma' \) (meaning \( \Gamma' \) has all declarations of \( \Gamma \), and in the same
order) and \( \Gamma \vdash a : A \) then \( \Gamma' \vdash a : A \).

Proof: By induction on the length of the derivation of \( \Gamma \vdash a : A \). We do the case \((\Pi\text{-intro})\)
here. All other cases are similar or straightforward.

Assume \( \Gamma, x : A \vdash b : B \) and \( \Gamma \vdash (\Pi x : A. B) : s \). We will show \( \Gamma' \vdash (\lambda x : A. b) : (\Pi x : A. B) \).
We may assume \( x \not\in \text{FV}(\Gamma') \cup \text{FV}(A) \) (Replacement). By Lemma 7.2.4.1 we have \( \Gamma, x : A \vdash ok \).
This must be derived with \((C\text{-var})\), so \( \Gamma' \vdash A : s \), so by IH \( \Gamma' \vdash A : s \) and since \( \Gamma', x : A \) is
a pseudocontext we have \( \Gamma', x : A \vdash ok \). By using the IH on \( \Gamma, x : A \vdash b : B \) we now have
\( \Gamma', x : A \vdash b : B \). Directly by IH, we have \( \Gamma' \vdash (\Pi x : A. B) : s \). Using the \((\Pi\text{-intro})\) rule we
obtain \( \Gamma' \vdash (\lambda x : A. b) : (\Pi x : A. B) \). \( \Box \)

Lemma 7.2.4.6 (Substitution) If \( \Gamma \vdash c : C \) then

- if \( \Gamma, y : C, \Gamma' \vdash ok \) then \( \Gamma, \Gamma'[y := c] \vdash ok \)
- if \( \Gamma, y : C, \Gamma' \vdash a : A \) then \( \Gamma, \Gamma'[y := c] \vdash a[y := c] : A[y := c] \)

Proof: By simultaneous induction on the derivation of \( \Gamma, y : C, \Gamma' \vdash ok \) and \( \Gamma, y : C, \Gamma' \vdash a : A \)
respectively. Use \(\leq\text{(Substitution)}\) (Lemma 7.2.3.3) for the subtyping judgments occurring as
premises in the \((\text{subsum})\) and \((\text{BII\text{-elim})}\) rule. \( \Box \)

Lemma 7.2.4.7 (Bounded-Substitution) If \( \Gamma \vdash c : C \) and \( \Gamma \vdash c \leq c' \) then

- if \( \Gamma, y \leq c' : C, \Gamma' \vdash ok \) then \( \Gamma, \Gamma'[y := c] \vdash ok \)
- if \( \Gamma, y \leq c' : C, \Gamma' \vdash a : A \) then \( \Gamma, \Gamma'[y := c] \vdash a[y := c] : A[y := c] \)

Proof: By simultaneous induction on the derivation of \( \Gamma, y \leq c' : C, \Gamma' \vdash ok \) and the derivation
of \( \Gamma, y \leq c' : C, \Gamma' \vdash a : A \) respectively. Here we use \(\leq\text{Bounded-Substitution}\) (Lemma 7.2.3.8)
for the subtyping judgments occurring as premises in the \((\text{subsum})\) and \((\text{BII\text{-elim})}\) rule. \( \Box \)

Lemma 7.2.4.8 (Generation)

1. If \( \Gamma \vdash s : C \) then \( \Gamma \vdash s' \leq C \) with \((s : s') \in A \) for some \( s \).
2. If \( \Gamma \vdash x : C \) then \( \Gamma \vdash A \leq C \) with \( x : A \in \Gamma \) for some \( A \) or \( x \leq a : A \in \Gamma \) for some \( a \) and
   \( A \).
7.2. META-THEORY

3. If \( \Gamma \vdash (\Pi x : A \cdot B) : C \) then \( \Gamma \vdash s_3 \leq C \) and \( \Gamma \vdash A : s_1 \) and \( \Gamma, x : A \vdash B : s_2 \) for some \((s_1, s_2, s_3) \in \mathcal{R}\).

4. If \( \Gamma \vdash (\lambda x : A \cdot b) : C \) then \( \Gamma \vdash (\Pi x : A \cdot B) \leq C \) and \( \Gamma, x : A \vdash b : B \) and 
   \( \Gamma \vdash (\Pi x : A \cdot B) : s \) for some \( B \) and \( s \).

5. If \( \Gamma \vdash (\Pi x \leq a : A \cdot B) : C \) then \( \Gamma \vdash s_3 \leq C \) and \( \Gamma \vdash A : s_1 \) and \( \Gamma, x \leq a : A \vdash B : s_2 \) for some triple \((s_1, s_2, s_3) \in \mathcal{R}\).

6. If \( \Gamma \vdash (\lambda x \leq a : A \cdot b) : C \) then \( \Gamma \vdash (\Pi x \leq a : A \cdot B) \leq C \) and \( \Gamma, x \leq a : A \vdash b : B \) and 
   \( \Gamma \vdash (\Pi x \leq a : A \cdot B) : s \) for some \( B \) and \( s \).

7. If \( \Gamma \vdash b \ a' : C \) then either 
   - \( \Gamma \vdash B[x := a'] \leq C \) and \( \Gamma \vdash b : (\Pi x : A \cdot B) \) and \( \Gamma \vdash a' : A \) for some \( A \) and \( B \), or 
   - \( \Gamma \vdash B[x := a'] \leq C \) and \( \Gamma \vdash b : (\Pi x \leq a : A \cdot B) \) and \( \Gamma \vdash a' : A \) and \( \Gamma \vdash a' \leq a \) for some \( a \), \( A \) and \( B \).

**Proof:** Every clause is proved by induction on the derivation, and each proof is very similar: there is one case that leads directly to the conclusion by reflexivity of \( \leq \), and the only other possible case is (subsum). (The only exception is that for the application there are two cases that lead directly to the conclusion.) We illustrate this for the fourth clause (ordinary \( \lambda \)-abstraction).

For the fourth clause the last step in the derivation can only be (\( \Pi \)-intro) or (subsum). The first case is trivial by (\( \leq \)-refl), so we now consider the other case. Suppose \( \Gamma \vdash (\lambda x : A \cdot b) : C \) is derived, using

\[
\frac{\text{(subsum)}}{\Gamma \vdash (\lambda x : A \cdot b) : C' \quad \Gamma \vdash C : s \quad \Gamma \vdash C' \leq C}{\Gamma \vdash (\lambda x : A \cdot b) : C}
\]

We apply the IH on \( \Gamma \vdash (\lambda x : A \cdot b) : C' \). This delivers the correct conclusion, but with \( C' \) instead of \( C \). Now we apply transitivity of \( \leq \) (Lemma 7.2.3.6) and we are done. \( \square \)

**Lemma 7.2.4.9 (Correctness of Types)** If \( \Gamma \vdash a : A \) then \( A \equiv s \) or \( \Gamma \vdash A : s \) for some sort \( s \).

**Proof:** By induction on the derivation of \( \Gamma \vdash a : A \). Use Generation and the substitution properties for cases (\( \Pi \)-elim) and (\( \Pi I \)-elim) (Lemmas 7.2.4.8, 7.2.4.6 and 7.2.4.7). \( \square \)

**Theorem 7.2.4.10 (Subject Reduction)**

- If \( \Gamma \vdash a : A \) and \( a \triangleright_\beta a' \) then \( \Gamma \vdash a' : A \).
- If \( \Gamma \vdash ok \) and \( \Gamma \triangleright_\beta \Gamma' \) then \( \Gamma' \vdash ok \).
- If \( \Gamma \vdash a : A \) and \( \Gamma \triangleright_\beta \Gamma' \) then \( \Gamma' \vdash a : A \).
Proof: By simultaneous induction on the derivation. All cases go straightforward by IH, except cases (II-elim) and (BII-elim) of the first clause. We treat here the last case, the case (II-elim) is similar.

Let the last step in the derivation be

\[
\frac{\Gamma \vdash b : (\Pi x \leq e : E. B) \quad \Gamma \vdash c' : E \quad \Gamma \vdash c' \leq e}{\Gamma \vdash b \downrel{\sigma} c' : B[x := c']}
\]

So \(a \equiv b \downrel{\sigma} c'\) and \(A \equiv B[x := c']\). Since \(a \perp \sigma \ a'\) we have three cases.

1. \(a' \equiv b' \downrel{\sigma} c'\) with \(b \downrel{\sigma} b'\): This case is straightforward. By IH \(\Gamma \vdash b' : (\Pi x \leq e : E. B)\), so using (BII-elim) we get \(\Gamma \vdash b' \downrel{\sigma} c' : B[x := c']\), which is the same as \(\Gamma \vdash a' : A\).

2. \(a' \equiv b \downrel{\sigma} c'' \downrel{\sigma} c''': By IH we have \(\Gamma \vdash c'' : E\). By \(\leq\)-Conversion-closed (Theorem 7.2.2.6) on \(\Gamma \vdash c' \leq e\) we get \(\Gamma \vdash c'' \leq e\), so with (BII-elim) \(\Gamma \vdash b \downrel{\sigma} c'' : B[x := c'']\). Using Correctness of Types, we have \(B[x := c'] \equiv s\) or \(\Gamma \vdash B[x := c'] : s\) for some \(s\). In the first case, it is easy to show that \(B[x := c'] \equiv B[x := c'']\) and hence \(\Gamma \vdash b \downrel{\sigma} c'' : B[x := c']\) and we are done. In the second case, \(\Gamma \vdash B[x := c''] \leq B[x := c']\) follows from \(B[x := c'] = B[x := c'']\) by Conv-admissible (Lemma 7.2.3.1) and we use (subsum) to conclude \(\Gamma \vdash b \downrel{\sigma} c'' : B[x := c']\).

3. \(b \equiv \lambda x \leq c : C. d\) and \(a' \equiv d[x := c']\): We treat this case in detail below.

So the premises of the (BII-elim) rule are

(1) \(\Gamma \vdash (\lambda x \leq c : C. d) : (\Pi x \leq e : E. B)\)
(2) \(\Gamma \vdash c' : E\)
(3) \(\Gamma \vdash c' \leq e\)

By Generation (Lemma 7.2.4.8) on 1 there exist \(D\) and \(s\) such that

(4) \(\Gamma \vdash (\Pi x \leq c : C. D) \leq (\Pi x \leq e : E. B)\)
(5) \(\Gamma, x \leq c : C \vdash d : D\)
(6) \(\Gamma \vdash (\Pi x \leq c : C. D) : s\)

By Generation on 6

(7) \(\Gamma \vdash C : s'\)

By \(\leq\)-Generation-left-right (Lemma 7.2.2.9) on 4

(8) \(c = \pi e\)
(9) \(C = \pi E\)
(10) \(\Gamma, x \leq c : C \vdash d \leq B\)

By Conv-admissible on 9 we have \(\Gamma \vdash E \leq C\), using this with 2 and 7 and (subsum) we get

(11) \(\Gamma \vdash c' : C\)

By \(\leq\)-Conversion-closed (Theorem 7.2.2.6) on 3 and 8

(12) \(\Gamma \vdash c' \leq c\)

By Bounded-Substitution (Lemma 7.2.4.7) on 5, 11, 12

(13) \(\Gamma \vdash d[x := c'] : D[x := c']\)

By \(\leq\)-Bounded-Substitution (Lemma 7.2.3.8) on 10, 12

(14) \(\Gamma \vdash D[x := c'] \leq B[x := c']\)

By Correctness of Types (Lemma 7.2.4.9) on 1
7.2. META-THEORY

(15) $\Gamma \vdash (\Pi x \leq e : E. B) : s''$

By Generation (Lemma 7.2.4.8) on 15

(16) $\Gamma, x \leq e : E \vdash B : s''$

By Bounded-Substitution (Lemma 7.2.4.7) on 16, 2, 3

(17) $\Gamma \vdash B[x := c'] : s''$

Now use (subsum) on 13, 14, 17

(18) $\Gamma \vdash d[x := c'] : B[x := c']$

This is the same as $\Gamma \vdash a' : A$, so we are done with this case.

Subject Reduction has an important consequence on subtyping derivations, namely that the restricted subtyping rules do not introduce untypable terms. In other words, if the terms in the conclusion are typable, then all terms in the derivation are typable (Lemma 7.2.4.13). In order to prove this, we need the following two results:

Lemma 7.2.4.11 If $\Gamma \vdash x a_1 \ldots a_n : A$ and $x \leq c : C \in \Gamma$ then $\Gamma \vdash c a_1 \ldots a_n : A$.

Proof: By straightforward induction on the derivation.

Lemma 7.2.4.12 If $\Gamma \vdash A' \leq A$ and $\Gamma \vdash A' : s$ then both

- if $\Gamma, x : A, \Gamma' \vdash b : B$ then $\Gamma, x : A', \Gamma' \vdash b : B$, and

- if $\Gamma, x : A, \Gamma' \vdash ok$ then $\Gamma, x : A', \Gamma' \vdash ok$.

Proof: By simultaneous induction on the derivation of $\Gamma, x : A, \Gamma' \vdash b : B$ and $\Gamma, x : A, \Gamma' \vdash ok$ respectively.

Lemma 7.2.4.13 Suppose $\Upsilon$ is a derivation for $\Gamma \vdash a \leq b$, and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ hold. Then for all subderivations $\Upsilon'$ of $\Upsilon$, where $\Upsilon'$ shows $\Gamma' \vdash c \leq d$ for some $c$ and $d$, there are $C, D$ such that $\Gamma' \vdash c : C$ and $\Gamma' \vdash d : D$.

Proof: By induction on $\Upsilon$.

We want to stress here that this lemma only holds for the restricted rules ($\vdash$), and not for the liberal rules ($\eta$). This is in contrast to most other lemmas and theorems which hold regardless of which version of the subtyping rules are used. This difference is caused by two facts. First, this lemma is concerned with subderivations and most other lemmas are not. Second, there are fewer $\vdash$ subderivations than $\eta$ subderivations, even though the set of derivable judgments for $\vdash$ and $\eta$ is the same.

This lemma has the following consequence. The ($\leq\Pi\text{app}$) rule is not used in subtyping derivations with typable terms in the conclusion. This follows from the fact that the conclusion of the ($\leq\Pi\text{app}$) rule contains two terms that are never typable, since the terms consist of an application of a $\Pi$-type to one or more arguments.
7.2.5 Minimal Typing

For ordinary functional PTSs, we have Uniqueness of Typing, which says that a term has only one type, modulo β-conversion (see Theorem 2.1.2.8). We do not have unique types in \( \text{PTS} \subseteq s \), since by the subsumption rule a term can have different types. We will show that we do have a weaker property, Minimal Typing. This means that every typable term \( a \) has a minimal type \( A \), which means that if \( B \) is a type of \( a \), \( B \) is a supertype of \( A \). Formally:

**Definition 7.2.5.1** Term \( a \) has minimal type \( A \) in \( \Gamma \), written as \( \Gamma \vdash_m a : A \), if \( \Gamma \vdash a : A \) and

\[
\Gamma \vdash a : B \implies \Gamma \vdash A \leq B \text{ for all } B.
\]

Minimal Typing is important for type-checking, since the problem "does term \( a \) have type \( B \)" can then be split into the simpler problems "compute a minimal type \( A \) for \( a \)" and "is \( A \) a subtype of \( B \)". By definition, minimal types are unique up to equivalence, i.e. \( \Gamma \vdash_m a : A \) and \( \Gamma \vdash_m a : A' \) implies \( \Gamma \vdash A \leq A' \) and \( \Gamma \vdash A' \leq A \). Assuming antisymmetry of \( \leq \), minimal types are unique up to \( \beta \)-conversion.

Like Uniqueness of Types for PTSs, Minimal Typing cannot be expected to hold for all \( \text{PTS} \subseteq s \). We restrict ourselves to the functional ones, defined as follows.

**Definition 7.2.5.2** A \( \text{PTS} \subseteq \lambda(S, A, R, S \subseteq, R \subseteq) \) is functional if

\[
(s:s') \in A \text{ and } (s:s'') \in A \implies s' \equiv s''
\]

\[
(s_1,s_2,s_3) \in R \text{ and } (s_1,s_2,s_3') \in R \implies s_3 \equiv s_3'
\]

\[
(s_1,s_2,s_3) \in R \subseteq \text{ and } (s_1,s_2,s_3') \in R \subseteq \implies s_3 \equiv s_3'
\]

However, Minimal Typing is not easily proved. A direct proof by induction on the structure of the term \( a \) fails, because of two problems.

1. We sometimes need the induction hypothesis for a type of \( a \), instead of a subterm.

   We solve this problem by first proving a weaker property called Weak Minimal Typing (Lemma 7.2.5.5), which is strong enough to replace the IH for the type of \( a \). The actual situation is a bit more subtle than described here, see the case "\( a \equiv \lambda X : C. d'' \)" in the proof of Minimal Typing (page 248). This problem is caused by the fact that in \( \text{PTS} \subseteq s \), we can have subtyping on more than one level.

2. If \( a \) is an application \( b \ c' \), we get by the IH a minimal type \( B \) of \( b \). But we are not interested in \( B \) itself, but in \( B' \), the least supertype of \( B \) that is a \( \Pi \)-type; only from \( B' \) we can calculate a minimal type of the application \( b \ c' \). How do we obtain this \( B' \)?

   This problem is solved by introducing a new kind of reduction \( \triangleright_{\text{whb} \sigma} \) (weak head beta sigma), which reduces \( B \) to \( B' \). For a precise account we refer to the case "\( a \equiv b \ c'' \)" in the proof of Minimal Typing (page 249).

The solutions to these problems determine the structure of this section: we first prove Weak Minimal Typing and some related properties, then define \( \text{whb} \sigma \) reduction and develop some theory about it, and only then we show Minimal Typing.

Most of the properties depend on the \( \text{PTS} \subseteq \) being functional, so we have the following convention.

**Convention:** In this section we consider only functional \( \text{PTS} \subseteq s \).
7.2. META-THEORY 245

Weak Minimal Typing

The Weak Minimal Typing property says common types of a term have a (common) lower bound.

**Definition 7.2.5.3** Terms $a$ and $b$ have a *lower bound* in $\Gamma$, $\Gamma \vdash a \sqcup b$, if there is a $c$ such that $\Gamma \vdash c \leq a$ and $\Gamma \vdash c \leq b$.

Using this notation, we express a few properties of subtyping.

**Lemma 7.2.5.4**

- If $\Gamma \vdash s_1 \sqcup s_2$ then $s_1 \equiv s_2$.
- If $\Gamma \vdash (\Pi x : A_1. B_1) \sqcup (\Pi x : A_2. B_2)$ then $\Gamma, x : A_1 \vdash B_1 \sqcup B_2$.
- If $\Gamma \vdash (\Pi x \leq a_1 : A_1. B_1) \sqcup (\Pi x \leq a_2 : A_2. B_2)$ then $\Gamma, x \leq a_1 \vdash B_1 \sqcup B_2$.
- Not $\Gamma \vdash (\Pi x : A_1. B_1) \sqcup (\Pi x \leq a_2 : A_2. B_2)$.
- If $\Gamma \vdash A \sqcup s$ then $\Gamma \vdash A \leq s$.

**Proof:** We prove the first part; the other proofs are similar. Assume that $\Gamma \vdash C \leq s_1$ and $\Gamma \vdash C \leq s_2$. Proceed by induction on the derivation of $\Gamma \vdash C \leq s_1$.

These properties help in proving Weak Minimal Typing.

**Lemma 7.2.5.5 (Weak Minimal Typing)**

If $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$ then $\Gamma \vdash A \sqcup B$.

**Proof:** By induction on the structure of $a$. Use Generation (Lemma 7.2.4.8) on $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$. For some cases, we need Lemma 7.2.5.4, Trans-admissible (Lemma 7.2.3.6), and the substitution properties (Lemmas 7.2.3.3 and 7.2.3.8).

Note that if we read “$\equiv_{\beta}$” for “$\leq$”, then “$\sqcup$” is equal to “$\equiv_{\beta}$”, and we have the Uniqueness of Types property.

The following two lemmas relate the types of two terms that are in the subtype relation. This is convenient in the proof of Minimal Typing.

**Lemma 7.2.5.6** If $\Gamma \vdash a \leq b$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ then $\Gamma \vdash A \sqcup B$.

**Proof:** By induction on the subtyping derivation, using Lemmas 7.2.4.11, 7.2.5.4, 7.2.4.12 and 7.2.4.13, and Weak Minimal Typing.

**Lemma 7.2.5.7** If $\Gamma \vdash a : A$ and $\Gamma \vdash A \leq B$ and $\Gamma \vdash B : s$ then $\Gamma \vdash A : s$.

**Proof:** Use Correctness of Types (Lemma 7.2.4.9) on $\Gamma \vdash a : A$, and Lemma 7.2.5.6.
**whβσ-reduction**

We define $\triangleright_{whβσ}$ reduction as the union of $\triangleright_{whβ}$ and $\triangleright_{whσ}$. The $\triangleright_{whβ}$ reduction is the usual weak head restriction of $\triangleright_β$: $\triangleright_{whβ}$ reduces $a b_1 \ldots b_n$ to $a' b_1 \ldots b_n$ if $a$ is a $β$-redex which reduces to $a'$. The $\triangleright_{whσ}$ reduction reduces a term $x b_1 \ldots b_n$ to $c b_1 \ldots b_n$ if $c$ is the bound of $x$. This reduction is the weak head restriction of the so-called $Γ$-reduction found in [PS97, Che97]. We show basic properties of these reduction relations in Lemmas 7.2.5.11 through 7.2.5.13, then relate $\triangleright_{whβσ}$ and $≤$ in Lemmas 7.2.5.14 and 7.2.5.15, and prove Subject Reduction for $\triangleright_{whβσ}$ and some consequences in Lemmas 7.2.5.16 through 7.2.5.18. The theory for $\triangleright_{whβσ}$ will be re-used in Section 7.3, where we give a typing algorithm for $PTS≤ɔs$.

**Definition 7.2.5.8** The relation $\vdash _{whσ}$ is defined as follows:

\[ x \leq a : A \in Γ \quad \implies \quad Γ \vdash a \triangleright_{whσ} x \triangleright_{whσ} a \\
Γ \vdash a \triangleright_{whσ} a' \quad \implies \quad Γ \vdash a b \triangleright_{whσ} a' b \]

We say $a$ is in whσnf — weak head $σ$ normal form — in $Γ$, if there is no $b$ such that $Γ \vdash a \triangleright_{whσ} b$.  

**Definition 7.2.5.9** The relation $\triangleright_{whβ}$ is defined as usual:

\[(λx:A.b)\quad \triangleright_{whβ} \quad b[x:=a] \]

\[(λx≤a':A.b)\quad \triangleright_{whβ} \quad b[x:=a] \]

\[a\quad b \quad \triangleright_{whβ} \quad a'\quad b \; , \; \text{if} \; a \triangleright_{whβ} a' \]

We define the notion of whβnf as usual: $a$ is in whβnf, if there is no $b$ such that $a \triangleright_{whβ} b$.  

**Definition 7.2.5.10** $Γ \vdash a \triangleright_{whβσ} b$ is $Γ \vdash a \triangleright_{whσ} b$ or $a \triangleright_{whβ} b$. $Γ \vdash _{whβσ} b$ is the reflexive and transitive closure of $Γ \vdash _{whβσ} b$.

**Lemma 7.2.5.11** If $a \triangleright_β c$ and $c$ in whβnf, then for some $b$ in whβnf $a \triangleright_{whβ} b$ and $b \triangleright_β c$.

**Proof sketch:** Perform the following steps.

1. Show in untyped $λ$-calculus: If $a \triangleright_β c$ and $c$ is in whβnf, then for some $b$ in whβnf $a \triangleright_{whβ} b$, i.e. $a$ has a whβnf. This can be proved using the so-called standardization theorem [Bar84].

2. Transfer this result to $PTS≤ɔs$, using a translation from $PTS≤ɔ$ terms to untyped terms.

3. Show in $PTS≤ɔs$: If $a \triangleright_{whβ} b$ and $a \triangleright_β c$ then $b \triangleright_β d$ and $c \triangleright_{whβ} d$ for some $d$.

Now suppose $a \triangleright_β c$ and $c$ in whβnf. Then by 2 there is a $b$ with $a \triangleright_{whβ} b$ and $b$ in whβnf ($b$ is the whβnf of $a$). By 3 there is a $d$ with $b \triangleright_β d$ and $c \triangleright_{whβ} d$. Since $c$ is in whβnf, we have $d \equiv c$, so $b \triangleright_β c$.

**Lemma 7.2.5.12**

- If $a \triangleright_β s$ then $a \triangleright_{whβ} s$.

- If $a \triangleright_β x b_1 \ldots b_n$ then $a \triangleright_{whβ} x b'_1 \ldots b'_n$ and $b_i \triangleright_β b_i$ for some $b_1 \ldots b_n$.  


7.2. META-THEORY

- If \( a \xrightarrow{\beta} \Pi x : B.C \) then \( a \xrightarrow{\text{wh} \beta} \Pi x : B'.C' \) and \( B' \xrightarrow{\beta} B \) and \( C' \xrightarrow{\beta} C \) for some \( B', C' \).

- If \( a \xrightarrow{\beta} \Pi x \leq b : B.C \) then \( a \xrightarrow{\text{wh} \beta} \Pi x \leq b' : B'.C' \) and \( b' \xrightarrow{\beta} b, B' \xrightarrow{\beta} B \) and \( C' \xrightarrow{\beta} C \) for some \( b', B', C' \).

*Proof:* Easy, by using Lemma 7.2.5.11.

The following lemma says \( \triangleright_{\text{wh} \beta \sigma} \) is deterministic, i.e. reducing a term one step always results in the same term.

**Lemma 7.2.5.13 (CR_{wh \beta \sigma}, Unique wh \beta \sigma-normal forms)**

1. If \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} B \) and \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} C \) then \( B \equiv C \).

2. If \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} B \) and \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} C \) and \( B \) and \( C \) are in \( \text{wh} \beta \sigma \text{nf} \) in \( \Gamma \), then \( B \equiv C \).

*Proof:* The first proof is easy. The second proof is even easier: by 4 we have Church-Rosser for \( \text{wh} \beta \sigma \), and by Church-Rosser normal forms are unique.

The following two lemmas relate \( \triangleright_{\text{wh} \beta \sigma} \) and \( \leq \).

**Lemma 7.2.5.14** If \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} B \) then \( \Gamma \vdash A \leq B \).

*Proof:* By induction on the reduction sequence.

**Lemma 7.2.5.15**

- If \( \Gamma \vdash A \leq s \) then \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} s \).

- If \( \Gamma \vdash A \leq (\Pi x : B.C) \) then \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} (\Pi x : B'.C') \) and \( \Gamma \vdash (\Pi x : B'.C') \leq (\Pi x : B.C) \).

- If \( \Gamma \vdash A \leq (\Pi x \leq b : B.C) \) then \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} (\Pi x \leq b' : B'.C') \) and \( \Gamma \vdash (\Pi x \leq b' : B'.C') \leq (\Pi x \leq b : B.C) \).

*Proof:* By induction on the NR-height of the derivation, using \( \leq \)-Generation (Lemma 7.2.2.7) and Lemma 7.2.5.12.

The following three lemmas relate \( \triangleright_{\text{wh} \beta \sigma} \) and typing.

**Lemma 7.2.5.16 (Subject Reduction for wh \beta \sigma)**

If \( \Gamma \vdash a \triangleright_{\text{wh} \beta \sigma} a' \) and \( \Gamma \vdash a : A \) then \( \Gamma \vdash a' : A \).

*Proof:* By induction on the length of the reduction sequence. Use Subject Reduction (Theorem 7.2.4.10) and Lemma 7.2.4.11.

**Lemma 7.2.5.17 (Type Reduction for wh \beta \sigma)** If \( \Gamma \vdash a : A \) and \( \Gamma \vdash A \triangleright_{\text{wh} \beta \sigma} A' \) then \( \Gamma \vdash a : A' \).

*Proof:* By Correctness of Types and Lemma 7.2.5.14 and Subject Reduction for wh \beta \sigma (Lemma 7.2.5.16).
Lemma 7.2.5.18 If $\Gamma \vdash a : A$ and $\Gamma \vdash a \not\in \text{wh}^{\leqslant} B$ and $B$ is not a sort, then $\Gamma \vdash B : s$ for some $s$.

Proof: Easy, by Correctness of Types (Lemma 7.2.4.9) and Subject Reduction for $\text{wh}^{\beta} B$ (Lemma 7.2.5.16). □

Theorem 7.2.5.19 (Minimal Typing for functional $PTS^{\leqslant} s$)
If $a$ is typable in $\Gamma$, there is a type $M$ with $\Gamma \vdash a : M$.

Proof: By induction on the structure of $a$. Use Generation (Lemma 7.2.4.8). For every case we have two parts.

1. Find an $M$ such that $\Gamma \vdash a : M$.
2. Show that this $M$ is minimal, i.e. $\Gamma \vdash a : B \implies \Gamma \vdash M \leq B$ for all $B$.

We treat four cases, viz. $a \equiv s$, $a \equiv \Pi x : C. D$, $a \equiv \lambda x : C. d$ and $a \equiv b \cdot c'$.

$a \equiv s$. By Generation $(s : s') \in A$ for some $s'$. Take $M \equiv s'$. We know $\Gamma \vdash ok$, so by (axiom)

$\Gamma \vdash a : M$.

Now take a $B$ with $\Gamma \vdash a : B$. By Generation $\Gamma \vdash s'' \leq B$ for some $s''$ with $(s : s'') \in A$.

By functionality $s' \equiv s''$, so $\Gamma \vdash M \leq B$.

$a \equiv \Pi x : C. D$. By Generation $\Gamma \vdash C : s_1$ and $\Gamma, x : C \vdash D : s_2$ for some $(s_1, s_2, s_3) \in R$.

Take $M \equiv s_3$. By (II-form) we have $\Gamma \vdash a : M$.

Now take a $B$ with $\Gamma \vdash a : B$. By Generation we have $\Gamma \vdash s'_3 \leq B$ and $\Gamma \vdash C : s'_1$ and

$\Gamma, x : C \vdash D : s'_2$ for some $(s'_1, s'_2, s'_3) \in R$. Using Weak Minimal Typing (Lemma 7.2.5.5) twice, we get $\Gamma \vdash s_1 \cup s'_1$ and $\Gamma, x : C \vdash s_2 \cup s'_2$.

By Lemma 7.2.5.4 $s_1 \equiv s'_1$ and $s_2 \equiv s'_2$.

By functionality, $s_3 \equiv s'_3$. Hence $\Gamma \vdash M \leq B$.

Instead of Weak Minimal Typing (Lemma 7.2.5.5), we could have used the IH instead. However, in the following case we have to rely on the lemma.

$a \equiv \lambda x : C. d$. By Generation we have for some $D$ and $s$:

(1) $\Gamma, x : C \vdash d : D$
(2) $\Gamma \vdash (\Pi x : C. D) : s$
By IH on 1 we have for some $M_1$
(3) $\Gamma, x : C \vdash M_1 \leq D$
and hence
(4) $\Gamma, x : C \vdash M_1 \leq D$
By Generation on 2
(5) $\Gamma \vdash C : s_1$
(6) $\Gamma, x : C \vdash D : s_2$
(7) $(s_1, s_2, s_3) \in R$
Using Lemma 7.2.5.7 on 4, 3 and 6
(8) $\Gamma, x : C \vdash M_1 : s_2$
With rule (II-form) on 5, 8 and 7
(9) $\Gamma \vdash (\Pi x : C. M_1) : s_3$
Now use rule (II-intro) on 3 and 9
7.2. **META-THEORY**

(10) $\Gamma \vdash (\lambda x : C. d) : (\Pi x : C. M_1)$

So choose $M \equiv \Pi x : C. M_1$, and we have

(11) $\Gamma \vdash a : M$

We have stated in the beginning of this section on Minimal Typing that we need the property of Weak Minimal Typing before we can prove Minimal Typing. We have used Weak Minimal Typing to infer 8, through lemma Lemma 7.2.5.7 which relies on Weak Minimal Typing.

It seems impossible to obtain 8 by using the IH, instead of using Weak Minimal Typing (directly or indirectly). Let us illustrate this with the following reasoning, which instead of (4) assumes the much stronger $M_1 \Rightarrow^\beta D$, but still fails.

We need to show $\Gamma, x : C \vdash M_1 \leq s_2$. By Correctness of Types on 3 we have $\Gamma, x : C \vdash M_1 : s_2'$ for some $s_2'$ (the case $M_1 \equiv s_2'$ for some $s_2'$ is straightforward).

So we now need to show $s_2' \equiv s_2$. By Subject Reduction on $M_1 \Rightarrow^\beta D$, we have $\Gamma, x : C \vdash D : s_2'$. Now we would like to apply the IH on $D$, which would give us $\Gamma, x : C \vdash D \leq s_2$ and $\Gamma, x : C \vdash S \leq s_2'$. By Lemma 7.2.5.4 we would have $s_2 \equiv s_2'$ and we would be done. However, we cannot apply the IH on $D$, since $D$ is not a subterm of $a$.

Even if we could apply the induction hypothesis, we have the problem that we have only $\Gamma, x : C \vdash M_1 \leq D$ and not $M_1 \Rightarrow^\beta D$. This indicates that Weak Minimal Typing is really necessary to prove 8 (via Lemma 7.2.5.7).

Now we return to the actual proof and show the minimality of this $M$. Take a $B$ with $\Gamma \vdash a : B$. By Generation

(12) $\Gamma \vdash (\Pi x : C. D') \leq B$

(13) $\Gamma, x : C \vdash d : D'$

By 3 and 13

(14) $\Gamma, x : C \vdash M_1 \leq D'$

and by $(\leq \Pi)$

(15) $\Gamma \vdash (\Pi x : C. M_1) \leq (\Pi x : C. D')$

By Trans-admissible (Lemma 7.2.3.6) on 15 and 12 and our choice for $M$

(16) $\Gamma \vdash M \leq B$

$a \equiv b' c'$: By Generation, we have two cases. We treat only the case $\Gamma \vdash b : (\Pi x : C. D)$ here; the case $\Gamma \vdash b : (\Pi x \leq c : C. D)$ is similar. So we have

(1) $\Gamma \vdash b : (\Pi x : C. D)$

(2) $\Gamma \vdash c' : C$

By IH on 1

(3) $\Gamma \vdash c_1 b \vdash M_1$

By 3 and 1

(4) $\Gamma \vdash M_1 \leq (\Pi x : C. D)$

By Lemma 7.2.5.15 on 4

(5) $\Gamma \vdash M_1 \Rightarrow^\beta (\Pi x : C'. D')$
(6) $\Gamma \vdash (\Pi x : C', D') \leq (\Pi x : C, D)$
By Lemma 7.2.5.14 on 5
(7) $\Gamma \vdash M_1 \leq (\Pi x : C', D')$
By Type Reduction for $\text{wh}\beta\sigma$ (Lemma 7.2.5.17) on 3 and 5
(8) $\Gamma \vdash b : (\Pi x : C', D')$
By $\leq$-Generation-left-right (Lemma 7.2.2.9) on 6
(9) $\Gamma \vdash C \leq C'$
By Correctness of Types on 8 and Generation
(10) $\Gamma \vdash C' : s'$
By (subsum) on 2,9,10
(11) $\Gamma \vdash c' : C'$
By (II-elim) on 8,11
(12) $\Gamma \vdash b \, c' : D'[x := c']$

Take $M \equiv D'[x := c']$, and we have $\Gamma \vdash a : M$.

Now we show the minimality of $M$. Take a $B$ with $\Gamma \vdash b \, c' : B$. By Generation, we again have two cases.

a. $\Gamma \vdash b : (\Pi x : E. F)$ and $\Gamma \vdash F[x := c'] \leq B$ and $\Gamma \vdash c' : E$, or
b. $\Gamma \vdash b : (\Pi x \leq e : E. F)$

In case b, we have by Weak Minimal Typing and 1 that $\Gamma \vdash (\Pi x : C. D) \cup (\Pi x \leq e : E. F)$. This leads to a contradiction by Lemma 7.2.5.4. So we are in case a, and we have:

(13) $\Gamma \vdash F[x := c'] \leq B$
(14) $\Gamma \vdash b : (\Pi x : E. F)$
(15) $\Gamma \vdash c' : E$

By 3 and 14
(16) $\Gamma \vdash M_1 \leq (\Pi x : E. F)$
By Lemma 7.2.5.15 on 16 and using Unique $\text{wh}\beta\sigma$-normal form (Lemma 7.2.5.13) on 5
(17) $\Gamma \vdash (\Pi x : C', D') \leq (\Pi x : E. F)$
By $\leq$-Generation-left-right (Lemma 7.2.2.9) on 17
(18) $\Gamma, x : E \vdash D' \leq F$
By $\leq$-Substitution (Lemma 7.2.3.3) on 18 and 15
(19) $\Gamma \vdash D'[x := c'] \leq F[x := c']$
and finally, by Trans-admissible (Lemma 7.2.3.6) on 19 and 13 and by definition of $M$
(20) $\Gamma \vdash M \leq B$

This concludes this case. $\square$
7.2. META-THEORY

7.2.6 Relation to P Ts$

In this section we show that a P Ts can be seen as a P T S$ with $S$ and $R$ empty.

Theorem 7.2.6.1
Consider the P Ts $\lambda S = \lambda (S, A, R)$ and the P T S$ $ \lambda S$ = $\lambda (S, A, R, \emptyset, \emptyset)$. Then

- if $\Gamma \vdash S ok$ then $\Gamma \vdash S$ ok, and
- if $\Gamma \vdash S a : A$ then $\Gamma \vdash S a : A$.

Proof: Easy, since the P Ts rules form a subset of the P T S$ rules, apart from rule (conv), which can easily be imitated by (subsum), using Conv-admissible (Lemma 7.2.3.1).

Lemma 7.2.6.2 If $S$ = $\emptyset$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ and $\Gamma \vdash a \leq b$ then $a = b$.

Proof: By induction on the derivation of $\Gamma \vdash a \leq b$. Note that by Lemma 7.2.4.13 all sub-derivations are also well-typed.

Theorem 7.2.6.3
Consider the P Ts $\lambda S = \lambda (S, A, R)$ and the P T S$ $ \lambda S$ = $\lambda (S, A, R, \emptyset, \emptyset)$. Then

- if $\Gamma \vdash S ok$ then $\Gamma \vdash S ok$, and
- if $\Gamma \vdash S a : A$ then $\Gamma \vdash S a : A$.

Proof: By simultaneous induction on the structure of the derivation. Use Lemma 7.2.6.2 for the (subsum) case.

By Theorem 7.2.6.1 and 7.2.6.3 typing in the P Ts $\lambda (S, A, R)$ is equivalent to typing in the P T S$ $ \lambda (S, A, R, \emptyset, \emptyset)$. So any P Ts can be seen as a P T S$ with $S \equiv R \equiv \emptyset$. In other words, the framework of P Ts includes the framework of P Ts.

We conjecture that adding subtyping and bounded quantification is a conservative extension, provided that there are no more bounded quantifications than ordinary quantifications.

Conjecture 7.2.6.4 (Conservativity) Consider the P Ts $\lambda S = \lambda (S, A, R)$ and the P T S$ $ \lambda S$ = $\lambda (S, A, R, S, R)$ with $R \subseteq R$. Suppose $\Gamma$ is a $\lambda S$ pseudocontext and $a$ and $A$ are $\lambda S$ pseudoterms. Then

- $\Gamma \vdash S a : A$ $\iff$ $\Gamma \vdash S a : A$
- $\Gamma \vdash S a : -$ $\iff$ $\Gamma \vdash S a : -$
- $\Gamma \vdash S - : A$ $\iff$ $\Gamma \vdash S - : A$
7.3 Type-Checking Algorithm

A type-checking algorithm is essential for implementation of $PTS^<s$, e.g. in Yarrow. It must be able to decide the following two problems:

- $\Gamma \vdash a : A ?$ – Given $\Gamma$, $a$ and $A$, has $a$ type $A$ in $\Gamma$?
- $\Gamma \vdash a : ?$ – Given $\Gamma$ and $a$, what is a minimal type of $a$ in $\Gamma$, or is $a$ not typable?

These two problems are related. If we solve the second problem, we have also solved the first one, since when given $\Gamma$ and $a$, we can compute a minimal type $A'$ of $a$, and then check if $A' \leq A$, since we have Minimal Typing. So we concentrate on the second problem. In order to solve it, we must also solve the following problems:

- $\Gamma \vdash \text{ok} ?$ – Given $\Gamma$, is it well-formed?
- $\Gamma \vdash a \leq b ?$ – Given $\Gamma$, $a$ and $b$, is $a$ a subtype of $b$ in $\Gamma$?

The type-checking problem in ordinary $PTS$s is already difficult to solve. In [Pol92] a natural algorithm is given, but it works only on a small set of the so-called semi-full $PTS$s (from the $\lambda$-cube, only $\lambda P$ and $\lambda C$ are semi-full). Another approach is chosen by both [BJMP94] (section 6) and [Sev96], that give similar algorithms, that work for all functional $PTS$s, but both algorithms use an auxiliary algorithm. Considering the fact that $PTS^<s$ have roughly twice as many typing rules as $PTS$s, this approach is unworkable here. We choose for the approach given in [Pol93], that works for all so-called bijective $PTS$s, which includes all systems in the $\lambda$-cube and $\lambda \omega_L$. The rest of the introduction explains this approach, adapted to $PTS^<s$.

The structure of the typing algorithm for a term $a$ is as follows: consider the structure of $a$, and determine what the last step in a typing derivation of $\Gamma \vdash a : A$ could be. This typically leads to a number of premises, which are typing problems for subterms of $a$. Solve these problems by recursion; this gives types for these subterms. A type of $a$ is computed by combining these types.

The problem with the original typing rules is that given a term $a$, there always two rules that could be used as last step in a derivation. We could use the rule specific for the structure of $a$, but we could also use the subsumption rule. Therefore, we present new derivation rules, notated with $\text{sd}$, without subsumption rule. The subscript sd stands for syntax-directed: the last step in a typing derivation for $\Gamma \vdash a : A$ is uniquely determined by the structure (syntax) of $a$. But we cannot heedlessly leave out the subsumption rule; sometimes it is essential for the next step in a derivation. So we integrate the subsumption rule into the other rules at all places where it is essential. By only allowing subsumption where it is essential, the term is never assigned a bigger type than necessary, so the sd rules deliver a minimal type. From these syntax-directed rules we can easily make an algorithm. (In fact, our sd rules are not completely syntax-directed, but this is easy to remedy in the algorithm.)

We provide also syntax-directed versions of the subtyping rules, where the shapes of $a$ and $b$ determine which rule must be used to derive $\Gamma \vdash a \leq b$, and sd versions of the well-formedness rules, where the shape of $\Gamma$ determines which rule must be used to derive $\Gamma \vdash \text{ok}$. In both cases, it is not so hard to find the sd rules, so we concentrate here on the typing rules.
7.3. TYPE-CHECKING ALGORITHM

Of course, we want the syntax-directed typing rules and the original rules to be equivalent, so we have to prove:

soundness: \( \Gamma \vdash_{sd} a : A \implies \Gamma \vdash a : A \)
completeness: \( \Gamma \vdash a : A \implies \exists A'. \Gamma \vdash_{sd} a : A' \) and \( \Gamma \vdash A' \leq A \)

This is the usual definition of completeness in presence of subtyping: the original rules can assign many types \( A \) to a term \( a \), but the sd rules assign only one type \( A' \) to \( a \), a minimal type. It is straightforward to prove soundness, but completeness is problematic, as we will show in Section 7.3.1. We use the following tricks, from [Pol93], to tackle this problem.

1. Consider syntax-directed typing judgments with not only the type, but also the sort of the term, so those judgments have the form

\[ \Gamma \vdash_{sd} a : A : s \]

2. Integrate the II-formation rule into the syntax-directed II-introduction rule.

We have to pay a price for the extended judgments. First, we need to compute the sort of each term. In order to compute the sort of a variable, we must put its sort in the context. And we can only compute the sort of an application from the sorts of the function and the argument, if we put a restriction on the set \( R \) of rules: \( R \) must be bijective. Second, we need to extend \( S \) and \( A \) in order for every term to have a sort.

For efficiency reasons, the syntax-directed typing algorithm does not check whether the context is well-formed (the original rules do, see Lemma 7.2.4.1). Instead, the algorithm assumes the context is well-formed, and keep this invariant through recursive calls. So before type-checking a term, we need to check separately that the context is well-formed.

The syntax-directed subtyping rules are very similar to the restricted subtyping rules (Definition 7.2.1.1), with an important difference: we have banished the \( \leq \)-red rule, since it can be applied at any moment in a derivation. Instead, the sd subtyping rules expect their arguments to be in \( \beta \) normal-form; so any judgment \( \Gamma \vdash a \leq b \) in the typing rules must be replaced by \( \Gamma \vdash_{sd} \beta n(a) \leq \beta n(b) \). We will use ordinary contexts in the sd subtyping rules, since we do not need the additional typing information offered by extended contexts.

In practice, reducing a term to normal form can be very time consuming (in particular since definitions must also be unfolded in \( PTS \leq s \) with definitions). Therefore it is important in practice to reduce only lazily, so only perform reduction when it is really necessary, i.e. to see the constructor of terms. However, the description of optimized sd subtyping rules is beyond the scope of this chapter.

In order to show termination of the subtyping algorithm, we adapt an approach similar to [PS97, Che97]. This means we introduce \( \sigma \)-reduction (which is an extended version of the wh-\( \sigma \)-reduction of the previous section), and assume the \( PTS \leq s \) is SN\( _{\beta \sigma} \), i.e. strongly normalizing with respect to the combination of \( \beta \) and \( \sigma \)-reduction. We do not prove SN\( _{\beta \sigma} \) for any \( PTS \leq s \). Just as for \( PTS \), strong normalization results for \( PTS \leq s \) are very hard to prove, and do not hold for all \( PTS \leq s \).

Section 7.3.1 gives a technical discussion why the proof of completeness for the typing judgment is problematic, and motivates the tricks indicated above. Section 7.3.2 defines the syntax-directed typing rules. In Section 7.3.3 we prove soundness, and in Section 7.3.4 completeness, with respect to the original judgments. This holds only for a class of \( PTS \leq s \),
viz. the bijective $PTS_{\leq s}$ which are $SN_{\beta}$. Section 7.3.5 tells how to interpret the $sd$ rules as an algorithm, and Section 7.3.6 shows for bijective $SN_{\beta}$ $PTS_{\leq s}$ that the subtyping and the typing algorithms are terminating, and hence type-checking is decidable for these $PTS_{\leq s}$.

### 7.3.1 Completeness is Problematic

As noted by [Pol92, BJMP94, Sev96, Pol93], proofs of completeness of syntax-directed typing rules for $PTS$s (and also for $PTS_{\leq s}$) are problematic because of the (II-intro) rule:

\[
\frac{\Gamma, x : B \vdash c : C \quad \Gamma \vdash (\Pi x : B. C) : s}{\Gamma \vdash (\lambda x : B. c) : (\Pi x : B. C)}
\]

The most naive $sd$ version of this rule is obtained by just replacing $\vdash$ by $\vdash_{sd}$.

\[
\frac{\Gamma, x : B \vdash_{sd} c : C \quad \Gamma \vdash_{sd} (\Pi x : B. C) : s}{\Gamma \vdash_{sd} (\lambda x : B. c) : (\Pi x : B. C)}
\]

A proof of completeness by induction on $\Gamma \vdash a : A$ fails in this case. By IH on $\Gamma, x : B \vdash c : C$ we know $\Gamma, x : B \vdash_{sd} c : C'$ and $\Gamma, x : B \vdash C' \leq C$ for some $C'$, and by IH on $\Gamma \vdash (\Pi x : B. C) : s$ we know $\Gamma \vdash_{sd} (\Pi x : B. C) : S$ and $\Gamma \vdash S \leq s$. We cannot apply (sd-II-intro1) now, since we need $\Gamma \vdash_{sd} (\Pi x : B. C') : s$ and we have $\Gamma \vdash_{sd} (\Pi x : B. C) : S$. There are two mismatches here. First, the mismatch between $C'$ and $C$, and second, the mismatch between $s$ and $S$.

The second mismatch is solved by using the more reasonable rule

\[
\frac{\Gamma, x : B \vdash_{sd} c : C \quad \Gamma \vdash_{sd} (\Pi x : B. C) : s \quad \Gamma \vdash_{sd} S \leq s}{\Gamma \vdash_{sd} (\lambda x : B. c) : (\Pi x : B. C) ; S}
\]

Now a proof by induction fails because we still have the first mismatch: we need to prove $\Gamma \vdash_{sd} (\Pi x : B. C') : S$ and we have $\Gamma \vdash_{sd} (\Pi x : B. C) : S$.

The first solution that comes to mind to solve this problem is to show $\Gamma \vdash (\Pi x : B. C') : s$, and to apply the IH to that judgment. This leaves the problem of finding an appropriate induction measure, so that we indeed can use the IH. We cannot do induction on the derivation, because all derivations of $\Gamma \vdash (\Pi x : B. C') : s$ can be longer than the derivation of $\Gamma \vdash (\Pi x : B. C) : s$ (since $C'$ can be a bigger term than $C$). We think it is impossible to define such an induction measure.

Another solution seems to be the development of some meta-theory for the $sd$ typing judgment, so that from $\Gamma, x : B \vdash_{sd} c : C'$ and $\Gamma, x : B \vdash C' \leq C$ and $\Gamma \vdash_{sd} (\Pi x : B. C) : S$ we can derive that $\Gamma \vdash_{sd} (\Pi x : B. C') : S$ holds. Unfortunately, this turns out to be infeasible: for the $sd$ judgment, the proofs of the Correctness of Types and Subject Reduction properties appear to be mutually dependent.

The problem with (sd-II-intro2) is that it suffers from the type-as-subject problem: the type in one premise occurs in the subject of another premise. For (sd-II-intro2) the type is $C$ in premise $\Gamma \vdash_{sd} c : C$, and the subject is $(\Pi x : B. C)$ in premise $\Gamma \vdash_{sd} (\Pi x : B. C) : S$. So we want to avoid the type-as-subject problem, and get rid of the premise $\Gamma \vdash_{sd} (\Pi x : B. C) : S$. This judgment has been derived with (sd-II-form), so we integrate this formation rule into (sd-II-intro2), which yields

\[
\frac{\Gamma, x : B \vdash_{sd} c : C \quad \Gamma \vdash_{sd} B : S_1 \quad \Gamma \vdash S_1 \leq s_1}{\Gamma \vdash_{sd} (\lambda x : B. c) : (\Pi x : B. C)}
\]

\[
\frac{\Gamma, x : B \vdash_{sd} c : C \quad \Gamma \vdash_{sd} B : S_2 \quad \Gamma, x : B \vdash S_2 \leq s_2}{\Gamma \vdash_{sd} (\lambda x : B. c) : (\Pi x : B. C)}
\]

\[
\frac{\Gamma, x : B \vdash_{sd} c : C \quad \Gamma \vdash_{sd} B : S_3 \quad \Gamma \vdash S_3 \leq s_3}{\Gamma \vdash_{sd} (\lambda x : B. c) : (\Pi x : B. C)}
\]

\[
(s_1, s_2, s_3) \in \mathcal{R}
\]
A proof of completeness by induction still fails, now because of the premise \( \Gamma, x : B \vdash \Gamma, x : S_2 \); also (sd-II-intro3) suffers from the type-as-subject problem. We get rid of the premise \( \Gamma, x : B \vdash \Gamma, x : C : S_2 \) by considering judgments of the form \( \Gamma \vdash a : A \) instead of \( \Gamma \vdash a : A \).

In other words, we compute not only the type of the term, but also its sort. The premises \( \Gamma, x : B \vdash \Gamma, x : C : C, \Gamma, x : B \vdash \Gamma, x : S_2 \) and \( \Gamma, x : B \vdash \Gamma, x : S_2 \leq s_2 \) are now combined into one premise \( \Gamma, x : B \vdash \Gamma, x : C : s_2 \). So the sd introduction rule is now:

\[
\frac{\Gamma, x : B \vdash \Gamma, x : C : s_2 \quad \Gamma \vdash \Gamma, x : S_1 : s_1' \quad \Gamma, x : B \vdash \Gamma, x : S_1 \leq s_1}{\Gamma \vdash \Gamma, x : B \vdash \Gamma, x : (\Pi x : B, C) : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R}
\]

This does not have the type-as-subject problem, and now the proof of completeness goes well. By IH on \( \Gamma, x : B \vdash c : C \) we have \( \Gamma, x : B \vdash c : C' \) : \( s \) and \( \Gamma, x : B \vdash C' \leq C \). By using the IH on the other premises, we can show that \( \Gamma \vdash \Gamma, x : B \vdash S_1 : s_1' \) and \( \Gamma, x : B \vdash S_1 \leq s_1 \) and \( (s_1, s_2, s_3) \in \mathcal{R} \) and \( s = s_2 \), so we have all the premises of the (sd-II-intro) rule. Summarizing, by the use of the extended sd judgments, we get for free that \( C \), the computed type of \( c \), has type \( s_2 \). By the integration of \( \Pi \) formation in (sd-II-intro), we can use \( C : s_2 \) to check that \( \Pi x : B, C \) holds.

So we have used two tricks:

1. The integration of the \( \Pi \)-formation rule into the \( \Pi \)-introduction rule.

2. Using sd typing judgments that work with triples (term, type and sort) instead of pairs.

The second trick was motivated by the problem still present after using the first trick. The reader might wonder if, however, the second trick alone is sufficient. This would lead to the following rule:

\[
\frac{\Gamma, x : B \vdash \Gamma, x : C : s' \quad \Gamma \vdash \Gamma, x : (\Pi x : B, C) : s : s'' \quad \Gamma \vdash \Gamma, x : S \leq s}{\Gamma \vdash \Gamma, x : B \vdash \Gamma, x : (\Pi x : B, C) : s} \quad (s_1, s_2, s_3) \in \mathcal{R}
\]

This rule suffers also from the type-as-subject problem, so we need both tricks.

The type-as-subject problem is solved in the algorithm of [Pol92] by replacing the premise \( \Gamma \vdash (\Pi x : B, C) : S \) by a simple syntactical condition on \( C \), and in [BJMP94] and [Sev96] by checking this condition in a slightly different derivation system. The type-as-subject problem occurs at a number of other places. In the proof of Minimal Typing of \( PTS \leq s \) we encountered it (page 244, problem 1). It also occurs when one tries to prove the so-called Expansion Postponement property [BJMP94, Pol98].

### 7.3.2 Syntax-Directed Rules

Most definitions of this section are adapted from [Pol93]. Before we give the typing rules, we specify for which systems the typing rules are meant.

**Definition 7.3.2.1** A \( PTS \leq \lambda(S, A, \mathcal{R}, S \leq, \mathcal{R} \leq) \) is bijective if it is functional and

\[
\begin{align*}
(s_1, s_2, s_3) \in \mathcal{R} \text{ and } (s_1, s'_2, s_3) \in \mathcal{R} & \implies s_2 \equiv s'_2 \\
(s_1, s_2, s_3) \in \mathcal{R} \leq \text{ and } (s_1, s'_2, s_3) \in \mathcal{R} \leq & \implies s_2 \equiv s'_2
\end{align*}
\]
Note that all systems where all rules \((\mathcal{R} \cup \mathcal{R}^\leq)\) have the form \((s_1, s_2, s_3)\) arc bijective. In particular, all our example systems in Section 7.1.3 are bijective.

**Convention:** In Section 7.3 we consider only bijective \(PTS^\leq_s\) which are \(SN_p\).

The restriction to strongly normalizing systems is necessary, since the sd subtyping rules expect the terms to be in normal form.

As indicated above, we need to extend the sorts and axioms to make sure every typable term has a sort. We do this by adding a sort \(\top\) which is the type of all hitherto untypable sorts. In this way, the terms that did not have a sort before, have sort \(\top\) now.

**Definition 7.3.2.2**

- \(S_\top = S \cup \top\).
- \(A_\top = A \cup \{(s: \top) \mid s \in S \land \neg \exists s', (s: s') \in A\}\).
- \(A_\top(s)\) is the unique \(s' \in S_\top\) for which \((s: s') \in A_\top\), if \(s \in S\). By the definition of \(A_\top\), the function \(A_\top(\_ )\) is total on \(S\). \(\Box\)

Our contexts are extended, so that not only the type of a variable, but also its sort is registered. Since ordinary variable declarations now consist of *triples* (variable, type and sort), we denote the set of these extended contexts by \(C_3\).

**Definition 7.3.2.3 (Extended pseudocontexts)**
The set of extended pseudocontexts \(C_3\) of a \(PTS^\leq\) \(\lambda(S, A, R, S^\leq, R^\leq)\) is defined by

- \(\epsilon \in C_3\),
- \(\Delta, x:A:s \in C_3\) if \(\Delta \in C_3, A \in T, s \in S_\top, x \in V\) and \(x \notin FV(\Delta) \cup FV(A)\),
- \(\Delta, x \leq a : A : s \in C_3\) if \(\Delta \in C_3, a \in T, A \in T, s \in S_\top, x \in V\) and \(x \notin FV(\Delta) \cup FV(a) \cup FV(A)\).

Here \(\epsilon\) denotes the empty context, and \(FV(\epsilon) = \emptyset, FV(\Delta, x:A:s) = FV(\Delta) \cup \{x\} \cup FV(A)\), and \(FV(\Delta, x \leq a : A : s) = FV(\Delta) \cup \{x\} \cup FV(a) \cup FV(A)\). \(\Box\)

**Convention:** \(s, s'\) etc. range over \(S_\top\) (and not just over \(S\)). \(\Delta\) ranges over \(C_3\).

The erasure of an extended pseudocontext throws away the sorts, so we obtain an ordinary context.

**Definition 7.3.2.4** The *erasure* of an extended pseudocontext \(|\Delta|\) is a context, defined as follows:

\[
\begin{align*}
|\epsilon| &= \epsilon \\
|\Delta, x:A:s| &= |\Delta|, x:A \\
|\Delta, x \leq a : A : s| &= |\Delta|, x \leq a : A
\end{align*}
\]
7.3. TYPE-CHECKING ALGORITHM

Now we give the syntax-directed rules for well-formedness of contexts and for typing. We have to integrate the subsumption rule with any premise $\Gamma \vdash a : A$ where certain demands on the type $A$ are applicable, by replacing this premise with other sd premises. This transformation depends on the particular demands on $A$, which fall into three categories. So each category can be considered as a transformation step from the original rules to sd rules.

1. The type $A$ must be a sort or a $\Pi$-type. But the sd rules deliver a minimal type, say $A'$, which might not have this form. Therefore we need a least supertype of $A'$ which does have this form. By Lemma 7.2.5.15 we obtain such a supertype by $wh_\beta \sigma$ reduction. So we replace the original premise $\Gamma \vdash a : A$ by $\Delta \vdash a : A' : s$ and $\Delta \vdash A' \gg wh_\beta \sigma \ A$.

2. The type $A$ is already determined by another premise. A minimal type of $a$, say $A'$, might be a subtype of $A$. So we replace the premises $\Gamma \vdash a : A$ by $\Delta \vdash a : A' : s$ and $\Delta \vdash \beta n(f(A')) \leq \beta n(f(A))$.

3. The type $A$ occurs in the term to be checked (in the conclusion). Just as in category 2, a minimal type $A'$ of $a$ might be a subtype of $A$. But we also need to check that $A$ is well-formed. (If we do not check this, $A$ might be untappable, and hence $A$ might be non $SN_\beta$.) Therefore we replace $\Gamma \vdash a : A$ by $\Delta \vdash a : A' : s$ and $\Delta \vdash A : S : s'$ and $\Delta \vdash \beta n(f(A')) \leq \beta n(f(A))$.

Definition 7.3.2.5 (Well-formedness of contexts)

\[
\begin{align*}
\text{(C-empty)} & \quad \frac{}{\Delta \vdash a : ok} \\
\text{(C-var)} & \quad \frac{\Delta \vdash a : ok \quad \Delta \vdash A : S : s' \quad \Delta \vdash S \gg wh_\beta \sigma \ s}{\Delta, x : A : s \vdash a : ok} \\
\text{(C-Bvar)} & \quad \frac{\Delta \vdash a : ok \quad \Delta \vdash A : A' : s \quad \Delta \vdash A : S : s' \quad \Delta \vdash \beta n(f(A')) \leq \beta n(f(A))}{\Delta, x \leq a : A : s \vdash a : ok} \\
\end{align*}
\]

Discussion The rules (C-var) and (C-Bvar) have a premise $\Delta \vdash a : ok$. Such a premise did not appear in the original rules. Here, this premise is necessary because the other premises, which are sd typing judgments, do not enforce well-formedness of the context (the original typing rules did).

- The premise $\Gamma \vdash A : s$ in the original (C-var) rule falls into category 1. By applying the corresponding transformation step, we obtain the sd (C-var) rule.
- The premise $\Gamma \vdash a : A$ in the original (C-Bvar) rule falls into category 3. By applying the corresponding transformation, we get amongst others premise $\Delta \vdash a : A' : s$ for the sd (C-Bvar) rule. Since this already determines the sort of $a$, we can ignore the original $\Gamma \vdash A : s$ premise.
Definition 7.3.2.6 (Unbounded typing rules)

\[
\begin{align*}
\text{(axiom)} & \quad \frac{\Delta \vdash_{sd} s_1 : s_2 : A_T(s_2)}{(s_1 : s_2) \in A} \\
\text{(var)} & \quad \frac{\Delta \vdash_{sd} x : A : s}{x : A : s \in \Delta} \\
\text{(II-form)} & \quad \frac{\Delta \vdash_{sd} A : S_1 : s' \vdash \Delta \vdash S_1 \Rightarrow_{wh,\beta_\sigma} s_1}{\Delta, x : A : s_1 \vdash_{sd} B : S_2 : s' \vdash \Delta, x : A : s_2 \Rightarrow_{wh,\beta_\sigma} s_2} \\
& \quad \frac{\Delta \vdash_{sd} (\Pi x : A. B) : s_3 : A_T(s_3)}{(s_1, s_2, s_3) \in \mathcal{R}} \\
\text{(II-intro)} & \quad \frac{\Delta \vdash_{sd} A : S_1 : s_1 \vdash \Delta \vdash S_1 \Rightarrow_{wh,\beta_\sigma} s_1}{\Delta, x : A : s_1 \vdash_{sd} b : B : s_2} \\
& \quad \frac{\Delta \vdash_{sd} (\lambda x : A. b) : (\Pi x : A. B) : s_3}{(s_1, s_2, s_3) \in \mathcal{R}} \\
\text{(II-elim)} & \quad \frac{\Delta \vdash_{sd} b : B' : s_3 \vdash \Delta \vdash B' \Rightarrow_{wh,\beta_\sigma} (\Pi x : A. B)}{\Delta \vdash_{sd} a : A' : s_1 \vdash \Delta \vdash b : B[x := a] : s_2} \\
& \quad \frac{\Delta \vdash_{sd} \beta nf(A') \leq \beta nf(A)}{(s_1, s_2, s_3) \in \mathcal{R}}
\end{align*}
\]

\[
\begin{align*}
\text{Definition 7.3.2.7 (Bounded typing rules)}
\text{(Bvar)} & \quad \frac{\Delta \vdash_{sd} x : A : s}{x \leq a : A : s \in \Delta} \\
\text{(BII-form)} & \quad \frac{\Delta \vdash_{sd} A : S_1 : s_1 \vdash \Delta \vdash b : B[x := a] : s_2 \vdash \Delta \vdash_{sd} \beta nf(A') \leq \beta nf(A)}{\Delta \vdash_{sd} (\Pi x \leq a : A. B) : s_3 : A_T(s_3)} \\
& \quad \frac{\Delta \vdash_{sd} A : S_1 : s_1 \vdash \Delta, x \leq a : A : s_1 \vdash_{sd} B : S_2 : s_2}{(s_1, s_2, s_3) \in \mathcal{R}_\leq} \\
\text{(BII-intro)} & \quad \frac{\Delta \vdash_{sd} A : S_1 : s_1 \vdash \Delta, x \leq a : A : s_1 \vdash_{sd} b : B : s_2}{\Delta \vdash_{sd} (\lambda x \leq a : A. b) : (\Pi x \leq a : A. B) : s_3} \\
& \quad \frac{\Delta \vdash_{sd} b : B' : s_3 \vdash \Delta \vdash B' \Rightarrow_{wh,\beta_\sigma} (\Pi x \leq a : A. B)}{\Delta \vdash_{sd} a' : A' : s_1 \vdash \Delta \vdash b : B[x := a'] : s_2} \\
& \quad \frac{\Delta \vdash_{sd} \beta nf(A') \leq \beta nf(A)}{(s_1, s_2, s_3) \in \mathcal{R}_\leq}
\end{align*}
\]

\[
\begin{align*}
\text{Discussion} & \quad \text{As we already announced in the introduction, the typing rules do not check that the context is well-formed. This reveals itself in the sd rules (var), (axiom) and (Bvar), where no premise } \Delta \vdash_{sd} ok \text{ appears. Now we turn to the individual rules, which are obtained from the original ones by applying the transformation steps given above.}
\end{align*}
\]

- In the sd (axiom) rule, we compute the type of } s_2 \text{ (and hence, the sort of } s_1), \text{ by applying the function } A_T \text{ to } s_2.
- The sd (II-form) rule is obtained from the original (II-form) rule by applying transformation step 1 twice.
7.3. TYPE-CHECKING ALGORITHM

- The sd (II-intro) rule is discussed in the introduction. Note it can be considered as the application of transformation step 1 to the following reformulation of the original (II-intro) rule:

\[
\begin{align*}
\text{(II-intro') } & \quad \Gamma, x : A \vdash b : B \quad \Gamma, x : A \vdash B : s_2 \quad \Gamma \vdash A : s_1 \\
& \quad \frac{}{\Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B)} (s_1, s_2, s_3) \in \mathcal{R}
\end{align*}
\]

where the demand \( \Gamma \vdash B : s_2 \) is ignored, since it is already covered by \( \Delta \vdash b : B : s_2 \).

- The sd (II-elim) rule is obtained by applying transformation step 1 on premise \( \Gamma \vdash b : B \) and step 2 on premise \( \Gamma \vdash a : A \). Note that \( s_2 \) is uniquely determined by \( s_1 \) and \( s_3 \) since the \( PTS \subseteq \) is bijective.

- The sd (BII-form) rule is obtained by applying transformation steps 1 and 3.

- The sd (BII-intro) rule is obtained in a similar way as the sd (II-intro) rule, but now we use transformation step 3.

- The sd (BII-elim) rule is obtained by applying transformation steps 1 and 2. Again, the sort \( s_2 \) is uniquely determined by \( s_1 \) and \( s_3 \) since the \( PTS \subseteq \) is bijective.

We can show right away that the sd typing rules give a unique type and sort.

**Lemma 7.3.2.8 (Uniqueness of Types for \( \vdash_{sd} \))**

If \( \Gamma \vdash_{sd} a : A \vdash_{sd} a : A' \vdash_{sd} s' \) then \( A \equiv A' \) and \( s \equiv s' \). (Actually both judgments even have the same derivation, if we ignore the subtyping derivations, but we do not need this.)

**Proof:** By induction on the structure of \( a \). Most cases are straightforward, using the IH, since only one rule can be used considering the shape of \( a \). The exceptions are \( a \equiv x \) (both (var) and (Bvar) could be the last rule in a derivation) and \( a \equiv b \) (both (II-elim) and (BII-elim) could be used). By simple arguments we can show that in both judgments the same rule must be used, and the same type and sort is assigned to \( a \).

**Definition 7.3.2.9 (Subtyping rules)**

\[
\begin{align*}
(\leq\text{-refl}) & \quad \Gamma \vdash_{sd} b \leq b \\
(\leq\text{-II}) & \quad \Gamma \vdash_{sd} A' \leq A \quad \Gamma, x : A' \vdash_{sd} B \leq B' \\
& \quad \frac{}{\Gamma \vdash_{sd} (\Pi x : A. B) \leq (\Pi x : A'. B')}
\end{align*}
\]

\[
\begin{align*}
(\leq\text{-BII}) & \quad \Gamma, x \leq a : A \vdash_{sd} B \leq B' \\
& \quad \frac{}{\Gamma \vdash_{sd} (\Pi x \leq a : A. B) \leq (\Pi x \leq a : A. B')}
\end{align*}
\]

\[
\begin{align*}
(\leq\text{-\lambda}) & \quad \Gamma, x : A \vdash_{sd} b \leq b' \\
& \quad \frac{}{\Gamma \vdash_{sd} (\lambda x : A. b) \leq (\lambda x : A. b')}
\end{align*}
\]

\[
(\leq\text{-transvar}) \quad \Gamma \vdash_{sd} \beta n(a_{c_1} \ldots c_n) \leq b \\
& \quad \frac{}{\Gamma \vdash_{sd} x_{c_1} \ldots c_n \leq b} \quad x : a \vdash \Gamma, n \geq 0
\]

**Discussion** As we have announced in the introduction, we remove the \( (\leq\text{-red}) \) rule, and assume \( \Gamma \vdash_{sd} a \leq b \) is only called when \( a \) and \( b \) are in \( \beta \) normal-form. The \( (\leq\text{-transvar}) \) rule
has to be adapted, so that this is kept invariant. Since we do not need to develop meta-theory for the sd subtyping judgment, and this judgment will only be used with well-typed terms, the original ($\leq$-Happ) rule is not necessary any more.

### 7.3.3 Soundness

The proof of soundness of the syntax-directed rules proceeds in three stages:

1. soundness of the subtyping judgment (Lemma 7.3.3.1),

2. soundness of the typing judgment (Lemma 7.3.3.6), and

3. soundness of the well-formedness judgment (Lemma 7.3.3.7).

When proving soundness of the typing judgment, we are confronted with the following problem. The typing rules for $\Delta \vdash_{sd} a : A : s$ do not check the extended context $\Delta$ is well-formed. But we do need this proviso, since the original typing rules do check the context is well-formed. We cannot express $\Delta$ is well-formed by $\Delta \vdash_{sd} \text{ok}$ here, since we do not know the soundness of $\Delta \vdash_{sd} \text{ok}$ yet. Therefore we introduce a new kind of judgment, $\Delta \vdash_3 \text{ok}$, that says $\Delta$ is well-formed, using the original typing rules.

**Lemma 7.3.3.1 ($\leq$-Soundness)** If $\Gamma \vdash_{sd} a \leq b$ then $\Gamma \vdash a \leq b$.

**Proof:** By straightforward induction on the derivation. \[\square\]

**Definition 7.3.3.2** The predicate $\vdash_3 \text{ok}$ on extended contexts is defined as follows:

- **(C-empty)**
  
  \[
  \frac{}{\varepsilon \vdash_3 \text{ok}}
  \]

- **(C-var)**
  
  \[
  \frac{\Delta \vdash_3 \text{ok} \quad |\Delta| \vdash A : s}{\Delta, x : A : s \vdash_3 \text{ok}}
  \]

- **(C-Bvar)**
  
  \[
  \frac{\Delta \vdash_3 \text{ok} \quad |\Delta| \vdash a : A \quad |\Delta| \vdash A : s}{\Delta, x \leq a : A : s \vdash_3 \text{ok} \quad s \in S^\leq}
  \]

The next lemma states some simple properties of this judgment.

**Lemma 7.3.3.3**

- If $\Delta \vdash_3 \text{ok}$ then $|\Delta| \vdash \text{ok}$.

- If $\Delta, \Delta' \vdash_3 \text{ok}$ then $\Delta \vdash_3 \text{ok}$.

**Proof:** Both by straightforward induction on the derivation. \[\square\]

The following two lemmas are convenient for proving soundness of the typing judgment.

**Lemma 7.3.3.4** If $\Gamma \vdash \text{ok}$ and $s \in S$ then $\Gamma \vdash s : A_\tau(s)$ or $(s:A_\tau(s)) \in A_\tau \setminus A$.

**Proof:** Easy. \[\square\]
Lemma 7.3.3.5 If $\Gamma \vdash A \leq B$ and $\Gamma \vdash A : s$ and $\Gamma \vdash B : C$ then $\Gamma \vdash B : s$.

Proof: By Lemma 7.2.5.6 $\Gamma \vdash C \cup s$, so by Lemma 7.2.5.4 $\Gamma \vdash C \leq s$, so by Lemma 7.2.5.15 $\Gamma \vdash C \Rightarrow_{wh\beta\sigma} s$. In combination with $\Gamma \vdash B : C$ Type Reduction for $wh\beta\sigma$ (Lemma 7.2.5.17) gives $\Gamma \vdash B : s$. \qed

Lemma 7.3.3.6 (Soundness of Typing) If $\Delta \vdash a : A : s$ and $\Delta \vdash ok$ then $|\Delta| \vdash a : A$ and $|\Delta| \vdash A : s$ or $(A : s) \in A_T \setminus A$.

Proof: By induction on the $\vdash_{td}$ derivation. We treat here two cases, for which the proof is long, viz. (BII-form) and (BII-elim).

(BII-form): We have $a \equiv \Pi x \leq b : B, C, A \equiv s_3$ and $s \equiv A_T(s_3)$. The premises of the rule are:

1. $\Delta \vdash_{td} b : B' : s_1$
2. $\Delta \vdash_{td} B : S_1 : s_1'$
3. $|\Delta| \vdash_{td} \betanf(B') \leq \betanf(B)$
4. $\Delta, x \leq b : B : s_1 \vdash_{td} C : S_2 : s_2'$
5. $|\Delta|, x \leq b : B \vdash S_2 \Rightarrow_{wh\beta\sigma} s_2$
6. $(s_1, s_2, s_3) \in R^S$

By IH on 1

7. $|\Delta| \vdash b : B'$
8. $|\Delta| \vdash B' : s_1$ or $(B' : s_1) \not\in A_T \setminus A$

By 6 we know $s_1 \in S$, so $(B' : s_1) \not\in A_T \setminus A$. Hence gives 8 gives

9. $|\Delta| \vdash B' : s_1$

By IH on 2

10. $|\Delta| \vdash B : S_1$

By $\leq$-Soundness (Lemma 7.3.3.1) and $\leq$-Conversion-closed (Theorem 7.2.2.6) on 3

11. $|\Delta| \vdash B' \leq B$

By Lemma 7.3.3.5 on 10, 9 and 11

12. $|\Delta| \vdash B : s_1$

By (subsum) on 7, 11 and 12

13. $|\Delta| \vdash b : B$

By 6

14. $s_1 \in S^S$

By 13, 12, 14 and the assumption $\Delta \vdash ok$ we have

15. $\Delta, x \leq b : B : s_1 \vdash ok$

Now IH on 4

16. $|\Delta|, x \leq b : B \vdash C : S_2$

By Subject Reduction for $wh\beta\sigma$ (Lemma 7.2.5.16) on 16 and 5

17. $|\Delta|, x \leq b : B \vdash C : s_2$

By (BII-form) on 12, 17 and 6

18. $|\Delta| \vdash (\Pi x \leq b : B, C) : s_3$, i.e. $|\Delta| \vdash a : A$

By Lemma 7.2.4.1
(19) \(|\Delta| \vdash \text{ok}\)

By Lemma 7.3.3.4 on 19 and \(s_3 \in \mathcal{S}\)

(20) \(|\Delta| \vdash s_3 : A_\top(s_3)\) or \((s_3 : A_\top(s_3)) \in A_\top \setminus A\)

By 18 and 20 we are done with this case.

(BII-elim): We have \(a \equiv c \ b', A \equiv C[x := b']\) and \(s \equiv s_2\). The premises of the rule are:

1. \(\Delta \vdash c : C'\)
2. \(|\Delta| \vdash C' \supset \text{wh}_\beta\sigma\) \((\Pi x \leq b : B. C)\)
3. \(\Delta \vdash b' : B'\)
4. \(|\Delta| \vdash b' \leq \beta \text{nf}(B)\)
5. \((s_1, s_2, s_3) \in \mathcal{R}^\leq\)
6. \(|\Delta| \vdash b' \leq \beta \text{nf}(b)\)

By III on 1

7. \(|\Delta| \vdash c : C'\)
8. \(|\Delta| \vdash C' : s_3\) or \((C':s_3) \in A_\top \setminus A\)

By Type Reduction for wh\(\beta\)\(\sigma\) (Lemma 7.2.5.17) on 7 and 2

9. \(|\Delta| \vdash c : (\Pi x \leq b : B. C)\)

By 5 \(s_3 \in \mathcal{S}\), so \((C':s_3) \notin A_\top \setminus A\), so by 8

10. \(|\Delta| \vdash C' : s_3\)

By Subject Reduction for wh\(\beta\)\(\sigma\) (Lemma 7.2.5.16) on 10 and 2

11. \(|\Delta| \vdash (\Pi x \leq b : B. C) : s_3\)

By Generation (Lemma 7.2.4.8) on 11

12. \(|\Delta| \vdash B : s_3'\)
13. \(|\Delta| \vdash x \leq b : B \vdash C : s_3''\)
14. \(|\Delta| \vdash s_3'' \leq s_3\)
15. \((s_3', s_3'', s_3') \in \mathcal{R}^\leq\)

By IH on 3

16. \(|\Delta| \vdash b' : B'\)
17. \(|\Delta| \vdash B' : s_1\) or \((B':s_1) \notin A_\top \setminus A\)

By 5 \(s_1 \in \mathcal{S}\), so \((B':s_1) \notin A_\top \setminus A\), so by 17

18. \(|\Delta| \vdash B' : s_1\)

By \(\leq\)-Soundness (Lemma 7.3.3.1) and \(\leq\)-Conversion-closed (Theorem 7.2.2.6) on 4

19. \(|\Delta| \vdash B' \leq B\)

By Lemma 7.2.5.6 on 19, 18 and 12, and using Lemma 7.2.5.4

20. \(s_1 \equiv s_1'\)

By \(\leq\)-Generation (Lemma 7.2.2.7) on 14

21. \(s_3 \equiv s_3'\)

By bijectivity of the \(\text{PTS}\) on 5, 15, 20 and 21

22. \(s_2 \equiv s_2'\)

By (subsum) on 16, 19 and 12

23. \(|\Delta| \vdash b' : B\)

By \(\leq\)-Soundness (Lemma 7.3.3.1) and \(\leq\)-Conversion-closed (Theorem 7.2.2.6) on 6
7.3. TYPE-CHECKING ALGORITHM

(24) $|\Delta| \vdash b' \leq b$
By (BII elim) on 9, 24 and 23

(25) $|\Delta| \vdash c \ b' : C[x := b']$, i.e. $\Gamma \vdash a : A$
By 13 and 22

(26) $|\Delta|, x \leq b : B \vdash C : s_2$
By Bounded Substitution on 26, 24 and 23

(27) $|\Delta| \vdash C[x := b'] : s_2$, i.e. $\Gamma \vdash A : s$

By 25 and 27 we are done with this case.

Lemma 7.3.3.7 (ok-Soundness) If $\Delta \vdash a$ ok then $\Delta \vdash a$ ok.

Proof: By induction on the derivation, using Soundness of Typing (Lemma 7.3.3.6).

Theorem 7.3.3.8 (Soundness) If $\Delta \vdash a : A : s$ and $\Delta \vdash a$ ok then $|\Delta| \vdash a : A$.

Proof: By ok-Soundness and Soundness of Typing (Lemmas 7.3.3.7 and 7.3.3.6).

7.3.4 Completeness

The proof of completeness of the syntax-directed rules proceeds in three stages:

1. completeness for the subtyping judgment,
2. completeness for the typing judgment, and
3. completeness for the well-formedness judgment.

The next lemma helps proving $\leq$-Completeness.

Lemma 7.3.4.1 If $\Gamma \vdash a \leq b$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ then $\Gamma \vdash a \leq b$.

Proof: By straightforward induction on the derivation.

Lemma 7.3.4.2 ($\leq$-Completeness)
If $\Gamma \vdash a \leq b$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ then $\Gamma \vdash \betanf(a) \leq \betanf(b)$.

Proof: By induction on the subtyping derivation, using the limitation to $PTS \leq s$ which are $SN\beta$. Lemma 7.3.4.1 is necessary for cases ($\leq$-II) and ($\leq$-BII). Note that by Lemma 7.2.4.13 all terms in premises are also well-typed in their context.

We have to take the normal-forms of $a$ and $b$ in the sd judgment; for a counterexample take $a$ and $b$ convertible but not syntactically equal: then $\Gamma \vdash a \leq b$ but not $\Gamma \vdash a \leq b$. The restriction to well-typed terms and well-formed contexts ensures we can take the normal-form, since we are in a $PTS \leq s$ which is strongly normalizing.

The following three lemmas are useful for proving Completeness of Typing.

Lemma 7.3.4.3 If $\Gamma \vdash A \leq A'$ and $(A : s) \in A \setminus A$ then not $\Gamma \vdash A' : B$. 
Proof: The assumption \((A:s) \in A_T \setminus A\) implies that \(A \in S\) and \(\neg \exists s', (s:s') \in A\). Since \(A\) is a sort and \(\Gamma \vdash A \leq A'\) we have by \(\leq\)-Generation (Lemma 7.2.2.7) that \(A =_\beta A'\), so \(A' \trianglerighteq _\beta A\). Suppose towards a contradiction that \(\Gamma \vdash A' : B\). By Subject Reduction \(\Gamma \vdash A : B\), and by Generation (Lemma 7.2.4.8) \((A:s'') \in A\) for some \(s''\). This is in contradiction with our assumption.

\[\square\]

**Lemma 7.3.4.4** If \(\Delta \trianglerighteq _{\text{ld}} A : S : s'\) and \(|\Delta| \vdash S \trianglerighteq _{\text{whb} \sigma} s\) and \(\Delta \trianglerighteq _{\text{h}} ok\) then \(|\Delta| \vdash A : s\).

**Proof:** By Soundness of Typing and Type Reduction for \(\text{whb} \sigma\) (Lemma 7.2.5.17).

\[\square\]

**Lemma 7.3.4.5** If \(\Delta \trianglerighteq _{\text{h}} ok\) and \(\Delta \trianglerighteq _{\text{ld}} a : A : s'\) and \(|\Delta| \trianglerighteq _{\text{ld}} bnf(A) \leq bnf(B)\) and \(|\Delta| \vdash B : s\) then \(s \equiv s'\) and hence \(\Delta \trianglerighteq _{\text{ld}} a : A : s\).

**Proof:** By \(\leq\)-Soundness (Lemma 7.3.3.1) and \(\leq\)-Conversion-closed (Theorem 7.2.2.6) we have \(|\Delta| \vdash A \leq B\). By Soundness of Typing (Lemma 7.3.3.6) on \(\Delta \trianglerighteq _{\text{ld}} a : A : s'\) we have \((A:s') \in A_T \setminus A\) or \(|\Delta| \vdash A : s'\). The former case leads to a contradiction using Lemma 7.3.4.3. So we have \(|\Delta| \vdash A : s'\). Now use the assumption \(|\Delta| \vdash B : s\) and Lemma 7.2.5.6 to show \(|\Delta| \vdash s' \cup s\) and by Lemma 7.2.5.4 \(s \equiv s'\).

\[\square\]

The next lemma is Completeness of Typing. We have the proviso that the extended context is well-formed.

**Lemma 7.3.4.6** (Completeness of Typing)
If \(|\Delta| \vdash a : A\) and \(\Delta \trianglerighteq _{\text{h}} ok\) then \(\Delta \trianglerighteq _{\text{ld}} a : B : s\) for some \(B, s\) and \(|\Delta| \trianglerighteq _{\text{ld}} bnf(B) \leq bnf(A)\).

**Proof:** By induction on the derivation of \(|\Delta| \vdash a : A\). We treat here only the case (\(\Pi\)-intro). We have \(a \equiv \lambda x : C. d\) and \(A \equiv \Pi x : C. D\). The premises of the rule are:

1. \(|\Delta|, x : C \vdash d : D\)
2. \(|\Delta| \vdash (\Pi x : C. D) : s_3\)

By IH on 2 and \(\Delta \trianglerighteq _{\text{h}} ok\)

3. \(\Delta \trianglerighteq _{\text{ld}} (\Pi x : C. D) : S_3 : s_3'\)
4. \(|\Delta| \trianglerighteq _{\text{ld}} bnf(S_3) \leq bnf(s_3)\)

By \(\leq\)-Soundness and \(\leq\)-Conversion-closed (Theorem 7.2.2.6)

5. \(|\Delta| \vdash S_3 \leq s_3\)

The sd (\(\Pi\)-form) rule must be used to derive 3, so

6. \(\Delta \trianglerighteq _{\text{ld}} C : S_1 : s_1'\)
7. \(|\Delta| \vdash S_1 \trianglerighteq _{\text{whb} \sigma} s_1\)
8. \(\Delta, x : C : s_1 \trianglerighteq _{\text{ld}} D : S_2 : s_2'\)
9. \(|\Delta|, x : C : S_2 \trianglerighteq _{\text{whb} \sigma} s_2\)
10. \((s_1, s_2, S_3) \in R\)

By Lemma 7.3.4.4 on \(\Delta \trianglerighteq _{\text{h}} ok\), 6 and 7

11. \(|\Delta| \vdash C : S_1\)
12. \(|\Delta| \vdash C : S_1 \trianglerighteq _{\text{h}} ok\)

By 12 we can use IH on 1, so for some \(s'\)
• The conventions concerning meta-variables apply also to the programming variables in the algorithms.

• We ignore $\alpha$-conversion. In an actual implementation, we have to explicitly deal with it, of course.

• $M$ and $N$ is the same as $\text{let } 0k \equiv M \text{ in } N$. So there is a definite order in the evaluation of $M$ and $N$ (this construct is similar to the conditional and found in some programming languages).

• Mathematical operations like $\equiv$ and $\in$ deliver $0k$ or Fail.

• The algorithm $\text{wh}\beta\sigma\text{nf}(\Gamma; a)$ computes the $\text{wh}\beta\sigma\text{nf}$ of $a$ in $\Gamma$ (by Lemma 7.2.5.13 this normal form is unique). This algorithm may not terminate. Similarly, we assume an algorithm $\beta\text{nf}(\_)$.

Now we give the definition and specification for the subtyping, typing and well-formedness algorithms.

**Definition 7.3.5.1** The subtyping algorithm Sub gets as argument a pseudocontext $\Gamma$ and pseudoterms $a$ and $b$, and returns either $0k$ or Fail, or it does not terminate. The definition closely follows the sd subtyping rules in Definition 7.3.2.9. The algorithm is defined as follows.

### 7.3. TYPE-CHECKING ALGORITHM

(13) $\Delta, x : C : s_1 \vdash_d d : D' : s'$

(14) $|\Delta|, x : C \vdash_d \beta\text{nf}(D') \leq \beta\text{nf}(D)$

By Lemma 7.3.4.4 on 8 and 9

(15) $|\Delta|, x : C \vdash D : s_2$

Now we are going to use the fact that the sd typing rules work with triples: by 13 the type of $D'$ is available. By the argument given in Lemma 7.3.4.5 we know that this type is equal to $s_2$. Because of the integration of $\Pi$-formation in ($\Pi$-intro), we can use $D' : s_2$ to check that the $\Pi$-type is well-formed.

By Lemma 7.3.4.5 on 12, 13, 14 and 15 we have $s' \equiv s_2$ and hence

(16) $\Delta, x : C : s_1 \vdash_d d : D' : s_2$

By 10 $S_3 \equiv S$, so by $\leq$-Generation on 5

(17) $S_3 \equiv s_3$

And hence by 10

(18) $(s_1, s_2, s_3) \in R$

Now by sd (II-intro) on 6, 7, 16 and 18

(19) $\Delta \vdash_d (\lambda x : C \cdot d) : (\Pi x : C \cdot D') : s_3$

So we take $B \equiv \Pi x : C \cdot D'$ and $s \equiv s_3$. Now we still have to show $|\Delta|, \vdash_d \beta\text{nf}(B) \leq \beta\text{nf}(A)$, i.e. $|\Delta| \vdash_d \beta\text{nf}(\Pi x : C \cdot D') \leq \beta\text{nf}(\Pi x : C \cdot D)$.

By 11 $\beta\text{nf}(C)$ exists, so by Lemma 7.3.4.1 on 14

(20) $|\Delta|, x : \beta\text{nf}(C) \vdash_d \beta\text{nf}(D') \leq \beta\text{nf}(D)$

By $(\leq\text{-refl})$

(21) $|\Delta|, \vdash_d \beta\text{nf}(C) \leq \beta\text{nf}(C)$

By $(\leq\text{-II})$ on 21 and 20
the constructs. This is important to understand the algorithms correctly.

- Each algorithm can deliver Fail to indicate that the corresponding property does not hold. All constructs propagate Fail.

- An important construct is case. The meaning of this construct is the same as in functional programming languages: the body of the case-construct consists of a list of cases, with a pattern possibly containing new programming variables, and a value in which these variables are bound. If none of the cases match with the given term, Fail is delivered as result.

- let $p \equiv t$ in $M$ is the same as

  ```
  case t of
    p then M
  endcase.
  ```

So first $t$ is evaluated, and if it fails or does not match with $p$, the whole let expression fails.

- The construct “determine $v$ such that $P$ in $M$” tries to find values for the programming variable(s) $v$ such that $P$ holds. If $P$ does not hold for any value, Fail is returned. This construct is in principle indeterministic and incomputable, but we'll see that in our algorithms, we can effectively compute whether such a $v$ exists, and if so, what its unique value is.
7.3. TYPE-CHECKING ALGORITHM

Note that the order of the let's in each case is chosen in such a way, that each term is only reduced after it is type-checked. For example, in the case $c \equiv \lambda z : a . b$ the second line let $(s_1, s') \equiv \text{Typ}(\Delta; A)$ in... type-checks $A$, before the normal form of $A$ is taken in the third line $\text{Sub}( \Delta; \beta \text{nf}(A'); \beta \text{nf}(A))$.

All determine constructions are deterministic by the definition of extended pseudocontext and by the bijectivity of the $PTS^S$ we consider here.

Lemma 7.3.5.4 $\Delta \vdash_{rd} a : A : s \iff \text{Typ}(\Delta; a) = (A, s)$.

Proof: Both implications are proved by straightforward induction. Use Lemma 7.3.5.2. □

Definition 7.3.5.5 The well-formedness algorithm $\mathsf{Wf}$ gets as argument an extended pseudocontext $\Delta$, and returns either $\mathsf{Ok}$ or $\mathsf{Fail}$, or it does not terminate. The definition closely follows the sd well-formedness rules in Definition 7.3.2.5. It is defined as follows.

\[
\begin{align*}
\mathsf{Wf}(\Delta') \equiv \\
\begin{cases} 
\epsilon & \text{then } \mathsf{Ok} \\
\Delta, x : A : s & \text{then } \mathsf{Wf}(\Delta) \text{ and} \\
\text{let } (s, s') \equiv \text{Typ}(\Delta; A) \text{ in} \\
s \equiv \text{wh} \beta \text{nf}(|\Delta|; S) \\
\Delta, x \leq a : A : s_1 & \text{then } \mathsf{Wf}(\Delta) \text{ and} \\
\text{let } (A', s_2) \equiv \text{Typ}(\Delta; a) \text{ in} \\
\text{let } (s, s') \equiv \text{Typ}(\Delta; A) \text{ in} \\
\text{Sub}( |\Delta|; \beta \text{nf}(A'); \beta \text{nf}(A)) \text{ and} \\
s_1 \equiv s_2 \text{ and } s_1 \in S^S 
\end{cases}
\end{align*}
\]

endcase □

Lemma 7.3.5.6 $\Delta \vdash_{sd} \mathsf{Ok} \iff \mathsf{Wf}(\Delta) = \mathsf{Ok}$.

Proof: Both implications are proved by straightforward induction. Use Lemmas 7.3.5.2 and 7.3.5.4. □

7.3.6 Termination

In this section we show that the algorithms given in the previous section always terminate for a certain class of $PTS^S$. An important corollary is that typing is decidable for this class. For typing and well-formedness termination are relatively straightforward. In order to prove termination of the subtyping algorithm we need to find a well-founded and decreasing measure in the recursion.

A simple measure like the sum of the size of the terms does not work, because of the last case in the subtyping algorithm: the term $x \ c_1 \ldots c_n$ is replaced in the recursive call by $\beta \text{nf}(a \ c_1 \ldots c_n)$, where $a$ is the bound of $x$. In the process of replacing a variable by its bound and $\beta$-normalizing, the term can get bigger. Therefore we introduce $\sigma$-reduction, an extension of $\text{wh}$ with some compatibility rules, and use as measure a combination of the size of the terms and the maximum number of $\beta \sigma$ reduction steps the terms can make. In the case above, this maximum number of $\beta \sigma$ steps decreases, because replacing a variable...
by its bound is one \(\sigma\) step, and \(\beta\)-normalizing leads to zero or more \(\beta\) steps. This approach is also used in [PS97] for \(F^\infty\) and in [Che97] for \(\lambda S^\le\). The final result is that all bijective \(PTS^\le\)s which are \(SN_{\beta\sigma}\) have a terminating subtyping algorithm, and hence these systems have decidable typing.

**Definition 7.3.6.1 (\(\sigma\)-reduction)** The \(\sigma\)-reduction relation \(\vdash \rightarrow_{\sigma} \subseteq \Gamma \times T \times T\) is the smallest relation such that

- \(\Gamma_1, x \leq a : A, \Gamma_2 \vdash x \rightarrow_{\sigma} a,\)

and that is closed under the following compatibility rules

- if \(\Gamma, x \leq a : A \vdash b \rightarrow_{\sigma} b'\) then \(\Gamma \vdash (\Pi x \leq a : A. b) \rightarrow_{\sigma} (\Pi x \leq a : A. b')\)

- if \(\Gamma, x : A \vdash b \rightarrow_{\sigma} b'\) then \(\Gamma \vdash (\Pi x : A. b) \rightarrow_{\sigma} (\Pi x : A. b')\),
  \(\Gamma \vdash (\lambda x : A. b) \rightarrow_{\sigma} (\lambda x : A. b')\)

- if \(\Gamma \vdash a \rightarrow_{\sigma} a'\) then \(\Gamma \vdash (a b) \rightarrow_{\sigma} (a' b)\)
  \(\Gamma \vdash (\Pi x : a. b) \rightarrow_{\sigma} (\Pi x : a'. b)\).

\(\Gamma \vdash a \rightarrow_{\beta\sigma} b\) is \(\Gamma \vdash a \rightarrow_{\sigma} b\) or \(a \rightarrow_{\beta} b\). \(\Gamma \vdash \rightarrow_{\beta\sigma}\) is the reflexive and transitive closure of \(\Gamma \vdash \rightarrow_{\beta\sigma}\).

Note that \(\rightarrow_{\sigma}\) is an extension of \(\rightarrow_{\whe}\), and we do not have all the compatibility rules. We have only those (compatibility) rules for \(\rightarrow_{\sigma}\) that correspond to subtyping rules. For example, we have \(\Gamma \vdash a \rightarrow_{\sigma} a' \implies \Gamma \vdash (a b) \rightarrow_{\sigma} (a' b)\) because of the \((\leq\text{-app})\) rule; we do not have \(\Gamma \vdash (b a) \rightarrow_{\sigma} (b a')\) because we have no subtyping rule with conclusion \(\Gamma \vdash (b a) \leq (b a')\). We conjecture that our proof of termination would also work with the full set of compatibility rules, but we rather keep \(\rightarrow_{\sigma}\) as small as possible to make the notion \(SN_{\beta\sigma}\) as weak as possible.

**Definition 7.3.6.2**

- A term \(a\) is in \(\beta\sigma\ nf\) \(\Gamma\), if there is no \(b\) such that \(\Gamma \vdash a \rightarrow_{\beta\sigma} b\).

- A term \(a\) is \(SN_{\beta\sigma}\) in \(\Gamma\) if every \(\beta\sigma\) reduction path starting with \(a\) in \(\Gamma\) is finite.

- A \(PTS^\le\ \lambda S^\le\) is \(SN_{\beta\sigma}\) if \(a\) is \(SN_{\beta\sigma}\) in \(\Gamma\) for every context \(\Gamma\) and term \(a\) typable in \(\Gamma\) in \(\lambda S^\le\).

- \(\text{steps}_{\Gamma}(a)\) is the maximum number of steps of \(a\) to reach a \(\beta\sigma\ nf\) \(\Gamma\). This function is only defined if \(a\) is \(SN_{\beta\sigma}\) in \(\Gamma\).

The next lemma proves properties of steps that are necessary for the termination of the subtyping algorithm.

**Lemma 7.3.6.3**

1. If \(a\) is \(SN_{\beta\sigma}\) in \(\Gamma\) and \(\Gamma \vdash a \rightarrow_{\beta\sigma} b\) in \(n\) steps, then \(\text{steps}_{\Gamma}(a) \geq n + \text{steps}_{\Gamma}(b)\)

2. \(\text{steps}_{\Gamma}(A) \leq \text{steps}_{\Gamma}(\Pi x : A. B)\)

3. \(\text{steps}_{\Gamma, x : A}(B) \leq \text{steps}_{\Gamma}(\Pi x : A. B)\)
7.3. TYPE-CHECKING ALGORITHM

4. \(
\text{steps}_{\text{\mathcal{T}}, x \leq a : A}(B) \leq \text{steps}_{\text{\mathcal{T}}}(\Pi x \leq a : A. B)
\)

5. \(
\text{steps}_{\text{\mathcal{T}}, x = A}(b) \leq \text{steps}_{\text{\mathcal{T}}}(\lambda x : A. b)
\)

6. \(
\text{steps}_{\text{\mathcal{T}}, x : A}(B) = \text{steps}_{\text{\mathcal{T}}, x : A'}(B)
\)

7. \(
\text{steps}_{\text{\mathcal{T}}}(a c_1 \ldots c_n) < \text{steps}_{\text{\mathcal{T}}}(x c_1 \ldots c_n) \text{ if } x \leq a : A \in \Gamma
\)

8. \(
\text{steps}_{\text{\mathcal{T}}}(\beta\text{nf}(a)) \leq \text{steps}_{\text{\mathcal{T}}}(a)
\)

where we assume that every occurrence of steps is well-defined in parts 2 through 8.

Proof: All proofs are easy.

1. We know \(b\) is also \(\text{SN}_{\beta\sigma}\). Take \(m = \text{steps}_{\text{\mathcal{T}}}(b)\), so \(b\) reduces in \(m\) steps to a \(\beta\sigma\) nf in \(\Gamma\). Then we can reduce \(a\) in \(n + m\) steps to a \(\beta\sigma\) nf in \(\Gamma\). Hence \(\text{steps}_{\text{\mathcal{T}}}(a)\), the maximum number of steps of \(a\) to reach a \(\beta\sigma\) nf, is at least \(n + m\). So \(\text{steps}_{\text{\mathcal{T}}}(a) \geq n + \text{steps}_{\text{\mathcal{T}}}(b)\).

2. We first prove the following property. If \(\Gamma \vdash A \triangleright_{\beta\sigma} A'\) then \(\Gamma \vdash \Pi x : A. B \triangleright_{\beta\sigma} \Pi x : A'. B\).

This follows from the definition of \(\triangleright_{\beta}\) and \(\triangleright_{\sigma}\).

Now suppose \(\text{steps}_{\text{\mathcal{T}}}(A) = n\). Then there is a reduction path \(A \triangleright_{\beta\sigma} A_1 \triangleright_{\beta\sigma} \ldots A_n\) in \(\Gamma\) consisting of \(n\) steps. By the property just shown, there is also a reduction path \(\Pi x : A. B \triangleright_{\beta\sigma} \Pi x : A_1. B \triangleright_{\beta\sigma} \ldots \Pi x : A_n. B\) of \(n\) steps. By property we have that \(\text{steps}_{\text{\mathcal{T}}}(\Pi x : A. B) \geq n + \text{steps}_{\text{\mathcal{T}}}(\Pi x : A_n. B)\), so \(\text{steps}_{\text{\mathcal{T}}}(\Pi x : A. B) \geq \text{steps}_{\text{\mathcal{T}}}(A)\).

3,4,5. Similar to 2.

6. First prove the following lemma by induction on \(B\):

\[
\text{If } \Gamma, x : A, \Gamma' \vdash B \triangleright_{\sigma} B' \text{ then } \Gamma, x : A', \Gamma' \vdash B \triangleright_{\sigma} B'.
\]

By the technique used in 2, we have \(\text{steps}_{\text{\mathcal{T}}, x : A}(B) \leq \text{steps}_{\text{\mathcal{T}}, x : A'}(B)\). By symmetry between \(A\) and \(A'\), we may replace \(\leq\) by =.

7. If \(x \leq a : A \in \Gamma\), then \(\Gamma \vdash x \triangleright_{\sigma} a\), so \(\Gamma \vdash x c_1 \ldots c_n \triangleright_{\beta\sigma} a c_1 \ldots c_n\), so by property 1 \(\text{steps}_{\text{\mathcal{T}}}(a c_1 \ldots c_n) < \text{steps}_{\text{\mathcal{T}}}(x c_1 \ldots c_n)\).

8. We have \(a \triangleright_{\beta} \beta\text{nf}(a)\), so \(\Gamma \vdash a \triangleright_{\beta\sigma} \beta\text{nf}(a)\), so by property 1 \(\text{steps}_{\text{\mathcal{T}}}(a) \geq \text{steps}_{\text{\mathcal{T}}}((\beta\text{nf}(a))\).

\(\square\)

Definition 7.3.6.4 The size of a term \(a\), \(\text{size}(a)\), is the number of occurrences of variables and sorts in \(a\).

\(\square\)

Lemma 7.3.6.5

1. \(\text{size}(a) \geq 1\)

2. If \(a\) is a subterm of \(b\) and \(a \neq b\) then \(\text{size}(a) < \text{size}(b)\).

Proof: Easy.

\(\square\)

Convention: In this section, we consider only \(\text{PTS} \leq_{\delta}\) which are \(\text{SN}_{\beta\sigma}\).

Lemma 7.3.6.6 If \(c\) and \(d\) are well-typed in \(\Gamma\), then \(\text{Sub}(\Gamma; c; d)\) terminates.
Proof: By induction on the lexicographically ordered pair

$$(\text{steps}_T(c) + \text{steps}_T(d), \text{size}(c) + \text{size}(d)),$$

This measure is well-defined, since $c$ and $d$ are well-typed in $\Gamma$, and we have a $SN_{\beta_\eta}$ \(PTS\). We proceed by case distinction on the structure of the algorithm. We treat here the two hardest cases (for $c \neq d$).

$c \equiv \Pi x : A, B$ and $d \equiv \Pi x : A', B'$: By Correctness of Types, $c$ and $d$ are typable in $\Gamma$, so by Generation (Lemma 7.2.4.8) $A$ and $A'$ are also well-typed in $\Gamma$. The following two calculations show we can apply the IH for $A$ and $A'$. They are justified by Lemmas 7.3.6.3 and 7.3.6.5.

\[
\begin{align*}
\text{steps}_T(A) + \text{steps}_T(A') &\leq \\
\text{steps}_T(\Pi x : A, B) + \text{steps}_T(\Pi x : A', B') &= \\
\text{steps}_T(c) + \text{steps}_T(d)
\end{align*}
\]

and

\[
\begin{align*}
\text{size}(A) + \text{size}(A') &< \\
\text{size}(\Pi x : A, B) + \text{size}(\Pi x : A', B') &= \\
\text{size}(c) + \text{size}(d)
\end{align*}
\]

So $\text{Sub}(\Gamma; A'; A)$ terminates. If the result is $\text{Fail}$, then $\text{Sub}(\Gamma; c; d) \equiv \text{Fail}$ and we are done. So we assume the result is $\text{Ok}$. By Lemma 7.3.5.2 we have $\Gamma \vdash A' \leq A$ and hence $\Gamma \vdash A' \leq A$.

By using Generation on the facts that $c$ and $d$ are typable, we get $\Gamma, x : A \vdash B : s_1$ and $\Gamma, x : A' \vdash B' : s_2$. By Lemma 7.2.4.12 we also have $\Gamma, x : A' \vdash B : s_1$. So $B$ and $B'$ are well-typed in $\Gamma, x : A'$. The following two calculations show we can apply the IH also for $B$ and $B'$.

\[
\begin{align*}
\text{steps}_{T, x : A'}(B) + \text{steps}_{T, x : A'}(B') &= \\
\text{steps}_{T, x : A}(B) + \text{steps}_{T, x : A'}(B') &\leq \\
\text{steps}_{T}(\Pi x : A, B) + \text{steps}_{T}(\Pi x : A', B') &= \\
\text{steps}_{T}(c) + \text{steps}_{T}(d)
\end{align*}
\]

and

\[
\begin{align*}
\text{size}(B) + \text{size}(B') &< \\
\text{size}(\Pi x : A, B) + \text{size}(\Pi x : A', B') &= \\
\text{size}(c) + \text{size}(d)
\end{align*}
\]

So $\text{Sub}(\Gamma, x : A'; B; B')$ terminates, and hence $\text{Sub}(\Gamma; c; d)$ terminates.

$c \equiv x \ c_1 \ldots c_n$: We assume $x \leq a : A \in \Gamma$ for some $a, A$. (If not, $\text{Sub}(\Gamma; c; d)$ returns $\text{Fail}$ and terminates.) By assumption, $x \ c_1 \ldots c_n$ is a term in $\Gamma$, so by Correctness of Types, $x \ c_1 \ldots c_n$ is typable in $\Gamma$, so by Lemma 7.2.4.11 also $a \ c_1 \ldots c_n$ is typable in $\Gamma$, so this term is $SN_{\beta_\eta}$. Hence $\beta_\eta(a \ c_1 \ldots c_n)$ exists, and is well-typed in $\Gamma$ by Subject Reduction. We show the measure decreases in the following calculation, that uses Lemma 7.3.6.3.

\[
\begin{align*}
\text{steps}_T(\beta_\eta(a \ c_1 \ldots c_n)) + \text{steps}_T(d) &\leq \\
\text{steps}_T(a \ c_1 \ldots c_n) + \text{steps}_T(d) &< \\
\text{steps}_T(x \ c_1 \ldots c_n) + \text{steps}_T(d) &= \\
\text{steps}_T(c) + \text{steps}_T(d)
\end{align*}
\]

By IH $\text{Sub}(\Gamma; \beta_\eta(a \ c_1 \ldots c_n); d)$ terminates, and so $\text{Sub}(\Gamma; c; d)$ terminates.
Lemma 7.3.6.7 If $\Delta \vdash_{td} ok$ then $\text{Typ}(\Delta; c)$ terminates.

Proof: By induction on the structure of $c$. We treat here case $c \equiv \Pi x \leq a : A. B$, all other cases are similar or simpler.

By IH $\text{Typ}(\Delta; a)$ terminates. If it returns Fail, then the term let $(A', s_1) \equiv \text{Typ}(\Delta; a)$ in ... also returns Fail, so $\text{Typ}(\Delta; c)$ returns Fail, and terminates. So we assume $\text{Typ}(\Delta; a)$ returns $(A', s_1)$. By Lemma 7.3.6.4 $\Delta \vdash_{td} a : A' : s_1$, and by Soundness $\Delta \vdash a : A'$. So $A'$ is well-typed in $|\Delta|$, so $A'$ is SN$_\beta$, so $\beta\text{nf}(A')$ terminates, and also $\beta\text{nf}(A')$ is well-typed in $|\Delta|$.

By similar reasoning, we can assume $\text{Typ}(\Delta; A) \equiv (S_1, s'_1)$ and hence $\Delta \vdash_{td} A : S_1 : s'_1$ and $\beta\text{nf}(A)$ terminates and $\beta\text{nf}(A)$ is well-typed in $|\Delta|$. So we know by Lemma 7.3.6.6 that $\text{Sub}(|\Delta|; \beta\text{nf}(A'); \beta\text{nf}(A))$ terminates; we assume the result is $0k$. By Lemma 7.3.6.2 $|\Delta| \vdash_{td} \beta\text{nf}(A') \leq \beta\text{nf}(A)$. We can also assume $s_1 \in S^\leq$.

Now we have all the premises of the (C-Bvar) rule, so $\Delta, x \leq a : A : s \vdash_{td} ok$. Now by IH $\text{Typ}(\Delta, x \leq a : A : s ; B)$ terminates, we assume it returns $(S_2, s'_2)$. By similar reasoning as above, we show that $S_2$ is well-typed in $|\Delta|, x \leq a : A$, so $S_2$ is SN$_\beta$ in this context, so $S_2$ is also SN$_{wh, \beta}$ in this context, so $\text{wh} \beta, \text{nf}(|\Delta|, x \leq a : A; S_2)$ terminates. The determine construct also terminates, so $\text{Typ}(\Delta; (\Pi x \leq a : A. B))$ terminates. □

Lemma 7.3.6.8 $\text{Wf}(\Delta)$ terminates.

Proof: By induction on the structure of $\Delta$. The proofs of the individual cases are similar to those in Lemma 7.3.6.7. □

7.3.7 Summary of Type-Checking

We have given syntax-directed derivation rules, and shown these to be equivalent to the original derivation rules for $PTS^\leq s$ that are bijective and SN$_\beta$. The sd rules were transformed into proper algorithms, that terminate for $PTS^\leq s$ that are SN$_{\beta r}$. Let us review how to check $\Gamma \vdash a : A$, using the algorithms. This is done by the following four steps, which all have to succeed.

1. Find a corresponding $\Delta$ ($|\Delta| \equiv \Gamma$) and check $\text{Wf}(\Delta) \equiv 0k$. By inspecting the $\text{Wf}$ algorithm, we see there is at most one choice for $\Delta$; each sort $s$ in a declaration $x : B : s$ or $x \leq b : B : s$ is determined by $b$ and $B$. In other words, calculating $\Delta$ from $\Gamma$ and computing $\text{Wf}(\Delta)$ can be done simultaneously.

2. Compute $\text{Typ}(\Delta; a)$, which should deliver a pair $(A', s)$ of a minimal type $A'$ of $a$ and the sort $s$ of $a$.

3. Check $A$ is a term in $|\Delta|$. This holds, iff $A \equiv s'$ for some $s' \in S$, or $\text{Typ}(\Delta; A)$ returns a pair $(B, s')$.

4. Check $\text{Sub}(\Gamma; \beta\text{nf}(A'); \beta\text{nf}(A)) \equiv 0k$.

In practice, e.g. in the proof-assistant Yarrow, we maintain an extended context $\Delta$, and not just a context $\Gamma$. So the sorts of all declared variables are computed once and forever. This is more efficient that recomputing the sorts all the time.
Corollary 7.3.7.1 Type-checking is decidable for $PTS \leq s$ which are bijective and $SN_{\beta\sigma}$.

For a given $PTS \leq s$, it is easy to check if it is bijective. However, to check whether $SN_{\beta\sigma}$ holds is quite another matter. We conjecture that if $SN_{\beta}$ holds, also $SN_{\beta\sigma}$ holds, but we do not think it is feasible to prove this in general. Proving $SN_{\beta\sigma}$ for $\lambda C \leq s$ (from which $SN_{\beta\sigma}$ for all example systems in Section 7.1.3 follows) might be done as follows.

1. Prove $SN_{\beta}$ for $\lambda C \leq s$ from $SN_{\beta}$ for $\lambda C$, using a reduction preserving translation. This translation might proceed as in [CG92] or [BCGS91], but the translation is more complicated here, because of lifted subtyping. On the other hand, we do not need to prove the translation is "coherent".

2. Prove $SN_{\sigma}$ for $\lambda C \leq s$.

3. Combine the results, along the lines of the proof of $SN_{\beta\sigma}$ for $P^w \leq s$ in [PS97].
7.4 Conclusions

7.4.1 Related Work

We compare our work with six systems with subtyping: kernel-Fun [CW85], $F_{\leq}^w$ [PS97], $F_{\leq}^\lambda$ [Com95], $\lambda P_{\leq}$ [AC96b], $\lambda C_{\leq}$ as defined in [Che97] and $F_{\leq}^C$ [CG97]. These are all systems of the $\lambda$-cube [Bar92] extended with subtyping. For each system, the comparison is divided into the following three parts:

**Calculus** Which terms can be formed, what are the typing and subtyping rules, and what are the derivable judgments? A detailed comparison has been given in the corresponding subsections of 7.1.3, so in this section we only summarize these differences. We also use the terminology defined in Section 7.1.3.

A common difference is that we consider a whole family of type systems, where each of the references considers one specific system. As a consequence, [CW85, PS97, Com95, AC96b, CG97] make a natural syntactical distinction between programs (in those accounts called terms) and datatype-constructors (types) and kinds, and a distinction between type derivations for each of these.

**Meta-theory** How was the meta-theory developed? The main problem in our first attempt to define $PTS_{\leq}s$ and to develop its meta-theory was the mutual dependency between the typing and subtyping judgments. We solved this by adopting the design decision to define the subtyping judgment on pseudoterms, so that the subtyping rules do not depend on the typing rules. We will discuss for the selected references whether this problem occurs, how they solve this, and why we cannot follow their approach for $PTS_{\leq}s$.

The second problem is the intractability of the subtyping rules. All subtyping systems mentioned above have alternative rules of some sort, that are more tractable because these do not include the transitivity rule. They all have an alternative rule ($\leq$-transvar) instead to handle the absence of transitivity in combination with variables. However, the solutions for the combination of transitivity and conversion differ widely. There are also quite some differences in the proofs of equivalence between original and subtyping rules (admissibility of transitivity is an important part of all these proofs). We will not explain all these differences because they are quite technical. ([Che97] sheds some lights on these matters.)

The third problem we had was to prove Minimal Typing. We had a technical subproblem caused by subtyping on different levels, which we solved by introducing Weak Minimal Typing. This subproblem is new and does not occur in the literature. Another subproblem, occurring in all existing subtyping systems, is to find a way to compute minimal types. We will discuss the individual solutions below.

**Algorithms** What are the algorithms for computing types and deciding subtyping? We remark here that none of the references use triples in the syntax-directed typing judgments as we do (they do not need to). However, in [CG92, PS97, Com95, AC96b, CG97] every level of terms (programs and datatype-constructors) has its own typing algorithm, just as it has its own typing judgment. We can make the same distinction by fixing a sort in our syntax-directed judgments, e.g. $\Gamma \vdash n : A : *$ corresponds to the judgment con-
cerning programs (also known as the typing judgment), and \( \Gamma \vdash a : A : \Box \) corresponds to the judgment concerning datatype-constructors (the kinding judgment).

The algorithms should terminate. Termination of the typing algorithm is usually straightforward, so we will focus on termination of the subtyping algorithm. We proved this assuming Strong Normalization of \( \beta \sigma \) reduction; we discuss the proofs found for the individual systems below.

Let us now consider the individual systems.

- The calculus kernel-Fun (the pure fragment of Fun [CW85]) is equal to \( \lambda 2 \leq \), but \( \lambda 2 \leq \) does not have the Top type of Kernel-Fun. Kernel-Fun in its turn is equal to \( F_{\leq} [CG92] \), except for the more liberal \((\leq \Pi)\) rule present in the latter. Section 7.1.3.2 elaborates on these differences.

The theory developed in [CG92] has widely influenced all of the other discussed systems. Still, we ignore the meta-theoretical part and the algorithmic part of the comparison, because those are treated below for the more general \( F_{\leq}^\omega \), and accounts of the meta-theory for the second-order calculus use the undecidable \( F_{\leq} \).

- The calculus \( F_{\leq}^\omega [PS97] \) is equal to our \( \lambda \omega \leq \), but \( \lambda \omega \leq \) does not have the Top types of \( F_{\leq}^\omega \). (Section 7.1.3.3).

As noted in the introduction of this chapter, [PS97] first develops meta-theory about typing datatype-constructors, then about the subtyping judgment and finally about typing programs. This cannot be done in general for PTS\(\leq\)s, since typing for the various categories of terms is mutually dependent. In [PS97] the Minimal Typing property is proved using the typing algorithm, whereas we prove this property separately, because our typing algorithm works only for the bijective SN\(\beta\sigma\) PTS\(\leq\)s. The typing algorithm given for \( F_{\leq}^\omega \) uses a variant of our \( \sigma \)-reduction to find minimal types.

The final subtyping algorithms are the same, except for the Top-types. The termination of the subtyping algorithm is shown using SN\(\beta\sigma\) in a similar way as we do. The difference is that they actually prove SN\(\beta\sigma\) for their system. The typing algorithms are essentially the same.

- The system \( F_{\leq}^\omega [CG97] \) is equal to our \( \lambda \omega \leq^+ \), except that we do not have Top-types and subtyping on bounded operator abstractions (Section 7.1.3.4).

The meta-theory developed in [CG97] follows a quite different approach than other works mentioned here and our work; by giving a typed operational semantics they solve the mutual dependence between the typing and subtyping judgments occurring in \( F_{\leq}^\omega \). We do not know whether this approach is applicable to PTS\(\leq\)s. Minimal Typing is not proved for \( F_{\leq}^\omega \), as it is the same as in [PS97].

The subtyping algorithm of [CG97] is similar to our algorithm. One difference is that they postpone applications of the \((\leq \text{refl})\) rule (as in [AC96b]). Another difference is that our algorithm relates terms in normal form, whereas theirs works with weak head normal forms, and postpones reduction as long as possible, so it is more efficient. Termination of subtyping is not proved, but is (allegedly) similar as in [Com95]. No typing algorithm (for programs) is given, as it is the same as for [PS97].
• The calculus $F_\subseteq^\omega$ [Com95] is an extension of $F_\subseteq^\omega$ with intersection types (although $F_\subseteq^\omega$ was developed independently of $F_\subseteq^\omega$). Intersection types are the greatest lower bounds with respect to the subtype ordering, and can be used to model both overloading of functions and certain aspects of multiple inheritance.

Similarly to $F_\subseteq^\omega$, the meta-theory of $F_\subseteq^\omega$ can be neatly split into three stages, although each of the stages become more complex because of intersection types. Transitivity elimination (i.e. admissibility of transitivity) is proved by defining a reduction relation on subtyping derivations. The Minimal Typing property is proved in conjunction with the typing algorithm (like [PS97] and unlike us).

The subtyping algorithm of [Com95] is essentially the same as ours, besides having additional clauses for intersection types. Its termination is proved in a different way, viz. by introducing an auxiliary type-constructor + and so-called +-reduction for this type-constructor. Termination is proved using an induction measure based on this form of reduction. There are similarities between +-reduction on the $\sigma$-reduction found here and in [PS97], but we prefer the latter approach, since it does not need an additional type-constructor. The typing algorithm of $F_\subseteq^\omega$ is at some points more complicated than ours, because it has to deal with intersection types.

• The calculus $\lambda P_\subseteq$ [AC96b] is equal to our calculus $\lambda P_\subseteq$ when considering the derivable typing judgments concerning programs (Section 7.1.3.5). The differences in both the derivable subtyping judgments and the typing judgments concerning datatype-constructors are inconsequential.

Unlike the two systems Fun and $F_\subseteq^\omega$ discussed above, $\lambda P_\subseteq$ has mutual dependencies between programs and datatype-constructors, i.e. between "terms" and "types". They split the $\beta$ reduction relation into $\beta_1$, reduction on programs, and $\beta_2$, reduction on type-constructors. In this way, they are able to prove first some essential meta-theory about typing for type-constructors, e.g. Subject Reduction with respect to $\beta_2$, then develop the theory for subtyping, and conclude with theory for typing programs, e.g. Subject Reduction with respect to $\beta_1$. This careful distinction of different kinds of reduction is very hard in $PTS_\subseteq$, since we cannot syntactically distinguish programs and type-constructors (these notions themselves make little sense in the general $PTS_\subseteq$s). Just as in [PS97, Com95] the Minimal Typing property is proved using the typing algorithm, whereas we prove this property separately. Minimal types are computed in a similar way as in [PS97].

The subtyping algorithm for $\lambda P_\subseteq$ is similar to our algorithm, the main difference being that they postpone applications of the ($\subseteq$-refl) rule as long as possible, whereas we apply this rule as soon as possible. The reason is that their subtyping algorithm is defined for $\beta_2$-normal forms, instead of $\beta$-normal forms. So the algorithmic version of ($\subseteq$-refl) has to check for $\beta_1$ equality, which is expensive. Since our algorithm does work with $\beta$-normal forms, we can just test for syntactical equality, which is cheap (or more accurately, we already paid the costs when $\beta$-normalizing the terms). Termination of the algorithm is proved in a similar way as in [Com95], using +-reduction. Their typing algorithm is essentially the same as ours.

• The system $\lambda C_\subseteq$ as described in [Che97] is equal to our $\lambda C_\subseteq^\omega$ (Section 7.1.3.7). Both formulations have no bounded quantifications.
Here, programs and type-constructors are also mutually dependent, but the "trick" of splitting $\beta$-reduction in [AC96b] is not feasible any more, because there would be four kinds of reduction. But the typing judgments occurring in subtyping rules all have the simple form $\Gamma \vdash A : s$. Using this in combination with the specific rules $R$ of $\lambda C \leq \gamma$, enough meta-theory for typing can be done before subtyping is examined. This method does not work for $PTS \leq s$, since terms involved in the subtype relation are not always typable with a sort. Minimal Typing is not treated.

The algorithmic derivation rules of [Che97] are our reformulated subtyping rules (Definition 7.2.1.2) minus $(\leq \Pi \text{-app})$, $(\leq \text{-BII})$ and $(\leq \lambda)$. No concrete algorithms are given. Termination of these algorithmic rules is proved in a similar way as [PS97] and we do, but $SN_{\beta E}$ is shown. So [Che97] has decidable subtyping, whereas we have this result only under the condition that $SN_{\beta E}$ holds. No typing algorithm is given in [Che97].

### 7.4.2 Summary and Conclusion

In this chapter we defined the framework of Pure Type Systems with subtyping, an extension of the $PTS$s with subtyping, bounded quantification and lifted subtyping. We do not have subtyping on sorts (e.g. as in [Luo98]), or coercive subtyping (as in [Bar96, Luo99]). Coercive subtyping means that subtyping between existing types can be defined with functions serving as coercions, and that these coercions are inserted at the appropriate places by the typing algorithm. Coercive subtyping is in particular suited to formalization of mathematics.

For this thesis, the main application of the $PTS \leq s$ is the system $\lambda \omega_{\omega}^F$. In the following chapters we show how the features of $\lambda \omega_{\omega}^F$ are used to write object-oriented programs and reason about such programs. But $\lambda \omega_{\omega}^F$ is not the only application of $PTS \leq s$. Many existing type systems with subtyping can be seen as members of our framework, viz. $\lambda \rightarrow \leq s$, $F_{\leq s}$, $F_{\leq \omega}$, $\mathcal{F}_{\leq s}$, $\lambda P_{\leq s}$ and $\lambda C \leq \gamma$. Other members, like $\lambda C \leq$ (Section 7.1.3.6) are new systems which have promising features, both applicable in programming languages and in theorem proving.

We developed the meta-theory for $PTS \leq s$, including Subject Reduction, and Minimal Typing for functional systems. This was not easy:

- **The first problem was the definition of the subtyping rules.** In the usual definitions, the subtyping rules and typing rules are mutually dependent, which results in huge difficulties in proving theory general for $PTS \leq s$. Therefore we adopted the design decision that the subtyping rules are defined independent of the typing rules. This decision worked out well, since it allows us to develop the meta-theory for the subtyping judgment before the theory of the typing judgment.

- **The second problem was that the subtyping rules in themselves are intractable, as usual.** Therefore we had to give a reformulation of the subtyping rules (Definition 7.2.1.2), that behaves better, as is usual in systems with subtyping. The two sets of subtyping rules need to be proved equivalent. The hardest part of this proof is the admissibility of transitivity (transitivity elimination). We were able to prove this using an unusual induction measure and the property $\leq \text{-Conversion-closed}$ (Theorem 7.2.2.6). This theorem says that in a subtyping judgment terms may be replaced by convertible ones, without making the derivation of the judgment essentially longer. One aspect of the good behaviour of the reformulated rules is that they do not introduce untypable terms: if the terms in
the conclusion are typable, so are all terms in the premises of the rule (Lemma 7.2.4.13). This property is important in our proof of Minimal Typing.

- The third problem was how to prove Minimal Typing. One part of the solution is introducing $\omega$-reduction, which is quite usual in subtyping systems. Another part of the solution is to prove Minimal Typing in two stages, by introducing an auxiliary lemma called Weak Minimal Typing. This part is new compared to both ordinary $PTS$s and existing subtyping systems.

Our design decision, i.e. the subtyping rules do not depend on the typing rules, has two drawbacks. First, a consequence of the decision is that $\preceq$ is defined for all pseudoterms instead of only for (typable) terms. Similarly, the meta-theory for the subtyping judgment is done for pseudoterms. This forced us to introduce a weird rule, ($\preceq$-IIapp), to have equivalence between the original rules and the reformulated rules on pseudoterms. This rule is weird, since it only relates untypable terms. So, even though we are ultimately only interested in subtyping on terms, we are forced to consider the meta-theory for pseudoterms, and in particular forced to consider this strange ($\preceq$-IIapp) rule, even though it is clearly useless for terms. Indeed, we showed in Lemma 7.2.4.13 that ($\preceq$-IIapp) is never used in sensible subtyping derivations. This situation is reminiscent to that of $\beta$ reduction, which is also defined on pseudoterms, and Lemma 7.2.4.13 then corresponds to Subject Reduction. Second, the design decision makes it hard to extend the $PTSs$ with some features, like $Top$-types (Section 7.1.3.2) and subtyping on bounded operator abstractions (Section 7.1.3.4). For $\lambda\omega_s^<$, these extensions make little sense, so for our purposes this drawback has no effect. Anyhow, the meta-theory is much more difficult if the system does not comply to the design decision, and we doubt that we could have developed the theory without it.

The last technical part of this chapter is the development of a type-checking algorithm for $PTSs$ that are bijective and $SN_{\beta\omega}$. This algorithm extends that of [Pol93] with a subtyping algorithm and $\omega$-reduction; both appear in a similar form in [PS97] and [AC96b], which in their turn stem from work on $F_{\leq}$ (e.g. see [CG92]). The main work here lies in reformulating the typing rules, so they become syntax-directed. Showing completeness of our syntax-directed typing rules for $PTSs$ is as hard as for $PTSs$ in [Pol93], but much harder than for existing systems with subtyping. The subtyping rules were already tamed before, so little work is needed there. It is easy to convert the reformulated rules into a proper algorithm. The main omission in this part is the proof of $SN_{\beta\omega}$ for several concrete $PTSs$ (this property does not hold for all systems), but we have reason to believe it holds for $\lambda\omega_s^<$. 
Chapter 8

$\lambda\omega^+_L \subseteq$, Subtyping Added

In this chapter we define the programming logic $\lambda\omega^+_L \subseteq$, which is an extension of $\lambda\omega^+_L$ with subtyping. We define $\lambda\omega^+_L \subseteq$ as a PTS$\subseteq$ (Chapter 7) with additional typing rules for records and existential types, as given in Chapter 6, and with new subtyping rules for records and existential types. Here the generality of the work on PTS$\subseteq$s pays off: we are able to give a compact definition of $\lambda\omega^+_L \subseteq$.

Section 8.1 gives the syntax of $\lambda\omega^+_L \subseteq$, and Section 8.2 briefly discusses the semantics. We show in Section 8.3 some consequences of subtyping to the logic. Although we introduce subtyping in order to model OOP, it can be used for other things as well; in Section 8.4 we show a simple example of a program using subtyping, and how this subtyping appears in the specification.

8.1 Syntax

We define $\lambda\omega^+_L \subseteq$ in two steps. First we introduce the programming language $\lambda\omega^+_L$, and then the entire programming logic $\lambda\omega^+_L \subseteq$. Since the purely logical part remains unchanged, viz. $\lambda\omega_p$, we do not discuss it again.

8.1.1 The Programming Language $\lambda\omega^+_L \subseteq$

The programming language $\lambda\omega^+_L \subseteq$ is an extension of $\lambda\omega^+_L$ with subtyping.

Definition 8.1.1.1 The system $\lambda\omega^+_L \subseteq$ is the PTS$\subseteq$ specified by

$S = \{*, \square s\}, A = \{*, s : \square s\}, R = \{(\square s, \square s), (\square s, *), (*, *), (s, *)\}, S \subseteq = \{\square s\}, R \subseteq = \{(\square s, *), \}

extended with the constructions for records and existential types as in Chapter 6 (Definitions 6.1.1.1, 6.1.1.2, 6.1.2.1), and with the following subtyping rules for records and existential types.

(\leq\text{-width}) \quad \Gamma \vdash \{l_1 : A_1, \ldots, l_k : A_k\} \leq \{l_1 : A_1, \ldots, l_n : A_n\} \quad k \geq n

(\leq\text{-depth}) \quad \text{for all } \Gamma \vdash A_i \leq B_i \quad \Gamma \vdash \{l_1 : A_1, \ldots, l_n : A_n\} \leq \{l_1 : B_1, \ldots, l_n : B_n\}
This definition directly shows that this system is an extension of $\lambda \omega_1^+$ and $\lambda \omega^\leq$.

Having $\Box_s \in S^\leq$ means that we have subtyping on datatype-constructors, since datatype-constructors live in kinds, and kinds live in $\Box_s$. So we can declare subtypes of datatype-constructors in the context, e.g. the context $\text{Int} : *_s, \text{Nat} \leq \text{Int} : *_s$ is well-formed, and any term of type $\text{Nat}$ will also have type $\text{Int}$. Another example: the context $\text{Vehicle} : *_s \rightarrow *_s, \text{CarI} \leq \text{Vehicle} : *_s \rightarrow *_s$ is well-formed, and in this context, and any term of type $\text{CarI} T$ will also have type $\text{Vehicle} T$ (where $T$ is some datatype).

Before discussing $\mathcal{R}^\leq$, we discuss the subtyping rules on record types and existential types. These are as usual (e.g. see [Car86, CW85]). There are two subtyping rules for records. The first rule, $(\leq\text{-width})$, is the most essential subtyping rule for records. If record type $B$ has all the fields of $C$ (and possibly some more), $B$ is a subtype of $C$. This is justified by the fact that all operations permitted on a record $c$ of type $C$ are also permitted on a record $b$ of type $B$; any field available in $c$ is also available in $b$. So it is sound to consider $b$ to have also type $C$ by subsumption. This rule $(\leq\text{-width})$ introduces a new source of subtyping. In $PTS^\leq s$ every interesting subtyping judgment originates from subtype declarations, but now we can have interesting subtyping judgments even though there are no subtype declarations, e.g. $\text{Nat} : *_s \vdash \{1 : \text{Nat}, m : \text{Nat}\} \leq \{1 : \text{Nat}\}$.

By the rule $(\leq\text{-depth})$ subtyping propagates (covariantly) through the construction of records. The two subtyping rules for records can be combined into one rule:

$$ (\leq\text{-record}) \frac{\Gamma \vdash A_i \leq B_i \quad \text{for all } i \leq k}{\Gamma \vdash \{l_1 : A_1, \ldots, l_k : A_k\} \leq \{l_1 : B_1, \ldots, l_n : B_n\} \quad k \geq n} $$

We prefer the original presentation for clarity, but $(\leq\text{-record})$ is better suited for implementation.

By rule $(\leq\Sigma)$ subtyping propagates (covariantly) through the construction of existential types.

Note that we comply to the design decision of the previous chapter: there are no typing premises in the subtyping rules. Therefore we expect that all results for subtyping in Chapter 7 extend straightforwardly to the extended set of subtyping rules.

Now we discuss the bounded rule $(\Box_s, *) \in \mathcal{R}^\leq$. This makes it possible to write bounded abstractions over a kind in a program, just as $(\Box_s, *) \in \mathcal{R}$ makes it possible to write polymorphic abstractions over a kind in a program. These bounded abstractions are programs of the form $(\lambda X : A : Ik, b)$ where $A$ is a datatype-constructor and $Ik$ is a kind. Such a program expects a datatype-constructor (which is a subtype of $A$) as argument.

For example, the program $(\lambda X : 1 : \text{Nat} \vdash *_s, \lambda x : X. x : 1)$ expects a datatype which is a subtype of $\{1 : \text{Nat}\}$ as argument. The only concrete types that fulfill this requirement are record types with a field $1 : \text{Nat}$. Given such an argument, the program produces the function that selects the 1 field.

Programs of the form $(\lambda X : A : Ik, b)$ are called bounded polymorphic programs. The types of these programs are bounded polymorphic types, also known as bounded quantifications, of
the form \((\Pi X \leq A : IK. B)\). For example,

\[
(\lambda X : \{1 : \text{Nat}\} : \ast_\perp, \lambda x : X. x \cdot 1) : (\Pi X \leq \{1 : \text{Nat}\} : \ast_\perp, X \rightarrow \text{Nat}).
\]

The bounded rule \((\Box, \ast_\perp)\) allows the formation of these bounded polymorphic types:

\[
\frac{\Pi \text{-form} \quad \Gamma \vdash IK : \Box, \Gamma, X \leq A : IK \vdash B : \ast_\perp}{\Gamma \vdash (\Pi X \leq A : IK. B) : \ast_\perp}
\]

This kind of abstraction in programs can be used to define generic programs that behave uniformly for the class of subtypes of \(A\). One example is defined above. The main reason to introduce these abstractions is to model object-oriented programs, see Chapter 9.

The reader might wonder why we do not have bounded \(\Sigma\)-types in analogy with bounded \(\Pi\)-types. The reason is that we will not need them for our model of (simple) objects. But bounded existential types can be useful for objects, e.g. in order to model different levels of encapsulation (hiding), see [PT93].

### 8.1.2 The Programming Logic \(\lambda \omega^+\le\)

The programming logic \(\lambda \omega^+\le\) is an extension of \(\lambda \omega^+\) and includes \(\lambda \omega^+\le\).

**Definition 8.1.2.1** The system \(\lambda \omega^+\le\) is the \(PTS\le\) specified by

\[
\begin{align*}
S & = \{\ast_\perp, \Box, \ast_p, \Box_p\} \\
A & = \{\ast_\perp : \Box, \ast_p : \Box_p\} \\
R & = \{\ (\Box, \Box), \ (\ast_\perp, \ast_\perp), \\
& \quad (\Box, \ast_p), (\ast_\perp, \ast_p), (\Box, \ast_p), (\Box, \Box_p), (\ast_\perp, \Box_p), (\ast_\perp, \ast_p), \ast_p,p, (\ast_p, \ast_p) \} \\
S^\le & = \{\Box\} \\
R^\le & = \{(\Box, \ast_\perp), (\Box, \Box_p), (\Box, \ast_p)\},
\end{align*}
\]

extended with the constructions for labelled products (records) and existential types as in Chapter 6 (Definitions 6.1.1.1, 6.1.1.2, 6.1.2.1), and with the subtyping rules as in Definition 8.1.1.1.

The sorts (\(S\)), axioms (\(A\)) and rules (\(R\)) are just as in \(\lambda \omega^+\) and \(\lambda \omega^+\le\), so \(\lambda \omega^+\le\) is an extension of these systems. The subtyping sorts (\(S^\le\)) are as in \(\lambda \omega^+\le\), and the bounded rules (\(R^\le\)) are an extension of those in \(\lambda \omega^+\le\), so \(\lambda \omega^+\le\) includes this programming language.

The bounded rule \((\Box, \ast_p) \in R^\le\) allows universal quantification over all subtypes of a datatype-constructor. This results in propositions of the form \((\forall X \leq A : IK. P)\) where \(A\) is a datatype- constructor and \(IK\) is its kind. For example, given a datatype \(a\), the proposition \(\forall B \leq A : \ast_\perp. \forall x, y : B. x =_B y \Rightarrow x =_A y\) expresses that equality on \(B\) implies equality on \(A\) for all subtypes \(B\) of \(A\). The formation rule of this universal quantification is

\[
\frac{\Pi \text{-form} \quad \Gamma \vdash IK : \Box, \Gamma, X \leq A : IK \vdash P : \ast_\perp}{\Gamma \vdash (\forall X \leq A : IK. P) : \ast_\perp}
\]

Often, such propositions are used to express properties of bounded polymorphic programs, as we will see in Section 8.4.
CHAPTER 8. $\lambda\omega_L^\pm \leq$, SUBTYPING ADDED

The bounded rule $(\Box_s, \Box_p) \in \mathcal{R}^\leq$ allows bounded polymorphic predicates. These predicates are of the form $(\lambda X \leq A : \mathcal{K}. P)$ and have type $(\Pi X \leq A : \mathcal{K}. \mathcal{P})$. For example, in order to express that a list of records with a field $\text{id} : \text{Nat}$ is sorted on the $\text{id}$ field, we need a bounded polymorphic predicate $\text{Sorted.id} : \Pi X \leq [\{ \text{id} : \text{Nat} \}] : *_s. \text{List} X \rightarrow *_p$. The formation rule for these predicates is:

\[
\frac{\Gamma \vdash \mathcal{K} : \Box_s \quad \Gamma, X \leq A : \mathcal{K} \vdash \mathcal{P} : \Box_p}{\Gamma \vdash (\Pi X \leq A : \mathcal{K}. \mathcal{P}) : \Box_p}
\] (BII-form)

In Section 8.4, we will define such a predicate $\text{Sorted.id}$.

The system $\lambda\omega_L^\pm \leq$ consists of three parts, which are clearly identified in the specification.

- $\lambda\omega_L^\pm \leq$ for the programs and their datatypes, with subtyping, bounded polymorphism and records and existential types. This part is specified by
  
  \[
  \{ (\Box_s, \Box_s), (\Box_s, *_s), (*_s, *_s) \} \subseteq \mathcal{R},
  \]

- $\Box_s \in \mathcal{S}^\leq$,

- $(\Box_s, *_s) \in \mathcal{R}^\leq$.

- $\lambda\omega_p$ for propositions and their proofs. This part is specified by
  
  \[
  \{ (\Box_p, \Box_p), (\Box_p, *_p), (*_p, *_p) \} \subseteq \mathcal{R}.
  \]

- All possible dependencies of propositions and proofs on programs and types. This part is specified by
  
  \[
  \{ (\Box_s, \Box_p), (\Box_s, *_p), (*_s, \Box_p), (*_s, *_p) \} \subseteq \mathcal{R},
  \]

- \[
  \{ (\Box_s, \Box_p), (\Box_s, *_p) \} \subseteq \mathcal{R}^\leq.
  \]

The system $\lambda\omega_L^\pm \leq$ is not the most general extension of $\lambda\omega_L$ in the $\mathcal{PTS}^\leq$ framework. It does not have:

- $*_s \in \mathcal{S}^\leq$ or $*_p \in \mathcal{S}^\leq$

  It does not make sense to have subtyping on these sorts, just as $* \in \mathcal{S}^\leq$ does not make sense for the systems in the $\lambda$-cube, as discussed in Section 7.1.3.9.

- $\Box_p \in \mathcal{S}^\leq$

  As a result, we cannot declare a variable as a subtype of a proposition, and there are no bounded quantifications on propositions. We do not see how this might be used, so we ignore this possibility.

  However, our choice of $\Box_p \not\in \mathcal{S}^\leq$ does not mean there is no subtyping on propositions. For example, if $\text{Nat} \leq \text{Int}$ and $P : \text{Int} \rightarrow *_p$, we have

\[
(\forall n : \text{Int}. P \, n) \leq (\forall n : \text{Nat}. P \, n)
\]

because of subtyping rule ($\leq$-$\Pi$). So subtyping on the level of datatypes can "seep through" to the level of propositions. This is an important consequence of our definition of $\mathcal{PTS}^\leq$s. Let us briefly discuss what it means for two propositions to be in the subtype relationship: if $P \leq Q$, any inhabitant $p$ of $P$ is also an inhabitant of $Q$ by the subsumption rule, and hence $P \implies Q$. So the subtyping relation on propositions may be considered as a fragment of the impliciation relation $\implies$.
8.2. SEMANTICS OF $\lambda \omega^+_L$ $\leq$

- $(\square, \square) \in R^\leq$

As a result, we do not have bounded datatype-constructors. There is no use for this rule, because the formation of datatype-constructors does not depend on subtyping (see Section 7.1.3.4).

Although we formally introduced $\lambda \omega^+_L \leq$ as a $PTS \leq$, we will also be using the definition mechanism of $DPTS$s. We do not formally introduce $DPT\leq_s$ ($PTS$ with definitions and subtyping), since the rules of $DPTS$s and $PTS \leq s$ can be straightforwardly combined, and we do not expect any problems in the development of the meta-theory for $DPT\leq_s$.

Meta-theory

We will not develop meta-theory for $\lambda \omega^+_L \leq$. In the previous chapter we proved several meta-theoretical results for a class of $PTS \leq$, including $\lambda \omega^+_L$ (i.e. $\lambda \omega^+_L \leq$ without records and existential types). These results include Subject Reduction and Minimal Typing. We conjecture that these properties hold also for $\lambda \omega^+_L \leq$, because records and existential types seem to be innocent extensions, and the meta-theory of Chapter 7 can probably be extended straightforwardly to cope with them.

Whether the property of Strong Normalization holds for $\lambda \omega^+_L \leq$ is a bit more dubious, since we did not prove SN for $\lambda \omega^+_L$ (see Section 7.3.7). We simply assume that $SN_\theta$ for $\lambda \omega^+_L \leq$ holds. The issue of type-checking is not really complicated by the additional constructs and (sub)typing rules.

8.2 Semantics of $\lambda \omega^+_L \leq$

We will not investigate the semantics of this system, since any model for (an extension of) $\lambda \omega$ can be extended straightforwardly to a model of (an extension of) $\lambda \omega_L$, see [Pol94]. An example of a PER model for an extension of $\lambda \omega$, with subtyping can be found in [Com95] (although this model does not contain records, but so-called intersection types).

8.3 Axioms and Library

We do not need any additional axioms for subtyping. Important propositions using subtyping can be provied within the system, and belong to the library. Section 8.3.1 shows such a lemma and its proof in detail. Section 8.3.2 explains some subtleties concerning the combination of Leibniz’ equality and subtyping.

8.3.1 Lemmas using Subtyping

We give one example of a lemma using subtyping, namely that equality on a subtype implies equality on the supertype:

$$E_{\text{eq} \_\_}E_{\text{eq} \_\_} := \ldots : \forall B: \_ \_ . \forall A \leq B : \_ \_ . \forall x, y: A . x =_A y \implies x =_B y$$
An elementary proof is as follows:

\[
\begin{array}{l}
1. \quad B : *_z \\
2. \quad A \leq B : *_z \\
3. \quad x : A \\
4. \quad y : A \\
5. \quad x =_A y \\
6. \quad P : B \rightarrow *_p \\
7. \quad Q := \lambda z : A. P z : A \rightarrow *_p \\
8. \quad \forall P : A \rightarrow *_p. P x \implies P y \\
9. \quad Q x \implies Q y \\
10. \quad P x \implies P y \\
11. \quad P x \implies P y \\
12. \quad \forall P : B \rightarrow *_p. P x \implies P y \\
13. \quad x =_B y \\
14. \quad \forall B : *_z. \forall A \leq B : *_z. \forall x, y : A. x =_A y \implies x =_B y
\end{array}
\]

The essential step is in line 7, where a predicate \( Q : A \rightarrow *_p \) is formed from \( P : B \rightarrow *_p \) in a trivial way. We use subsumption to give the variable \( z \) type \( B \), so \( P \) can be applied to \( z \). The rest of this proof is simply going through the definition of \( = \).

The introduction of \( Q \) is not necessary; because of the \((\leq \Pi)\) rule, \( P \) has also type \( A \rightarrow *_p \), so we can use \( \forall E \) directly on \( 8 \) and \( P \) to obtain \( P x \implies P y \). In fact, the proof can be simplified even further, since any proof of \( x =_A y \) is also a proof of \( x =_B y \), because the former proposition is a subtype of the latter. (The reader can check this by unfolding the definition of \( = \).) This leads to the following short but less informative proof:

\[
\begin{array}{l}
1. \quad B : *_z \\
2. \quad A \leq B : *_z \\
3. \quad x : A \\
4. \quad y : A \\
5. \quad x =_A y \\
6. \quad x =_B y \\
7. \quad \forall B : *_z. \forall A \leq B : *_z. \forall x, y : A. x =_A y \implies x =_B y
\end{array}
\]

8.3.2 Subtleties Concerning Equality and Subtyping

It is easy to show that the converse of \( Eq\_sub\_Eq\_super \) does not hold, because the following two properties constitute a counterexample:

1. \( \{ 1 = 0 \} =_0 \{ 1 = S 0 \} \). Proof: Because of axiom \textit{ind.rec} in Definition 6.2.1 we have the lemma \( \forall x : \emptyset. x =_0 \emptyset \). Applying this lemma to \( \{ 1 = 0 \} \) and \( \{ 1 = S 0 \} \), we obtain \( \{ 1 = 0 \} =_0 \emptyset \) and \( \{ 1 = S 0 \} =_0 \emptyset \). We are done by symmetry and transitivity.

2. \( \neg(\{ 1 = 0 \} =_0 \{ 1 = S 0 \} ) \). The proof is easy.
These two properties seem contradictory, by the following, flawed reasoning:

We have \( \{1 = 0\} = \emptyset \{1 = S \ 0\} \), and we may substitute equals for equals in any proposition, so in particular we can substitute \( \{1 = S \ 0\} \) for \( \{1 = 0\} \) in inequality 2, so we have \( \neg \{1 = S \ 0\} = \emptyset \{1 = \text{Nat} \ 0\} \{1 = S \ 0\} \). But by reflexivity we have \( \{1 = S \ 0\} = \emptyset \{1 = \text{Nat} \ 0\} \{1 = S \ 0\} \), so we have a contradiction.

The flaw lies in the substitution of \( \{1 = S \ 0\} \) for \( \{1 = 0\} \), as we will explain. In \( \lambda \omega^+ \) — i.e. without subtyping — the substitution of equals for equals is justified by the following lemma.

**Lemma 8.3.2.1 (for \( \lambda \omega_L^+ \))**

If \( \Gamma \vdash \ldots : a =_A b \) and \( \Gamma \vdash \ldots : Q[z := a] : \star_p \) then \( \Gamma \vdash \ldots : Q[z := b] \).

**Proof sketch:** Suppose \( p_{eq} \) is a proof for \( a =_A b \) and \( p_a \) is a proof for \( Q[z := a] \). It is easy to show that \( \Gamma \vdash p_{eq} : a =_A b \) implies that \( \Gamma \vdash a : A \). The main works lies in showing that since \( \Gamma \vdash a : A \) and \( \Gamma \vdash Q[z := a] : \star_p \), that \( \Gamma \vdash \lambda z : A. Q : A \rightarrow \star_p \). From this, we can easily derive by application of the typing rules and the definition of \( =_0 \), that \( \Gamma \vdash p_{eq} (\lambda z : A. Q) p_a : Q[z := b] \).

But in \( \lambda \omega_L^{+ \leq} \), this lemma does not hold. For example, take \( A \equiv \{\}\), \( a \equiv \{1 = 0\} \), \( b \equiv \{\} \) and \( Q \equiv (z.1 =_{\text{Nat}} 0) \). It is easy to show that all premises of the lemma hold, but the conclusion will certainly not hold, since we have \( Q[z := b] \equiv (\{\} \cdot 1 =_{\text{Nat}} 0) \), and this term is not typable, hence it cannot have an inhabitant.

The failure of this lemma for \( \lambda \omega_L^{+ \leq} \) lies in the fact that \( \lambda \omega_L^{+ \leq} \) does not possess the Uniqueness of Types property. The equality \( a =_A b \) implies that both \( a \) and \( b \) have type \( A \). But \( a \) may also have another type, say \( C \), which is a strict subtype of \( A \). The term \( Q[z := a] \) may only be well-typed because of this more informative typing \( a : C \). Then \( Q[z := b] \) is typable if \( b \) does have this type \( C \). (Without subtyping, \( C \) and \( A \) must be the same by Uniqueness of Types.) This suggests the following lemma for \( \lambda \omega_L^{+ \leq} \), that has the additional premise that \( A \) is a minimal type of \( a \).

**Lemma 8.3.2.2**

If \( \Gamma \vdash \ldots : a =_A b \) and \( \Gamma \vdash \ldots : Q[z := a] : \star_p \) and \( \Gamma \vdash a : A \) then \( \Gamma \vdash \ldots : Q[z := b] \).

**Proof sketch:** Similar to Lemma 8.3.2.1.

So we can substitute \( b \) for \( a \) if \( a =_A b \) and \( A \) is a minimal type for \( a \).

Now we can indicate the flaw in the reasoning given above: we have \( \{1 = 0\} = \emptyset \{1 = S \ 0\} \), but \( \{\} \) is not a minimal type for \( \{1 = 0\} \), so we may not substitute \( \{1 = S \ 0\} \) for \( \{1 = 0\} \).

This section shows that Leibniz’ equality and subtyping interact in subtle ways. First, the validity of an equality \( a =_A b \) may depend on the value of \( A \) (as shown with the example above). Second, we may replace equals for equals only under an additional condition.

In an early version of Yarrow with subtyping, the rewrite tactic did not consider the second subtlety, and admitted the flawed reasoning given above up to the point where the proof term was type-checked for admittance to the context; only then the proof was rejected since it was not typable.
8.4 Example: Sorting of Records

This example is adapted from Section 4.7, where we specified and defined a program that sorts a list of numbers. Here we define a bounded polymorphic programs sortId, that given the type of elements $B \leq \{ \text{id} : \text{Nat} \}$, sorts a list of Bs on the value of the id field. The purpose of this example is showing how the different forms of bounded II-types are used in various places in the specification, definition and correctness-proof of the sortId program.

First, we define the bounded polymorphic predicate $OrderedId$, that states that a list is ordered on its id field, and give some properties of this predicate.

$$
\text{OrderedId} := \lambda B \leq \{ \text{id} : \text{Nat} \} : *_s, \lambda l : \text{List } B. \forall P : \text{List } B \rightarrow *_p.
\]
$$

$$
P (\text{nil } B) \implies
\]

$$
(\forall a : B. \forall m : \text{List } B. P m \implies
\]

$$
(\forall b : B. \text{Elem}_B b m \implies a \cdot \text{id} \leq b \cdot \text{id}) \implies
\]

$$
P (a ; m) \implies
\]

$$
\]

$$
\text{OrderedId}_{\text{nil}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \text{OrderedId}_B (\text{nil } B)
\]

$$
\text{OrderedId}_{\text{cons}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall a : B. \forall m : \text{List } B. \text{OrderedId}_B m \implies
\]

$$
(\forall b : B. \text{Elem}_B b m \implies a \cdot \text{id} \leq b \cdot \text{id}) \implies
\]

$$
\text{OrderedId}_B (a ; m)
\]

$$
\text{OrderedId}_{\text{singleton}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall m : B. \text{OrderedId}_B (\text{singleton } m)
\]

$$
\text{OrderedId}_{\text{exh}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall a : B. \forall m : \text{List } B. \text{OrderedId}_B m \implies
\]

$$
1 =_{\text{List } B} \text{nil } B \lor
\]

$$
(\exists a : B. \exists m : \text{List } B. 1 =_{\text{List } B} a ; m \land
\]

$$
(\forall b : B. \text{Elem}_B b m \implies a \cdot \text{id} \leq b \cdot \text{id}) \land
\]

$$
\text{OrderedId}_B m
\]

$$
\text{OrderedId}_{\text{cons.}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall a : B. \forall m : \text{List } B. \text{OrderedId}_B (a ; m) \implies
\]

$$
(\forall b : B. \text{Elem}_B b m \implies a \cdot \text{id} \leq b \cdot \text{id}) \land \text{OrderedId}_B m
\]

In order to type $\text{OrderedId}$, we need the fact that $(\square, \square) \in R^\leq$. The easiest way to see this, is to consider the type of the predicate. In order to formulate the properties of $\text{OrderedId}$, we need the rule $(\square, *_p) \in R^\leq$. Note that we use the symbol $\leq$ for the subtyping ordering, and $\preceq$ for the usual ordering on numbers.

The predicate $\text{Perm}$ of Section 4.7 is already polymorphic in the ordinary sense, so we use this predicate and its properties without any modification. We continue with the bounded polymorphic insertId program, which inserts a record in a list, so that the ordering on the id field is preserved.

$$
\text{insertId} := \lambda B \leq \{ \text{id} : \text{Nat} \} : *_s. \lambda n : B. \text{primrecList} (\text{singleton } n)
\]

$$
(\text{head} : B. \text{tail}, \text{insert}\_\text{tail} : \text{List } B
\]

$$
(\text{if} (\text{leq } n \cdot \text{id} \text{ head}\_\text{id}) (n ; \text{head} ; \text{tail}) (\text{head} ; \text{insert}\_\text{tail}))
\]

$$
: \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. B \rightarrow \text{List } B \rightarrow \text{List } B
\]

$$
\text{insertId}_{\text{nil}} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall m : B.
\]

$$
\text{insertId} m (\text{nil } B) =_{\text{List } B} \text{singleton } m
\]

$$
\text{Le}\_\text{insertId} := \ldots : \forall B \leq \{ \text{id} : \text{Nat} \} : *_s. \forall m, n : B. \forall l : \text{List } B. m \cdot \text{id} \preceq n \cdot \text{id} \implies
\]

$$
\text{insertId} m (n ; l) =_{\text{List } B} m ; n ; l
\]
8.4. EXAMPLE: SORTING OF RECORDS

\[
\text{Gt\textunderscore insertId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall m, n: B. \forall l: \text{List B}. n \cdot \text{id} < m \cdot \text{id} \implies \text{insertId}_m (n; l) = \text{insertId}_n m \cdot \text{insertId}_m l
\]

\[
\text{Elem\textunderscore insertId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall m, n: B. \forall ns: \text{List B}.
\quad \text{Elem}_B m (\text{insertId}_n ns) \implies m = n \lor \text{Elem}_B m ns
\]

\[
\text{OrderedId\textunderscore insertId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall m, n: B. \forall l: \text{List B}. \text{OrderedId}_B l \implies \text{OrderedId}_B (\text{insertId}_m l)
\]

\[
\text{Perm\textunderscore insertId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall m, n: B. \forall l: \text{List B}.
\quad \text{Perm}_B (\text{insertId}_m l) (m; l)
\]

For the definition of \text{insertId} we need that \((\Box, *_\lambda) \in \mathcal{R}^<\).

Using \text{insertId} and its properties, it is straightforward to define \text{sortId} and show its correctness.

\[
\text{sortId} := \ldots \quad \lambda B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda. \text{primreclist} (\text{nil B})
\quad (\lambda \text{head}: B. \lambda \text{tail}, \text{sort\textunderscore tail}: \text{List B}.
\quad \text{insertId}_m \text{head} \text{sort\textunderscore tail})
\quad : \Pi B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda. \text{List B} \to \text{List B}
\]

\[
\text{sortId\textunderscore nil} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda. \text{sortId} (\text{nil B}) = \text{nil B}
\]

\[
\text{sortId\textunderscore cons} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall m, n: B. \forall l: \text{List B}.
\quad \text{sortId} (m; l) = \text{insertId}_m (\text{sortId}_l)
\]

\[
\text{OrderedId\textunderscore sortId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall l: \text{List B}. \text{OrderedId}_B (\text{sortId}_l)
\]

\[
\text{Perm\textunderscore sortId} := \ldots \quad \forall B \subseteq \{ \text{id}: \text{Nat} \} : *_\lambda, \forall l: \text{List B}. \text{Perm}_B (\text{sortId}_l)
\]

The proofs of all properties are very similar to the proofs in Section 4.7. It is possible to specify and define a more general program \text{sortGen}: \Pi B: *_\lambda, \forall l: \text{List B}. \text{sortGen} l : \text{List B} \to \text{List B}, that is polymorphic and that has the ordering to be used as additional parameter, and define both \text{sort} and \text{sortId} in terms of \text{sortGen}. In practice, this is a more elegant approach, but since our sorting programs serve as examples of the use of the various systems rather than as elements of a practical and general library, we have not taken this approach.
Chapter 9

Encoding Objects

In this chapter we work out the existential model of [PT94] sketched in Chapter 5. We do this by applying this model to the example program given there, which is repeated in Figure 9.1. We summarize here briefly our explanation of OOP, as given in Section 5.1 (for the explanation of the example program we refer to Section 5.2). Then we give the 6 steps of the existential model.

An object is the combination of a state and a set of methods that operate on the state. This combining is called aggregation. The state, consisting of a set of instance variables, cannot be accessed directly, but only through the methods (encapsulation). Objects are generated by classes. A class describes the type of state, the initial state, and implementation of the methods. A class can be defined from scratch, but also as an extension of an existing class. The new class (the subclass) inherits the instance variables and the methods of the existing class (the superclass), and typically adds some instance variables and methods. Methods of the superclass can also be overridden (redefined) in the subclass. So an object of a subclass has all the functionality (methods) of objects of its superclass. The mechanism of subtyping therefore allows an object of a subclass to be used at any place where an object of the superclass is expected. The implementation of a method may use other methods of the class, by self reference. There are a few options for the precise meaning of self reference. We have chosen for late binding, i.e. self always refers to the methods of the used object, and not necessarily to the definitions in which self was used.

In the rest of this chapter we develop the existential model for OOP in 6 steps, where each step corresponds with a section. In each step a few notions given above are modelled, by expressing (parts of) the example program in $\lambda\omega_L$ or one of its extensions. These steps are as follows.

1. Aggregation The instance variables and methods are combined (aggregated) into one value, which forms an object. We define so-called invocation functions that model the invocation of a method by selecting the method from the object and let it work on the state of the object. The initial values for the instance variables and the implementation of the methods, together with the corresponding types form a class. In order to model aggregation we need records, so we use system $\lambda\omega^+_L$.

2. Encapsulation We hide the state of the object, so that the state is only accessible through the methods. Encapsulation is achieved by using existential types. We pack the type of the state and the aggregated value (of step 1) into an other value, which will henceforth
class Point is
   vars x : Nat = 0
with
   getX = state.x,
   setX = \n:Nat. state{x=n},
   bump = self.setX state (S (self.getX state))
end

class ColPoint from Point is
   vars c : Colour = red
with
   getC = state.c,
   setC = \col:Colour. state{c=col},
   setX = \n:Nat. let state' := super.setX state n in
         self.setC state' blue
end

Figure 9.1: An example of object-oriented programming

represent an object. From this package, only the methods are accessible, whereas the
type of the state is abstract. The type of such object is an existential type, provided in
$\lambda \omega^+_L$.

3. Subtyping This is the possibility to use an object generated by a subclass whenever
objects of the superclass are expected. In particular, the invocation functions of a class
must also work on objects of subclasses. We need the subtyping mechanism as provided
by system $\lambda \omega^0_L$ to model this.

4. Inheritance We define methods in such a way that a subclass can inherit methods from
a superclass. We also show how method overriding is modelled. In order to model
inheritance we need a special form of subtyping called width-subtyping.

5. Self reference We introduce the fixed point combinator to model that the implementa-
tion of one method can use other methods of the same class.

6. Late binding By applying the fixed point combinator (introduced in step 5) at an other
place we model late binding.

The models we give in the first three steps use type systems which are presented in previous
chapters of this thesis. In the last three steps, we introduce new extensions of the type system,
but we will not consider theoretical aspects of these extensions. Sections 9.1 through 9.6 work
out the six steps. Section 9.7 gives a summary of the model, Section 9.8 discusses how we
intend to find proof rules for objects using the model, and we give a comparison to related
work in Section 9.9.
9.1 Aggregation, in $\lambda\omega_L^+$

In a traditional (non OO) style, we would write the piece of code for manipulating points as follows. We would define PointRep, the type representing the state, as a record type, the initial state pointInit, and three functions that manipulate the state.

\[
\begin{align*}
\text{PointRep} & := \{ x : \text{Nat} \} \\
\text{pointInit} & := \{ x = 0 \} \\
\text{getX} & := \lambda \text{st}: \text{PointRep}. \text{st} \cdot x \\
\text{setX} & := \lambda \text{st}: \text{PointRep}. \lambda n: \text{Nat}. \{ x = n \} \\
\text{bump} & := \lambda \text{st}: \text{PointRep}. \{ x = S \text{ st} \cdot x \}
\end{align*}
\]

But in OOP the state and the methods are combined into a single entity, viz. an object. We do this in two steps.

First we combine the methods into one value, using a record. Each field of the record is one method, and the label is the name of the method, in the same way as we combine the operations of an ADT (Section 6.3.1).

\[
\begin{align*}
\text{pointMet} & := \{ \text{getX} = \lambda \text{st}: \text{PointRep}. \text{st} \cdot x, \\
& \quad \text{setX} = \lambda \text{st}: \text{PointRep}. \lambda n: \text{Nat}. \{ x = n \}, \\
& \quad \text{bump} = \lambda \text{st}: \text{PointRep}. \{ x = S \text{ st} \cdot x \} \}
\end{align*}
\]

Note that in the definition of the bump field we have unfolded the definition of setX and getX, otherwise we would have a recursive definition, which is not allowed in our type system. We will fix this in Section 9.5 where we model self reference.

It is convenient to abbreviate the type of this record. As preparation for the next section, we do this abstracted from the representation type PointRep. This abstracted type is called the interface; applied to any representation type for the state the interface gives the type of the record of methods. The interface PointI for points is defined as follows.

\[
\begin{align*}
\text{PointI} & := \lambda R : *_s. \{ \text{getX} : R \to \text{Nat}, \\
& \quad \text{setX} : R \to \text{Nat} \to R, \\
& \quad \text{bump} : R \to R \}
\end{align*}
\]

So we have \text{pointMet} : PointI PointRep.

The second step is the pairing of the initial state with the methods, which yields the initial object.

\[
\begin{align*}
\text{newPoint} & := \{ \text{state} = \text{pointInit}, \text{met} = \text{pointMet} \} \\
& := \{ \{ \text{state} : \text{PointRep}, \text{met} : \text{PointI PointRep} \} \}
\end{align*}
\]
Again, it is convenient to abbreviate this type.

\[
\text{Point} := \{\text{state:PointRep, met:PointIPointRep}\}
\]

So Point is the type of all objects of the class of points.

Now we turn towards modelling method invocation. First we consider the \text{getX} method. If \(p\) is a point, the invocation \(p \gg \text{getX}\) means we select \(\text{getX}\) from \(p\)'s methods, and apply this function on \(p\)'s state, i.e. \(p\cdot\text{met}.\text{getX}\ p\cdot\text{state}\). Abstracting over \(p\), we get the following invocation function \text{point'getX}. (The apostrophe in our names, e.g. point'getX, is not an operation.)

\[
\text{point'getX} := \lambda p:\text{Point}. p\cdot\text{met}.\text{getX}\ p\cdot\text{state}
\]

: \text{Point} \rightarrow \text{Nat}

Now we consider the \text{bump} method. The invocation \(p \gg \text{bump}\) should give a new Point value, including methods. So the new state is paired with \(p\)'s methods. The new state is obtained by applying \text{bump} from \(p\)'s methods to \(p\)'s state. So \(p \gg \text{bump}\) is modelled as \text{point'bump} \(p\) where the invocation function is defined as follows.

\[
\text{point'bump} := \lambda p:\text{Point}. \{\text{state} = p\cdot\text{met}.\text{bump}\ p\cdot\text{state}, \text{met} = p\cdot\text{met}\}
\]

: \text{Point} \rightarrow \text{Point}

For \text{setX} the situation is similar, but we have to abstract over \text{setX}'s argument \(n\).

\[
\text{point'setX} := \lambda p:\text{Point}. \lambda n:\text{Nat}. \{\text{state} = p\cdot\text{met}.\text{setX}\ p\cdot\text{state} n, \text{met} = p\cdot\text{met}\}
\]

: \text{Point} \rightarrow \text{Nat} \rightarrow \text{Point}

So \(p \gg \text{setX} n\) is translated to \text{point'setX} \(p\ n\). The first example of the use of objects in Section 5.2, \((\text{newPoint} \gg \text{setX} (\text{S } 0)) \gg \text{getX}\) is translated to

\[
\text{point'getX} (\text{point'setX} \text{newPoint} (\text{S } 0)),
\]

which reduces to \text{S } 0.

Let us give a brief summary of the model given in this section. A class consists of the following 4 parts:

- an interface,
- a representation type for the state,
- an initial state, and
- an implementation of the methods.

Out of the four components of a class we make an object, by aggregation of the initial value and the methods. We model invocation of methods using invocation functions. In the following sections, we will modify only the invocation functions, the way we model the implementation of the methods, and how we make an object from the 4 components.

We ignored coloured points in this section. The first deficit of this model we will fix is the absence of encapsulation: now the state of an object is directly accessible, e.g. \text{newPoint-state\cdot x} is well-typed and reduces to 0.
9.2.  

Encapsulation, in $\lambda \omega_L^+$

In this section we incorporate encapsulation in our model by using the existential types present in $\lambda \omega_L^+$ (Chapter 6). We hide the actual representation type PointRep in the initial point object by using the pack construct as follows.

\[
\text{newPoint} := \text{pack} \{\text{PointRep}, \{\text{state} = \text{pointInit}, \text{met} = \text{pointMet}\}\} : \Sigma X : *_x. \{\text{state} : X, \text{met} : \text{PointIX}\}
\]

where we retain (only) PointI, PointRep, pointInit and pointMet from the previous section. In contrast to the previous model, now we cannot access the state directly. Again, we abbreviate the type of point objects to Point.

\[
\text{Point} := \Sigma X : *_x. \{\text{state} : X, \text{met} : \text{PointIX}\} : *_x
\]

In this definition, the only thing specific for points is the interface PointI. Abstraction over this interface yields the type constructor Object, as follows.

\[
\text{Object} := \lambda I : *_x \rightarrow *_x. \Sigma X : *_x. \{\text{state} : X, \text{met} : \text{IX}\} : (*_x \rightarrow *_x) \rightarrow *_x
\]

So Point can be rewritten as Object PointI. In general, objects belonging to class C have type Object CI, where CI is the interface for C.

In order to invoke methods on a point, we have to unpack the point. First we consider the invocation function for the getX method, which unpacks the point and then proceeds as in the previous section.

\[
\text{point'getX} := \lambda p : \text{Point}. \text{unpack} p \text{ as } (X, x) \text{ in } x \cdot \text{met} \cdot \text{getX} x \cdot \text{state} : \text{Point} \rightarrow \text{Nat}
\]

The bump method should deliver a new object of type Point, so after unpacking, calculating the new state, and pairing this state with the methods, the invocation function should pack this pair into a Point object.

\[
\text{point'bump} := \lambda p : \text{Point}. \text{unpack} p \text{ as } (X, x) \text{ in } \\
\quad \quad \quad \text{pack} \{X, \{\text{state} = x \cdot \text{met} \cdot \text{bump} x \cdot \text{state}, \text{met} = x \cdot \text{met}\}\} \text{ in Point} : \text{Point} \rightarrow \text{Point}
\]

The invocation for setX is defined similarly.

\[
\text{point'setX} := \lambda p : \text{Point}. \lambda n : \text{Nat}. \text{unpack} p \text{ as } (X, x) \text{ in } \\
\quad \quad \quad \text{pack} \{X, \{\text{state} = x \cdot \text{met} \cdot \text{setX} x \cdot \text{state} n, \text{met} = x \cdot \text{met}\}\} \text{ in Point} : \text{Point} \rightarrow \text{Nat} \rightarrow \text{Point}
\]

These invocation functions are used in the same way as in previous section, so we have

\[
\text{point'getX} (\text{point'setX} \text{newPoint} (S 0)) \triangleright_S S 0
\]

As explained in Chapter 6, the existential type $\Sigma X : *_x. T$ has as elements pairs $(A, b)$ with $b : B[X := A]$. So we can make elements of type Point with a different representation type.
For example, we take as representation type a two field record \( \{ x : \text{Nat}, \text{bumped} : \text{Bool} \} \), where bumped keeps track whether the point was ever bumped.

\[
\begin{align*}
\text{PointRep2} & \ := \ \{ x : \text{Nat}, \text{bumped} : \text{Bool} \} \\
\text{pointInit2} & \ := \ \{ x = 0, \text{bumped} = \text{false} \} \\
& \quad : \ \text{PointRep2} \\
\text{pointMet2} & \ := \ \{ \ \begin{array}{l}
\text{getX} = \lambda \text{st} : \text{PointRep2}. \text{st}.x, \\
\text{setX} = \lambda \text{st} : \text{PointRep2}. \lambda n : \text{Nat}. \{ x = n, \text{bumped} = \text{st}.\text{bumped} \}, \\
\text{bump} = \lambda \text{st} : \text{PointRep2}. \{ x = S \text{st}.x, \text{bumped} = \text{true} \}
\end{array} \} \\
& \quad : \ \text{PointRep2} \\
\text{newPoint2} & \ := \ \text{pack} \langle \text{PointRep2}, \{ \text{state} = \text{pointInit2}, \text{met} = \text{pointMet2} \} \rangle \\
& \quad : \ \text{Point}
\end{align*}
\]

Since the methods have the same interface, we have the same type of objects, and we can use the same invocation functions \( \text{point}'\text{setX} \) and so on. We will not further explore the possibility to have other representations for one type of objects.

As preparation for the next section, we consider how to model the class of coloured points. For the moment, we are only interested in the interface \( \text{ColPointI} \), the type of objects \( \text{ColPoint} \), and the type of the initial coloured point object \( \text{newColPoint} \).

\[
\begin{align*}
\text{ColPointI} & \ := \ \lambda \text{R} : \ast_s. \{ \ \begin{array}{l}
\text{getX} : \text{R} \rightarrow \text{Nat}, \\
\text{setX} : \text{R} \rightarrow \text{Nat} \rightarrow \text{R}, \\
\text{bump} : \text{R} \rightarrow \text{R}, \\
\text{getC} : \text{R} \rightarrow \text{Colour}, \\
\text{setC} : \text{R} \rightarrow \text{Colour} \rightarrow \text{R}
\end{array} \} \\
& \quad : \ \ast_s \\
\text{ColPoint} & \ := \ \text{Object} \ \text{ColPointI} \\
& \quad : \ \ast_s \\
\text{newColPoint} & \ := \ \ldots \ : \ \text{ColPoint}
\end{align*}
\]

We assume we have declared in the context a datatype \( \text{Colour} : \ast_s \), with elements \text{red}, \text{white} and \text{blue}.

One shortcoming of the present model is that we cannot invoke a method (for example \( \text{point}'\text{getX} \)) of a superclass (the class of points) on an object (e.g. \( \text{newColPoint} \)) of a subclass (the class of coloured points). For example, the expression \( \text{point}'\text{getX} \text{newColPoint} \) is not well typed, since \( \text{newColPoint} \) has type \( \text{Object ColPointI} \) instead of \( \text{Object PointI} \). In the next section we will fix this.

### 9.3 Subtyping, in \( \lambda \omega^+_L \leq \)

We extend the type system \( \lambda \omega^+_L \) with subtyping, that is, we move to the system \( \lambda \omega^+_{L} \leq \) defined in Chapter 8. The subtyping rules state that

\[
\{ l_1 : A_1, \ldots, l_k : A_k \} \leq \{ l_1 : A_1, \ldots, l_n : A_n \}
\]

if \( k \geq n \) (rule \((\leq\text{-width})\) in Section 8.1).
In particular, we have

\[
\begin{align*}
& \| \text{getX} : R \rightarrow \text{Nat}, \\
& \text{setX} : R \rightarrow \text{Nat} \rightarrow R, \\
& \text{bump} : R \rightarrow R, \\
& \text{getC} : R \rightarrow \text{Colour}, \\
& \text{setC} : R \rightarrow \text{Colour} \rightarrow R \|
\end{align*}
\]

\[
\begin{align*}
& \| \text{getX} : R \rightarrow \text{Nat}, \\
& \text{setX} : R \rightarrow \text{Nat} \rightarrow R, \\
& \text{bump} : R \rightarrow R \|
\end{align*}
\]

(i)

for any datatype \( R \).

In \( \lambda^< \subseteq \ldots \) we have lifted subtyping (rule \((\leq \cdot \lambda))\), that is \( \Gamma \vdash (\lambda X: *, A) \leq (\lambda X: *, B) \) if \( \Gamma, X: *, \Gamma \vdash A \leq B \). This means that by (i) we have

\[ \text{ColPointI} \leq \text{PointI}, \]

where \( \text{ColPointI} \) and \( \text{PointI} \) are defined as in the previous section. We say that the interface \( \text{ColPointI} \) refines \( \text{PointI} \). In general, if \( D \) is a subclass of \( C \), the interface of \( D \) refines (is a subtype of) the interface of \( C \).

By the structural subtyping rules for records and existential types (rules \((\leq \cdot \text{depth}))\) and \((\leq \cdot \Sigma))\), and using the definition of \( \text{Object} \), this means that

\[ \text{Object ColPointI} \leq \text{Object PointI}. \]

and hence \( \text{ColPoint} \leq \text{Point} \). By the subsumption rule it follows that any value with type \( \text{ColPoint} \) now also has type \( \text{Point} \). So the object \( \text{newColPoint} \) has also type \( \text{Point} \), so we can apply the invocation function \( \text{point}' \text{getX} \) to this object. For example, the expression \( \text{point}' \text{getX} \text{newColPoint} \) is well-typed.

The expression \( \text{point}' \text{bump} \text{newColPoint} \) is also well-typed, but with type \( \text{Object PointI} \). This is clearly not desirable, since we have lost type information; bumping a coloured point should result in a new coloured point, and not in an ordinary point. Let us consider the type of \( \text{point}' \text{bump} \).

\[ \text{point}' \text{bump} : \text{Point} \rightarrow \text{Point}. \]

We want \( \text{point}' \text{bump} \) to be applicable to \( \text{Point} \)s (which should deliver values of type \( \text{Point} \)), and to be applicable to \( \text{ColPointI} \)s (which should deliver values of type \( \text{ColPointI} \)). More general, \( \text{point}' \text{bump} \) should be applicable to objects of any subclass of points, and deliver values of the same subclass. So given an interface \( I \) which refines \( \text{PointI} \), \( \text{point}' \text{bump} \) should take an object of type \( \text{Object I} \) and deliver a value of type \( \text{Object I} \). This is expressed in \( \lambda^< \subseteq \ldots \) by a bounded quantification, as follows,

\[ \text{point}' \text{bump} : \Pi I \leq \text{PointI}: *, *, \text{Object I} \rightarrow \text{Object I}. \]

So \( \text{point}' \text{bump} \) is a polymorphic function, and takes as first argument the interface \( I \), which should be a subtype of \( \text{PointI} \). We can now form the expression \( \text{point}' \text{bump} \text{ColPointI} \) with type \( \text{Object ColPointI} \rightarrow \text{Object ColPointI} \), i.e. \( \text{ColPoint} \rightarrow \text{ColPoint} \). So the invocation of the \( \text{bump} \) method on \( \text{newColPoint} \) is now written as \( \text{point}' \text{bump} \text{ColPointI} \text{newColPoint} \) and has the correct type \( \text{ColPoint} \).
The definition of point 'bump so that it has the desired type is a simple generalization of the definition in the previous section.

\[
\text{point'\ bump} \ := \ \lambda I \leq \text{PointI}: \ast_s \rightarrow \ast_s. \ \lambda p: \text{Object I}. \\
\text{unpack p as } \langle x, x \rangle \text{ in} \\
\text{pack } \langle x, \{\text{state} = x \cdot \text{met}.\text{bump} \ x \cdot \text{state}, \text{met} = x \cdot \text{met}\} \rangle \text{ in Object I} \\
: \ \Pi I \leq \text{PointI}: \ast_s \rightarrow \ast_s, \ \text{Object I} \rightarrow \text{Object I}
\]

Using exactly the same mechanism, we make point 'setX polymorphic.

\[
\text{point'setX} \ := \ \lambda I \leq \text{PointI}: \ast_s \rightarrow \ast_s. \ \lambda p: \text{Object I}. \ \lambda n: \text{Nat}. \\
\text{unpack p as } \langle x, x \rangle \text{ in} \\
\text{pack } \langle x, \{\text{state} = x \cdot \text{met}.\text{setX} x \cdot \text{state} n, \text{met} = x \cdot \text{met}\} \rangle \text{ in Object I} \\
: \ \Pi I \leq \text{PointI}: \ast_s \rightarrow \ast_s, \ \text{Object I} \rightarrow \text{Nat} \rightarrow \text{Object I}
\]

And in order to be consistent, we adapt the definition of point 'getX too (even though it's not really necessary).

\[
\text{point'getX} \ := \ \lambda I \leq \text{PointI}: \ast_s \rightarrow \ast_s. \ \lambda p: \text{Object I}. \\
\text{unpack p as } \langle x, x \rangle \text{ in} \ x \cdot \text{met}.\text{getX} \ x \cdot \text{state} \\
: \ \Pi I \leq \text{PointI}: \ast_s \rightarrow \ast_s, \ \text{Object I} \rightarrow \text{Nat}
\]

We also give the invocation functions for the new methods of coloured points, which also use bounded quantifications, since the user might want to define subclass of coloured points.

\[
\text{colPoint'getC} \ := \ \lambda I \leq \text{colPointI}: \ast_s \rightarrow \ast_s. \ \lambda p: \text{Object I}. \\
\text{unpack p as } \langle x, x \rangle \text{ in} \ x \cdot \text{met}.\text{getC} \ x \cdot \text{state} \\
: \ \Pi I \leq \text{colPointI}: \ast_s \rightarrow \ast_s, \ \text{Object I} \rightarrow \text{Colour}
\]

\[
\text{colPoint'setC} \ := \ \lambda I \leq \text{colPointI}: \ast_s \rightarrow \ast_s. \ \lambda p: \text{Object I}. \ \lambda \text{col}: \text{Colour}. \\
\text{unpack p as } \langle x, x \rangle \text{ in} \\
\text{pack } \langle x, \{\text{state} = x \cdot \text{met}.\text{setC} x \cdot \text{state} \text{col}, \text{met} = x \cdot \text{met}\} \rangle \text{ in Object I} \\
: \ \Pi I \leq \text{colPointI}: \ast_s \rightarrow \ast_s, \ \text{Object I} \rightarrow \text{Colour} \rightarrow \text{Object I}
\]

At this moment, we already model objects, classes, encapsulation, interfaces and subtyping between interfaces. The only concepts missing are inheritance of classes, self reference and late binding.

### 9.4 Inheritance, with Width-Subtyping

In Section 9.4.1 we give part of the model, and explain why we need width-subtyping. In Section 9.4.2 we formally introduce width-subtyping, and in Section 9.4.3 we use it in the model.

#### 9.4.1 The Model

A subclass inherits two things from its superclass: a set of instance variables and a set of methods. First we model inheritance of instance variables, which goes straightforward, and
then the inheritance of methods, for which an extension of our type system is necessary. Our model of the inheritance of methods differs from the existential model given in [PT94].

The subclass of coloured points inherits the instance variable $x : \text{Nat}$ from its superclass, and adds an instance variable $c : \text{Colour}$. So we define

\[
\text{ColPointRep} := \{x : \text{Nat}, c : \text{Colour}\}
\]

The initial value for $x$ is also inherited, so we define

\[
\text{colPointInit} := \{x = \text{pointInit}\cdot x, \\
\quad c = \text{red}\}
\]

: \text{ColPointRep}

Now we consider how to model inheritance of methods, i.e. how to define \text{colPointMet}, the record with implementations for coloured points, in terms of \text{pointMet}. Since the class of coloured points inherits the methods \text{getX} and \text{bump} from its superclass, we would like to write the following.

\[
\text{colPointMet} := \{\text{getX} = \text{pointMet}\cdot \text{getX}, \\
\text{bump} = \text{pointMet}\cdot \text{bump}, \\
\text{setX} = \ldots, \\
\text{getC} = \lambda \text{state} : \text{ColPointRep}. \text{state}\cdot c, \\
\text{setC} = \lambda \text{state} : \text{ColPointRep}. \lambda \text{col} : \text{Colour}. \{x = \text{state}\cdot x, c = \text{col}\}\}
\]

: \text{ColPoint}\text{I}\text{ColPointRep}

(for the moment, we do not consider the \text{setX} method, which is overridden.) But this definition does not have the indicated type: the \text{bump} field has type \text{PointRep} \to \text{PointRep} instead of \text{ColPointRep} \to \text{ColPointRep}. Abbreviating the definition of \text{pointMet}\cdot \text{bump} to \text{bmp}, so \text{bmp} = \lambda \text{state} : \text{PointRep}. \{x = S \text{state}\cdot x\}, we see that the origin of this typing problem lies in the body of this definition: \text{bmp} makes a new record, always consisting of one field. Intuitively, we want \text{bmp} to have as argument a state with an \emph{x} field of type \text{Nat}, and to deliver the same state with the \emph{x} field updated, and retaining all other fields. So \text{bmp} should be a so-called polymorphic record update: given any record type \emph{R} with a field \emph{x} : \text{Nat} (hence “polymorphic”), and a record of type \emph{R}, it should deliver a record of type \emph{R} with an updated \emph{x} field. There are several ways to extend the second-order \lambda-calculus with polymorphic record updates (e.g. [HP96, Zwa95, Pol97]), and we have chosen the approach of [Pol97], since it is the simplest. This extension consists of an update operation and a new form of subtyping, called width-subtyping. Width-subtyping is really necessary, since ordinary subtyping is not sufficient to type update operations. It is possible to encode record updating in the programming language, as done in the standard version of the existential model [PT94], but this encoding is awkward. We prefer record updating as a primitive, even though that means deviating slightly from this standard version.

A record update is of the form $r\{l := a\}$, e.g. we will write \text{state}\{x := S (\text{state}\cdot x)\} for the body of \text{bmp}. If \text{state} = \{x = 0, c = \text{red}\}, this body will be equal to \{x = S 0, c = \text{red}\}. Now we know what the body of \text{bmp} should be, we consider the complete definition of \text{bmp}. It should be polymorphic, so that it can be applied to any value of type \emph{R}, where \emph{R} is a record type with an \emph{x} field of type \text{Nat}, and produce a new value of the same type \emph{R}. This suggests
we should define \texttt{bmp} as follows.

\[
\texttt{bmp} := \lambda R \subseteq \text{PointRep} : \ast, \lambda \text{state} : R. \text{state}(x := S(\text{state} \cdot x)) \\
: \Pi R \subseteq \text{PointRep} : \ast, R \rightarrow R
\]

But this is not a good definition. The subtyping demand \( R \subseteq \text{PointRep} \) does not ensure \( R \) has an \( x \) field of type \( \text{Nat} \). By depth subtyping for records, \( R \) may also have an \( x \) field with a more specific type than \( \text{Nat} \). For example, if \( \text{Even} \), the set of even numbers, is a subtype of \( \text{Nat} \), then also \( \emptyset x : \text{Even} \emptyset \leq \emptyset x : \text{Nat} \emptyset \). This clearly leads to a problem: \texttt{bmp} applied to type \( \emptyset x : \text{Even} \emptyset \) and value \( \{ x = e \} \) for some \( e : \text{Even} \) gets type \( \emptyset x : \text{Even} \emptyset \), but reducing this expression would lead to the value \( \{ x = S e \} \), of which the \( x \) field is odd.

Here the width-subtyping comes in. This is a new relation \( \subseteq \) on record types. We have \( A \subseteq B \) only if \( A \) has all the fields of \( B \) (and possibly more), so we do not have depth subtyping for \( \subseteq \). So the proper definition of \texttt{bmp} is as follows:

\[
\texttt{bmp} := \lambda R \subseteq \text{PointRep}. \lambda \text{state} : R. \text{state}(x := S(\text{state} \cdot x)) \\
: \Pi R \subseteq \text{PointRep}. R \rightarrow R
\]

So in analogy with bounded quantifications which put an ordinary subtyping demand on a type variable, we have width-bounded quantifications, which put a width-subtyping demand on a type variable.

The problem with the type \( \emptyset x : \text{Even} \emptyset \) we sketched above does not occur any longer, since \( \emptyset x : \text{Even} \emptyset \) is not a width-subtype of \( \emptyset x : \text{Nat} \emptyset \). On the other hand, we can apply \texttt{bmp} to \texttt{ColPointRep}, since \texttt{ColPointRep} \( \subseteq \) \texttt{PointRep}.

**Remark 9.4.1.1** Now we have two forms of subtyping, viz. ordinary subtyping \((\leq)\), and width-subtyping \((\subseteq)\). It might seem simpler to have just one form of subtyping, by using only \( \subseteq \) and removing \( \leq \). There are two reasons why we retain \( \leq \). First, the property of Minimal Typing (see Section 7.2.5) is lost if \( \leq \) is removed, since \( \subseteq \) has no structural rules, e.g. for \( \rightarrow \) types. Second, since \( \subseteq \) does not have depth subtyping for records, we do not have \texttt{ColPoint} \( \subseteq \) \texttt{Point}, so a coloured point object can not longer be considered as a point. Since the feature to consider an object of a subclass as an object of a superclass is essential in OOP, we need both forms of subtyping.

Before we elaborate on the model that uses record updates and width-subtyping, we give a more formal account of the extension of our type system with these notions, adapted from [Pol97].

### 9.4.2 Syntax and Typing Rules for Width-Subtyping

The terms are extended with record update and width-bounded abstractions and quantifications as follows.

**Definition 9.4.2.1** The set of pseudoterms is extended with:

\[
T ::= \ldots \mid T(l := T) \mid \lambda V \subseteq T. T \mid \Pi V \subseteq T. T
\]

In order to type width-bounded quantifications, it is necessary to allow width-subtyping declarations in contexts.
Definition 9.4.2.2 The set of pseudocontexts is extended with
\[ \Gamma, X \subseteq A \in \mathcal{C} \text{ if } \Gamma \in \mathcal{C}, A \in T, X \in V \text{ and } X \notin \text{FV}(\Gamma) \cup \text{FV}(A), \]
and the notion FV for contexts is extended with \( \text{FV}(\Gamma, X \subseteq A) = \text{FV}(\Gamma) \cup \{X\} \cup \text{FV}(A) \). \( \Box \)

We have to introduce two new reduction rules. One for width-bounded abstractions, similar to reduction for ordinary and bounded abstractions, and one to capture the computational effect of performing a record update.

Definition 9.4.2.3 The reduction rules are extended with:
\[ (\lambda X \subseteq A. \ b) \ B \ \triangleright_{\beta} \ b[X := B] \]
\[ (a\{\{l := b\}\})\cdot\ l' \ \triangleright_{\beta} \begin{cases} \ a\cdot l & \text{if } l \neq l' \\ \ b & \text{if } l = l' \end{cases} \]
and all the compatibility rules. \( \Box \)

The width-subtyping rules (Definition 9.4.2.4 below) are essentially a subset of the ordinary typing rules. We only have reflexivity (modulo \( \beta \)-conversion), transitivity, width-subtyping by declaration, and \( (\subseteq \text{-width}) \) for width-subtyping between record types. So we do not have e.g. structural rules for arrow, existential and polymorphic types, or lifted width-subtyping for type constructors. Note that \( \subseteq \) is defined for all pseudoterms, just as \( \leq \) is defined for all pseudoterms.

Definition 9.4.2.4 The width-subtyping rules are defined by:
\[
\begin{align*}
(\subseteq\text{-refl}) & \quad B =_{\beta} B' \quad \frac{}{\Gamma \vdash B \subseteq B'} \\
(\subseteq\text{-trans}) & \quad \Gamma \vdash A \subseteq B \quad \Gamma \vdash B \subseteq C \quad \frac{}{\Gamma \vdash A \subseteq C} \\
(\subseteq\text{-var}) & \quad \frac{}{\Gamma \vdash X \subseteq A} \\
(\subseteq\text{-width}) & \quad \frac{\Gamma \vdash \llbracket \ l_1 : A_1, \ldots, l_m : A_m \rrbracket \llbracket \ l_1 : A_1, \ldots, l_n : A_n \rrbracket \ \text{m} \geq n}{m \geq n} 
\end{align*}
\]

The most interesting new typing rule (Definition 9.4.2.5 below) is (Rec-update). It says that if \( a : A \) and \( A \subseteq \llbracket \ l : B \rrbracket \), i.e. \( A \) has an \( l \) field of type \( B \), we may update the \( l \) field of \( a \) with \( b : B \). The rules (WII-form), (WII-intro) and (WII-elim) are the width-subtyping analogies of (BII-form) through (BII-elim) (Section 7.1.1). We need the premise \( \Gamma \vdash A' : * \) in rule (WII-elim) since \( A' \subseteq A \) does not guarantee \( A' \) is a term (\( \subseteq \) is defined for all pseudoterms). The differences with the rules for ordinary bounded quantifications are that we do not write the type of the width-bound since it is always \( * \), and that the formation is not parametrized by a set \( \mathcal{R} \subseteq \). Rule (Wvar) expresses that any variable declared as width-subtype is a datatype, and (C-width) says only record types may be used as width-bound. One difference with [Pol97] is that we do not have a direct subsumption rule for \( \leq \); instead we say with rule \( (\subseteq \leq) \) that \( A \leq B \) follows from \( A \subseteq B \), so that we may use the ordinary subsumption rule to infer \( a : B \).
from $a : A$ and $A \sqsubseteq B$. This rule ($\leq \sqsubseteq$) is only necessary to derive $\Gamma \vdash X \leq B$ whenever $X \sqsubseteq B \in \Gamma$, since the width-subtyping rules (apart from $\sqsubseteq$-var) form a subset of the ordinary subtyping rules.

**Definition 9.4.2.5** The typing, well-formedness and subtyping rules are extended with:

(Rec-update) $\Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma \vdash A \sqsubseteq \{l : B\}$

\[ \Gamma \vdash \{l := b\} : A \]

(WI-form) $\Gamma, X \sqsubseteq A \vdash B : *$

\[ \Gamma \vdash (\Pi X \sqsubseteq A. B) : * \]

(WI-intro) $\Gamma, X \sqsubseteq A \vdash b : B \quad \Gamma \vdash (\Pi X \sqsubseteq A. B) : *$

\[ \Gamma \vdash (\lambda X \sqsubseteq A. b) : (\Pi X \sqsubseteq A. B) \]

(WI-elim) $\Gamma \vdash b : (\Pi X \sqsubseteq A. B) \quad \Gamma \vdash A' : * \quad \Gamma \vdash A' \sqsubseteq A$

\[ \Gamma \vdash b A' : B[X := A'] \]

(Wvar) $\Gamma \vdash ok$

\[ \Gamma \vdash X : * \quad X \sqsubseteq A \in \Gamma \]

(C-width) $\Gamma \vdash A : * \quad \Gamma \vdash A \sqsubseteq \emptyset$

\[ \Gamma, X \sqsubseteq A \vdash ok \]

($\leq \sqsubseteq$) $\Gamma \vdash A \sqsubseteq B$

\[ \Gamma \vdash A \leq B \]

\[ \Box \]

**Definition 9.4.2.6** The system $\lambda \omega^+_{\leq \sqsubseteq}$ is the extension of $\lambda \omega^+_{\leq}$ with the rules given in Definitions 9.4.2.1, 9.4.2.2, 9.4.2.3, 9.4.2.4 and 9.4.2.5.

\[ \Box \]

We will not formally develop the meta-theory for $\lambda \omega^+_{\leq \sqsubseteq}$, but only conjecture that the meta-theory for $PTS_{\sqsubseteq}$ can extended without major problems to a theory for $\lambda \omega^+_{\leq \sqsubseteq}$. We think so, because $\sqsubseteq$ is a much simpler relation than the already present $\leq$, and the subtyping algorithm is straightforwardly extended with rule ($\leq \sqsubseteq$). Although we could easily have implemented $\lambda \omega^+_{\leq \sqsubseteq}$ in Yarrow, we have not actually done this, so none of the following formal definitions have been mechanically type-checked.

### 9.4.3 On with the Model

Now that we have given the syntax and typing rules for $\lambda \omega^+_{\leq \sqsubseteq}$, in which we can make polymorphic record updates, we use this system for our model. We could define `pointMet` as a record where fields `bump` and `setX` are defined as such polymorphic record updates, but factorizing over the bounded abstraction leads to a more concise and more convenient definition, as follows. (We retain all definitions from the previous section, except for `pointMet`, `newPoint`, `colPointMet` and `newColPoint`.)

```
pointMet := \lambda R \sqsubseteq PointRep. \{getX = \lambda state:R.state.x, setX = \lambda state:R.\_n:Nat.state(x := n), bump = \lambda state:R.state(x := S state.x)\}
```

\[ \Pi R \sqsubseteq PointRep. PointI R \]
newPoint := pack {PointRep, {state = pointInit, met = pointMet PointRep}}
             : Object PointI

In the definition of the getx method, we use that R ⊆ PointRep implies R ⊆ PointRep, so
state has not only type R, but by subsumption also type PointRep, and hence we may select
the x field of state. Note that in the definition of the initial point object, pointMet is applied
to pointRep, so that the methods in the object have the proper, non-polyomorphic type. The
polymorphism in pointMet is only used for inheritance, as follows.

colPointMet := λR ⊆ ColPointRep. let super := pointMet R.
({getx = super.getx,
  bump = super.bump,
  setx = λstate:R. λn:Nat. let state' := super.setx state n.
                    state'{c := blue},
  getc = λstate:R. state.c,
  setc = λstate:R. λcol:Colour. state{c := col})
  : R ⊆ ColPointRep. ColPointI R

newColPoint := pack {ColPointRep,
                      {state = colPointInit, met = colPointMet ColPointRep}}
              : Object ColPointI

Also colPointMet is defined using a width-bounded abstraction over R ⊆ ColPointRep, since
the user might define subclasses of coloured points. We introduce the definition super for the
methods of the superclass, applied to type R. This definition is well-typed, since by transitivity
R ⊆ PointRep follows from R ⊆ ColPointRep and ColPointRep ⊆ PointRep. The getx and
bump methods for coloured points are inherited and are obtained by simply selecting the
Corresponding field from super. The method setx is not inherited, but overridden, but still
super is used for referring to the method of the superclass.

So we use width-subtyping to make it possible to let a subclass inherit methods, where
this subclass has a larger state, i.e. more instance variables. Ordinary subtyping is used to
apply invocation functions to objects with more methods.

Remark 9.4.3.1 For a truly faithful model of OOP, we need a record extension operation
on types, ⊕, and on values, +. The operation A ⊕ {m : B} is only admitted if A is
convertible to some concrete record type, and a + {m = b} is only admitted if the type of a is
a concrete record type. Using these operations, we can define ColPointRep and colPointInit
as extensions of the corresponding terms for ordinary points, as follows.

ColPointRep := PointRep ⊕ {c : Colour}
              : *

colPointInit := pointRep + {c = red}
              : ColPointRep

For a similar definition of the methods colPointMet, we have to take overriding into account:
the expression a + {m = b} is defined to be an update operation if label m occurs in a, e.g.
{1 = a, m = b, n = c} + {m = d} reduces to {1 = a, m = d, n = c}, and similarly for the
datatype level. Thus we can define

\[ \text{ColPointI} \quad := \quad \lambda R.\:*_s.\text{PointIR} + \\
\quad \{ \text{getC} : R \rightarrow \text{Colour} \} + \\
\quad \{ \text{setC} : R \rightarrow \text{Colour} \rightarrow R \} + \\
\quad \{ \text{setX} : R \rightarrow \text{Nat} \rightarrow R \} \]

\[ \quad : \quad *_s \rightarrow *_s \]

\[ \text{colPointMet} \quad := \quad \lambda R \subseteq \text{ColPointRep}. \text{let super} := \text{pointMet R.} \]
\[ \text{super} + \]
\[ \quad \{ \text{setX} = \lambda \text{state} : R. \lambda n : \text{Nat}. \text{let state'} := \text{super} \cdot \text{setX} \text{ state n.} \} \]
\[ \quad \text{state'}[c := \text{blue}] \} + \]
\[ \quad \{ \text{getC} = \lambda \text{state} : R. \text{state} \cdot c \} + \]
\[ \quad \{ \text{setC} = \lambda \text{state} : R. \lambda \text{col} : \text{Colour}. \text{state}[c := \text{col}] \} \]
\[ \quad : \quad \Pi R \subseteq \text{ColPointRep}. \text{ColPointIR} \]

where + and + associate to the left. In general, such a record extension is a troublesome operation (see Section 9.9), but here the extension is innocent, since the involved types here are always concrete record types (and not type variables, for instance). In fact, by this restriction to concrete record types, our extension operations can be considered as just syntactic sugar.

\[ \square \]

9.5 Self Reference, with the Fixed Point Combinator

In the example given in Figure 9.1, the bump method for points is defined using the getx and setx methods accessed by self reference. In the previous sections we ignored the self reference in this example and unfolded the definitions of getx and setx. In this section we do model this self reference.

Consider the definition of pointMet given in the previous section. We would like to define the bump field as

\[ \text{bump} = \lambda \text{state} : R. \text{self} \cdot \text{setX} \text{ state (S (self} \cdot \text{getX} \text{ state))} \]

where self of type PointIR is equal to pointMet R. But since we are defining pointMet, this means we have a recursive definition. In typed lambda calculi, recursive definitions are not admitted, and are usually emulated using the fixed point combinator Y. We formally introduce this combinator before we proceed with the model.

The fixed point combinator

**Definition 9.5.1** The system \( \lambda \omega^{+ \subseteq Y} \) is the extension of \( \lambda \omega^{+ \subseteq \Pi} \) with \( \Pi A: *_s. (A \rightarrow A) \rightarrow A \) and the following reduction rule.

\[ Y A f \triangleright_{\beta} f (Y A f) \]

We will leave the type argument of \( Y \) implicit.

By adding the fixed point combinator, we lose Strong Normalization. Furthermore, it is incompatible with many of the lemmas for datatypes given in Chapter 4. For example, we
9.5. SELF REFERENCE, WITH THE FIXED POINT COMBINATOR

have the lemma ∀m : Nat. m ≠ S m. But in presence of the fixed point combinator we have
Y S = S (Y S), which is a counterexample to this lemma.

So if we want to find proof rules for objects (with self reference), using the fixed point
combinator as presented here makes things a lot more complicated, and it would be nice to
have an alternative. But we proceed and use this combinator to show how self reference can
be modelled in an extension of ⇨ω ≤, and leave finding a sensible alternative as future work.

The model

Using the fixed point combinator, we model self reference as follows. The collection of methods
pointMet is a width-bounded abstraction over R ⊆ PointRep as before, and the body is the
fixed point over the function that given self:PointIR, i.e. a set of methods, returns a set
of methods in which self can be used.

pointMet := λR ⊆ PointRep. Y (λself:PointIR.
{getX = λstate:R. state·x,
setX = λstate:R. λn:Nat. state{x := n},
bump = λstate:R. self·setX state (S (self·getX state)))
) : ΠR ⊆ PointRep. PointIR

This definition reduces to the definition given in the previous section: given some R ⊆ PointRep, the term pointMet R reduces as follows, where the subterm (λself:PointIR {...})
is abbreviated to selfFun.

pointMet R

⇒ β6 Y selfFun

⇒ β6 {getX = ..., setX = ...,
bump = λstate:R. (Y selfFun)·setX state (S ((Y selfFun)·getX state))
⇒ β6 {getX = ..., setX = ...,
bump = λstate:R. state{x := S state·x}}

The methods for coloured points, in which setX is overridden and uses self reference are
defined similarly.

{getX = super·getX,
bump = super·bump,
setX = λstate:R. λn:Nat. let state′ := super·setX state n.
self·setC state′ blue,
getC = λstate:R. state·c,
setC = λstate:R. λcol:Colour. state[c := col]})
) : ΠR ⊆ ColPointRep. ColPointIR

Also colPointMet is convertible to colPointMet as given in the previous section. So we still have

colPoint ′getC ColPointI (point ′bump ColPointI newColPoint) ⇒ β6 red .

This means we have not modelled late binding yet (see the end of Section 5.2).
9.6 Late Binding

In the model of the previous section, self always refers to the group of methods in which self is used, because self references are immediately resolved by taking the fixed point. However, by the concept of late binding, the actual meaning of self in a definition of a method should only be determined when we know for which class the method is used. For example, if we invoke the inherited bump method on a coloured point, self-setX in the definition of bump should refer to the setX method for coloured points. In order to model this, we abstract the group of methods pointMet over the group of methods which should be substituted for self.

\[
\text{pointMet} := \lambda R \subseteq \text{PointRep}. \lambda \text{self} : \text{PointIR}.
\begin{align*}
& \{ \text{getX} = \lambda \text{state} : \text{R}. \text{state}-x, \\
& \text{setX} = \lambda \text{state} : \text{R}. \lambda n : \text{Nat}. \text{state}(x := n), \\
& \text{bump} = \lambda \text{state} : \text{R}. \text{self}-\text{setX state}(S(\text{self-getX state})) \}
\end{align*}
\]

Where self references, since we know to what methods self should refer, namely those for ordinary points.

\[
\text{newPoint} := \text{pack } \{ \text{state} = \text{pointInit}, \text{met} = Y(\text{pointMet PointRep}) \}
\]

: Object PointIR

This newPoint is convertible to newPoint as defined in Section 9.4. But consider the group of methods for coloured points and the initial coloured point, defined as follows.

\[
\text{colPointMet} := \lambda R \subseteq \text{ColPointRep}. \lambda \text{self} : \text{ColPointIR}. \text{let super} := \text{pointMet R self}. \\
\begin{align*}
& \{ \text{getX} = \text{super-getX}, \\
& \text{bump} = \text{super-bump}, \\
& \text{setX} = \lambda \text{state} : \text{R}. \lambda n : \text{Nat}. \text{state}' := \text{super-setX state} n. \\
& \text{self-setC state}' \text{blue}, \\
& \text{getC} = \lambda \text{state} : \text{R}. \text{state}-c, \\
& \text{setC} = \lambda \text{state} : \text{R}. \lambda \text{col} : \text{Colour}. \text{state}(c := \text{col}) \}
\end{align*}
\]

: \Pi R \subseteq \text{ColPointRep}. \text{ColPointIR} \rightarrow \text{ColPointIR}

\[
\text{newColPoint} := \text{pack } \{ \text{ColPointRep}, \\
\begin{align*}
& \{ \text{state} = \text{colPointInit}, \text{met} = Y(\text{colPointMet ColPointRep}) \}
\end{align*}
\]

: Object ColPointIR

Again, self references are only resolved when we define the initial coloured point object, by calculating the fixed point of colPointMet ColPointRep. By the definition of super in colPointMet, viz. pointMet R self, we use a version of pointMet where self refers to the methods of coloured point objects. We illustrate this for the bump method with the following reduction sequence, starting with the bump field of the methods in newColPoint.

\[
(Y(\text{colPointMet ColPointRep}))-\text{bump}
\]

\[
\beta\eta \quad ((\text{self} ; \ldots ; \text{let super} := \text{pointMet ColPointRep self}. \{ \text{bump} = \text{super-bump}, \ldots \}))
\]

\[
(Y(\text{colPointMet ColPointRep}))-\text{bump}
\]

\[
\beta\eta \quad (\text{pointMet ColPointRep}(Y(\text{colPointMet ColPointRep}))-\text{bump}
\]

\[
\eta \quad \text{let self} := (Y(\text{colPointMet ColPointRep})).
\begin{align*}
\lambda \text{state} : \text{ColPointRep}. \text{self-setX state}(S(\text{self-getX state}))
\end{align*}
\]
The last term clearly shows that for self in the definition of bump the setX and getX methods as defined for coloured points are taken. For example, we now have

\[ \text{co} \text{Point } ' \text{set} \text{ColPoint} \text{I } \text{point} ' \text{bump} \text{ColPoint} \text{I } \text{new} \text{ColPoint} >_{\beta_5} \text{blue}, \]

just as the example in the end of Section 5.2 indicates.

Now our model is complete.

9.7 Summary

In this section we summarize the model for OOP built up in the previous 6 sections. First, we define the generic type constructor Object, which expects an interface and returns the type of the objects belonging to classes with that interface.

\[ \text{Object } := \lambda I:s_r \to *_s. \sum X:*_s. \{ \text{state} : X, \text{met} : I X \} \]

\[ : (s_r \to *_s) \to *_s, \]

The record type and existential type present here are used to model aggregation and encapsulation respectively.

Now we consider the four parts defined for each class, illustrated with the points example.

- The interface.

\[ \text{PointI } := \lambda R:*_s. \{ \text{getX} : R \to \text{Nat}, \text{setX} : R \to \text{Nat} \to R, \text{bump} : R \to R \} \]

\[ : *_s \to *_s, \]

The definition of the interface determines the invocation functions, which are used to model the invocation of a method on an object. For example, for PointI we have the invocation functions point'getX, point'setX and point'bump, and the last one is defined as follows.

\[ \text{point'} \text{bump } := \lambda I \leq \text{PointI} : *_s \to *_s. \lambda p: \text{Object I.} \]

\[ \text{unpack p as } \langle X, x \rangle \text{ in } \]

\[ \text{pack } \langle X, \{ \text{state } = x \cdot \text{met} \cdot \text{bump } x \cdot \text{state}, \text{met } = x \cdot \text{met} \} \rangle \text{ in } \text{Object I} \]

\[ : \Pi I \leq \text{PointI} : *_s \to *_s. \text{Object I } \to \text{Object I} \]

Each invocation function is polymorphic; its type is a bounded quantification. This ensures we can use the same invocation functions for objects with a more refined interface, and in particular for objects belonging to any subclass. The interface and the invocation functions are independent of the representation of the state. The following three parts of a class do depend on the representation.

- The representation type for the state, giving the instance variables and their types.

\[ \text{PointRep } := \{ x : \text{Nat } \} \]

\[ : *_s, \]

- The initial state, giving the initial values for the instance variables.

\[ \text{pointInit } := \{ x = 0 \} \]

\[ : \text{PointRep} \]
• The collection of methods.

\[
\text{pointMet} := \lambda R \subseteq \text{PointRep}. \lambda \text{self:PointIR}.
\]
\[
\begin{align*}
\text{getX} &= \lambda \text{state:R}. \text{state}\cdot x, \\
\text{setX} &= \lambda \text{state:R}. \lambda n:\text{Nat}. \text{state}\{x := n\}, \\
\text{bump} &= \lambda \text{state:R}. \text{self}\cdot \text{setX state}(S(\text{self}\cdot \text{getX state}))
\end{align*}
\]
\[
: R \subseteq \text{PointRep}. \text{PointIR} \rightarrow \text{PointIR}
\]

This collection is abstracted over two things. First, it is abstracted over the actual representation type \(R\), which is a width-subtype of the representation type for (the state of) points. This ensures methods can be inherited and used for objects with a larger state. Second, it is abstracted over the methods which should be used for \text{self}, in order to model late binding.

The initial object is defined in terms of the initial state and the collection of methods, where this collection is instantiated with the representation type, and self references are resolved.

\[
\text{newPoint} := \text{pack}(\text{PointRep}, \\
\{\text{state} = \text{pointInit}, \text{met} = Y(\text{pointMet PointRep})\})
\]
\[
: \text{Object PointI}
\]

Let us stress the difference in use between ordinary subtyping (\(\leq\)) and width-subtyping (\(\subseteq\)). Ordinary subtyping allows more refined interfaces, i.e. objects with more \textit{methods}, and width-subtyping allows extension of the state, i.e. objects with more \textit{instance variables}.

If a class is defined by inheritance as subclass of an existing class (the superclass), each of the four parts is an extension of the corresponding part of the superclass. For the interface, representation type and initial state this is straightforward by extension of the respective records, illustrated as follows.

\[
\text{ColPointI} := \lambda R:*_s. \begin{cases}
\text{getX} : R \rightarrow \text{Nat}, \\
\text{setX} : R \rightarrow \text{Nat} \rightarrow R, \\
\text{bump} : R \rightarrow R, \\
\text{getC} : R \rightarrow \text{Colour}, \\
\text{setC} : R \rightarrow \text{Colour} \rightarrow R
\end{cases}
\]
\[
: *_s \rightarrow *_s
\]

\[
\text{ColPointRep} := \begin{cases}
\{x : \text{Nat}, c : \text{Colour}\}
\end{cases}
\]
\[
: *_s
\]

\[
\text{colPointInit} := \{x = \text{pointInit}\cdot x, \\
c = \text{red}\}
\]
\[
: \text{ColPointRep}
\]

The type for each new method in the interface also determines a new invocation function (e.g. for \text{ColPointI} the functions \text{colPoint_get} and \text{colPoint_setC}).

The collection of methods of a subclass is also a record extension of the collection of methods of its superclass, but we have to abstract over the representation type and the self
methods of the subclass. Furthermore, methods may be overridden.

\[
\text{colPointMet} \triangleq \lambda R \subseteq \text{ColPointRep}. \lambda \text{self:ColPoint1 R}. \text{let super:=pointMet R self.}
\]

\[
\{ \text{getX = super-getX,} \\
\quad \text{bump = super-bump,} \\
\quad \text{setX = } \lambda \text{state:R. ln:Nat. let state':=super-setX state n.} \\
\quad \quad \quad \text{self-setC state 'blue,} \\
\quad \text{getC = } \lambda \text{state:R. state-c,} \\
\quad \text{setC = } \lambda \text{state:R. \lambda col:Colour. state\{c := col\}}\}
\]

\[
: \Pi R \subseteq \text{ColPointRep. ColPoint1R } \rightarrow \text{ColPoint1R}
\]

The initial object for a subclass is defined in exactly the same way as the initial object for any class, namely by combining the four parts.

\[
\text{newColPoint} \triangleq \text{pack } \{ \text{ColPointRep,} \\
\quad \{ \text{state = colPointInit, met = Y (colPointMet ColPointRep)}\}\}
\]

\[
: \text{Object ColPoint1}
\]

9.8 Proof Rules for Objects

Our goal is to provide proof rules for objects. By the model for OOP presented in this chapter, which is essentially the model of [PT94], we get a clear overview over the extensions of \(\lambda \omega_L\) usable to model objects, namely records, existential types, subtyping, width-subtyping and the fixed point combinator. Our idea is to first obtain separate proof rules for each of these extensions, and then to combine these rules to obtain proof rules for objects, in the same 6 steps as our development of the model (Sections 9.1 through 9.6). In this thesis, we have studied proof rules for records and existential types (Chapter 6), and for ordinary subtyping (Chapter 8), but we did not develop rules for width-subtyping and the fixed point combinator.

So, first, we are ready to use the proof rules presented in Chapter 6 to develop proof rules for our model of objects as presented in step 2 (encapsulation). Unfortunately, we do not have the time to actually do this here, but we expect that the resulting proof rules will answer the following questions, amongst others.

- What is the notion of equality between objects? The notion of parametricity of Chapter 6 will play an important role here, in order to show that objects with different states but with the same behaviour may be considered equal.

For example, given an interface \(I\) and two objects

\[
o \equiv \text{pack } \{ \text{Rep, } \{ \text{state = s, met = m}\}\}
\]

\[
o' \equiv \text{pack } \{ \text{Rep', } \{ \text{state = s', met = m'}\}\, ,
\]

parametricity for existential types (Section 6.4.3) gives us

\[
(\exists(\sim):\text{Rep } \rightarrow \text{Rep' } \rightarrow s \sim s' \wedge \text{Sim}_I \text{Rep Rep'}(\sim) m m')
\]

\[
\Rightarrow 
\]

\[
o =_{(\text{object } I)} o'.
\]

Interestingly enough, the notion of equality between objects so obtained seems to be the same as in the co-algebraic approach (e.g. [Rei95]), discussed in Section 5.5.
• What is the meaning of quantifications over all objects belonging to class? The notion of invariant (Chapter 6) will probably play a big role here.

Second, we can combine these proof rules with subtyping, to obtain rules for objects as presented in step 3. These new rules should give an answer to the following question.

• What should be the relation between specifications for objects of a subclass and for objects of its superclass?

Third, we should explore the proof rules for width-subtyping, and combine these with rules for objects as in step 3, in order to obtain rules for inheritance.

Fourth, we have to find an alternative for using the fixed point combinator, as this combinator complicates logical matters too much (see Section 9.5). Furthermore, we should find proof rules for this alternative, and combine them in order to obtain rules for objects as in step 5 and 6.

By this stepwise approach we hope to obtain well justified proof rules for objects in a type-theoretical setting.

9.9 Related Work

We discuss here several variants of the existential object model [PT94]; for other approaches on theory for objects we refer to Section 9.5.

The only difference between our model and [PT94] is that we use width-subtyping (Section 9.4), where [PT94] avoids this form of subtyping at the cost of a more complicated model (so they use only the type system $\omega^+ \leq$ with the fixed point combinator). For example, in [PT94] the collection of methods for points is written as follows.

$$
\text{pointMet} := \lambda R: \text{PointRep}. \lambda \text{get}: R \rightarrow \text{PointRep}. \lambda \text{put}: R \rightarrow \text{PointRep} \rightarrow R.
$$

$$
\begin{align*}
\text{get} & = \lambda \text{state}: R. (\text{get state}) \cdot \text{x}, \\
\text{set} & = \lambda \text{state}: R. \lambda n: \text{Nat}. \text{put state} \{x = n\}, \\
\text{bump} & = \ldots
\end{align*}
$$

$$
\Pi R: \text{PointRep}. (R \rightarrow \text{PointRep}) \rightarrow (R \rightarrow \text{PointRep} \rightarrow R) \rightarrow \text{PointRep}
$$

So our width-bounded abstraction ($\lambda R \sqsubseteq \text{PointRep}. b$) is replaced by

$$
(\lambda R: \text{PointRep}. \lambda \text{get}: R \rightarrow \text{PointRep}. \lambda \text{put}: R \rightarrow \text{PointRep} \rightarrow R. b),
$$

and in the body a field selection $\text{state} \cdot \text{x}$ is replaced by $(\text{get state}) \cdot \text{x}$, and a record update $\text{state} \{x := n\}$ is replaced by $\text{put state} \{x = n\}$. When this collection of methods is used for inheritance, viz. to build a coloured point object, the collection is instantiated as follows.

$$
\begin{align*}
R & := \text{ColPointRep} \text{ (just as we do),} \\
\text{get} & := \lambda cr: \text{ColPointRep}. \{x = cr \cdot x\}, \text{ and} \\
\text{put} & := \lambda cr: \text{ColPointRep}. \lambda r: \text{PointRep}. \{x = r \cdot x, c = cr \cdot c\}.
\end{align*}
$$

So this put function implements a record update for $\text{ColPointRep}$ with the fields of a $\text{PointRep}$, and this get function implements the coercion of values of type $\text{ColPointRep}$ to values of type $\text{PointRep}$. This coercion is implicit in our model, where $\text{ColPointRep} \sqsubseteq \text{PointRep}$, so any value of type $\text{ColPointRep}$ may also be considered as having type $\text{PointRep}$. We prefer
our model with width-subtyping because it is simpler and more direct. Therefore we also expect simpler proof rules.

A disadvantage of the calculus we use to model OOP is that we have two kinds of subtyping, namely ordinary subtyping ($\leq$) and width-subtyping ($\leq^w$). The reasons why $\leq^w$ alone is not sufficient are explained in Remark 9.4.1.1: if we have just $\leq$ and no $\leq^w$, we lose the property of Minimal Typing, and furthermore objects of a subclass can no longer be considered as objects of its superclass (e.g. we have not $\text{ColPoint} \leq^w \text{Point}$).

One attempt to improve this situation is the introduction of positive subtyping (denoted here by $\leq^+$) in [HP96]. In contrast to $\leq$, there are structural rules for $\leq^+$, for example $A \to C \leq^+ A \to D$ if $C \leq^+ D$, but in contrast to $\leq$, $\leq^+$ is not contravariant in the domain of $\to$-types, so $A \to C \leq^+ B \to D$ implies $A = B$. Because of the structural rules for $\leq^+$, the calculus with $\leq^+$ has the property of Minimal Typing. However, [HP96] still has the problem that objects of a subclass cannot be considered as objects of its superclass. For a more detailed discussion of the differences between $\leq^+$ and $\leq$ see [Pol97].

Another calculus which simplifies the model by having only one form of subtyping is described in [HP98]. In their calculus, $F_{\leq}^D$, Minimal Typing holds and objects of a subclass can be considered as objects of its superclass. Furthermore, $F_{\leq}^D$ is a second-order system (like $\lambda_2$) instead of a higher-order system (like $\lambda_\omega$). In some aspects, this makes the system much simpler, e.g. because there is no reduction on types. But for our goal, namely to provide proof rules in some extension of $\omega_L$, this advantage is not relevant, since we have reduction on types anyway (e.g. in propositions). In other aspects, e.g. the subtyping rules, $F_{\leq}^D$ is much more complicated than $\omega_L^{+\leq}$.

In our calculus $\omega_L^{+\leq}$ and the calculi of [HP96, HP98], the polymorphic record update is a primitive. Many earlier approaches provided a polymorphic record update in terms of other primitives, e.g. record extension and field removal. One reason for this is that some form of record extension is also necessary to directly model inheritance of methods, i.e. the extension of the record of methods with new methods (see Remark 9.4.3.1). We discuss two of those approaches here.

The calculus in [CM91] has both record extension and field removal. The polymorphic record update function that increases the $x$ field of its argument by one has the following type in this system:

$$\Pi R \leq \{ l : A | x : \text{Nat} \} : R \to R \setminus x | x : \text{Nat},$$

where "$\setminus x$" is the removal of the $l$ field from a record type, and "$l : A$" is the extension of a record type with the $l$ field of type $A$. The bounded quantification in this type cannot be restricted to those types $R$ for which $R \setminus x | x : \text{Nat}$ will be equal to $R$ (i.e. to the width-subtypes of $\{ l : A \} \text{Nat}$). This more complicated type for polymorphic record updates is the major obstacle for using this calculus (and many others based on extension of records, e.g. [HP91, Car92]) to model OOP.

The calculus $F_\#$ defined in [Zwa95] has a very general form of record concatenation, which is usable for both extension and updating of records, but has no field removal. $F_\#$ has two relations on types, namely ordinary subtyping ($\leq$) and compatibility (denoted by $\#$). This compatibility relation holds between record types $A$ and $B$ if $A$ and $B$ have the same type on common fields, e.g. any two disjoint record types are compatible, $\{ l : A, m : B \}$ and $\{ m : B, n : C \}$ are compatible for any $l$, $m$, $n$, $A$, $B$ and $C$, but $\{ l : \text{Nat} \}$ and $\{ x : \text{Bool} \}$ are not compatible. Two record values can only be concatenated if their types are compatible,
so the typing rule for concatenation (notated with "with") is

\[
\Gamma \vdash r : R \quad \Gamma \vdash s : S \quad \Gamma \vdash R \# S \\
\overline{\Gamma} \vdash (r : R) \text{ with } (s : S) : R \text{ With } S
\]

(with-intro)

For technical reasons, the concatenation operation with is decorated with the types of the arguments. The type of the concatenation of \(r : R\) and \(s : S\) is the concatenation \(R \text{ With } S\) of their respective types (in \(\text{Zwa95}\) we wrote \(R \land S\)). If \(R\) and \(S\) are disjoint, the meaning of the concatenation \((r : R)\text{ with } (s : S)\) is just the union of all fields of \(r\) and \(s\), e.g.

\[
\left(\{(x = 0, y = 0) : \prod x : \text{Nat}, y : \text{Bool} \}\right) \text{ with } \left(\{(z = 0) : \prod z : \text{Nat} \}\right) = \{(x = 0, y = 0, z = 0)\}
\]

If \(R\) and \(S\) are not disjoint, we use the values in the record \(s\) (on the right-hand side of with) for the common fields, e.g.

\[
\left(\{(x = 0, y = 0) : \prod x : \text{Nat}, y : \text{Bool} \}\right) \text{ with } \left(\{(x = S 0) : \prod x : \text{Nat} \}\right) = \{(x = S 0, y = 0)\}
\]

By this overwriting of the common fields by the record on the right-hand side we can implement a polymorphic record update, as follows.

\[
\lambda R \leq \left\{ \prod x : \text{Nat} \right\}; R \# \left\{ \prod x : \text{Nat} \right\}. \lambda r : R. (r : R) \text{ with } \left(\{(x = S (r \cdot x)) : \prod x : \text{Nat} \}\right)
\]

\[
\Pi R \leq \left\{ \prod x : \text{Nat} \right\}; R \# \left\{ \prod x : \text{Nat} \right\}. R \rightarrow R \text{ With } \left\{ \prod x : \text{Nat} \right\}
\]

Here a so-called restricted abstraction (\(\lambda R \leq \left\{ \prod x : \text{Nat} \right\}; R \# \left\{ \prod x : \text{Nat} \right\}. b\)) and restricted quantification are used. In this restriction abstraction we impose both a subtyping demand (\(R \leq \left\{ \prod x : \text{Nat} \right\}\)) and a compatibility demand (\(R \# \left\{ \prod x : \text{Nat} \right\}\)) on \(R\). The former is necessary to type the selection \(r \cdot x\), and the latter is necessary to allow the concatenation. The combination of both demands ensures \(R\) has an \(x\) field of type \(\text{Nat}\) (if \(R\) would have an \(x\) field with as type a real subtype of \(\text{Nat}\), \(R\) fails the compatibility demand). In general, the compatible subtypes are exactly the width-subtypes, i.e. \((A \leq B \text{ and } A \# B) \Leftrightarrow A \subseteq B\). A final word about the polymorphic record update we gave. Its type may be simplified to

\[
\Pi R \leq \left\{ \prod x : \text{Nat} \right\}; R \# \left\{ \prod x : \text{Nat} \right\}. R \rightarrow R,
\]

since \(R \text{ With } \left\{ \prod x : \text{Nat} \right\}\) is a subtype of \(R\).

The advantage of \(\mathbb{F}_\#\) over the calculus with \(\leq\) and \(\sqsubseteq\) is that it includes a very general form of record concatenation, which is usable for direct modelling of inheritance of methods, but also for modelling more advanced features of some object-oriented languages, like multiple inheritance and template classes [Str86]. The major disadvantage of \(\mathbb{F}_\#\) is its complexity. The relation \(\#\) is much more complicated (proof-theoretically) than \(\sqsubseteq\), and the presence of concatenation of record types (With) makes \(\leq\) more complex. For a basic model of OOP with proof rules the additional expressive power of \(\mathbb{F}_\#\) is not necessary.
Chapter 10

Conclusions

10.1 Goal and Approach

Our goal was to find proof rules for Object-Oriented Programming in the programming logic $\lambda \omega_L$ [Pol94] or an extension. We have made several choices in our approach to this goal. Here we list those choices, and in Sections 10.2 through 10.7 we will elaborate on them.

- Our starting point is the programming logic $\lambda \omega_L$. This choice is already present in the formulation of our goal.
- We have chosen for machine support for the development of the theory in the system, instead of developing the theory by hand.
- We built our own proof assistant Yarrow, instead of using existing systems such as Coq and LEGO.
- We have chosen the existential model [PT94] to encode OOP features.
- We have used the reductionistic approach, by considering the ingredients of the existential model in isolation, before trying to combine them. This led to separate chapters on subtyping and ADTs, which will be treated separately below.
- We consider general $PTS_s$ instead of just $\lambda \omega_0^s$.

We summarize our achievements in Section 10.8. Sections 10.9 and 10.10 discuss related and future work, respectively. We conclude this chapter with a personal note on the work which led to this thesis, and a pointer to Yarrow and the theory developed in Yarrow, both available on the world wide web.

10.2 The Programming Logic $\lambda \omega_L$

First, we briefly recapitulate the main principle underlying $\lambda \omega_L$ and then discuss our experiences.

The main principle underlying $\lambda \omega_L$ is that programs and proofs are separate entities, that can be simultaneously developed. This is similar to the approach of deliverables [BM90], where programs and proofs of their correctness are formally combined into a pair. Our approach
stands in contrast to the approach of program extraction (Section 4.8), where only the proof is given and the program is extracted from the proof.

From the main principle follows the definition of $\lambda\omega_L$ as a PTS, in which programs and proofs are separate: proofs can depend on programs but not the other way around. In Poll's thesis this separation proved to be useful since the semantics could be developed into stages: first for the programming language, and then for the logic.

We discuss our experiences with $\lambda\omega_L$ in two parts, first concerning the separation of programs and proofs, and then concerning the definition of $\lambda\omega_L$ as a PTS.

In this thesis, we made a number of extensions to $\lambda\omega_L$. In most cases, only the syntax of the programming language was extended; the necessary adaptations to the logic were expressed as additional axioms, without changing the syntax of the logic. (The only exception is the introduction of subtyping, which gave rise to bounded universal quantifications, see Chapter 8). This difference in need for extension shows that the separation of programs and proofs is sensible and useful.

During the development of the theory in $\lambda\omega_L$, there was no single occasion where programs dependent on proofs were necessary or useful. This suggests the separation of programs and proofs is justified and not restrictive.

Although programs and proofs are separate entities, Poll proposed to develop them simultaneously. This simultaneous construction could not be implemented in Yarrow, because of technical problems (Section 3.2.2). Therefore we define programs and their proofs in two separate steps.

We give several remarks concerning the definition of $\lambda\omega_L$ as a PTS.

- Since $\lambda\omega_L$ is a PTS, we can consider $\lambda\omega_L^\leq$, its extension with subtyping, as a $PTS^\leq$. Or, historically more correct, the extension of $\lambda\omega_L$ with subtyping gave rise to a general framework of $PTS^\leq$, which is also applicable to other purposes. So by the definition of our system as an instance of a framework, any extension of the system suggests a (possibly) useful extension of the framework and vice versa. This cross-fertilization would have been much harder to obtain if our system were defined from scratch as "yet another type system".

- $\lambda\omega_L$ is defined as a bare PTS, which does not include inductive types. Inductive types could have been useful for two things, namely algebraic datatypes and inductively defined predicates. For a more practical use of $\lambda\omega_L$, algebraic datatypes are important. But since we were more interested in general proof rules than the development of complex programs, just three basic datatypes sufficed (booleans, natural numbers and lists). These can easily be postulated without extension of $\lambda\omega_L$. Inductively defined predicates, such as $\leq$ and Ordered, can be encoded in $\lambda\omega_L$, without need for additional axioms. For more practical use of our system some syntactic sugar, providing this encoding, would be useful.

- $\lambda\omega_L$ is defined as a PTS with a certain choice of abstractions and quantifications that can be formed (regulated through the parameters of the PTS). How good was this choice? In $\lambda\omega_L$ powerful quantifications are admitted (e.g. over all predicates on numbers), and such quantifications have been very useful. Still, at some points we needed quantifications not possible in $\lambda\omega_L$, in particular over all kinds ($\forall K : \Box$, ...) or over all prop-kinds ($\forall P : \Box_p$, ...), see Remark 4.5.2. These quantifications over kinds and
prop-kinds could be added to $\lambda\omega_L$ by adapting the parameters of the $PTS$. However, in practice we only used a few instantiations of these quantifications, so adding these quantifications to $\lambda\omega_JL$ is not really necessary.

A related matter is the need for quantifications over all record types in system $\lambda\omega_L^+$ ($\forall \langle l_1 : A_1, \ldots, l_n : A_n \rangle \ldots$), see Remark 6.2.2. We cannot easily add such quantifications to $\lambda\omega_JL^+$, e.g. by changing the parameters of the system.

- The abstraction mechanism in $\lambda\omega_JL$ is very powerful, but the syntax is very strict. Because of the strict syntax, the formal notation of some properties is quite awkward (e.g. the properties of $WB$ in Section 6.6.3). For practical use of $\lambda\omega_JL$ for proving program correctness syntactic sugar is very useful, if not necessary (e.g. see Section 6.9.2).

- Polymorphic predicates that are defined by induction on the structure of a datatype cannot be formally defined within $\lambda\omega_JL$. An example of such a predicate is $Sim$ (see Section 6.7.2). This problem cannot be solved by introducing inductive definitions, and occurs in many type-theoretical systems. Fortunately, this problem can be worked around by introducing such predicates axiomatically.

- A major consequence of using $\lambda\omega_JL$ is that we exclude other approaches to develop theory for OOP, for example object calculi [AC96a] and the co-algebraic approach of, for example, [Rei95]. See Section 5.5 for more on these approaches.

### 10.3 Machine Support

In Chapter 3 we claimed that machine support in the form of a proof assistant is needed for carrying out non-trivial proofs in system $\lambda\omega_L$. We present our working experience with machine support, and in that light consider this claim.

Our first experience in using machine support is that it takes a considerable effort to build up some boring basic theory about natural numbers and lists (Chapter 4). The formalization of the basic theory and the development of the example of sorting of lists have been simultaneous: during the development of this example we often needed some basic lemmas, which we proved and inserted in the basic theory before proceeding with the example. During the formalization of the examples for ADTs (Chapter 6), we only occasionally needed new lemmas belonging to the basic theory, and we could concentrate on the interesting part, namely the proof rules for ADTs. So the rigorous formalization of examples in a proof assistant requires some effort to get started, but once this has been done, one can concentrate on the problem at hand, without being much distracted by proofs of boring lemmas.

Although machine support was already essential for full formalization of the example of sorting, the formalization itself proceeded straightforwardly: no problems occurred and no new insights were necessary. Then we started formalizing the example of the ADT for bags. Only when we tried to actually give formal proofs we encountered problems and came to the insight that we needed quotients and subsets. Also later, when we tried to generalize from the example of bags to general ADTs, we were confronted with unforeseen problems, which led to another insight (namely that the specification of an ADT must satisfy certain conditions). We are quite sure we could not have obtained these insights without the strict formalization.

We support the claim that machine support is necessary by showing the sheer size of the formalization of the general theory for ADTs (in Yarrow). The entire context consists of
about 220 definitions and declarations, and is in total written form about 180000 characters long, and the longest proof term is about 11000 characters. So only by machine support it is possible to carry out strict formalization, which makes it clear which assumptions and mechanisms underlie intuitive proof rules.

10.4 Yarrow

In the introduction of Chapter 3 we gave three reasons for building our own proof assistant Yarrow, instead of using existing provers such as Coq and LEGO.

1. Existing proof assistants cannot deal with arbitrary PTSs, in particular $\lambda \omega_L$.

2. Existing assistants are hard to extend.

3. We were dissatisfied with deficiencies of Coq and LEGO, in particular the absence of forward reasoning.

Let us see — in retrospective — whether these reasons are valid.

The first reason for building Yarrow was the following. Existing proof assistants are not based on PTSs, but on different type theories. The theories of Coq and LEGO subsume $\lambda \omega_L$, but are much more powerful and complex: they allow more abstractions to be formed and they have inductive types. By basing Yarrow on arbitrary PTSs we are sure to remain within the borders of $\lambda \omega_L$.

In our experience, this is not a very good reason: in Yarrow we never entered terms using abstractions not allowed in $\lambda \omega_L$, and of course we never entered inductive types. In other words, it was always clear to us what was permitted in $\lambda \omega_L$ and what was not.

However, we will give an other advantage of a proof assistant for PTSs over proof assistants like Coq and LEGO for more complicated type systems, under the assumption that theory in $\lambda \omega_L$ is to be developed. In Coq Leibniz' equality is usually defined with inductive types. Therefore Coq's rewriting tactics are based on this definition of equality. Since $\lambda \omega_L$ does not have inductive types, we must use the second-order encoding of equality. Using this encoding in Coq renders the rewriting tactics useless. In Yarrow, we based the rewriting tactics on an arbitrary relation with the proper elimination laws, so we can easily use equality in proofs. We could only have obtained the same effect in Coq by extending Coq with another set of rewriting tactics suitable for the second-order encoding. So, different type systems require different tactics, and since Coq's type system includes far more features than PTSs, some of their tactics are unsuitable for developing theory in the PTS $\lambda \omega_L$. LEGO has a similar approach to rewriting tactics as Yarrow, based on an arbitrary relation with the proper elimination laws. However, its rewriting tactics are far less powerful, since a quantified equality must be instantiated before it can be used for rewriting, whereas in Coq and Yarrow the instantiation is usually automatically found by pattern matching.

The second reason we gave for building Yarrow was that existing proof assistants are hard to extend with features we need, such as records and subtyping.

We extended Yarrow with these two features (we emulated the existential types of Chapter 6 with their second-order encodings). For records, this extension was easy, but the introduction of subtyping was a major modification, since important parts of the type checking algorithm (see Section 7.3), the matching algorithm, and a number of tactics had to be revised. Extending Coq or LEGO would have been much harder. The main problem is the
extension of their type theories with subtyping, which is even more complicated than for PTSs because of the complexity of the theories. Another problem is that Coq is a very large program, so a major modification as adding subtyping would be a very laborious job. So indeed, by the size of (the theories of) Coq and LEGO, they are hard to extend, especially with radical concepts such as subtyping.

We should note that we have used subtyping only in the short Chapters 8 and 9. Chapter 6 gives the most extensive treatment of proof rules, and in this chapter we use records and existential types, but no subtyping.

The third reason for building Yarrow was the dissatisfaction with certain deficiencies of Coq and LEGO, in particular the inability to use forward reasoning. In Yarrow we implemented a tactic, called "Forward", that supports this. Let us describe how useful this tactic was.

In the total development of the general theory for ADTs (Chapter 6) there are about 1700 applications of tactics. The three most used tactics were "Intro" (280 times), "Apply" (240 times), and "Assumption" (230 times), and we used "Forward" about 80 times. These numbers alone show that Forward is useful. Moreover, we used Forward mainly when we could not proceed easily with other tactics (e.g. Apply); we estimate that we would have to replace each application of Forward by on average three other tactics with large arguments, if we would not have Forward. Since the usefulness of a tactic not only depends on how often it is used, but also how hard it is to emulate or avoid it, Forward turns out to be a very useful tactic. Hence it is fair to consider the absence of forward reasoning in Coq and LEGO a deficiency, and reason 3 to build Yarrow to be valid.

10.5 The Existential Model

We set out to find proof rules for OOP by encoding object-oriented features in an extended PTS. Here we take a critical look at this basic assumption, and discuss whether an extension of \( \lambda \omega_L \) is a suitable formalism for finding proof rules for OOP.

For our encoding, we chose the existential model, because it promised the simplest proof rules. (A well-known alternative, the recursive record model [CHC90, BCP97], is more complicated because it uses recursive types.) Two remarks about our model. First, it fixes the kind of OOP languages we consider to the common class-based languages, and restricts us to the basic OOP features. Second, even within the choice of the existential model, there are some variants possible concerning the encoding of the polymorphic record update. We have opted for the use of width-subtyping, in contrast to explicit get and put functions as in [PT94] and the compatibility relation of [Zwa95].

Since the existential model is the simplest encoding of OOP, our critical look at our basic assumption boils down to the question whether this model is a suitable means of finding proof rules for OOP in an extension of \( \lambda \omega_L \). We split this question into two parts, namely concerning the complexity and the faithfulness of the model.

Is the model too complicated?

Although we chose the least complex model, it is far from trivial. It uses five extensions of Pure Type Systems, namely records, existential types, subtyping, width subtyping, and the fixed point combinator, so at first glance the model might seem (overly) complicated.
However, each of the five extensions in itself is relatively simple, and they are independent of each other. Now that we are familiar with the extensions, we consider the model not too complicated. One should take into account that the meaning of most OOP concepts is not so simple. So we think the complexity of the model is reasonable. For example, the feature of late binding occurring in several OOP languages is simply modelled by taking the fixed point at another place.

Of course, the standard notation of our extended PTS is too complicated for practical use. But it is not hard to add syntactic sugar to allow a shorter, simpler notation for objects and classes, just as we suggested syntactic sugar for ADTs (Section 6.9.2).

Is the model faithful?

The model does not cater for more advanced OOP features present in some languages such as multiple inheritance or different levels of hiding. We ignore these features, but concentrate on the two main features present in all practical OOP languages which we do not model.

First, all practical languages allow non-termination, whereas we assume termination of all calculations (reductions). It is easy to extend the programming language with non-termination, simply by adding the fixed point combinator. But for the programming logic, this is quite a different matter; many axioms and proof rules become (much) more complicated if one adds non-termination, in particular those for abstract datatypes. Still, it is useful to obtain OOP proof rules based on termination, since often large parts of programs do not rely on the fixed point combinator (i.e. possibly non-terminating computations), even if written in languages which allow non-termination. We think it is feasible to obtain OOP proof rules that handle non-termination, but they would be considerably more complicated than without non-termination.

Second, all practical languages are imperative (have assignments in some form), whereas we model a purely functional language. Again, it is not so hard to extend the programming language with imperative features (for a relatively new and clean approach, see [LJ95]), but finding proof rules for programming languages with these features will be complicated and may be infeasible. This is mainly caused by aliasing, i.e. different variables (more precise: variable names) can point to the same object, so changing the object $a$ indicated by a variable may also change the object $b$ indicated by another variable (namely if $a$ and $b$ are the same objects). The same problem of aliasing makes it hard to extend the usual Dijkstra-Foare proof rules for small imperative languages with rules for pointers. Still, proof rules for our model of OOP should be suitable for parts of programs in which aliasing plays no role (e.g. parts which could be easily rewritten in a functional OOP language).

All considered, we feel that our approach is suitable for developing OOP proof rules, although their direct practical use is limited.

10.6 Abstract Datatypes

We gave an extensive conclusion about our treatment of abstract datatypes as existential types in Section 6.9. We will summarize it here. First, extending the syntax of $\lambda L$ with existential types is a straightforward addition of rules from the literature, but the derivation of proof rules for existential types (used for ADTs) has been a complicated job.
Second, a proper formalization of ADTs in type theory is only possible with machine support, as indicated in Section 10.3. Without the support it is infeasible to find general proof rules.

Third, there are important differences between our approach and the approach of algebraic specifications (e.g., [GTW78]). We admit far more specifications than in the algebraic approach, where specifications always consist of universally quantified equalities. However, the notation used in the algebraic approach is far more concise and elegant than our notation. This is caused by the formal type-theoretical framework we use, in which encapsulation is formally modelled, as opposed to the partially informal, set theoretical framework used for algebraic specifications, in which little attention is given to encapsulation. But the type-theoretical notation makes bindings explicit, which allows a formal notion of encapsulation (with existential types), and also of equality of implementations. From this we can derive proof rules for correct implementations, using quotients and subsets. In the algebraic approach [GTW78] the notion of correct implementation is defined using quotients and subsets, and this definition is only justified by appealing to the intuition. For practical use of our approach it is necessary to sugar the notation. As shown in Section 6.9 (in particular, Figure 6.7 on page 204), this is quite well possible, and this improved notation is as concise as in the algebraic approach.

10.7 Subtyping

Just as any extension of $\lambda_o^E$, the extension with subtyping has two aspects: syntax (including meta-theory) and proof rules. For subtyping, the latter part is relatively straightforward (see Chapter 8), but the extension of the syntax has been difficult. Therefore, the syntax will be the subject of the rest of this section. First, we give a brief history of the work reflected in Chapter 7, that treats the framework of Pure Type Systems with subtyping, and then motivate our choice for treating the whole framework instead of just $\lambda_o^E$.

It was clear from the start that the definition of $\lambda_o^{E_\leq}$ as the instantiation of a general scheme was far more concise than a definition specific for $\lambda_o^F$. Even though we gave a schematic definition of $PTS^{E_\leq}$, we tried to develop the meta-theory just for $\lambda_o^F$ (not for all $PTS^{E_\leq}$), so we could exploit the properties of $\lambda_o^F$. After many failed attempts, which were sometimes very lengthy, we found a way to define the framework in such a way that we could develop the meta-theory for $\lambda_o^F$. This definition is based on a simple but rigorous design decision, namely to eliminate the mutual dependency between the typing and subtyping judgments. Once this decision was taken, it turned out that we did not need to restrict ourselves to the single system $\lambda_o^F$, but that we could develop the theory for the whole framework. The meta-theory includes a type checking algorithm for a wide range of $PTS^{E_\leq}$, which has been implemented in Yarrow, so proofs for programs with subtyping can be made in our proof assistant.

The generality of the framework is not just a theoretical nicety, because the $PTS^{E_\leq}$ have more applications than just $\lambda_o^F$. Many existing type systems with subtyping can be seen as members of our framework, e.g. $F_{\leq}$ and $F_{\leq}$. Other members are new systems which have promising features, both applicable in programming languages and theorem proving.

So there are two reasons for treating the framework of $PTS^{E_\leq}$ instead of just $\lambda_o^F$. First, $\lambda_o^F$ is more concisely expressed as member of the framework than on its own. Second, the framework is interesting in its own right, and it did not cost additional effort to develop the theory for the whole framework.
10.8 Achievements

Our goal was to find proof rules for Object-Oriented Programming. We did not reach this destination, because of lack of time and the length of the path to this goal. But we covered quite some ground, as marked by Chapters 3, 6 and 7 and we have our goal in view, as shown in Chapter 9.

Fortunately, the areas covered in the three chapters do not only provide a lengthy way to proof rules for OOP, but are also interesting in themselves.

- Yarrow (Chapter 3) is a general proof assistant for $PTS$s and $PTS^\le$s. In comparison with Coq and LEGO there are three important differences. First, Yarrow handles arbitrary $PTS$s. Second, it handles $PTS$s with subtyping. Third, it has tactics for forward reasoning.

Because of its direct manipulation of proof terms, it has been successfully used in an educational environment. The use and development of Yarrow has also shown that forward reasoning is both desirable and practically implementable, and the forward tactic would be a valuable extension of other well-known assistants such as Coq.

- We gave proof rules for abstract datatypes (Chapter 6) based on type theory. These proof rules are more general than existing proof rules in that we allow specifications to be written in a more powerful logic. Furthermore, we derive these proof rules from some basic axioms, instead of just formulating the rules without (theoretic) justification. The theoretical foundation of our proof rules is based on parametricity and the existence of quotients and subsets.

- We defined the framework of Pure Type Systems with Subtyping (Chapter 7). This is a breakthrough in the field of type systems with subtyping for two reasons. First, the framework cleans up the field since several existing type systems with subtyping, such as $F^\le$ [CW85] and $F^\le_{\omega}$ [PS97], are essentially instantiations of this framework. Second, the framework includes interesting new systems such as $\lambda C^\le$, the Calculus of Constructions with subtyping and bounded quantifications, and $\lambda \omega^\le_L$, our programming logic with subtyping.

10.9 Related Work

Here we discuss related work on proof rules for objects obtained by modelling OOP in type theory. Other approaches to develop theory for OOP are given in Section 5.5; since we have not obtained proof rules for objects, it is not possible to give a more detailed comparison here. Work related to other subjects treated in this thesis is discussed at the end of the corresponding chapters.

The only work on OOP proof rules in type theory, that we are aware of, is [HNS98]. Their approach is roughly the same as ours. It also uses the existential model in the setting of type theory, but with so-called positive subtyping [HP96] instead of ordinary and width subtyping. Furthermore, their formalization is carried out in LEGO, in which positive subtyping is not present as primitive, but can be encoded. The major difference is that programs and correctness proofs are paired into formal values, called deliverables [BM90], whereas in $\lambda \omega^\le_L$ programs and proofs are always separate entities (it is too early to give a more detailed comparison). Invariants and equality of objects are not treated.
10.10 Future Work

The most important future work is to obtain proof rules for objects, by combining proof rules for the ingredients of objects, as mentioned in Section 9.8. Other future work, e.g. on Yarrow, \( PTS \leq S \), or ADTs, is described in the corresponding chapters.

10.11 A Brief History

The first year of my work as a Ph.D. student I was introduced to Erik Poll's work [Pol94], and I designed a very general extension of \( \lambda 2 \) for record concatenation (see Section 9.9). This extension was far more general than necessary for modelling single inheritance; it could also handle multiple inheritance and even more advanced features, such as so-called mix-in classes. Because of this generality, it took large part of my second year to develop meta-theory for this system. Still, it was only a programming language, and I did not consider the programming logic which should be integrated in the system.

Simultaneously, I implemented a type checker for \( PTS \), just for fun. By continuous work on it in the evening hours, the type checker grew out to the proof assistant Yarrow (Chapter 3). This work was inspired by the need to formally develop proof rules, first for simple functional programs (Chapter 4), and much later for ADTs (Chapter 6). During the development of the formal theories, it became clear that rewriting tactics and forward reasoning are important features of a proof assistant.

When the work on the meta-theory of \( \lambda 2 \) with record concatenation was done, I visited Erik Poll in Canterbury, and we discussed the direction my work should take. After consulting with Kees Hemerik, my supervisor, I decided that I would first concentrate on extending \( \omega_L \) with subtyping, since subtyping is an essential ingredient of models for OOP. This work took most of my third year, and resulted in general framework of \( PTS \) with subtyping (Chapter 7). Then I started on proof rules for objects, but after a couple of discussions with Erik Poll, we realized these were not easily inferred, and I decided to concentrate first on formal proof rules for ADTs, which form one of the building blocks for objects. This was a lot more work than expected, so that I spent most of my fourth year working on this. At the same time, I started writing my thesis, which gradually kept growing in the number of pages, although the planned treatment of objects became less and less extensive. (First, I planned three different encodings of objects with proof rules, then I restricted myself to one encoding with proof rules, and then also the proof rules were scratched out.) In the last, busy year, my right hand started itching and feeling a little numb, and it turned out that I had developed a light form of Repetitive Strain Injury. By working less hours, virtually stopping the development of Yarrow, and by using speech recognition software, I was able to complete this thesis in the end.

Yarrow and the theory on the web

Both the sources and a compiled form of Yarrow are publicly available on the web via my personal home page (September 1999: http://www.cs.kun.nl/~janz/index.html). The Yarrow distribution includes a tutorial and some examples.

At the same place, all of the theory developed in \( \omega_L \) and its extensions is available.
Bibliography


323


J.H. Geuvers. Theory of constructions is not conservative over higher order logic. Technical report, Department of Computer Science, University of Nijmegen, The Netherlands, 1989.


Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.


BIBLIOGRAPHY


Index

; (cons operator on lists), 89
< (ordering on numbers), 93
□p (type of $*$p), 76
□, (type of $*$), 72
$3, 256
Σ-types, 112–114, 291
  strong, 113
  weak, 113
→-types, 24
$A, 256
$S, 256
∈, 82
∈$R, 82
∃, 79
∀, 78
= (Leibniz’ equality of programs), 83
≤ (ordering on numbers), 92
¬, 78
⇒, 77
$*$p (type of all propositions), 76
$*s (type of all datatypes), 72
≤, see subtyping

ν, 78
∧, 78
∃, see width-subtyping

β, 260
βp, 24
βd, 255–260
F, 27
$F, 219
$F, 27
$F, 219
$F, 207, 218
$F, 217
$C≤−, 221
$C≤, 220
  without bounded quantifications, 221
$C, 27
$ω++≤, 302

λωF, 283
λωF, 114
λωF, 221
λωF, 80
  consistency of, 86
λωp, 76
λωF, 281
λωF, 114
λωs, 72
λωF, 219
λωF, 218
  with bounded constructor abstraction, 219
λω, 27
  as logic, 76–79
  as programming language, 72–76
λ→≤, 216
λ→, 27
λF, 220
λF, 220
λ2, 217
λ2, 27
=⇒ (equality of prop-constructors), 85
▷, see ▷
  for contexts, 232
▷, 31
▷, 270
▷whθ, 246
▷whθ, 246
▷whθ, 246
▷, see ▷
⊕ (concatenation of lists), 95

abstract datatypes, 109–206
abstract specification, 140, 149, 157
abstraction function, 201
AC, 194
admissibility, 230, 234–239
ADTs, see abstract datatypes
aggregation, 103, 106, 116, 291, 293–294
algebraic theory of ADTs, 202
antisymmetry of ≤, 239
App-admissible, 235
application of a substitution, 45
Apply (tactic), 39
  general version, 46
  simple version, 45
Assumption (tactic), 44
AXiom, 85
axiom
  of choice, 193
  of parametricity, 132
axiom_choice, 194
axioms
  for booleans, 88
  for lists, 89
  for natural numbers, 88
  for predicate logic, 87
  for \(\lambda \omega \subseteq\), 285
  for \(\lambda \omega^L\), 115, 132, 167
  for \(\lambda \omega\), 86
axioms (\(\mathcal{A}\)), 22
backward reasoning, 37, 41
bags (example of ADT), 135–164
Basic, 183
basic (interfaces), 166, 183–185
bdRed (command), 37
behaviour, observational, 125–126
bijective, 30, 255
Bisim, 131, 183
bisimulation, 131
Bool, 88
booleans
  axioms, 88
  library, 91
bound, 213
bound variables, 22
bounded abstraction, 211
bounded constructor abstraction, 219
bounded declaration, 211
bounded polymorphic types, 282
bounded polymorphic predicates, 284
bounded polymorphic programs, 282
bounded quantification, 208, 211, 217
bounded rules (\(R \subseteq\)), 211
≤-Bounded-Substitution, 237
Bounded-Substitution, 240
Calculus of Constructions, 27, 98
  with subtyping, 220
Church numerals, 75
Church-Rosser
  for whβα, 247
  for DPTSs, 31
  for PTS≤s, 226
  for PTSt, 23
class, 103, 291
class-based languages, 107
classical logic, 86
co-algebras, 107
Comp, 180
Completeness, 266
≤-Completeness, 263
completeness
  of \(\lambda\), 263–266
  of restricted subtyping, 230
  of syntax-d. typing, 253–255, 264
  of syntax-directed subtyping, 263
Completeness of Typing, 264
concatenation of lists, 95
cons operation on lists (\(\cdot\)), 89
conservativity
  of \(\lambda \omega_L\) over \(\lambda \omega\), 81
  of DPTSs over PTSt, 32
  of PTS≤s over PTSt, 251
consistency of \(\lambda \omega_L\), 86
context, 28
  extended, 256
  global, 36
  local, 37
  total, 42
Conv-admissible, 235
≤-Conversion-closed, 232
Correctness of Types
  for DPTSs, 32
  for PTS≤s, 241
  for PTSt, 25
correctness proof
  of an ADT, 110
CR, see Church-Rosser
CT, see Correctness of Types
data abstraction, see encapsulation
INDEX

datatype, 72
   basic, 75
datatype-constructor, 72
decidability of type-checking, 30, 274
Def (command), 37
definition, 30–33
   global, 31
   local, 31
   parametric, 33, 79
descriptive, 11
Dom, 179
dom (domain)
   of a context, 212
   of a substitution, 45
DPTS, 30–33
Elem, 96
encapsulation, 109, 131
   in OOP, 103, 106, 291, 295–296
   using existential types, 120–124
   without existential types, 116–120
encoding objects, 291–312
engine (of Yarrow), 63–65
equivalence of subtyping rules, 234–239
erasure, 256
Exact (tactic), 40, 44
exis_QuotAlg, 167, 186
exis_QuotAlgHO, 194
exis_QuotSubsetAlg, 167, 186
exis_QuotSubsetAlgHO, 195
exis_QuotType, 192, 194
exis_SubsetAlg, 167, 186
exis_SubsetAlgHO, 195
exis_SubsetType, 194
exis_QuotAlgHag, 143
exis_QuotSubsetAlgHag, 159
exis_SubsetAlgHag, 151
existential model, 106, 291–312
existential quantification
   over datatype, 82
   over kind, 82
   over propkind, 79
existential types, see Σ-types
Exit (command), 40
extended pseudocontexts, 256
extensionality, 84
False, 78
false, 88
field (of a record), 111
first-order (interfaces), 166, 183–185
FirstO, 183
fixed point combinator, 292, 304
flag-style
   printing proofs in, 53
flags, 74, 78
formalization of languages, 11
Forward (tactic)
   general version, 48
   simple version, 47
forward reasoning, 39, 41, 47
free variables, see FV
Fudgets, 56
functional PTS≤s, 244
functional PTS, 26
FV (free variables)
   of a context, 22, 31, 212
   of a term, 22, 212
   of an extended context, 256
≤-Generation, 233
Generation for typing, 240
Generation lemma, 25
generation properties for subtyping, 230–234
[generic], 120, 123
global context, 36
global definition, 31
goal-directed, 37
Haskell, 56
head, 94
hiding, see encapsulation
higher-order λ-calculus, 27
   with intersection types, 219
   with subtyping, 207, 218
higher-order (interfaces), 166
HL (free holes), 42
hole, 37, 42
holes in goals, 52
if, 88
Image, 179
Implen...UserSpec, 143, 152, 190
implementation of an ADT, 110, 120
equality of, 125–126

335
implementor of an ADT, 110
implementor's specification, 140, 149, 157, 190–191
ImplemSpec, 140, 149, 157
Implicit (command), 53
implicit arguments, 52, 87
ind_rec, 115
ind_sigma, 115
inductive definitions, 55
infix notation, 52, 87
inhabited, 28
inheritance, 103, 106, 292, 298–304
insertId, 288
instance variable, 103, 291
interface (for objects), 293
interface of an ADT, 110, 116
Intro (tactic), 38, 44
Intros (tactic), 39
Inv, 165
invariant, 149, 165
Inverse, 179
invocation function, 291, 294
IsER, 179
IsERon, 180
IsInjection, 179
IsInvar, 183
IsInvarBag1, 149
IsQuotAlg, 166, 185
IsQuotAlgBag1, 143
IsQuotSubsetAlg, 166, 185
IsQuotSubsetAlgBag1, 159
IsQuotType, 192, 194
IsSubsetAlg, 166, 185
IsSubsetAlgBag1, 151
IsSubsetType, 194
IsSurjection, 179
IsZclosed, 180
isZero, 93
iterlist, 94
internat, 92
kind, 72
label, 111
lambda calculus
  higher-order, 27
    with intersection types, 219
    with subtyping, 207, 218
polymorphic, 27
with subtyping, 217
second-order, 27
with subtyping, 217
simply dependent
with subtyping, 220
simply typed, 27
with subtyping, 216
late binding, 105, 106, 292, 306–307
LeftC, 180
Leibniz' equality, 83
  for prop-constructors, 85
lemmas
    names of, 90
leq, 93
liberal Σ elimination, 113, 140
library
  for booleans, 91
  for lists, 94
  for natural numbers, 91
  for predicate logic, 90
  for \( \omega^+ \subseteq \), 285
  for \( \omega^+ \), 179
  for \( \omega_L \), 89
lifted subtyping, 208, 215
List, 89
lists
  axioms, 89
  library, 94
local context, 37
local definition, 31
LOGIC, 78
LOLPLUS, 115
main mode of Yarrow, 36–37
matching, 45
meta-theory
  for \( \omega^+ \subseteq \), 285
  for \( \omega_L \), 114
  for \( \text{PTS}^\subseteq \text{s} \), 225–251
  for \( \text{PTS}^\oplus \text{s} \), 28
  for \( \text{PTS}^\oplus \text{t} \), 25
meta-variables, 23
method, 103, 291
minimal type, 244
Minimal Typing, 248
INDEX

Weak, 245
minimal typing, 244–250
MkImplSpec, 176, 190
MkUserSpec, 176, 190
monads, 57–59
morphism, 143, 151

naive specification, 136, 140, 147, 149
NaiveSpec, 136, 140, 147, 149
Nat, 88
natural numbers
  axioms, 88
  library, 91
nil, 89
normative, 11
notation
  in Yarrow, 37
  infix, 52, 87
NR-height, 232

0 (zero), 88
Object, 295
object, 103, 291
object-based languages, 107
Object-Oriented Programming, 103–107
objects
  as primitives, 107
  encoding, 291–312
  proof rules, 309–310
observational behaviour, 125–126
ok-Completeness, 265
ok-Soundness, 263
OOP, see Object-Oriented Programming
operations of an ADT, 116
Ordered, 96
OrderedId, 288
OrE (tactic), 50
OrIL (tactic), 50
OrIR (tactic), 50
override (of methods), 103, 291, 292, 303

pack, 111
pack. Quot, 143
pack.Subset, 152
PAR, 132
parametric definition, 33, 79
parametricity, 125–134, 181–183
  axiom, 132

for existential types, 131
  general, 131–134
parentheses, omitting of, 23
par Pi, 132, 183
parSigma, 131, 133, 183
partial equivalence relation, see PER
partial proof term, 37
PER, 160, 199
PER model, 96, 132, 143, 152, 160, 167, 194, 199
Perm, 96
II-type, 229
polymorphic lambda-calculus, 27
  with subtyping, 217
polymorphic record update, see record update
Polynat, 75
pred, 91
predicate logic
  axioms, 87
  library, 90
primreclist, 89
primrecnat, 88
principle, 122, 190
print proofs, 53
product
  cartesian, 111
  labelled, 111
program, 72
program extraction, 98
proof, 77
proof assistants, 35
proof mode, 37–41
  formal treatment, 42–52
proof principle
  for using ADTs, 121
proof rules for ADTs, 175–177
  folklore, 201
  for higher-order ADTs, 199
proof rules for objects, 309–310
proof script, 41
proof task, 36
proof term
  partial, 37
prop-constructor, 77
prop-kind, 77
proposition, 77
Prove (command), 37
pseudocontexts
  extended, 256
  of a DPTS, 31
  of a PTS≤, 212
  of a PTS, 22
 pseudoterms
  of a DPTS, 30
  of a PTS≤, 212
  of a PTS, 22
 PTS, 22–30
  examples of, 27
 PTS≤, 207–279
  examples of, 216–222
  relation to PTSs, 251
 QUOTBag, 143
first-order, 192–194
higher-order, 194–195
  of a subset, 159–160
  higher-order, 198–199
quotient type, 142, 192
QUOTSUBSET, 167
reachable values, 163
record extension, 303
record update, 299–302
records, 111–112
recursive record model, 107
reduction
  β, 23, 212
    for contexts, 232
  σ, 270
whβ, 246
whβσ, 246
whσ, 246
δ, 31
replacement
  for pseudocontexts, 230
  for subtyping, 231
  for typing, 239
representation type, 116
Restr, 180
Rewrite (tactic), 51
RightC, 180
rules (R), 22
rules, bounded (R≤), 211
S (successor), 88
second-order λ-calculus, 27
  with subtyping, 217
self reference, 104, 106, 291, 292, 304–305
semantics
  of λω^+≤, 285
  of λω_L, 85
Sensible, 167, 187
sensible specification, 143, 152, 160, 167–169, 187
service routines (of Yarrow), 66
Sim, 130, 181
SimLeftC, 195
SimBag, 136
simply dependent λ-calculus
  with subtyping, 220
simply typed λ-calculus, 27
  with subtyping, 216
simply-typed (interfaces), 127
SimplyT, 182
simulation, 125, 126
  for BagI, 136
size of a term, 271
SN, see Strongly Normalizing
software architecture of Yarrow, 56–67
solution (of a matching problem), 46
sort, 97
sortId, 289
sorting, 96
  of records, 288
sorts (S), 22
sorts, subtyping (S≤), 211
Soundness, 263
≤-Soundness, 260
soundness
  of h_δ, 260–263
  of restricted subtyping, 230
  of syntax-directed subtyping, 260
  of syntax-directed typing, 253, 261
Soundness of Typing, 261
Spec_sens, 143, 152, 160
special tactics, 50–52
specification
  of a PTS≤, 211
of a PTS, 22
specification of an ADT, 110
abstract, 140, 149, 157
implementor's, 140, 149, 157, 190–191
naive, 136, 140, 147, 149
sensible, 143, 152, 160, 167–169, 187
user's, 121, 190–191
well-behaved, 169–175, 187–189
SR, see Subject Reduction
stack (example of ADT), 116–124
Start lemma, 25, 29
state (of Yarrow in proof mode), 42
steps, 270
strong elimination, 113
Strongly Normalizing, 26
βσ, 270
λC, 27
λω^≤, 285
λω_L^≤, 114
λω_L, 82
λω_s, 75
Sub (subtyping algorithm), 267
subclass, 103, 291
subgoal, 37
Subject Reduction
for whβσ, 247
for DPTSs, 32
for PTS≤s, 239–243
for PTSs, 26
subscript, datatypes as, 87, 180
first-order, 192–194
higher-order, 195–198
subset type, 151
SUBSET_BAGT, 151
Substitution, 240
substitution, 22, 45
for a hole, 42
≤-Substitution, 235
Substitution lemma, 25
subsumption, 208, 214
subtyping, 207–279
in OOP, 103, 106, 207–209, 291, 292, 296–298
subtyping algorithm, 267
subtyping on programs
why not, 221
subtyping rules, 214
for λω^≤ and λω_L^≤, 281
liberal, 230
restricted, 226, 229
subtyping sorts (S≤), 211
superclass, 103, 291
syntactic sugar
for ADTs, 203
in Yarrow, 52–53
syntax-directed rules, 226, 255–260
subtyping, 259
typing, 258
System (command), 36
system architecture, 59
tactic, 37, 43
special, 50–52
tail, 94
task, 37
term, 28
termination of type-checking, 269–273
Top, 217, 218
total context, 42
Trans-admissible, 236
transitivity elimination, 228, 236
True, 78
true, 88
Typ (typing algorithm), 267
typable, 28, 215
Type Reduction for whβσ, 247
type-as-subject problem, 254
type-checking algorithm
for PTS≤s, 252–274
for PTSs, 30
typing algorithm (Typ), 267
typing rules
for DPTSs, 31
for PTS≤s, 213–215
alternative, 222–224
for PTSs
∧, 27
∧_W (standard), 24–26
unification, 45
uniform behaviour, 131
Unique whβσ-normal forms, 247
Uniqueness of Types

*DPTS*, 32

*PTS*, 26

for $t_{ld}$ (*PTS* $\leq s$), 259

unpack, 111

*Use (command)*, 50

use of an ADT, 110

user interface, 60–63

user of an ADT, 110

user’s specification of an ADT, 121, 190–191

*UserSpec*, 121

UT, see Uniqueness of Types

*Var (command)*, 36

Var-admissible, 235

variables, fonts for, 86

*WB*, 170–174, 187–189

Weak Minimal Typing, 245

Weakening lemma, 25, 29

*WeakUserSpec*, 141, 150, 158

well-behaved specification, 169–175, 187–189

well-formedness rules

for *DPTS*, 31

for *PTS* $\leq s$, 213

for *PTS*, 27

well-typed, 28, 215

*Wf (well-formedness algorithm)*, 269

width-bounded quantification, 300

width-subtyping, 298–302

width-subtyping rules, 301

Yarrow, 35–70

a type-theoretical perspective, 42–55

a user’s perspective, 36–41

on the world-wide web, 321

software architecture, 56–67

*Z-closure*, 180
Summary

The Pure Type Systems (PTSs) form a family of typed λ-calculi. On one hand, a PTS can be seen as a concise but powerful functional programming language. On the other hand, a PTS can be seen as a logical system. The thesis of Erik Poll [Pol94] combines both interpretations, leading to a programming logic called λωL. This is a formal system for the design of and reasoning about programs, i.e. proving programs correct with respect to a specification.

This thesis continues the work of Poll, and is divided in two parts. The first part gives the basic formalism and its implementation in Yarrow. Yarrow is a tool we implemented providing machine support for PTSs in general, and λωL in particular. The use of Yarrow has had an important influence on the rest of this thesis. In the second part we give several extensions of λωL, and show how they provide support for object-oriented programming, a programming paradigm which has become very popular in the last decades.

Now we consider the parts in more detail.

Contents of Part I

The basic formalism is given in Chapters 2 (PTSs) and 4 (λωL). These two chapters are given for reference, and are largely copied from Poll's thesis.

Chapter 3 discusses Yarrow, the tool we implemented. First we discuss why software support is necessary for developing theory in a PTS, and why we developed our own tool instead of using existing software. Then we consider Yarrow from three perspectives. First, we show how Yarrow is used. Second, we discuss the theory behind interactive proof development in Yarrow. In particular, we describe forward reasoning, which is a natural way of proving, not present in existing systems similar to Yarrow. Third, we give the software architecture of Yarrow. The main points here are Yarrow's treatment of I/O in the purely functional language Haskell, and the communication between the user interface and the engine. This communication is interesting since both a textual and a graphical user interface have been implemented.

New in Chapter 4 is a modest library of theory, and the development of a small example program together with its specification and proof of correctness, all developed in Yarrow.

Contents of Part II

The goal of the second part is to extend λωL in order to handle Object-Oriented Programming (OOP). This programming paradigm has become so popular because it helps structuring and reuse of programs. Central concepts of this paradigm are:

Aggregation. This is the pairing of data and algorithms working on that data.
Encapsulation. This limits the access to data and encourages modular programming. Encapsulation is also known as data hiding or data abstraction.

Inheritance. By inheritance, algorithms can work on related data structures, so reuse of programs is facilitated.

In order to cater for these concepts, we need to make several extensions to $\lambdaomega$. For aggregation we need records, for encapsulation we need existential types, and for inheritance we need subtyping. The elaborations of these extensions form the body of Part II.

Chapter 5 briefly introduces OOP by explaining a small example program, and gives the central concepts of OOP with the necessary extensions of $\lambdaomega$ in more detail.

Chapter 6 defines $\lambdaomega^+$ as the extension of $\lambdaomega$ with records and existential types. The focus in this chapter lies on deriving proof rules for these notions, i.e. logical rules for proving correctness of programs that use records and existential types. This is quite an enterprise. First we consider the theory of parametricity, which extends the notion of equality between values of existential types. But parametricity is not enough. We need additional axioms, stating the existence of so-called quotient and subset types. Only then practical proof rules can be derived for abstract datatypes, which form the simplest application of existential types. The use of Yarrow was crucial for developing the complicated theory, and made the need for the additional axioms apparent.

In Chapter 7 we consider subtyping. Instead of extending only the particular PTS $\lambdaomega$ with subtyping, we have chosen to extend the whole framework of PTSs, leading to a framework of $PTS^S$s (Pure Type Systems with subtyping). The problem here is to show that we preserve the nice syntactical properties of PTSs. We solve the problem by adopting a novel and rigorous design decision, namely by defining the subtyping relation independent of typing.

In Chapter 8 we define $\lambdaomega_+^S$, i.e. $\lambdaomega^+$ extended with subtyping. We show some subtleties concerning the logical aspects of subtyping, and give a small example program.

Chapter 9 shows how the ingredients of records, existential types and subtyping can be used to model OOP. Our model, a slight variant of the well-known existential model, is presented step-wise, so that OOP features are modelled one at a time. We speculate on proof rules for objects, but do not formally develop them.

Contributions of this Thesis

The three main contributions of this thesis are:

- Yarrow (Chapter 3), because it supports arbitrary PTSs, forward reasoning, and subtyping.

- Proof rules for abstract datatype (Chapter 6). These rules are formally derived from the principle of abstraction, and justify and generalize both folklore proof rules and rules found in the field of algebraic specifications.

- The framework of Pure Type Systems with subtyping, including the development of its meta-theory (Chapter 7). The main attraction of the PTS framework is that many nice properties can be proved once for all its members. Similarly, our extension of this framework with subtyping is attractive because we prove nice properties for a class of systems with subtyping. This class includes several existing systems, and new systems, which can be useful in the field of programming and formal logic.
Samenvatting

De Pure Type Systemen (PTS\textsubscript{en}) vormen een familie van getypeerde $\lambda$-calculi. Een PTS kan enerzijds worden gezien als eenvoudige maar krachtige functionele programmeertaal, en anderzijds als logisch systeem. In het proefschrift van Erik Poll [Pol94] worden deze zienswijzen gecombineerd, wat leidt tot een programmeerlogica genaamd $\lambda\omega_L$. Dit systeem maakt het mogelijk om op een formele wijze programma's te ontwerpen en over programma's te redeneren, dat wil zeggen programma's correct te bewijzen ten aanzien van een specificatie.

Dit proefschrift is een vervolg op het onderzoek van Poll. Het bestaat uit twee delen. Het eerste deel geeft het basisformalisme en zijn implementatie in Yarrow. Yarrow is een programma dat praktische toepassing ondersteunt van PTS\textsubscript{en} in het algemeen, en van $\lambda\omega_L$ in het bijzonder. Het gebruik van Yarrow heeft een belangrijke invloed gehad op de rest van dit proefschrift. In het tweede deel geven we een aantal uitbreidingen van $\lambda\omega_L$, ter ondersteuning van onze modellering van object-georiënteerd programmeren. Dit is een stijl van programmeren die de laatste decennia erg populair is geworden.

We gaan nu nader op beide delen in.

Inhoud van Deel I

Het basisformalisme van dit proefschrift wordt gegeven in Hoofdstukken 2 (PTS\textsubscript{en}) en 4 ($\lambda\omega_L$). Deze hoofdstukken dienen voornamelijk ter introductie, en zijn grotendeels gekopieerd uit Polls proefschrift.

Hoofdstuk 3 bespreekt Yarrow, het programma dat we geïmplementeerd hebben. Eerst geven we aan waarom computerondersteuning nodig is voor de ontwikkeling van theorie binnen een PTS, en waarom we zelf zo'n programma hebben gemaakt in plaats van bestaande systemen te gebruiken. Dan beschouwen we Yarrow vanuit drie gezichtspunten.

Ten eerste laten we zien hoe Yarrow gebruikt wordt. Ten tweede geven we de theorie die aan het interactief ontwikkelen van bewijzen ten grondslag ligt. In het bijzonder beschrijven we voorwaarts redeneren, een natuurlijke manier van bewijzen, die aan soortgelijke programma's ontbreekt. En, ten derde, gaan we in op de software-architectuur van Yarrow. De hoofdpunten hier zijn de aanpak van I/O in de puur functionele taal Haskell, en de communicatie tussen de gebruikers-interface en de kern van Yarrow. Deze communicatie is interessant omdat zowel een tekstuele als een grafische gebruikers-interface geïmplementeerd zijn.

Nieuw in Hoofdstuk 4 is een bescheiden bibliotheek van theorie, en de ontwikkeling van een klein voorbeeldprogramma samen met zijn specificatie en correctheidsbewijs, alle in Yarrow ontwikkeld.
SAMENVATTING

Inhoud van Deel II

Het doel van het tweede deel is $\lambda_{L}$ zo uitbreiden dat object-georiënteerd programmeren (OOP) ondersteund wordt. Deze stijl is zo populair geworden omdat deze helpt bij het structureren en hergebruik van programma’s. De belangrijkste concepten van deze stijl zijn:

Aggregatie. Dit is het paren van data en algoritmen die op die data werken.

Inkapseling. Dit beperkt de toegang tot de data, en moedigt zo modulair programmeren aan.

Overerving. Hierdoor kunnen algoritmen op soortgelijke datastructuren werken, zodat hergebruik van programma’s beter mogelijk wordt.

Om deze concepten te ondersteunen moeten we een aantal uitbreidingen van $\lambda_{L}$ maken. Voor aggregatie hebben we records nodig, voor inkapseling existentiële typen, en voor overerving subtypering. Het uitwerken van deze uitbreidingen is het belangrijkste doel van Deel II.

Hoofdstuk 5 introduceert OOP kort, door de uitleg van een klein voorbeeldprogramma, en beschrijft de belangrijkste ideeën van OOP met de bijbehorende uitbreidingen van $\lambda_{L}$ in meer detail.

Hoofdstuk 6 definiërt $\lambda_{L}^{+}$ als de uitbreiding van $\lambda_{L}$ met records en existentiële typen. De nadruk in dit hoofdstuk ligt op het ontwikkelen van bewijsregels voor deze uitbreidingen, d.w.z. logische regels voor het correct bewijzen van programma’s die records en existentiële typen gebruiken. Dit is een hele onderneming. Eerst beschouwen we de theorie van parametriciteit, die de gelijkheid tussen waarden met existentiële typen uitbreekt. Maar parametriciteit is niet genoeg. We hebben extra axiomatiek nodig, die het bestaan van zogenaamde quotient en subset typen uitdrukken. Pas dan kunnen we praktische bewijsregels voor abstracte datatypen ontwikkelen. Abstracte datatypen vormen de eenvoudigste toepassing van existentiële typen. Het gebruik van Yarrow bleek van cruciaal belang te zijn. Dankzij Yarrow konden we de ingewikkelde theorie ontwikkelen, en werd de noodzaak van extra axiomatiek duidelijk.

In Hoofdstuk 7 beschouwen we subtypering. We hebben ervoor gekozen om niet alleen het $PTS\lambda_{L}$ uit te breiden, maar het hele raamwerk van $PTS_{en}$, hetgeen leidt tot het raamwerk van $PTS^{\leq}en$ (Pure Type Systemen met subtypering). Hier is het probleem om aan te tonen dat we de mooie syntactische eigenschappen van $PTS_{en}$ behouden. We lossen dit probleem op door een nieuwe en rigoureuze ontwerpbeslissing te nemen, namelijk door de subtyperingsrelatie onafhankelijk van typering te definiëren.

In Hoofdstuk 8 definiëren we $\lambda_{L}^{+\leq}$, d.w.z. $\lambda_{L}^{+}$ uitgebreid met subtypering. We laten enkele logische subtiliteiten zien die met subtypering samenhangen, en we geven een klein voorbeeldprogramma.

Hoofdstuk 9 toont hoe de ingrediënten records, existentiële typen en subtypering kunnen worden gebruikt om OOP te modelleren. Ons model, een licht afwijkende variant van het bekende existentiële model, wordt stapsgewijs gepresenteerd, zodat we de concepten van OOP één voor één modelleren. We speculeren over bewijsregels voor objecten, maar ontwikkelen die niet formeel.

Bijdragen van dit proefschrift

De voornaamste drie bijdragen van dit proefschrift zijn:
SAMENVATTING

- Yarrow (Hoofdstuk 3), omdat het willekeurige $PTS\text{en}$, voorwaarts redeneren en subtypering ondersteunt.

- Bewijsregels voor abstracte datatypen (Hoofdstuk 6). Deze regels zijn formeel afgeleid van het principe van inkapseling, en rechtvaardigen en generaliseren zowel informele bewijsregels als regels uit het gebied van algebraïsche specificaties.

- Het raamwerk van $PTS\text{en}$ met subtypering, met de ontwikkeling van de bijbehorende meta-theorie (Hoofdstuk 7). Het aantrekkelijke van het $PTS$ raamwerk is dat veel mooie eigenschappen in één keer kunnen worden bewezen, voor alle erin passende systemen. Op dezelfde manier is onze uitbreiding van dit raamwerk aantrekkelijk, omdat we de mooie eigenschappen voor een klasse systemen met subtypering kunnen bewijzen. Een aantal bestaande systemen behoort tot deze klasse, maar ook nieuwe systemen behoren daartoe, die nuttig kunnen zijn als programmeertalen of formele logica's.
Curriculum Vitae


September 1982 – August 1988  Secondary school (Gymnasium B), Ichthus College, Drachten.

September 1988 – August 1994  Master’s degree in computer science at the University of Groningen, The Netherlands.


From May 1999  Research assistant at the University of Nijmegen, The Netherlands.
Title in the IPA Dissertation Series


P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4


D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9


N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalization in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computer Science, TUE. 1996-12


B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01


P.A.M. van den Beuker. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01


J. Verric. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03


E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01


C.H.M. van Kemenade. *Recombinatve Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04


J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08


P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábán. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

Stellingen
behorende bij het proefschrift
Object-Oriented Concepts and Proof Rules
van
Jan Zwanenburg

1. Het gebruik van de term “meta-variable” in moderne literatuur over syste- temen voor bewijsontwikkeling is erg ongelukkig.


2. De heldere kern van Coq wordt verduisterd door een wildgroei aan uitbreidingen.

3. Het is hoogst merkwaardig dat tactieken voor het zgn. voorwaarts reden- neren nog niet in stellingenbewijzers als Coq en Lego zijn geimplemen- teerd.

4. is een nog net leesbaar bewijs van inconsistentie van systeem $\lambda U^-$. 


5. In de praktijk zijn van de oneindige typen-hiërarchie in Coq slechts drie niveaus nodig.
6. Voor het bedrijven van wiskunde binnen een totaal formele logica is een stellingenbewijzer net zo onontbeerlijk als een rekenmachine dat is voor trigonometrische berekeningen.

7. De gebruikelijke formulering maakt Pure Type Systemen slecht uitbreidbaar.

[1] Sectie 2.2 van dit proefschrift.

8. Het idee achter de Turing-test is het toepassen van het begrip extensionaliteit op de entiteiten mens en computer.


9. Noem een getal speciaal als elke permutatie van de cijfers een priemgetal oplevert. Noem een getal flauw als het slechts uit dezelfde cijfers bestaat. Het aantal niet-flauwe speciale getallen is eindig.

10. Er zijn kleuringen van het rationele vlak $\mathbb{Q}^2$ met twee kleuren, zodanig dat elk tweetal punten met onderlinge afstand 1 verschillend gekleurd is. Dit kan constructief worden bewezen.

11. Zij $P_0, P_1, \ldots, P_i, \ldots$ een rij van verzamelingen punten in het platte vlak, als volgt verkregen. $P_0$ bestaat uit 4 willekeurige punten, en $P_{i+1}$ is de verzameling van snijpunten van de lijnen door elk willekeurig paar punten uit $P_i$. Deze rij is met kans 1 strikt monotoon.

12. Een rechte lijn is een gedachtekrakiel.

13. Het kan geen kwaad af en toe te proberen het wiel opnieuw uit te vinden.

14. Het taboe op sex is even groot als de populariteit van flauwe moppen hierover.

15. De invloed van de verruiming van de openingstijden van de supermarkten op het verschijnsel RSI is ten onrechte onderbelicht.

16. Het is tegelijkertijd vloek en zegen dat een proefschrift geen handschrift meer is.