HAL
open science

A framework for defining computational higher-order logics

Ali Assaf

To cite this version:

Ali Assaf. A framework for defining computational higher-order logics. Computer Science [cs]. École polytechnique, 2015. English. NNT : . tel-01235303v1

HAL Id: tel-01235303

https://pastel.hal.science/tel-01235303v1

Submitted on 30 Nov 2015 (v1), last revised 16 Dec 2015 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École doctorale polytechnique (EDX)
Numéro 447
Spécialité informatique

Thèse de doctorat

de

l'École polytechnique

préparée à

Inria Paris-Rocquencourt

par

Ali Assaf

A framework for defining computational higher-order logics

Un cadre de définition de logiques calculatoires d'ordre supérieur

Présentée et soutenue à Paris, le 28 septembre 2015

Composition du jury:

Président	Dale Miller	LIX & Inria Saclay, France
Rapporteurs	Andrea Asperti	University of Bologna, Italie
	Herman Geuvers	University of Nijmegen, Pays-bas
	John Harrison	Intel Corporation, USA
Examinateurs	Brigitte Pientka	McGill University, Canada
	Hugo Herbelin	PPS & Inria Paris, France
Directeurs de thèse	Gilles Dowek	Inria Paris, France
	Guillaume Burel	ENSIIE, France

Abstract

The main aim of this thesis is to make formal proofs more universal by expressing them in a common logical framework. More specifically, we use the lambda-Pi-calculus modulo rewriting, a lambda calculus equipped with dependent types and term rewriting, as a language for defining logics and expressing proofs in those logics. By representing propositions as types and proofs as programs in this language, we design translations of various systems in a way that is efficient and that preserves their meaning. These translations can then be used for independent proof checking and proof interoperability. In this work, we focus on the translation of logics based on type theory that allow both computation and higher-order quantification as steps of reasoning.

Pure type systems are a well-known example of such computational higher-order systems, and form the basis of many modern proof assistants. We design a translation of functional pure type systems to the lambda-Pi-calculus modulo rewriting based on previous work by Cousineau and Dowek. The translation preserves typing, and in particular it therefore also preserves computation. We show that the translation is adequate by proving that it is conservative with respect to the original systems.

We also adapt the translation to support universe cumulativity, a feature that is present in modern systems such as intuitionistic type theory and the calculus of inductive constructions. We propose to use explicit coercions to handle the implicit subtyping that is present in cumulativity, bridging the gap between pure type systems and type theory with universes à la Tarski. We also show how to preserve the expressivity of the original systems by adding equations to guarantee that types have a unique term representation, thus maintaining the completeness of the translation.

The results of this thesis have been applied in automated proof translation tools. We implemented programs that automatically translate the proofs of HOL, Coq, and Matita, to Dedukti, a type-checker for the lambda-Pi-calculus modulo rewriting. These proofs can then be re-checked and combined together to form new theories in Dedukti, which thus serves as an independent proof checker and a platform for proof interoperability. We tested our tools on a variety of libraries. Experimental results confirm that our translations are correct and that they are efficient compared to the state of the art.

Keywords: Lambda calculus, dependent types, Curry-Howard correspondence, rewriting, deduction modulo

Résumé

L'objectif de cette thèse est de rendre les preuves formelles plus universelles en les exprimant dans un cadre logique commun. Plus précisément nous utilisons le lambda-Pi-calcul modulo réécriture, un lambda calcul équipé de types dépendants et de réécriture, comme langage pour définir des logiques et exprimer des preuves dans ces logiques. En représentant les propositions comme des types et les preuves comme des programmes dans ce langage, nous concevons des traductions de différents systèmes qui sont efficaces et qui préservent leur sens. Ces traductions peuvent ensuite être utilisées pour la vérification indépendante de preuves et l'interopérabilité de preuves. Dans ce travail, nous nous intéressons à la traduction de logiques basées sur la théorie des types qui permettent à la fois le calcul et la quantification d'ordre supérieur comme étapes de raisonnement.

Les systèmes de types purs sont un exemple notoire de tels systèmes calculatoires d'ordre supérieur, et forment la base de plusieurs assistants de preuve modernes. Nous concevons une traduction des systèmes de types purs en lambda-Pi calcul modulo réécriture basée sur les travaux de Cousineau et Dowek. La traduction préserve le typage et en particulier préserve donc aussi l'évaluation et le calcul. Nous montrons que la traduction est adéquate en prouvant qu'elle est conservative par rapport au système original.

Nous adaptons également la traduction pour inclure la cumulativité d'univers, qui est une caractéristique présente dans les systèmes modernes comme la théorie des types intuitionniste et le calcul des constructions inductives. Nous proposons d'utiliser des coercitions explicites pour traiter le sous-typage implicite présent dans la cumulativité, réconciliant ainsi les systèmes de types purs et la théorie des types avec univers à la Tarski. Nous montrons comment préserver l'expressivité des systèmes d'origine en ajoutant des équations pour garantir que les types ont une représentation unique en tant que termes, conservant ainsi la complétude de la traduction.

Les résultats de cette thèse ont été appliqués dans des outils de traduction automatique de preuve. Nous avons implémenté des programmes qui traduisent automatiquement les preuves de HOL, Coq, et Matita, en Dedukti, un vérificateur de types pour le lambda-Pi-calcul modulo réécriture. Ces preuves peuvent ensuite être revérifiées et combinées ensemble pour former de nouvelles théories dans Dedukti, qui joue ainsi le rôle de vérificateur de preuves indépendant et de plateforme pour l'interopérabilité de preuves. Nous avons testé ces outils sur plusieurs bibliothèques. Les résultats expérimentaux confirment que nos traductions sont correctes et qu'elles sont efficaces comparées à l'état des outils actuels.

Mots clés : Lambda calcul, types dépendants, correspondance de Curry-Howard, réécriture, déduction modulo

Big thanks

To Gilles and Guillaume: Thank you for supervising me during these three years of erring and stumbling around in the dark. I knew it would take more than a week ;-) Gilles, I greatly appreciated our discussions, which were often stimulating, and in which you always gave your full attention and engagement. Guillaume, although I spent less time working directly with you, your feedback and attention to detail have been very valuable. Thanks to both of you for having the patience and giving me the autonomy needed to bring this work to fruition. And most importantly, thank you for encouraging me to do it while pursuing my own ideas. I can tell you that it was very satisfying to tackle, and solve, the problem at the angle that was most interesting to me.

To Andrea, Herman, and John: I hope this manuscript was edible when I sent it to you. Thank you for accepting to review my thesis. I am humbled to have it reviewed by authors of the works on which it is based.

To Brigitte, Dale, and Hugo: Thank you for accepting my invitation. I am honored to have you as members of the thesis committee. I could not have wished for a different composition and, frankly, I am surprised we managed to find a date that fits everyone from the first try! Brigitte, I am especially glad that you were able to come and close the circle that we had started 6 years ago at McGill University.

To the Deducteammates: Who would have thought that a single team could have so many homonyms among its members? Seriously: 2×Raphaël, 2×Frédéric, 2×Guillaume, 2×Pierre, 2×Simon, ... And that's not even counting the last names! Anyway, I am glad to have been part of it. It was a pleasure to work with all of you. I will not forgive the lame attempts at geek jokes involving lambdas and pis, but I am grateful for the camaraderie and the good mood :-)

To Zak: My companion in arms, it has been a long journey and it is finally coming to an end. You are a great friend, classmate, and colleague. Thanks for the unforgettable nights of GoT. May we meet again on the battlefield.

To the Parsifalians: I felt that I was part of a second team. Thanks for the playful rivalry, the interesting discussions, and the annoying puzzles. The lunches at Polytechnique were so much better in your company.

To my parents, Roger and Hanane, who gave me life several times over: Thank you for your stubborn love and your eternal support. You have always given us the tools for success and happiness, and I did not always deserve it. I am sorry to have made you worry! I love you both.

In terms of dedication, it is clear to me that there is only one sensible possibility. Throughout the writing of this manuscript, from the first file creation up to the very last commit, there was a particular person on my mind.

To Cat, without whom this thesis would not have seen the light of day:
Baby Bea is alive and well. I signed up for a journey of the mind, it ended up being a journey of the heart and soul. As I told you once, I cherish the crossings of our paths. These lines, and all the others I have written, are not enough. Thanks for reminding me not to be afraid of pain, lest it lock away the kingdoms of the heart. Thanks for all the memories made in the warmest winter. As hard as it was, I know that I will look back and smile. You know it better than anyone: nothing lasts forever, even cold November rain. Grazie, Caterina. Ti voglio bene. Always.

This thesis has greatly benefited from the proofreading and writing tips of the following people: Bruno Bernardo, Caterina Urban, Gilles Dowek, Guillaume Burel, Hugo Herbelin, Olivier Danvy, Roger Assaf, and Ronan Saillard. While I am sure there are still many mistakes and typos lurking in the pages, I know it would have been much worse without their keen eyes. A big thanks to all of you.

Zurich, November 28, 2015

"You didn't think it was gonna be that easy, did you?"
"You know, for a second there, yeah, I kinda did."

Contents

Introduction **15**

1 Introduction **17**
1.1 Proof systems . . . 17
1.1.1 Formalization of mathematics . . . 18
1.1.2 Software verification . . . 18
1.2 Universality of proofs . . . 18
1.2.1 Computational reasoning . . . 20
1.2.2 Higher-order reasoning . . . 21
1.3 Type theory and logical frameworks . . . 21
1.3.1 The Curry–Howard correspondence . . . 22
1.3.2 The logical framework approach . . . 23
1.4 Computational higher-order logics . . . 23
1.4.1 Pure type systems . . . 23
1.4.2 Universes and cumulativity . . . 24
1.5 Contributions . . . 25
1.6 Preliminary notions and notations . . . 26

I Frameworks and embeddings **29**

2 The $\lambda\Pi$-calculus **31**
2.1 Definition . . . 32
2.2 As a first-order logic calculus . . . 32
2.3 As a logical framework . . . 36
2.4 Limitations of $\lambda\Pi$ . . . 37

3 The $\lambda\Pi$-calculus modulo rewriting **41**
3.1 Definition . . . 42
3.2 Basic properties . . . 45
3.2.1 Soundness properties . . . 45
3.2.2 Completeness properties . . . 46

3.3 Computational higher-order embeddings 50

II Pure type systems **53**

4 Pure type systems **55**
4.1 Definition 56
4.2 Examples of pure type systems 58
4.3 Basic properties 60
4.4 Inter-system properties 61
4.4.1 Subsystems 61
4.4.2 Morphisms 61

5 Embedding pure type systems **63**
5.1 Translation 64
5.2 Completeness 67
5.2.1 Preservation of substitution 67
5.2.2 Preservation of equivalence 67
5.2.3 Preservation of typing 68
5.3 Alternative embeddings 69
5.3.1 Embedding without Π in the rewriting at the level of types 69
5.3.2 Embedding without rewriting at the level of types 70

6 Conservativity **71**
6.1 Normalization and conservativity 72
6.2 Proof of conservativity 74
6.2.1 Conservativity of equivalence 75
6.2.2 Conservativity of typing 77

7 Application: translating HOL to Dedukti **79**
7.1 HOL 80
7.2 Translation 82
7.2.1 Translation of types 83
7.2.2 Translation of terms 83
7.2.3 Translation of proofs 84
7.3 Completeness 87
7.4 Implementation 88
7.4.1 Proof retrieval 88
7.4.2 Holide: HOL to Dedukti 88
7.4.3 Extensions 90

III Cumulative and infinite hierarchies **93**

8 Cumulative type systems **95**
8.1 Definition . . . 97
8.1.1 Specification and syntax . . . 97
8.1.2 Subtyping . . . 97
8.1.3 Typing . . . 98
8.2 Basic properties . . . 100
8.3 Inter-system properties . . . 100
8.3.1 Subsystems . . . 100
8.3.2 Morphisms . . . 101
8.3.3 Closures . . . 101
8.4 Principal typing . . . 102
8.4.1 Definition . . . 102
8.4.2 Derivation rules . . . 104
8.4.3 Soundness . . . 107
8.4.4 Completeness . . . 107
8.5 Examples of CTSs with principal typing . . . 108

9 Embedding cumulativity **111**
9.1 Translation . . . 113
9.2 Completeness . . . 115
9.2.1 Preservation of substitution . . . 115
9.2.2 Preservation of equivalence . . . 117
9.2.3 Preservation of typing . . . 118

10 Infinite universe hierarchies **119**
10.1 Predicative universes . . . 120
10.2 Impredicative universe . . . 122
10.3 Cumulativity . . . 123

11 Application: translating CIC to Dedukti **129**
11.1 Inductive types . . . 130
11.1.1 Inductive types and eliminators . . . 130
11.1.2 Eliminators in the $\lambda\Pi$-calculus modulo rewriting . . . 132
11.2 Other features . . . 134
11.2.1 Modules . . . 134
11.2.2 Local let definitions and local fixpoints . . . 134
11.2.3 Floating universes . . . 135
11.2.4 Universe polymorphism . . . 136
11.3 Implementation . . . 137
11.3.1 Coqine: Coq to Dedukti . . . 137
11.3.2 Krajono: Matita to Dedukti . . . 138

Conclusion **141**

12 Conclusion **143**
12.1 Related work . . . 143
12.1.1 Logical embeddings . . . 143
12.1.2 Proof interoperability . . . 144
12.2 Future work . . . 145
12.2.1 Improving the translations . . . 145
12.2.2 Designing new embeddings . . . 145
12.2.3 Interoperability in Dedukti . . . 146

Appendix **163**

A Proof details **165**
A.1 Proofs of Chapter 5 . . . 165
A.2 Proofs of Chapter 6 . . . 166
A.3 Proofs of Chapter 8 . . . 169
A.4 Proofs of Chapter 9 . . . 172

B Original presentations **177**
B.1 The $\lambda\Pi$-calculus . . . 177
B.2 Pure type systems . . . 180

Introduction

1

Introduction

1.1 Proof systems

Proof systems are programs that allow users to do mathematics on a computer. The process of writing proofs on a computer is not very different from that of writing programs: the user writes the statement of a theorem and its proof in a language that the computer understands. The computer reads the proofs and checks for its correctness, possibly guiding the user during the writing process. Once a proof is validated, it can be published, stored in databases, queried for by search engines, edited for revision, etc.

In order for the computer to understand it, a proof must be very detailed, explaining all the steps of reasoning, according to the rules of a precise formal system. Writing such formal proofs can be time-consuming, but it is very worthwhile because it allows us to leverage the incredible power of computers. On one hand, computers are much better than humans at meticulously applying precise rules and checking very long derivations, so we can prove things with a much lesser degree of errors. On the other hand, we can use their computational power to perform long calculations automatically in our proofs. Many proof systems have been developed over the years, and they are mainly used in two areas: the *formalization of mathematics*, and *software verification*.

1.1.1 Formalization of mathematics

Formalizing various mathematical theories inside proof systems requires a lot of work, because we need to detail the proofs much more than on paper, but it also increases our confidence. There are theorems whose proofs are so large, or that require exhaustive computations that are so complex, that they were not accepted by the mathematical community until they have been completely formalized and proved in a proof system. Such is the case of the *4-color theorem*, which states that any planar map can be colored such that no two adjacent regions have the same color, using only 4 colors. Stated in the 1850s, it was only proved in 1976 but the proof was not universally accepted because it used a computer program for doing large calculations. In 2005, the proof was formalized by Gonthier and Werner in the Coq proof assistant [Gon05, Gon08], which helped eliminate doubts in the proof. Other notable examples are the *Feit–Thompson theorem* [GAA+13] and the *Kepler conjecture* [HHM+10, Hal14]. These formalization efforts are typically accompanied by the development of *mathematical libraries* [GM10] that contain collections of well-thought definitions and lemmas that can be reused for further work.

1.1.2 Software verification

We can also use proof systems to verify programs and software systems, proving that they are bug-free and that they respect their specification. This is very useful for critical systems where small errors can lead to huge economical or even human losses, for example in industrial, medical, and financial systems. Again, the advantages are that computers are much better than humans at verifying large, complex derivations. The CompCert C compiler [Ler15] is a verified C compiler that has been proved correct in Coq with respect to the C specification, from the source language all the way down to the machine code. Other notable examples include the use of Isabelle/HOL in the formal verification of the seL4 operating system microkernel [KEH+09], the use of HOL Light by Intel in the verification of the design of new chips [Har09], and the use of PVS by NASA to verify airline control systems [Muñ03, MCDB03]. Together with model checking and abstract interpretation, proof systems constitute one of the pillars of *formal methods* in software verification.

1.2 Universality of proofs

Proofs in mathematics are universal, but that is not always the case in proof systems. Doing proofs on a computer requires a certain level of trust because we cannot verify most of these proofs ourselves, nor the systems in which they are done. When we write a proof in one system, we would like to check it independently and reuse it for the benefit of developments in another system. These two aspects of universality, *independent proof checking* and *proof interoperability*, are not always possible.

The obstacles can be many:

- The proofs are written in high-level proof *scripts*, partially proved by tactics or automatically proved by automated theorem provers. There is a lot of missing information that requires complex machinery to reconstruct and check.
- The system has complete proof *objects* but provides no way of accessing them or presenting them in a way that is readable or usable, or does not have a clear specification of their meaning or the logic behind them.
- We can retrieve proofs but we need to translate them in the language of the other systems in order to reuse them. For n systems, we need a number of translations of the order of n^2, unless the translations are composable.
- The translations are inefficient and cannot be composed well.
- The logical formalisms are widely different, and even sometimes incompatible.

A solution to these problems is to translate proofs to a common framework called a *logical framework*, which is used to:

1. Specify the language for writing proofs and the rules of the logic in which the proofs are carried.
2. Check the correctness of the proof, by verifying that each individual step of reasoning follows the rules of the logic.

The framework must be simple enough to be trustworthy but powerful enough to be able to express the proofs of various logics efficiently. The benefits of doing so are several:

- We develop tools to retrieve the proof objects from various systems and drive the effort of providing a better specification for these proof objects.
- We can recheck the proofs in the logical framework, thus increasing our confidence. Logical frameworks provide a venue for proof systems to produce proof objects that are relatively independent of the implementation of the system itself.
- We need a number of translations that is only linear in the number of systems, instead of being quadratic.
- It gives a level playing field in which to study logics. Expressing, or trying to express, different logics in the same framework gives us better insight on the nature of these logics, their similarities, and their differences.
- It paves the way for proof interoperability. By having the proofs of different systems in a common framework, it becomes easier to combine them to form larger theories, in the same way that libraries written in the C language can be linked with programs written in OCaml in low-level assembly languages.

In this thesis, we contribute to the design of these translations, called *logical embeddings*. There are various different logics and logical frameworks, based on a variety of different formalisms. In our work, we focus on those based on *type theory*. In particular, we use the $\lambda\Pi$-*calculus modulo rewriting* as our logical framework and we focus on the embedding of logics that allow *computational* and *higher-order* reasoning.

1.2.1 Computational reasoning

Computation inside proofs can reduce the size of proofs, sometimes by drastic amounts. Indeed, if a person asserts that the statement

$$2 + 2 = 4$$

is true, then we only need to compute the value of $2 + 2$ and compare it to the right hand-side. In this case, the result of $2 + 2$ is 4 and therefore the statement is true. On the other hand, the statement

$$2 + 2 = 5$$

is not true, and indeed the value 4 is not equal to 5. In practice, computation allows us to express the proof of the first statement using one step:

$$\frac{\overline{4 = 4}}{2 + 2 = 4}$$

which is smaller than a non-computational proof that uses the axioms of arithmetic:

$$\frac{\overline{2 + 2 = 3 + 1} \qquad \dfrac{\overline{3 + 1 = 4 + 0} \qquad \overline{4 + 0 = 4}}{3 + 1 = 4}}{2 + 2 = 4}\ .$$

Computation in proofs is especially relevant when we deal with logics that reason about programs, where these steps correspond to *program evaluation*. On larger or more complex examples, the gains become significant enough that the proofs of difficult theorems become tractable. This technique, called *proof by reflection* [BC04], has been famously and successfully used in the proof of the 4-color theorem. To express such proofs in a logical framework, it is thus important to design embeddings that preserve computation.

1.2.2 Higher-order reasoning

Higher-order reasoning allows additional forms of quantification and can add a lot of expressive power to logic. In first-order logic, the universal quantification $\forall$ is restricted to usual objects like natural numbers. It can already express a lot of mathematics, but it has its limits. For example, the induction principle:

$$[P(0) \wedge \forall n. [P(n) \Rightarrow P(n+1)]] \implies \forall n. P(n)$$

cannot be expressed with a finite number of axioms. If we allow ourselves to quantify over propositions and predicates, we can formulate it as a single axiom:

$$\forall P. [[P(0) \wedge \forall n. [P(n) \Rightarrow P(n+1)]] \implies \forall n. P(n)]$$

Similarly, it can be shown that we cannot prove $0 \neq 1$ in intuitionistic type theory and the calculus of constructions without the ability to quantify over types [Smi88]. To express such proofs in a logical framework, it is thus important to design embeddings that can deal with higher-order reasoning.

While computational embeddings and higher-order embeddings have been the subject of much previous work, doing both at the same time has rarely been studied before, and indeed the two are usually incompatible in weak logical frameworks. In this thesis, we design embeddings that allow both computation and higher-order reasoning. We proceed in several stages to achieve our end-goal, which is the translation of the calculus of inductive constructions, the formalism behind the Coq proof system.

1.3 Type theory and logical frameworks

Type theory [Rus08] is an alternative to set theory that was proposed at the beginning of the 20th century as a solution to avoid the paradoxes of naive set theory exhibited by Russell. The main idea behind it is to classify mathematical objects into different classes called *types* (for example the type nat of natural numbers, the type $\mathsf{nat} \to \mathsf{nat}$ of functions, etc.), that ensure that all objects have a meaning and that the range of universal quantification $\forall$ is bounded to a single type. This simple but powerful idea found many applications in both logic and computer science.

After many simplifications, Church [Chu40] reformulated type theory using the λ-calculus, in what is known today as *simple type theory* (STT). The main innovations in that theory are that propositions are represented as objects of just another type o, and that we can write functions that construct these objects in the language of the λ-calculus. This means that we can quantify over propositions (by quantifying over objects of type o) and over predicates (by quantifying over functions of type $\mathsf{nat} \to o$), which allows higher-order reasoning. For these reasons, this simple yet expressive system is also known as *higher-order logic* (HOL). It has been implemented in many proof assistants, including HOL4, HOL Light, ProofPower, and Isabelle.

1.3.1 The Curry–Howard correspondence

Over the years, type theory has evolved to give rise to many systems, all based more or less on the λ-calculus. In the end of the 1960s, a central idea that has emerged is the *Curry–Howard correspondence* [Cur34, dB68, How80], also known as the "propositions-as-types" principle, which highlights an isomorphism between *proofs* and *programs*. A proposition A in some logic $\mathcal{L}$ corresponds to a type $[\![A]\!]$ in some calculus $\lambda_{\mathcal{L}}$, and a proof of A in this logic corresponds to a program of type $[\![A]\!]$ in that calculus. This correspondence can be summarized by the statement:

$$\Gamma \vdash_{\mathcal{L}} A \iff \exists M.\ [\![\Gamma]\!] \vdash_{\lambda_{\mathcal{L}}} M : [\![A]\!]$$

where M is a straightforward encoding of the proof of A. The implication $A \Rightarrow B$ corresponds to the type of functions $A \to B$, and the various derivation rules of the logic are in one-to-one correspondence with the typing rules of the calculus.

By viewing proofs as programs in a typed functional language, *proof-checking* becomes equivalent to *type-checking*, and a lot of the knowledge and expertise that we have from programming languages can be applied to logic, and vice versa. Proofs are no longer only *about* programs but they also *are* programs themselves, and can be manipulated as such, quantified over, passed around, and executed. This beautiful duality has since been extended to many logics and systems:

Propositional logic	Simply typed λ-calculus
First-order logic	$\lambda\Pi$-calculus
Second-order logic	System F
Classical logic	λ-calculus with $\mathsf{call-cc}$

This correspondence highlights the computational content of proofs, which is important for *constructive logics*, i.e. logics that avoid classical reasoning and the use of the law of excluded middle ($A \vee \neg A$), because it allows the extraction of programs from proofs and allows us to view types as program specifications. *Intuitionistic type theory* (ITT) was developed by Martin-Löf [ML73, ML84] as a logic based solely on this idea, and was proposed as a foundation for constructive mathematics. It is implemented in the system Agda. The *calculus of constructions* (CC) was proposed by Coquand and Huet [CH88] in the late 1980s as a powerful calculus that allows impredicative reasoning. Later, the work of Luo [Luo90] and Coquand and Paulin [CP90] unified ITT and CC in what is known today as the *calculus of inductive constructions* (CIC). This powerful and expressive theory serves as a basis for several proof assistants including the famous proof system Coq and Matita. In this thesis, we make heavy use of the Curry–Howard correspondence and we treat proof systems as typed λ-calculi.

1.3.2 The logical framework approach

A logical framework in type theory is a system that can define various logics and express proofs in those logics in such a way that *provability* corresponds to *type inhabitation*. Instead of using a different calculus $\lambda_{\mathcal{L}}$ for every logic $\mathcal{L}$, we use a single calculus LF. A logic $\mathcal{L}$ is embedded as a signature $\Sigma_{\mathcal{L}}$ that defines the constructs, axioms, and rules of the logic, in such a way that a proposition A is provable in $\mathcal{L}$ if and only if its corresponding type is inhabited in this signature:

$$\Gamma \vdash_{\mathcal{L}} A \iff \exists M.\ \Sigma_{\mathcal{L}}, [\![\Gamma]\!] \vdash_{LF} M : [\![A]\!]\,.$$

We can thus use the type checker of the framework as a proof checker for different logics.

Several logical frameworks with varying degrees of power and expressivity have been proposed over the years. The approach was pioneered by de Bruijn [dB68] in the AUTOMATH system in the late 1960s, but it was popularized by Harper, Honsell, and Plotkin [HHP93] in the system LF, which is a relatively weak but expressive system that consists of a minimal lambda-calculus with dependent types, also known as the $\lambda\Pi$-*calculus*. Through the propositions-as-types principle, it can express a wide variety of logics. It is implemented in the system TWELF, which is a type-checker and meta-theorem prover for that calculus. It has been successfully used to embed a wide variety of theories, including some higher-order ones such as HOL and system F.

In our thesis, we use the $\lambda\Pi$-*calculus modulo rewriting* [CD07, BCH12], an extension of the $\lambda\Pi$-calculus with *term rewriting*. As we will see, the $\lambda\Pi$-calculus is powerful enough to express computational embeddings, such as for first-order logic, and higher-order embeddings, such as for higher-order logic. However, it cannot easily and efficiently express the proofs of theories which are at the same time computational *and* higher-order, such as intuitionistic type theory or the calculus of constructions. Therefore, a logical framework based on it cannot practically express the proofs of AGDA, COQ, and MATITA. The $\lambda\Pi$-calculus modulo rewriting, as we will show, can be seen as a good compromise between the $\lambda\Pi$-calculus and Martin-Löf's logical framework. It is implemented in the type checker DEDUKTI, which we use in our thesis to check the results of our translations. In Chapter 2, we recall the theory of the $\lambda\Pi$-calculus and discuss its strength and weaknesses. In Chapter 3, we present the $\lambda\Pi$-calculus modulo rewriting and its metatheory and show how it alleviates the weaknesses mentioned above.

1.4 Computational higher-order logics

1.4.1 Pure type systems

In the late 1980s, Berardi and Terlouw [Ber89, Ter89] introduced *pure type systems* as a general framework for describing various typed λ-calculi. The framework is general enough to include many popular systems such as the simply typed lambda-calculus, system F, the calculus of constructions, and even higher-order logic. Because of its generality, it is the perfect candidate for studying the embedding of various computational higher-order systems. We briefly recall the theory of pure type systems in Chapter 4.

This thesis stems from the work of Cousineau and Dowek [CD07], who presented in 2007 a general embedding of pure type systems in the $\lambda\Pi$-calculus modulo rewriting. We recall and revise their work in Chapter 5, where we show that the embedding preserves typing and computation. Chapter 6 is devoted to proving the conservativity of this embedding, a result which was missing from the original paper. More specifically, we show that an original formula is provable in the embedding only if it is provable in the original system, thus justifying the use of the $\lambda\Pi$-calculus modulo rewriting as a logical framework. Traditional techniques fail in this setting, so we propose a new approach based on reducibility to show that a proof in the embedding reduces to a proof in the original system. This work has been the subject of a publication at TLCA 2015 [Ass15], although the proof we present here is more general and can be adapted to the translations we present later in this thesis.

In Chapter 7, we show how we applied these ideas in the implementation of HOLIDE, a tool for the automatic translation of HOL proofs to Dedukti, and present experimental results obtained from translating the OPENTHEORY standard library. HOLIDE was started as an internship project, but we continued its development during this thesis. In particular, we implemented *term sharing*, an optimization that proved to be crucial for the efficiency of the translation. This work was presented at PxTP 2015 [AB15].

1.4.2 Universes and cumulativity

In type theory, a *universe* is a type whose elements are themselves types. In systems based on the Curry–Howard correspondence, universes allow higher-order quantification (e.g. over propositions or predicates) and are essential to higher-order reasoning. They are already present in pure type systems, where they are called *sorts*.

While a basic embedding of universes can already be found in the embedding of pure type systems, there are additional features related to universes that are not covered, namely *infinite universe hierarchies* and *universe cumulativity*. Universes are usually organized in an infinite hierarchy, where each universe is a member of the next one:

$$\mathsf{Type}_0 \in \mathsf{Type}_1 \in \mathsf{Type}_2 \in \cdots .$$

This stratification is required to avoid paradoxes that would arise from $\mathsf{Type} \in \mathsf{Type}$. It is similar to the distinction between *sets* (or small sets) and *collections* (or large sets) in set theory. Cumulativity expresses that each universe is *contained* in the next one:

$$\mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \mathsf{Type}_2 \subseteq \cdots .$$

Infinite hierarchies and cumulativity have become a common part of many logical systems such as intuitionistic type theory and the calculus of inductive constructions, implemented in COQ and MATITA.

We present techniques for embedding those two features in the $\lambda\Pi$-calculus modulo rewriting. To this end, we first generalize the framework of pure type systems to *cumulative type systems* (CTS) [Bar99, Las12], which we present in Chapter 8. We then extend the embedding of pure type systems to cumulative type systems. A major difficulty is that cumulativity in Coq and Matita is expressed implicitly. To this end, we base ourselves on a comparison of two popular formulations of universes in intuitionistic type theory, *à la Russell* (implicit) and *à la Tarski* (explicit), and their interaction with cumulativity. We propose to use *explicit coercions* to represent cumulativity. In particular, we show that, in order for the embedding to be complete, coercions need to satisfy a condition called *full reflection* that ensures that types have a unique representation inside universes. This idea was the subject of a publication at TYPES 2014 [Ass14], although under a different light, as a calculus of constructions with explicit subtyping. The embedding of cumulative type systems is the subject of Chapter 9. In Chapter 10, we focus on infinite hierarchies, and show how to represent them using a finite set of constants and rewrite rules.

This thesis finally culminates with an embedding of the calculus of inductive constructions, which we have implemented in two automated translation tools: Coqine, that translates the proofs of Coq to Dedukti, and Krajono, that translates the proofs of Matita to Dedukti. We present this translation along with experimental results obtained from translating various libraries in Chapter 11.

1.5 Contributions

The main contributions of this thesis can be summarized as follows.

1. We revisit the **general embedding of pure type systems** in the $\lambda\Pi$-calculus modulo rewriting of Cousineau and Dowek and prove that it is conservative.

2. We extend the embedding to express **cumulativity and infinite hierarchies**. We highlight the need to have a unique representation of types inside universes to preserve the completeness of the embedding.

3. We implement these theoretical results in practical **tools for the automatic translation** of the proofs of HOL, Coq, and Matita.

To this, we also add:

0. We provide a critical analysis of the various forms of embeddings in logical frameworks based on type theory, and discuss the features needed for the embedding of **computational higher-order logics**.

1.6 Preliminary notions and notations

In this thesis, we will consider various systems based on the λ-calculus. We assume the reader is familiar with the λ-calculus, and more specifically, some typed variant *à la Church*, where λ-abstractions are annotated by the type of their domain:

$$\lambda x : A \,.\, M.$$

In the tradition of pure type systems, we do not distinguish between terms and types, which therefore belong to the same syntactic category. The type of λ-abstractions are Π-types, also known as *dependent products* and *dependent function types*:

$$\Pi x : A \,.\, B$$

which we use to denote polymorphic types, dependent types, etc.

We use standard λ-calculus notation, with term application written as $M\,N$. We use x, y, z to denote variables, M, N, A, B to denote terms, $\mathsf{c1}, \mathsf{c2}, \mathsf{c3}$ to denote specific constants, and Γ, Δ, Σ to denote contexts. λ-abstractions associate to the right and applications associate to the left, with application taking precedence over λ-abstraction. For example,

$$\lambda x_1 : A_1 \,.\, \lambda x_2 : A_2 \,.\, M\,N_1\,N_2$$

stands for the term

$$\lambda x_1 : A_1 \,.\, (\lambda x_2 : A_2 \,.\, ((M\,N_1)\;N_2))\,.$$

We make free use of parentheses to disambiguate. We also use the following shortcut notations. We write $\lambda x \,.\, M$ (with no type annotation) instead of $\lambda x : A \,.\, M$ when the type A is obvious from the surrounding context. We use the following notations:

$$\begin{aligned}
\lambda x, y : A \,.\, M &\stackrel{\text{def}}{=} \lambda x : A \,.\, \lambda x : A \,.\, M \\
\Pi x, y : A \,.\, B &\stackrel{\text{def}}{=} \Pi x : A \,.\, \Pi y : A \,.\, B \\
\lambda \vec{x} : \vec{A} \,.\, M &\stackrel{\text{def}}{=} \lambda x_1 : A_1 \,.\, \cdots \lambda x_n : A_n \,.\, M \\
\Pi \vec{x} : \vec{A} \,.\, B &\stackrel{\text{def}}{=} \Pi x : A_1 \,.\, \cdots \Pi x_n : A_n \,.\, B \\
M\,\vec{N} &\stackrel{\text{def}}{=} M\,N_1 \cdots N_n
\end{aligned}$$

where $\vec{x} = x_1, \ldots, x_n$, $\vec{A} = A_1, \ldots, A_n$, and $\vec{N} = N_1, \ldots, N_n$. We write $A \to B$ instead of $\Pi x : A \,.\, B$ when the variable x does not appear in free in B.

As usual, the variable x is bound in M in the term $\lambda x : A \,.\, M$, and is bound in B in the term $\Pi x : A \,.\, B$. We write $\mathrm{FV}\,(M)$ for the set of free variables of a term M. Terms are identified up to *α-equivalence* and we use *capture-avoiding substitution*. A substitution is written as $M\,\{x_1 \backslash N_1, \ldots, x_n \backslash N_n\}$, where the term N_i replaces the free occurrences of the variable x_i in the term M, possibly α-renaming binders to avoid capturing free variables. If $\sigma = \{x_1 \backslash N_1, \ldots, x_n \backslash N_n\}$ is a substitution, we also sometimes write $\sigma\,(M)$ instead of $M\sigma$ for the substitution operation.

Terms are equipped with *reduction relations*, which we will write using $\longrightarrow$ (long arrows). The reduction relations that we will consider are *closed by the subterm relation*. A reduction relation is closed by the subterm relation when $M \longrightarrow M'$ implies $C[M] \longrightarrow C[M']$ for any term context C. In other words, evaluation is unrestricted, it can occur anywhere in the term, in any order. In the systems we will consider, the main reduction relation will be *β-reduction*, i.e. the smallest reduction relation $\longrightarrow_\beta$ that is closed by the subterm relation such that, for all A, M, N, we have

$$(\lambda x : A \,.\, M)\ N \longrightarrow_\beta M\,\{x \backslash N\}\,.$$

Some of the systems that we will consider also have *term rewriting*, i.e. a reduction relation induced by a set of *rewrite rules*. A rewrite rule is a pair of terms written $M \longmapsto N$. Given a set of rewrite rules R, the reduction relation $\longrightarrow_R$ is the smallest reduction relation that is closed by the subterm relation such that, for all $(M \longmapsto N) \in R$ and substitution σ, we have

$$\sigma\,(M) \longrightarrow_R \sigma\,(N)\,.$$

For any reduction relation $\longrightarrow_r$, we write $\longleftarrow_r$ for its symmetric reduction relation, $\longrightarrow_r^+$ for its transitive closure, $\longrightarrow_r^*$ for its reflexive transitive closure, and $\equiv_r$ for its reflexive symmetric transitive closure. We note that $\equiv_r$ is a *congruence*, i.e. an equivalence relation that is compatible with the structure of terms. We write $\longrightarrow_{r_1 r_2}$ for the union of the relations $\longrightarrow_{r_1}$ and $\longrightarrow_{r_2}$ and $\longrightarrow_{r_1}\longrightarrow_{r_2}$ for their composition.

For a good introduction to λ-calculus and its typed variations, we strongly recommend Barendregt's *Lambda calculi with types* [Bar92]. Chapters 2, 3, 4, and 8 contain mostly the recollection of standard material, so readers already familiar with their respective content can skip them, although we strongly recommend reading Sections 2.2–2.4 and 3.3 for an explanation of the main ideas behind this thesis.

I

Frameworks and embeddings

2

The $\lambda\Pi$-calculus

The $\lambda\Pi$ calculus is a typed λ-calculus with dependent types. Known under many names ($\lambda\Pi$, LF, λP), it was proposed in a seminal paper by Harper, Honsell, and Plotkin [HHP93] as a logical framework to define logics and express proofs in those logics. The calculus is small but powerful enough to express a wide variety of logics. There are two features that make this possible. First, λ-abstractions allow the convenient and efficient representation of binders using *higher-order abstract syntax* (HOAS). Second, Π-types allow the correct representation of dependencies. It gained a lot of popularity and was implemented in systems such as Twelf[1] [PS99], which has been successfully used as a logical framework in various projects [App01, SS06].

Our work draws heavily from the $\lambda\Pi$ tradition. The $\lambda\Pi$-calculus modulo rewriting is an extension of this system with rewrite rules, and our embeddings rely on the same principles. In this chapter, we recall the theory of $\lambda\Pi$ and show the main ideas behind its capability to express logics. In particular, we will show that there are two main categories of embeddings, along the lines of Geuvers [GB99]: the *computational* embeddings and the *higher-order* embeddings. We will show that they have different properties and that they are incompatible, thus justifying the extension with rewriting of the next chapter.

[1] http://twelf.org/

2.1 Definition

We choose a presentation close to pure type systems, which does not make a distinction between terms and types. The reason behind this choice is that it leads to less duplication in the syntax and typing rules. The presentation of Harper et al. [HHP93] uses separate syntactic categories for objects, types, and kinds, as well as a separate signature Σ for global constants to better differentiate between the declarations that are part of the logic specification, and those that are local variable declarations, i.e. free hypotheses and free term variables. We will not make these distinctions on the syntax level and instead rely on naming conventions to differentiate between the different classes. The original presentation can be found in Appendix B.1.

Definition 2.1.1 (Syntax)**.** The syntax of $\lambda\Pi$ is given by the following grammar:

$$\begin{array}{llll} \text{sorts} & s & \in \mathcal{S} & ::= \mathsf{Type} \mid \mathsf{Kind} \\ \text{terms} & M, N, A, B & \in \mathcal{T} & ::= x \mid s \mid \Pi x : A \,.\, B \mid \lambda x : A \,.\, M \mid M\, N \\ \text{contexts} & \Sigma, \Gamma & \in \mathcal{G} & ::= \varnothing \mid \Gamma, x : A \end{array}$$

We try to use letters M, N for objects, letters A, B for types, and the letter K for kinds. We try to use the letter Σ for the context of global declarations (i.e. constants) that are proper to the entire logic and Γ for the context of local declarations.

Definition 2.1.2 (Typing)**.** The typing relations

- $\Gamma \vdash M : A$, meaning that the term M has *type* A in the context Γ,
- $\Gamma \vdash \mathrm{WF}$, meaning that the context Γ is *well-formed*,

are derived by the rules in Figure 2.1. A term A is a *well-formed type* in Γ when $\Gamma \vdash \mathrm{WF}$ and either $\Gamma \vdash A : s$ or $A = s$ for some $s \in \mathcal{S}$. We write this as $\Gamma \vdash A\ \mathrm{WF}$. It is *inhabited* in Γ when $\exists M, \Gamma \vdash M : A$. We write $\Gamma \vdash M : A : B$ as a shortcut for $\Gamma \vdash M : A$ and $\Gamma \vdash A : B$. We sometimes write $\vdash_{\lambda\Pi}$ instead of $\vdash$ to disambiguate.

The $\lambda\Pi$-calculus enjoys many of the usual desirable properties for systems of typed lambda calculus, including confluence, subject reduction, strong normalization, and decidability of type checking. Since it is an instance of a pure type system and since we will not use it directly, we will postpone the presentation of these properties to the chapter on pure type systems (Chapter 4).

2.2 As a first-order logic calculus

The $\lambda\Pi$ calculus-can be seen as the calculus of minimal first-order logic through the Curry–Howard correspondence. First-order logic allows predicates and quantification over objects, which correspond naturally to dependent types: a universal quantification $\forall x.A\,(x)$ is interpreted as a dependent function type $\Pi x.A\,(x)$ and a proof of $\forall x.A\,(x)$ is interpreted as a function taking an object x and producing a proof of $A\,(x)$.

$$\frac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash \mathrm{WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{Type}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A \,.\, B : s}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}\ \textsc{Conv}$$

Figure 2.1 – Typing rules of the system $\lambda\Pi$

Consider a signature in first-order logic, with function symbols f of arity n_f and predicate symbols p of arity n_p. First, we declare in our context Σ a type $\iota : \mathsf{Type}$ which is the type of the objects of the logic. We then extend the signature with declarations $f : \iota^n \to \iota$ and $p : \iota^n \to \mathsf{Type}$. The propositions of minimal predicate logic then have a direct interpretation as types of the $\lambda\Pi$ calculus:

$$\begin{aligned}
[\![p\,(M_1, \ldots, M_n)]\!] &= p\,[M_1]\,\ldots\,[M_n] \\
[\![A \Rightarrow B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![\forall x.A]\!] &= \Pi x : \iota\,.\,[\![A]\!]
\end{aligned}$$

where $[M]$ is the straightforward curried representation of the objects of the logic:

$$\begin{aligned}
[x] &= x \\
[f\,(M_1, \ldots, M_n)] &= f\,[M_1]\,\ldots\,[M_n]
\end{aligned}$$

The contexts are interpreted as:

$$[\![A_1, \ldots, A_n]\!] = h_1 : [\![A_1]\!]\,, \ldots, h_n : [\![A_n]\!]$$

The proofs of minimal predicate logic correspond to the typing derivations in $\lambda\Pi$: for any proof of a statement $\Gamma \vdash A$, there is a term M which has type A in the context Γ. Moreover, the typing derivation of M mirrors the proof:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-INTRO} \qquad\qquad \frac{\Gamma, h : A \vdash M : B}{\Gamma \vdash \lambda h : A\,.\,M : A \to B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-ELIM} \qquad\qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}\ \textsc{App}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A}\ \forall\text{-INTRO}\ x \qquad\qquad \frac{\Gamma, x : \iota \vdash M : A}{\Gamma \vdash \lambda x : \iota\,.\,M : \Pi x : \iota\,.\,A}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A\,\{x \backslash N\}}\ \forall\text{-ELIM} \qquad\qquad \frac{\Gamma \vdash M : \Pi x : \iota\,.\,A \qquad \Gamma \vdash N : \iota}{\Gamma \vdash M\,N : A\,\{x \backslash N\}}\ \textsc{App}$$

Notice that both introduction rules are represented by the Lam rule and that both elimination rules are represented by the App rule. Formally, there is a translation function $[\pi]$ such that if π is a proof of $\Gamma \vdash A$ then $[\pi]$ has type $[\![A]\!]$ in the context $\Sigma, [\![\Gamma]\!]$:

$$\Gamma \overset{\pi}{\vdash} A \iff \Sigma, [\![\Gamma]\!] \vdash [\pi] : [\![A]\!]\,.$$

The term $[\pi]$ is an encoding of the proof and is closely related to its own typing derivation. The translation function is defined by induction on the proof:

$$\left[\frac{}{\Gamma, A, \Gamma' \vdash A}\ \textsc{Hyp}\right] \qquad = h_n \qquad \text{(where } n = |\Gamma|\text{)}$$

$$\left[\frac{\Gamma, A \overset{\pi}{\vdash} B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-INTRO}\right] \qquad = \lambda h_n : A \,.\, [\pi] \qquad \text{(where } n = |\Gamma|\text{)}$$

$$\left[\frac{\Gamma \vdash A \overset{\pi}{\Rightarrow} B \qquad \Gamma \overset{\pi'}{\vdash} A}{\Gamma \vdash B}\ \Rightarrow\text{-ELIM}\right] = [\pi]\ [\pi']$$

$$\left[\frac{\Gamma \overset{\pi}{\vdash} A}{\Gamma \vdash \forall x.A}\ \forall\text{-INTRO}\right] \qquad = \lambda x : \iota \,.\, [\pi]$$

$$\left[\frac{\Gamma \vdash \overset{\pi}{\forall} x.A}{\Gamma \vdash A\,\{x \backslash N\}}\ \forall\text{-ELIM}\right] \qquad = [\pi]\ [N]$$

The adequacy of this representation was proved by Geuvers and Barendsen [GB99].

This embedding is *computational*: cut elimination in the proofs of predicate logic corresponds to the evaluation of the corresponding proof terms. In the $\lambda\Pi$-calculus, it corresponds to β-reduction:

$$\frac{\dfrac{\Gamma, A \overset{\pi}{\vdash} B}{\Gamma \vdash A \Rightarrow B} \qquad \Gamma \overset{\pi'}{\vdash} A}{\Gamma \vdash B} \longrightarrow \overset{\pi\{h\backslash\pi'\}}{\Gamma \vdash B} \qquad (\lambda h : A \,.\, \pi)\ \pi' \longrightarrow_\beta \pi\,\{h\backslash\pi'\}$$

$$\frac{\dfrac{\Gamma \overset{\pi}{\vdash} A}{\Gamma \vdash \forall x.A}}{\Gamma \vdash A\,\{x\backslash N\}} \longrightarrow \overset{\pi\{x\backslash N\}}{\Gamma \vdash A} \qquad (\lambda x : \iota \,.\, \pi)\ N \longrightarrow_\beta \pi\,\{x\backslash N\}$$

We say that β-reduction is *essential*[2] in this embedding because a step of β-reduction is playing the role of a reduction step in the original system. With this embedding, the strong normalization of β-reduction in $\lambda\Pi$ shows that cut elimination terminates in minimal logic (and therefore that the logic is sound).

[2] *As opposed to* administrative. *This terminology comes from the theory of* continuation passing style *(CPS) translations [Plo75, DF92], where the translation produces intermediary β-redexes that do not correspond to any β-redex in the original program.*

2.3 As a logical framework

Instead of interpreting propositions as types, we declare a type of propositions $o : \mathsf{Type}$ and constructors of the type o representing implication and universal quantification:

$$\begin{aligned} \mathsf{imp} &: o \to o \to o, \\ \mathsf{forall} &: (\iota \to o) \to o. \end{aligned}$$

A predicate symbol p of arity n is now declared as $p : \iota^n \to o$. The propositions are now interpreted as terms of type o:

$$\begin{aligned} [p\,(M_1, \ldots, M_n)] &= p\,[M_1]\,\ldots\,[M_n] \\ [A \Rightarrow B] &= \mathsf{imp}\,[A]\;[B] \\ [\forall x.A] &= \mathsf{forall}\,(\lambda x : \iota\,.\,A)\,. \end{aligned}$$

Notice the use of *higher order abstract syntax* (HOAS) to represent binding in the translation of universal quantification. The terms of type o can be thought of as *codes* representing propositions. To interpret $\Gamma \vdash A$, we add $\mathsf{proof} : o \to \mathsf{Type}$ which takes the code of a proposition to the type of its proofs. We then add constructors for each of the inference rules of minimal predicate logic:

$$\begin{aligned} \mathsf{imp_intro} &: \Pi p, q : o\,.\,(\mathsf{proof}\,p \to \mathsf{proof}\,q) \to \mathsf{proof}\,(\mathsf{imp}\,p\,q)\,, \\ \mathsf{imp_elim} &: \Pi p, q : o\,.\,\mathsf{proof}\,(\mathsf{imp}\,p\,q) \to \mathsf{proof}\,p \to \mathsf{proof}\,q, \\ \mathsf{forall_intro} &: \Pi p : (\iota \to o)\,.\,(\Pi x : \iota\,.\,\mathsf{proof}\,(p\,x)) \to \mathsf{proof}\,(\mathsf{forall}\,p)\,, \\ \mathsf{forall_elim} &: \Pi p : (\iota \to o)\,.\,\mathsf{proof}\,(\mathsf{forall}\,p) \to \Pi x : \iota\,.\,\mathsf{proof}\,(p\,x)\,. \end{aligned}$$

A proof of A is now translated as a term of type $\mathsf{proof}\,[A]$. If we write $[\![A]\!]$ for $\mathsf{proof}\,[A]$, the translation becomes:

$$\left[\frac{}{\Gamma, A, \Gamma' \vdash A}\ \textsc{Hyp}\right] = h_n \quad (\text{where } n = |\Gamma|)$$

$$\left[\frac{\Gamma, A \overset{\pi}{\vdash} B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-INTRO}\right] = \mathsf{imp_intro}\,[A]\;[B]\;(\lambda h_n : [\![A]\!]\,.\,[\pi])$$

$$\left[\frac{\Gamma \vdash A \overset{\pi}{\Rightarrow} B \qquad \Gamma \overset{\pi'}{\vdash} A}{\Gamma \vdash B}\ \Rightarrow\text{-ELIM}\right] = \mathsf{imp_elim}\,[A]\;[B]\,[\pi]\;[\pi']$$

$$\left[\frac{\Gamma \overset{\pi}{\vdash} A}{\Gamma \vdash \forall x.A}\ \forall\text{-INTRO}\right] = \mathsf{forall_intro}\,(\lambda x : \iota\,.\,[A])\;(\lambda x : \iota\,.\,[\pi])$$

$$\left[\frac{\Gamma \vdash \overset{\pi}{\forall} x.A}{\Gamma \vdash A\,\{x \backslash N\}}\ \forall\text{-ELIM}\right] = \mathsf{forall_elim}\,(\lambda x : \iota\,.\,[A])\;[\pi]\;[N]$$

The embedding again satisfies the property

$$\Gamma \overset{\pi}{\vdash} A \iff \Sigma, [\![\Gamma]\!] \vdash [\pi] : [\![A]\!] \,.$$

This embedding is *not* computational, as it does not preserve reductions. Indeed, it is straighforward to see that the translation of a cut is a normal form. In fact, the translation of any proof is a normal form. There are well-typed non-normal forms that do not belong to the translation, such as

$$(\lambda p : o \,.\, \mathsf{imp_intro}\, p\, p\, (\lambda h : \mathsf{proof}\, p \,.\, h))\ (\mathsf{imp}\, q\, q)$$

but those can be reduced to a normal form such as

$$\mathsf{imp_intro}\, (\mathsf{imp}\, q\, q)\ (\mathsf{imp}\, q\, q)\ (\lambda h : \mathsf{proof}\, (\mathsf{imp}\, q\, q) \,.\, h)$$

which is the translation of a proof of $(q \Rightarrow q) \Rightarrow (q \Rightarrow q)$.

However, β-reduction is still very important because we need it for the HOAS representation: a substitution is represented by a function application (such as $p\, x$ in the type of forall_elim), and β reduction is necessary for it to give the expected result. We say that β-reduction is *administrative*[3] in this embedding because a step of β-reduction does not correspond to anything in the original system, and in fact β-equivalent terms represent the same proof. With this embedding, the strong normalization of β-reduction in $\lambda\Pi$ has absolutely no bearing on the original logic.

2.4 Limitations of $\lambda\Pi$

We saw two kinds of embeddings, which we call *computational embeddings* and *higher-order embeddings*. The first interprets propositions directly as types and can interpret the proofs of minimal first-order logic as well as systems that are strictly smaller (e.g. minimal predicate logic, simply typed λ-calculus, etc.). It is computational because it preserves reductions. The second interprets propositions as objects and has a predicate proof for the type of their proofs. It is not computational because it does not preserve reduction but, since it represents propositions as objects, it can express higher-order quantification and therefore interpret systems that are strictly larger than $\lambda\Pi$.

For example, we cannot interpret full predicate logic in the first approach because there is no counter-part for conjunction ($\wedge$) and disjunction ($\vee$) in $\lambda\Pi$. However, it is easy to extend the embedding of the second approach to include those:

$$\begin{aligned}
&\mathsf{and} && : o \to o \to o, \\
&\mathsf{and_intro} && : \Pi p, q : o \,.\, \mathsf{proof}\, p \to \mathsf{proof}\, q \to \mathsf{proof}\, (\mathsf{and}\, p\, q)\,, \\
&\mathsf{and_elim1} && : \Pi p, q : o \,.\, \mathsf{proof}\, (\mathsf{and}\, p\, q) \to \mathsf{proof}\, p, \\
&\mathsf{and_elim2} && : \Pi p, q : o \,.\, \mathsf{proof}\, (\mathsf{and}\, p\, q) \to \mathsf{proof}\, q,
\end{aligned}$$

[3]See Footnote 2.

$$
\begin{aligned}
&\mathsf{or} && : o \to o \to o,\\
&\mathsf{or_intro1} && : \Pi p, q : o \,.\, \mathsf{proof}\, p \to \mathsf{proof}\,(\mathsf{or}\, p\, q)\,,\\
&\mathsf{or_intro1} && : \Pi p, q : o \,.\, \mathsf{proof}\, q \to \mathsf{proof}\,(\mathsf{or}\, p\, q)\,,\\
&\mathsf{or_elim} && : \Pi p, q, r : o \,.\, \mathsf{proof}\,(\mathsf{or}\, p\, q) \to\\
& && \quad (\mathsf{proof}\, p \to \mathsf{proof}\, r) \to (\mathsf{proof}\, q \to \mathsf{proof}\, r) \to \mathsf{proof}\, r.
\end{aligned}
$$

Similarly, it is possible to embed higher-order systems such as system F and higher-order logic (HOL) in $\lambda\Pi$ using this approach. In fact, the embedding of system F is very close the embedding of minimal predicate logic given above: it is obtained by just changing forall to quantify over propositions instead of objects and changing the type of the other constants correspondingly. After renaming the constants to fit better with the language of system F, the embedding of the types becomes:

$$
\begin{aligned}
&\mathsf{type} && : \mathsf{Type},\\
&\mathsf{arrow} && : \mathsf{type} \to \mathsf{type} \to \mathsf{type},\\
&\mathsf{forall} && : (\mathsf{type} \to \mathsf{type}) \to \mathsf{type}.
\end{aligned}
$$

One should not confuse type (with a lowercase "t"), which is the type of (the representation of) the types of system F, and Type (with an uppercase "T"), which is the type of the types of $\lambda\Pi$. The representation of the types of system F as objects is precisely what allows us to express polymorphism. The embedding of terms becomes:

$$
\begin{aligned}
&\mathsf{term} && : \mathsf{type} \to \mathsf{Type},\\
&\mathsf{lam} && : \Pi a, b : \mathsf{type} \,.\, (\mathsf{term}\, a \to \mathsf{term}\, b) \to \mathsf{term}\,(\mathsf{arrow}\, a\, b)\,,\\
&\mathsf{app} && : \Pi a, b : \mathsf{type} \,.\, \mathsf{term}\,(\mathsf{arrow}\, a\, b) \to \mathsf{term}\, a \to \mathsf{term}\, b,\\
&\mathsf{biglam} && : \Pi p : (\mathsf{type} \to \mathsf{type}) \,.\, (\Pi x : \mathsf{type} \,.\, \mathsf{term}\,(p\, x)) \to \mathsf{term}\,(\mathsf{forall}\, p)\,,\\
&\mathsf{bigapp} && : \Pi p : (\mathsf{type} \to \mathsf{type}) \,.\, \mathsf{term}\,(\mathsf{forall}\, p) \to \Pi x : \mathsf{type} \,.\, \mathsf{term}\,(p\, x)\,.
\end{aligned}
$$

Again, this embedding is *not* computational because it does not preserve reduction: if $M \longrightarrow_\beta M'$ in system F then $[M] \not\longrightarrow_\beta [M']$ in $\lambda\Pi$. In fact, it is not possible to have a computational embedding of system F in $\lambda\Pi$ without extending the language, assuming of course that we are interested in an embedding that is correct, that is *sound* and *complete*.[4] In that sense, we cannot have an embedding that is both higher-order and computational.

Why is that a problem? Consider the calculus of constructions, which contains both higher-order quantification and dependent types. Because of higher-order quantification, we know we need to follow the second approach. At the same time, we need to account for the conversion rule:

$$
\frac{\Gamma \vdash M : A \qquad A \equiv B}{\Gamma \vdash M : B}\ \textsc{Conv}\,.
$$

[4]These notions will be defined more precisely later.

In the embedding, we must have $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$. However, since the embedding does not preserve reduction, we have $[\![A]\!] \not\equiv [\![B]\!]$. Therefore, $[\![\Gamma]\!] \nvdash [M] : [\![B]\!]$, which means the embedding is incomplete. As soon as we want to encode a logic that is at the same time computational *and* higher-order, we have to choose between a computational embedding (which would be incomplete because it cannot do higher-order quantification) or a higher-order embedding (which would be incomplete because it does not simulate computation), unless it is unsound.

There is a third alternative, which is to encode the equivalence of types as a relation equiv and have a constructor, called a *coercion*, for the conversion rule:

$$\mathsf{conv} : \Pi a, b : \mathsf{type} \,.\, \mathsf{equiv}\, a\, b \to \mathsf{term}\, a \to \mathsf{term}\, b\,.$$

However, these embeddings are very costly because they require encoding the proof of equivalence inside the proof term. These proofs can be quite large in practice. As a matter of fact, the technique of *proof by reflection* [BC04] relies precisely on making use of the computational power of the logic to avoid them. That technique has been successfully used in proof assistants, for example in Coq to build useful libraires such as ssreflect [GLR09] and prove important theorems such as the 4 color theorem [Gon05, Gon08].

To encode rich theories like the calculus of constructions or intuitionistic type theory, we therefore need a framework that is strictly larger than $\lambda\Pi$ that can express embeddings that are at the same time higher-order and computational. This is where the $\lambda\Pi$-calculus modulo rewriting comes in. By adding rewrite rules to the framework, we can recover the computational aspect of the logic, and thus ensure that the embedding is complete. We show how to do this in Chapter 3.

3

The $\lambda\Pi$-calculus modulo rewriting

As we saw in the previous chapter, the $\lambda\Pi$-calculus lacks some computational capability in higher-order embeddings. To recover expressivity, the equivalence relation in the conversion rule must be enhanced. In this regard, Martin-Löf's logical framework [NPS90, Luo94] is the ideal setting, being able to extend the conversion with typed equations

$$M \equiv N : A,$$

read as *the two terms M and N of type A are equivalent.* This formulation makes expressing theories a breeze and serves as the framework in which Martin-Lof's intuitionistic type theory is expressed [ML84, ML98].

However, it is a theoretical setting that is not well-suited for implementation. First, the equivalence is expressed as a typed judgement that is regarded as an integral part of the derivation tree, instead of as a side condition like in pure type systems:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B \text{ WF} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}\ \textsc{Conv}.$$

This has the advantage of guaranteeing "subject reduction" because the equivalence is only defined on terms of the same type, instead of on untyped terms like in pure type systems. On the downside, it is not well-suited for implementation as such, because keeping track of the types at each step is prohibitively expensive. In practice, an untyped version is implemented and shown to be equivalent, for example as shown by Adams [Ada06] and Siles and Herbelin [SH12] in the case of pure type systems.

More importantly though, solving equations in an arbitrary equation system is undecidable in general. As a result, the type-checking problem

$$\Gamma \vdash M : A$$

is undecidable for arbitrary theories, because the proof of equivalence $\Gamma \vdash A \equiv B$ is not encoded in M, A, or Γ.

This is where the $\lambda\Pi$-*calculus modulo rewriting* ($\lambda\Pi R$) comes in. The idea is very simple: orient the equations into rewrite rules to give direction to the computation. As long as the resulting rewrite system is well-behaved, that is confluent and strongly normalizing, type-checking is decidable and there is a correct and complete algorithm for type-checking. Historically, $\lambda\Pi R$ originated as a λ-calculus counterpart of *deduction modulo* through the Curry–Howard correspondence. Deduction modulo is an extension of first-order logic with rewrite rules on terms and atomic propositions that was proposed by Dowek et al. [DHK03, DW03] with the idea that some theories are better expressed as rewrite rules than as axioms. This is desirable from a constructive point of view as it gives a computational content to proofs and helps recovering the disjunction and witness property. Later, Cousineau and Dowek [CD07] introduced the calculus and showed that it can embed any functional pure type system, leading to its proposal as a universal proof framework. It has been implemented in the type-checker Dedukti[1] together with a number of translations from other systems [BB12, BCH12, Bur13, CD15, CH15].

The metatheory of the $\lambda\Pi$-calulus modulo rewriting has changed over the years since its original inception. In this chapter, we present $\lambda\Pi R$ in its modern incarnation due largely to the works of Saillard [Sai13, Sai15]. We give conditions on the rewrite rules that satisfy soundness and completeness, and show how the calculus allows computational embeddings of higher-order theories such as system F. The embedding of pure type systems will be given in Chapter 5.

3.1 Definition

Definition 3.1.1 (Syntax)**.** The syntax of $\lambda\Pi R$ is given by the following grammar:

$$\begin{array}{llll}
\text{sorts} & s & \in \mathcal{S} & ::= \mathsf{Type} \mid \mathsf{Kind} \\
\text{terms} & M, N, A, B & \in \mathcal{T} & ::= s \mid x \mid \Pi x : A \,.\, B \mid \lambda x : A \,.\, M \mid M\, N \\
\text{contexts} & \Sigma, \Gamma, \Delta & \in \mathcal{G} & ::= \varnothing \mid \Gamma, x : A \mid \Gamma, R \\
\text{rewrite rules} & R & & ::= \varnothing \mid R, M \longmapsto N
\end{array}$$

We write $\Gamma, x : A \coloneqq M$ as a shortcut for $\Gamma, x : A, x \longmapsto M$.

The notion of rewriting traditionally relies on the notion of *constants* and *variables*. Since constants are just represented as variables in the $\lambda\Pi$ calculus, the notion of rewriting needs to be adapted. We differentiate between constants and variables by saying that constants are variables that appear in the context.

[1] http://dedukti.gforge.inria.fr/

Definition 3.1.2. A variable x is free in Γ if $x \notin \mathrm{dom}\,(\Gamma)$.

Definition 3.1.3 (Rewriting)**.** A context Γ induces a reduction relation $\longrightarrow_\Gamma$ defined as the smallest relation that is closed by subterms such that if $M \longmapsto N \in \Gamma$ then for all substitution σ of variables free in Γ, $\sigma\,(M) \longrightarrow_\Gamma \sigma\,(N)$.

Definition 3.1.4 (Typing rules)**.** The typing relations

- $\Gamma \vdash M : A$, meaning that the term M has *type* A in the context Γ,
- $\Gamma \vdash \mathrm{WF}$, meaning that the context Γ is *well-formed*,

are derived by the rules in Figure 3.1. A term A is a *well-formed type* in Γ when $\Gamma \vdash \mathrm{WF}$ and either $\Gamma \vdash A : s$ or $A = s$ for some $s \in \mathcal{S}$. We write this as $\Gamma \vdash A\ \mathrm{WF}$. It is *inhabited* in Γ when $\exists M, \Gamma \vdash M : A$. We write $\Gamma \vdash M : A : B$ as a shortcut for $\Gamma \vdash M : A$ and $\Gamma \vdash A : B$. We sometimes write $\vdash_{\lambda\Pi R}$ instead of $\vdash$ to disambiguate.

Notice that the equivalence relation in the rule Conv is $\equiv_{\beta\Gamma}$ instead of $\equiv_\beta$, and that there is a new rule Rew for checking rewrite rules. The derivation rules for the judgement $\Gamma \vdash R\ \mathrm{WF}$ are not shown here and will be explained in the next section. The following illustrative example shows a small derivation in $\lambda\Pi R$ that uses conversion.

Example 3.1.5. Let Γ be the context containing the following type a, constant c of type a, predicate function f, and rewrite rule that defines f on the constant c:

$$\begin{aligned} a &: \mathsf{Type}, \\ c &: a, \\ f &: a \to \mathsf{Type}, \\ f\,c &\longmapsto \Pi y : a\,.\,f\,y \to f\,y\,. \end{aligned}$$

Then the term $M = \lambda x : f\,c\,.\,x\,c\,x$ is well-typed in Γ:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : f\,c \vdash x : f\,c}}{\Gamma, x : f\,c \vdash x : \Pi y : a\,.\,f\,y \to f\,y}\ \textsc{Conv} \qquad \dfrac{}{\Gamma, x : f\,c \vdash c : a}}{\Gamma, x : f\,c \vdash x\,c : f\,c \to f\,c} \qquad \dfrac{}{\Gamma, x : f\,c \vdash x : f\,c}}{\Gamma, x : f\,c \vdash x\,c\,x : f\,c}}{\Gamma \vdash M : f\,c \to f\,c}$$

Note that the rewrite system is terminating but that the term M would not be well-typed without the rewrite rule, <u>even if we replace all the occurrences of $f\,c$ by $\Pi y : a\,.\,f\,y \to f\,y$</u>. This example shows that rewriting is a non-trivial feature that cannot be thought of as just syntactic sugar.

$$\frac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash R\ \mathrm{WF}}{\Gamma, R \vdash \mathrm{WF}}\ \textsc{Rew}$$

$$\frac{\Gamma \vdash \mathrm{WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{Type}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A \,.\, B : s}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash M : B}\ \textsc{Conv}$$

Figure 3.1 – Typing rules of the system $\lambda\Pi R$

3.2 Basic properties

In general, the system is not well-behaved because of the arbitrary rewrite system $\longrightarrow_\Gamma$. Important properties such as subject reduction and type uniqueness do not hold in general and type checking is undecidable. However, as long as the rewrite system verifies some conditions, these properties hold and the system is well-behaved. The $\Gamma \vdash R\ \mathrm{WF}$ judgement ensures that the rewrite rules verify these conditions, so that the system is well-behaved. We devote the next sections to these conditions. We give the statements without proof, referring the reader to Saillard's PhD thesis [Sai15] for an exhaustive study of the metatheory.

Lemma 3.2.1 (Free variables)**.** *If* $\Gamma \vdash M : A$ *then* $\mathrm{FV}(M) \cup \mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$*. If* $\Gamma, x : A, \Gamma \vdash \mathrm{WF}$ *then* $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$ *and* $x \notin \mathrm{dom}(\Gamma)$*.*

Lemma 3.2.2 (Substitution)**.** *If* $\Gamma, x : A, \Gamma' \vdash M : B$ *and* $\Gamma \vdash N : A$ *then* $\Gamma, \Gamma' \{x \backslash N\} \vdash M \{x \backslash N\} : B \{x \backslash N\}$*.*

Lemma 3.2.3 (Weakening)**.** *If* $\Gamma \vdash M : A$ *and* $\Gamma' \vdash \mathrm{WF}$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash M : A$*.*

3.2.1 Soundness properties

The soundness of the system consists of two key properties: subject reduction and uniqueness of types. These properties do not always hold. They follow from two important conditions: *product compatibility* and *well-typedness of rules*.

Definition 3.2.4 (Product compatibility)**.** A congruence relation is *product compatible* when for all A_1, A_2, B_2, B_2,

$$\Pi x : A_1 \,.\, B_1 \equiv \Pi x : A_2 \,.\, B_2 \implies (A_1 \equiv A_2 \wedge B_1 \equiv B_2)\,.$$

A context Γ is product compatible when $\equiv_{\beta\Gamma}$ is product compatible. Product compatibility follows trivially from the confluence of $\longrightarrow_{\beta\Gamma}$. However, not all systems considered in practice satisfy confluence.

Definition 3.2.5 (Rule typing)**.** A rewrite rule $M \longmapsto N$ *is well-typed* in Γ if for any substitution σ of variables free in Γ,

$$\Gamma \vdash \sigma(M) : A \implies \Gamma \vdash \sigma(N) : A$$

Once these two conditions are met, the system is sound. In general, these properties are not decidable. There are some sufficient conditions that can guarantee them. Dedukti checks for some of these conditions, but does not check confluence for example. A sufficient but not necessary condition for a rule to be well typed will be given in the next section when we discuss the notion of *pattern* in Section 3.2.2.

In the following, let Γ be a product compatible context with well-typed rewrite rules.

Lemma 3.2.6 (Inversion)**.** *The following holds:*

$$\begin{array}{lll}
\Gamma \vdash x : C & \Rightarrow \exists A. & \Gamma \vdash \mathrm{WF} \wedge (x : A) \in \Gamma \wedge C \equiv A \\
\Gamma \vdash s_1 : C & \Rightarrow & \Gamma \vdash \mathrm{WF} \wedge s_1 = \mathsf{Type} \wedge C \equiv \mathsf{Kind} \\
\Gamma \vdash \Pi x : A \,.\, B : C & \Rightarrow \exists s. & \Gamma \vdash A : \mathsf{Type} \wedge \Gamma, x : A \vdash B : s \wedge C \equiv s \\
\Gamma \vdash \lambda x : A \,.\, M : C & \Rightarrow \exists B, s. & \Gamma, x : A \vdash M : B \wedge \Gamma \vdash \Pi x : A \,.\, B : s \wedge C \equiv \Pi x : A \,.\, B \\
\Gamma \vdash M\, N : C & \Rightarrow \exists A, B. & \Gamma \vdash M : \Pi x : A \,.\, B \wedge \Gamma \vdash N : A \wedge C \equiv B\,\{x \backslash N\}
\end{array}$$

Lemma 3.2.7 (Correctness of typing)**.** *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A$ WF*.*

Lemma 3.2.8 (Stratification)**.** *If* $\Gamma \vdash M : A$ *then either*

- $\Gamma \vdash A : \mathsf{Type}$*, in which case we say that* M *is an* object*,*
- $\Gamma \vdash A : \mathsf{Kind}$*, in which case we say that* M *is a* type*,*
- $A = \mathsf{Kind}$*, in which case we say that* M *is a* kind*.*

We try to use the letters M, N *for objects,* A, B *for types, and* K *for kinds. If* $\Gamma \vdash K : \mathsf{Kind}$ *then either* $K = \mathsf{Type}$ *or* $K = \Pi x : A \,.\, K'$ *for some type* A *and kind* K'*.*

Theorem 3.2.9 (Subject reduction)**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow^*_{\beta\Gamma} M'$ *then* $\Gamma \vdash M' : A$*.*

Theorem 3.2.10 (Type uniqueness)**.** *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *then* $A \equiv_{\beta\Gamma} B$*.*

Corollary 3.2.11. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M' : A'$ *and* $M \equiv_{\beta\Gamma} M'$ *then* $A \equiv_{\beta\Gamma} A'$*.*

Proof. By confluence $\exists M''$ such that $M \longrightarrow^* M''$ and $M' \longrightarrow^* M''$. By subject reduction, $\Gamma \vdash M'' : A$ and $\Gamma \vdash M'' : A'$. By uniqueness of types, $A \equiv A'$. □

3.2.2 Completeness properties

In $\lambda\Pi$, the problem of type checking is decidable because $\longrightarrow_\beta$ is confluent and strongly normalizing for well-typed terms. In $\lambda\Pi R$, the problem is generally undecidable because of the arbitrary rewrite rules, which can break either confluence or strong normalization. We focus now on confluence. We will discuss normalization later when we prove conservativity in Chapter 6.

Because we are in a λ-calculus setting, the usual notion of confluence of $\longrightarrow_{\beta\Gamma}$ is not sufficient. We need to switch to *higher-order rewriting* (HOR) [Nip91, MN98], i.e. rewrite rules that allow function variables to be applied on the left-hand side. This is especially crucial for $\lambda\Pi R$ as a logical framework because of the use of HOAS. However, the notion of rewriting in HOR needs to be adapted because $\longrightarrow_{\beta\Gamma}$ can easily become non-confluent, as shown by the following example.

Example 3.2.12 (Non-confluence of traditional rewriting in the presence of applied function variables)**.** Consider the rewrite rule

$$\mathsf{D}\,(\lambda x\,.\,\mathsf{exp}\,(f\,x)) \longmapsto \lambda x\,.\,\mathsf{D}\,f\,x \times \mathsf{exp}\,(f\,x)$$

for function derivation. Then $\mathsf{D}\,(\lambda x\,.\,\mathsf{exp}\,((\lambda y\,.\,y)\;x))$ is a critical pair for $\longrightarrow_{\beta\Gamma}$ since

$$\mathsf{D}\,(\lambda x\,.\,\mathsf{exp}\,x) \longleftarrow_{\beta} \mathsf{D}\,(\lambda x\,.\,\mathsf{exp}\,((\lambda y\,.\,y)\;x)) \longrightarrow_{\Gamma} \lambda x\,.\,\mathsf{D}\,(\lambda y\,.\,y)\;x \times \mathsf{exp}\,((\lambda y\,.\,y)\;x)$$

and it is not joinable.

The common solution is to consider *rewriting modulo β,* written $\longrightarrow_{\underline{\Gamma}}$: for a rewrite rule $L \longmapsto R$, we have $M \longrightarrow_{\underline{\Gamma}} N$ if there is a substitution σ such that $M \equiv_{\beta} L\sigma$ (instead of $M = L\sigma$) and $N = R\sigma$. This is justified by the key observation that

$$M \equiv_{\beta\underline{\Gamma}} M' \iff M \equiv_{\beta\Gamma} M',$$

which shows that $\longrightarrow_{\beta\underline{\Gamma}}$ is suitable for checking $\equiv_{\beta\Gamma}$. However, higher-order matching modulo β is undecidable in general. Therefore, the rewrite rules are restricted to a class of *patterns* for which matching modulo β is decidable. A standard class of patterns that satisfies this property was introduced by Miller [Mil91].

Definition 3.2.13 (Pattern)**.** A *pattern* in Γ is a term $P = c\,\vec{N}$ where $c \in \mathrm{dom}\,(\Gamma)$ and, for any variable x that is free in Γ, if x appears in P then it only appears in the form $x\,\vec{y}$ where $\vec{y}$ is a list of distinct variables that are bound in P. A rule is called a *pattern rule* in Γ if its left-hand side is a pattern in Γ.

Example 3.2.14. In the context

$$\begin{aligned}
\mathsf{Real} &: \mathsf{Type},\\
\mathsf{exp} &: \mathsf{Real} \to \mathsf{Real},\\
\mathsf{D} &: (\mathsf{Real} \to \mathsf{Real}) \to (\mathsf{Real} \to \mathsf{Real}),
\end{aligned}$$

the term $\mathsf{D}\,(\lambda x\,.\,\mathsf{exp}\,(f\,x))$ is a pattern but the term $\mathsf{D}\,(\lambda x\,.\,g\,(f\,x))$ is not (because g is a free variable that is not applied to a list of distinct bound variables).

Patterns are useful because they provide a good sufficient condition for a rule to be well-typed, as well as guarantee the decidability of type-checking.

Theorem 3.2.15 (Sufficient condition for well-typedness of rules)**.** *A rule $M \longmapsto N$ is well-typed in Γ if M is a pattern, and $\exists \Delta, A$ such that* $\mathrm{dom}\,(\Delta) \subseteq \mathrm{FV}\,(M)$ *and*

$$\Gamma, \Delta \vdash M : A \;\wedge\; \Gamma, \Delta \vdash N : A.$$

Theorem 3.2.16 (Decidability of type-checking)**.** *If all the rules of Γ are pattern rules and $\longrightarrow_{\beta\underline{\Gamma}}$ is confluent and strongly normalizing, then type-checking is decidable.*

This algorithm is implemented in Dedukti. Checking the confluence and strong normalization of $\longrightarrow_{\beta\underline{\Gamma}}$ is undecidable. Therefore, these properties are not checked by Dedukti. As a result, if these conditions fail, the algorithm will be incomplete. Note that, as long as product compatibility holds, this would not compromise the soundness of the algorithm: non-confluence can result in the algorithm giving false negatives and non-normalization can result in the algorithm not terminating.

To summarize, the role of the judgement $\Gamma \vdash R$ WF is to ensure that the rewrite rules are well-behaved. For soundness, this means:

1. every rule $M \longmapsto N \in R$ is well-typed in Γ.

2. $\equiv_{\beta\Gamma R}$ is product compatible,

For completeness, this means:

3. every rule $M \longmapsto N \in R$ is a pattern rule,

4. $\longrightarrow_{\beta\underline{\Gamma R}}$ is confluent,

5. $\longrightarrow_{\beta\underline{\Gamma R}}$ is strongly normalizing for well-typed terms.

Note that points 4 implies point 2. Not all of these properties are checked by Dedukti. In the rest of this thesis, we will not focus on precise derivation rules for the judgement $\Gamma \vdash R$ WF, and instead discuss these properties on a case by case basis.

Example 3.2.17 (Programming in $\lambda\Pi R$)**.** We define natural numbers and addition as:

$$\begin{aligned}
&\mathsf{nat} \ : \mathsf{Type},\\
&\mathsf{zero} : \mathsf{nat},\\
&\mathsf{succ} : \mathsf{nat} \to \mathsf{nat},\\
&\\
&\mathsf{plus} \ : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat},\\
&\mathsf{plus}\,\mathsf{zero}\,j \qquad \longmapsto j,\\
&\mathsf{plus}\,(\mathsf{succ}\,i)\ j \longmapsto \mathsf{succ}\,(\mathsf{plus}\,i\,j)\,.
\end{aligned}$$

We can make plus symmetric in its arguments by adding the following rewrite rules:

$$\begin{aligned}
&\mathsf{plus}\,i\,\mathsf{zero} \qquad \longmapsto i,\\
&\mathsf{plus}\,i\,(\mathsf{succ}\,j) \longmapsto \mathsf{succ}\,(\mathsf{plus}\,i\,j)\,.
\end{aligned}$$

Note that the rewrite system is still confluent and strongly normalizing. We define vectors

of length n and the append function as:

$$
\begin{aligned}
&\mathsf{vec} && : \Pi n : \mathsf{nat} \,.\, \mathsf{Type}, \\
&\mathsf{nil} && : \mathsf{vec}\,\mathsf{zero}, \\
&\mathsf{cons} && : \Pi n : \mathsf{nat} \,.\, \mathsf{nat} \to \mathsf{vec}\,n \to \mathsf{vec}\,(\mathsf{succ}\,n)\,.
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{append} : \Pi m, n : \mathsf{nat} \,.\, \mathsf{vec}\,m \to \mathsf{vec}\,n \to \mathsf{vec}\,(\mathsf{plus}\,m\,n)\,, \\
&\mathsf{append}\,\mathsf{zero}\,n\,\mathsf{nil}\,v && \longmapsto v, \\
&\mathsf{append}\,(\mathsf{succ}\,m)\;n\,(\mathsf{cons}\,m\,x\,u)\;v && \longmapsto \mathsf{cons}\,(\mathsf{plus}\,m\,n)\;x\,(\mathsf{append}\,m\,n\,u\,v)\,.
\end{aligned}
$$

The rewrite rules are well-typed thanks to the rewrite rules of plus. For the first rule, with the context $\Delta = n : \mathsf{nat}, v : \mathsf{vec}\,n$, the left-hand side has type $\mathsf{vec}\,(\mathsf{plus}\,\mathsf{zero}\,n)$ while the right-hand side has type $\mathsf{vec}\,n$, and the two types are convertible. For the second rule, with the context $\Delta = m : \mathsf{nat}, n : \mathsf{nat}, x : \mathsf{nat}, u : \mathsf{vec}\,m, v : \mathsf{vec}\,n$, the left-hand side has type $\mathsf{plus}\,(\mathsf{succ}\,m)\;n$ while the right-hand side has type $\mathsf{succ}\,(\mathsf{plus}\,m\,n)$, and the two types are convertible. Again, we can make the definition of append symmetric in its arguments by adding the rewrite rules:

$$
\begin{aligned}
&\mathsf{append}\,m\,\mathsf{zero}\,u\,\mathsf{nil} && \longmapsto u, \\
&\mathsf{append}\,m\,(\mathsf{succ}\,n)\;u\,(\mathsf{cons}\,n\,x\,v) && \longmapsto \mathsf{cons}\,(\mathsf{plus}\,m\,n)\;x\,(\mathsf{append}\,m\,n\,u\,v)\,.
\end{aligned}
$$

and these rules are also well-typed because of the similar rewrite rules on plus. We can add add a rewrite rule for the associativity of append:

$$\mathsf{append}\,(\mathsf{plus}\,m\,n)\;k\,(\mathsf{append}\,m\,n\,u\,v)\;w \longmapsto \mathsf{append}\,m\,(\mathsf{plus}\,n\,k)\;u\,(\mathsf{append}\,n\,k\,v\,w)$$

but it requires a similar rule on plus:

$$\mathsf{plus}\,(\mathsf{plus}\,m\,n)\;k \longmapsto \mathsf{plus}\,m\,(\mathsf{plus}\,n\,k)$$

for it to be well-typed.

3.3 Computational higher-order embeddings

In this section, we show how $\lambda\Pi R$ allows higher-order computational embeddings. Recall the higher-order embedding of system F from Section 2.3:

$$\begin{aligned}
&\mathsf{type} && : \mathsf{Type},\\
&\mathsf{arrow} && : \mathsf{type} \to \mathsf{type} \to \mathsf{type},\\
&\mathsf{forall} && : (\mathsf{type} \to \mathsf{type}) \to \mathsf{type},\\
&\\
&\mathsf{term} && : \mathsf{type} \to \mathsf{Type},\\
&\mathsf{lam} && : \Pi a, b : \mathsf{type} \,.\, (\mathsf{term}\, a \to \mathsf{term}\, b) \to \mathsf{term}\,(\mathsf{arrow}\, a\, b)\,,\\
&\mathsf{app} && : \Pi a, b : \mathsf{type} \,.\, \mathsf{term}\,(\mathsf{arrow}\, a\, b) \to \mathsf{term}\, a \to \mathsf{term}\, b,\\
&\mathsf{biglam} && : \Pi p : (\mathsf{type} \to \mathsf{type}) \,.\, (\Pi x : \mathsf{type} \,.\, \mathsf{term}\,(p\, x)) \to \mathsf{term}\,(\mathsf{forall}\, p)\,,\\
&\mathsf{bigapp} && : \Pi p : (\mathsf{type} \to \mathsf{type}) \,.\, \mathsf{term}\,(\mathsf{forall}\, p) \to \Pi x : \mathsf{type} \,.\, \mathsf{term}\,(p\, x)\,.
\end{aligned}$$

In order to recover the preservation of reduction, we can very simply add the rewrite rules:

$$\begin{aligned}
\mathsf{app}\, a\, b\, (\mathsf{lam}\, a\, b\, f)\ x &\longmapsto f\, x\\
\mathsf{bigapp}\, p\, (\mathsf{biglam}\, p\, f)\ a &\longmapsto f\, a.
\end{aligned}$$

It is easy to see that the translation now preserves reduction:

$$\begin{aligned}
[(\lambda x : A \,.\, M)\ N] &= \mathsf{app}\,[A]\ [B]\ (\mathsf{lam}\,[A]\ [B]\ (\lambda x \,.\, [M]))\ [N]\\
&\longrightarrow_\Gamma (\lambda x \,.\, [M])\ [N]\\
&\longrightarrow_\beta [M]\,\{x \backslash [N]\}\\
&= [M\,\{x \backslash N\}]
\end{aligned}$$

$$\begin{aligned}
[(\Lambda\alpha.M)\ A] &= \mathsf{bigapp}\,(\lambda\alpha \,.\, [B])\ (\mathsf{bigapp}\,(\lambda\alpha \,.\, [B])\ (\lambda x \,.\, [M]))\ [A]\\
&\longrightarrow_\Gamma (\lambda x \,.\, [M])\ [A]\\
&\longrightarrow_\beta [M]\,\{x \backslash [A]\}\\
&= [M\,\{x \backslash A\}]\,.
\end{aligned}$$

One step of β-reduction in system F is simulated by one step of Γ-reduction in the translation, followed by an administrative β-reduction that takes care of the substitution.

While this solution works and is enough to obtain a computational embedding, there is another solution that we will consider. The terms of an arrow type $A \to B$ in system F are represented by terms of type $\mathsf{term}\,(\mathsf{arrow}\, A\, B)$ in $\lambda\Pi R$. The constant app takes a term of this type and allows us to view it as a term of the arrow type $\mathsf{term}\, A \to \mathsf{term}\, B$. Similarly, the constant lam takes a function of type $\mathsf{term}\, A \to \mathsf{term}\, B$ and transforms it into a term of type $\mathsf{term}\,(\mathsf{arrow}\, A\, B)$. In fact, these two functions, lam and app, form an isomorphism between the two types:

$$\mathsf{term}\,(\mathsf{arrow}\, A\, B) \rightleftharpoons \mathsf{term}\, A \to \mathsf{term}\, B.$$

This is reflected in the rewrite rule expressing β which can be seen as identifying their composition with the identity:

$$\mathsf{app}\, a\, b\, (\mathsf{lam}\, a\, b\, f) \longmapsto f,$$

while the composition in the other direction actually expresses η-reduction:

$$\mathsf{lam}\, a\, b\, (\lambda x\,.\, \mathsf{app}\, a\, b\, m\, x) \longmapsto m.$$

The embedding we will consider relies on this idea and takes it further, by identifying the two types via a rewrite rule:

$$\mathsf{term}\, (\mathsf{arrow}\, A\, B) \;\longmapsto\; \mathsf{term}\, A \to \mathsf{term}\, B.$$

Now that the two types are equal, not only can we define lam and app as the identity functions

$$\begin{aligned} \mathsf{lam}\, a\, b\, f &\longmapsto f, \\ \mathsf{app}\, a\, b\, f &\longmapsto f, \end{aligned}$$

but we can get rid of them entirely! Indeed, we can now translate functions of type $\mathsf{term}\, (\mathsf{arrow}\, A\, B)$ directly as λ abstractions and applications directly as applications. The embedding becomes:

$$\begin{aligned} &\mathsf{type} \;\; : \mathsf{Type}, \\ &\mathsf{arrow} : \mathsf{type} \to \mathsf{type} \to \mathsf{type}, \\ &\mathsf{forall} \;\; : (\mathsf{type} \to \mathsf{type}) \to \mathsf{type}, \\ \\ &\mathsf{term} \;\; : \mathsf{type} \to \mathsf{Type}, \\ &\mathsf{term}\, (\mathsf{arrow}\, a\, b) \longmapsto \mathsf{term}\, a \to \mathsf{term}\, b \\ &\mathsf{term}\, (\mathsf{forall}\, p) \quad\;\; \longmapsto \Pi a : \mathsf{type}\,.\, \mathsf{term}\, (p\, a)\,. \end{aligned}$$

This context is well-formed, as both members of each rewrite rule are of type type, the first in the context $a : \mathsf{type}, b : \mathsf{type}$ and the second in the context $p : \mathsf{type} \to \mathsf{type}$ (Theorem 3.2.15), and the system is confluent. The translation of terms is now:

$$\begin{aligned} [x] \quad\quad\quad &= x \\ [\lambda x : A\,.\, M] &= \lambda x : [\![A]\!]\,.\, [M] \\ [M\, N] \quad\quad &= [M]\; [N] \\ [\Lambda \alpha . M] \quad &= \lambda \alpha : \mathsf{type}\,.\, [M] \\ [M\, \langle A \rangle] \quad &= [M]\; [A]\,. \end{aligned}$$

Notice the similarity with the direct embedding of first-order logic in Section 2.2. In some sense, this embedding is a unification of the two embeddings of Chapter 2, since if we kept lam, we would have

$$\mathsf{lam}\, [A]\; [B]\; (\lambda x\,.\, [M]) \equiv \lambda x : [\![A]\!]\,.\, [M]\,.$$

Finally, note that in this embedding β-reduction is simulated by β-reduction only:

$$\begin{aligned}
[(\lambda x : A \,.\, M)\ N] &= (\lambda x \,.\, [M])\ [N] \\
&\longrightarrow_\beta [M]\,\{x \backslash [N]\} \\
&= [M\,\{x \backslash N\}]
\end{aligned}$$

$$\begin{aligned}
[(\Lambda \alpha . M)\ A] &= (\lambda x \,.\, [M])\ [A] \\
&\longrightarrow_\beta [M]\,\{x \backslash [A]\} \\
&= [M\,\{x \backslash A\}]\,.
\end{aligned}$$

What are the benefits of this approach? The main advantage compared to the first one is a more compact representation of terms. Indeed, we no longer need to carry around explicit type annotations that are needed for the constructors lam, app, biglam, and bigapp. Because Dedukti is a minimalist system, it has no support for implicit arguments, so these annotations incur a significant cost in the size of the generated terms and the time it takes to check them. In fact, we can show that the size of the terms is quadratically larger without this optimization.

Example 3.3.1 (Quadratic blowup)**.** Consider the type $a^n \to a$ of size n. Its translation with the first approach is

$$\underbrace{\mathsf{arrow}\, a\, (\cdots (\mathsf{arrow}\, a\, a))}_{n}$$

of size $O\,(n)$. Consider the term $\lambda x_1 : a \,.\, \cdots \lambda x_n : a \,.\, x_n$ of size n. Its translation is

$$\underbrace{\mathsf{lam}\, a\, \underbrace{a^{n-1} \to a}_{n} \left(\lambda x_1 \,.\, \cdots \mathsf{lam}\, a\, \underbrace{a}_{1}\, (\lambda x_n \,.\, x_n) \right)}_{n}$$

of size $O\,(n^2)$. Similarly, the translation of the term $f\, x \cdots x$ of size n is also of size $O\,(n^2)$.

With the second approach, the size of the translation is $O\,(n)$ in both cases. We will therefore prefer it in our embeddings. However, there is no such thing as free lunch. Because β-reduction is simulated by β reduction, there is no more clear separation between administrative redexes and essential redexes. As a result, the proof of conservativity becomes much harder. We discuss this further in Chapter 6.

II

Pure type systems

4

Pure type systems

Pure type systems are a general class of typed λ calculi. They share a common syntax and have the same typing rules. Each of those calculi is parameterized by a specification that describes which types are allowed. The idea was first introduced by Berardi [Ber89] and Terlow [Ter89] under the name *generalized type systems* and later simplified, renamed, and popularized by Geuvers, Nederhof, and Barendregt in the early 90s [Bar92, GN91].

One of the reasons for the popularity of pure type systems is that several important systems of typed λ-calculus à la Church can be seen as specific instances. These include the simply typed lambda calculus, the $\lambda\Pi$-calculus, system F, the calculus of constructions, and even higher-order logic. Subtle differences between these systems can be reduced to differences in their specifications. Pure type systems provide a convenient way of describing a wide variety of calculi, allowing the classification and comparison of these systems with respect to one another.

Another reason for their popularity is that they fit very well in the Curry–Howard correspondence. Many logical systems correspond to pure type systems, so that the same classification and comparisons can be applied directly to them as well. For this reason, they are used as a basis for many proof assistants, such as Automath, Coq, and Matita. They also provide a link with intuitionistic type theory, on which Agda is based, which we will later explore in Chapters 8 and 9 when we treat cumulativity.

We now recall the theory of pure type systems. We refer to the work Geuvers and Nederhof [GN91] and Barendregt [Bar92] for a detailed and comprehensive presentation. Our presentation is a slightly adapted and modernized version of the original one. The main differences are that we only allow sort constants, we do not partition variables by sorts, and we use a separate context formation judgement $\Gamma \vdash \mathrm{WF}$ instead of the weakening rule. This presentation is closer to modern systems such as Coq. The original results still hold for this presentation. The original presentation can be found in Appendix B.2.

4.1 Definition

Definition 4.1.1 (Specification)**.** A pure type system (PTS) *specification* is a triple $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

- $\mathcal{S}$ is a set of constants called *sorts,*
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a relation called *axioms,*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a relation called *rules.*

We sometimes write (s_1, s_2) for the rule (s_1, s_2, s_2). The pure type system associated with a specification $\mathcal{P}$ is written $\lambda\mathcal{P}$. In the following, we assume a fixed specification $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

Definition 4.1.2 (Syntax)**.** The syntax of $\lambda\mathcal{P}$ is given by the following grammar:

$$\begin{array}{lll} \text{sorts} & s & \in \mathcal{S} \\ \text{terms} & M, N, A, B & \in \mathcal{T} ::= x \mid s \mid M\,N \mid \lambda x : A\,.\,M \mid \Pi x : A\,.\,B \\ \text{contexts} & \Gamma, \Delta & \in \mathcal{G} ::= \varnothing \mid \Gamma, x : A \end{array}$$

Definition 4.1.3 (Typing)**.** The typing relations

- $\Gamma \vdash M : A$, meaning that the term M has *type* A in the context Γ,
- $\Gamma \vdash \mathrm{WF}$, meaning the context Γ is *well-formed*,

are derived by the rules in Figure 4.1. A term A is a *well-formed type* in Γ when $\Gamma \vdash \mathrm{WF}$ and either $\Gamma \vdash A : s$ or $A = s$ for some $s \in \mathcal{S}$. We write this as $\Gamma \vdash A\ \mathrm{WF}$. It is *inhabited* in Γ when $\exists M, \Gamma \vdash M : A$. We write $\Gamma \vdash M : A : B$ as a shortcut for $\Gamma \vdash M : A$ and $\Gamma \vdash A : B$. We sometimes write $\vdash_{\lambda\mathcal{P}}$ instead of $\vdash$ to disambiguate.

Definition 4.1.4 (Top-sorts)**.** A sort s is a *top-sort* if $\nexists s' \in \mathcal{S} : (s, s') \in \mathcal{A}$. The set of top-sorts is written $\mathcal{S}^\top$.

Remark 4.1.5. It follows that if $\Gamma \vdash A\ \mathrm{WF}$ then either $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$ or $A = s$ for some $s \in \mathcal{S}^\top$, but not both.

$$\frac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A \,.\, B : s_3}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}\ \textsc{Conv}$$

Figure 4.1 – Typing rules of the system $\lambda\mathcal{P}$

Definition 4.1.6 (Functional PTS)**.** A PTS specification is *functional* when $\mathcal{A}$ and $\mathcal{R}$ are functional relations, that is when:

- $((s_1, s_2) \in \mathcal{A} \wedge (s_1, s_2') \in \mathcal{A}) \implies s_2 = s_2'$
- $((s_1, s_2, s_3) \in \mathcal{R} \wedge (s_1, s_2, s_3') \in \mathcal{R}) \implies s_3 = s_3'$

When a specification is functional, the relations $\mathcal{A}$ and $\mathcal{R}$ are partial functions. We write $\mathcal{A}(s_1)$ (resp. $\mathcal{R}(s_1, s_2)$) for the sort s_2 such that $(s_1, s_2) \in \mathcal{A}$ (resp. s_3 such that $(s_1, s_2, s_3) \in \mathcal{R}$) when they exist.

Definition 4.1.7 (Full PTS)**.** A PTS specification is *full* when the relation $\mathcal{R}$ is total, that is when:

$$\forall s_1, s_2 \in \mathcal{S} \times \mathcal{S}, \exists s_3 \in \mathcal{S}, (s_1, s_2, s_3) \in \mathcal{R}$$

When a specification is functional and full, the relation $\mathcal{R}$ is a total function.

Definition 4.1.8 (Complete PTS)**.** A PTS specification is *complete* when it is full and has no top-sorts (i.e. $\mathcal{S}^\top = \varnothing$). When a specification is functional and complete, the relations $\mathcal{A}$ and $\mathcal{R}$ are total functions.

4.2 Examples of pure type systems

Many systems of typed λ calculus à la Church can be described as pure type systems. Historically, the sorts of these systems have been given many names, such as $\mathsf{Type}, \mathsf{Kind}$ in $\lambda\Pi$-calculus, $\mathsf{Prop}, \mathsf{Type}$ in Coq, and $*, \Box$ in the literature on pure type systems [Bar92, GN91]. For conciseness and to avoid confusion, we will use the symbolic notation $*, \Box$ in these examples.

Example 4.2.1.

- The **simply typed λ-calculus** corresponds to the PTS $\lambda\!\!\rightarrow$ given by the specification:

$$(\rightarrow) \begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *) \end{bmatrix}$$

- The **$\lambda\Pi$-calculus** corresponds to the PTS λP given by the specification:

$$(P) \begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *), (*, \Box) \end{bmatrix}$$

  The rule $(*, \Box)$ allows *dependent types*, i.e. types that depend on terms, e.g. the type of vectors

$$\mathsf{Vec} : \mathbb{N} \rightarrow *.$$

- **System F** corresponds to the PTS $\lambda 2$ given by the specification:

$$(2)\begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *), (\Box, *) \end{bmatrix}$$

  The rule $(\Box, *)$ allows *polymorphism*, i.e. terms that depend on types, e.g. the polymorphic identity function

$$\lambda\alpha : * . \lambda x : \alpha . x : \Pi\alpha : * . \alpha \to \alpha.$$

- **System $\mathbf{F}_\omega$** corresponds to the PTS $\lambda\omega$ given by the specification:

$$(\omega)\begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *), (\Box, *), (\Box, \Box) \end{bmatrix}$$

  The rule $(\Box, \Box)$ allows *type operators*, i.e. types that depend on types, e.g. the type of lists

$$\mathsf{list} : \Pi\alpha : * . * .$$

- The **calculus of constructions** corresponds to the PTS λC given by the specification:

$$(C)\begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *, *), (*, \Box), (\Box, *), (\Box, \Box) \end{bmatrix}$$

  It is the union of the all the previous systems. It is a full PTS.

- **Minimal intuitionistic higher-order logic** can be expressed as the PTS λHOL given by the specification:

$$(HOL)\begin{bmatrix} \mathcal{S} = *, \Box, \triangle \\ \mathcal{A} = (*, \Box), (\Box, \triangle) \\ \mathcal{R} = (*, *), (\Box, *), (\Box, \Box) \end{bmatrix}$$

  The rule $(*, *)$ corresponds to *implication* $(\Rightarrow)$, the rule $(\Box, *)$ corresponds to *universal quantification* $(\forall)$, and the rule $(\Box, \Box)$ corresponds to the *functional arrow* of simple types $(\to)$.

- The **universal pure type system** $\lambda *$ is given by the specification:

$$(*)\begin{bmatrix} \mathcal{S} = * \\ \mathcal{A} = (*, *) \\ \mathcal{R} = (*, *) \end{bmatrix}$$

It is universal in the sense that there is a PTS *morphism* (see Section 4.4.2) from any other PTS to it. This PTS is not strongly normalizing and all its types are inhabited [Bar92]. It is therefore inconsistent when viewed as a logic.

- Girard's **system U** can be expressed as the PTS λU given by the specification:

$$(U)\begin{bmatrix} \mathcal{S} = *, \Box, \triangle \\ \mathcal{A} = (*, \Box), (\Box, \triangle) \\ \mathcal{R} = (*, *), (\Box, *), (\Box, \Box), (\triangle, *), (\triangle, \Box) \end{bmatrix}$$

  This system can be viewed as an extension of higher-order logic with polymorphic types. Historically, Girard [Gir72] proved that this pure type system is inconsistent and used it to show that $\lambda *$ is inconsistent.

- Coquand's **system U$^-$** can be expressed as the PTS λU^- given by the specification:

$$(U^-)\begin{bmatrix} \mathcal{S} = *, \Box, \triangle \\ \mathcal{A} = (*, \Box), (\Box, \triangle) \\ \mathcal{R} = (*, *), (\Box, *), (\Box, \Box), (\triangle, \Box) \end{bmatrix}$$

  It is a subsystem of system U. Coquand [Coq91] showed that this system is also inconsistent. Hurkens [Hur95] provides another proof.

4.3 Basic properties

We state the following important properties without proof. Unless otherwise stated, the complete proofs can be found in the works of Geuvers and Nederhof [GN91] and Barendregt [Bar92].

Theorem 4.3.1 (Confluence)**.** *If $M \equiv_\beta M'$ then $\exists M''$ such that $M \longrightarrow_\beta^* M''$ and $M' \longrightarrow_\beta^* M''$.*

Lemma 4.3.2 (Free variables)**.** *If $\Gamma \vdash M : A$ then $\mathrm{FV}(M) \cup \mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$. If $\Gamma, x : A, \Gamma \vdash \mathrm{WF}$ then $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$ and $x \notin \mathrm{dom}(\Gamma)$.*

Lemma 4.3.3 (Substitution)**.** *If $\Gamma, x : A, \Gamma' \vdash M : B$ and $\Gamma \vdash N : A$ then $\Gamma, \Gamma'\{x \backslash N\} \vdash M\{x \backslash N\} : B\{x \backslash N\}$.*

Lemma 4.3.4 (Weakening)**.** *If $\Gamma \vdash M : A$ and $\Gamma' \vdash \mathrm{WF}$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash M : A$.*

Lemma 4.3.5 (Inversion)**.** *he following holds:*

$$\begin{array}{lll}
\Gamma \vdash x : C & \Rightarrow \exists A. & \Gamma \vdash \mathrm{WF} \wedge (x : A) \in \Gamma \wedge C \equiv A \\
\Gamma \vdash s_1 : C & \Rightarrow \exists s_2. & \Gamma \vdash \mathrm{WF} \wedge (s_1, s_2) \in \mathcal{A} \wedge C \equiv s_2 \\
\Gamma \vdash \Pi x : A . B : C & \Rightarrow \exists s_1, s_2, s_3. & \Gamma \vdash A : s_1 \wedge \Gamma, x : A \vdash B : s_2 \wedge (s_1, s_2, s_3) \in \mathcal{R} \wedge C \equiv s_3 \\
\Gamma \vdash \lambda x : A . M : C & \Rightarrow \exists B, s. & \Gamma, x : A \vdash M : B \wedge \Gamma \vdash \Pi x : A . B : s \wedge C \equiv \Pi x : A . B \\
\Gamma \vdash M\, N : C & \Rightarrow \exists A, B. & \Gamma \vdash M : \Pi x : A . B \wedge \Gamma \vdash N : A \wedge C \equiv B\{x \backslash N\}
\end{array}$$

Lemma 4.3.6 (Correctness of typing)**.** *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A$ WF.

Theorem 4.3.7 (Subject reduction)**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_\beta^* M'$ *then* $\Gamma \vdash M' : A$.

Theorem 4.3.8 (Strengthening [Jut93])**.** *If* $\Gamma, x : A, \Gamma' \vdash M : B$ *and* $x \notin \mathrm{FV}(\Gamma') \cup \mathrm{FV}(M) \cup \mathrm{FV}(B)$ *then* $\Gamma, \Gamma' \vdash M : B$.

When the pure type system is functional, terms have a unique type up to β-equivalence.

Theorem 4.3.9 (Uniqueness of types)**.** *Let* $\mathcal{P}$ *be a functional specification. If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *then* $A \equiv_\beta B$.

Corollary 4.3.10. *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M' : A'$ *and* $M \equiv_\beta M'$ *then* $A \equiv_\beta A'$.

Proof. By confluence $\exists M''$ such that $M \longrightarrow^* M''$ and $M' \longrightarrow^* M''$. By subject reduction, $\Gamma \vdash M'' : A$ and $\Gamma \vdash M'' : A'$. By uniqueness of types, $A \equiv_\beta A'$. □

4.4 Inter-system properties

4.4.1 Subsystems

Definition 4.4.1 (Subsystem)**.** A PTS $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is a *subsystem* of $\mathcal{P}' = (\mathcal{S}', \mathcal{A}', \mathcal{R}')$ when $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{A} \subseteq \mathcal{A}'$, $\mathcal{R} \subseteq \mathcal{R}'$, and $\mathcal{C} \subseteq \mathcal{C}'$. We write $\mathcal{P} \subseteq \mathcal{P}'$.

Lemma 4.4.2. *If* $\mathcal{P} \subseteq \mathcal{P}'$ *then* $\Gamma \vdash_{\lambda\mathcal{P}} M : A \implies \Gamma \vdash_{\lambda\mathcal{P}'} M : A$.

Lemma 4.4.3 (Compactness)**.** *If* $\Gamma \vdash_{\lambda P} M : A$ *then there is a finite sub-system* $\mathcal{P}' \subseteq \mathcal{P}$ *such that* $\Gamma \vdash_{\lambda\mathcal{P}'} M : A$.

4.4.2 Morphisms

Definition 4.4.4 (Morphism)**.** Let $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ and $\mathcal{P}' = (\mathcal{S}', \mathcal{A}', \mathcal{R}')$ be two PTSs. A function $\varphi : \mathcal{S} \to \mathcal{S}'$ is a *PTS morphism* when:

1. $\forall (s_1, s_2) \in \mathcal{A}, (\varphi(s_1), \varphi(s_2)) \in \mathcal{A}'$,
2. $\forall (s_1, s_2, s_3) \in \mathcal{R}, (\varphi(s_1), \varphi(s_2)) \in \mathcal{R}'$,

The function φ can be extended to all terms and contexts of $\mathcal{P}$ as follows:

$$\begin{aligned}
\varphi(x) &= x \\
\varphi(M\,N) &= \varphi(M)\ \varphi(N) \\
\varphi(\lambda x : A\,.\,M) &= \lambda x : \varphi(A)\,.\,\varphi(N) \\
\varphi(\Pi x : A\,.\,B) &= \Pi x : \varphi(A)\,.\,\varphi(B)
\end{aligned}$$

We write $\varphi : \mathcal{P} \to \mathcal{P}'$ to say that φ is a morphism between $\mathcal{P}$ and $\mathcal{P}'$.

Lemma 4.4.5. *If $\mathcal{P} \subseteq \mathcal{P}'$ then the identity function is a morphism from $\mathcal{P}$ to $\mathcal{P}'$, i.e.* $\mathrm{id} : \mathcal{P} \to \mathcal{P}'$.

Morphisms preserve β-reductions, β-equivalence, and typing.

Lemma 4.4.6. *If $\varphi : \mathcal{P} \to \mathcal{P}'$ then*

1. $M \longrightarrow_\beta M' \iff \varphi(M) \longrightarrow_\beta \varphi(M')$,
2. $M \equiv_\beta M' \iff \varphi(M) \equiv_\beta \varphi(M')$,
3. $\Gamma \vdash_{\mathcal{P}} M : A \implies \Gamma \vdash_{\mathcal{P}'} \varphi(M) : \varphi(A)$

Corollary 4.4.7. *If $\varphi : \mathcal{P} \to \mathcal{P}'$ and $\mathcal{P}'$ is weakly (resp. strongly) normalizing then $\mathcal{P}$ is weakly (resp. strongly) normalizing.*

5

Embedding pure type systems

In this chapter, we show how to embed pure type systems in the $\lambda\Pi$-calculus modulo rewriting. The original translation and the proof that it preserves typing are due to Cousineau and Dowek [CD07]. The embedding is restricted to functional PTSs, as these satisfy uniqueness of types (Theorem 4.3.9). Indeed, because $\lambda\Pi R$ also satisfies uniqueness of types, it is not clear how to embed non-functional PTSs. However, this is not a real limitation because in practice all useful PTSs are functional.[1] We recall the translation here and reprove the preservation of typing in full detail. Our main contribution on this topic is a proof of the conservativity of the embedding which we will give in the next chapter.

[1] In some systems, such as the calculus of inductive constructions and intuitionistic type theory with universes à la Russell, terms can have more than one type; for example

$$\vdash \mathsf{Type}_i : \mathsf{Type}_{i+j}$$

for all $i, j \in \mathbb{N}$. However, this is *not* due to non-functionality but to another notion called *cumulativity*. We show how to treat cumulativity in Chapter 9.

5.1 Translation

In the following, let $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a fixed functional PTS specification. The main difficulty in the translation is that types in pure type systems can be manipulated as regular objects, for example taken as an argument or passed as an argument to a function, as long as they follow the rules of the specification. However, in the $\lambda\Pi$ and $\lambda\Pi R$, only *objects* (i.e. terms whose type is in Type) can be manipulated by functions.

We use an idea similar to universes *à la Tarski* in intuitionistic type theory. There are two popular ways of presenting universes: the *Russell style* and the *Tarski style* [ML84, Pal98]. In the Russell style, we make no distinction between terms and types, similar to pure type systems:

$$\frac{}{\vdash \mathsf{U}_i \text{ type}} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash A \text{ type}} \qquad \frac{}{\Gamma \vdash \mathsf{U}_i : \mathsf{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i \qquad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \Pi x : A \,.\, B : \mathsf{U}_i}$$

In the Tarski style, we distinguish between types, such as U_i and $\Pi x : A \,.\, B$, and the terms of type U_i, which are *codes* that represent types. There is a code u_i that represents U_i and $\pi x : A.B$ that represents $\Pi x : A \,.\, B$. To decode them, there is an operator T_i that takes an element of type U_i and returns a type.

$$\frac{}{\vdash \mathsf{U}_i \text{ type}} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash \mathsf{T}_i\,(A) \text{ type}} \qquad \frac{}{\Gamma \vdash \mathsf{u}_i : \mathsf{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i \qquad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \pi x : A.B : \mathsf{U}_i}$$

The decoding is implemented by the equations

$$\begin{aligned} \mathsf{T}_{i+1}\,(\mathsf{u}_i) &\equiv \mathsf{U}_i, \\ \mathsf{T}_i\,(\pi x : A.B) &\equiv \Pi x : \mathsf{T}_i\,(A) \,.\, \mathsf{T}_i\,(B)\,. \end{aligned}$$

To represent pure type systems in $\lambda\Pi R$, we therefore declare for every sort s a type U_s and a function T_s that decodes the elements of type U_s as types. A type A in s will have two representations, a *term representation* $[A]$ as an element in U_s, and a *type representation* $[\![A]\!]$. The decoding function T_s makes the link between the two:

$$\mathsf{T}_s\,[A] \equiv [\![A]\!]\,.$$

The embedding is illustrated in Figure 5.1.

Definition 5.1.1. Let $\Sigma_{\mathcal{P}}$ be the signature containing the declarations

$$\begin{array}{lll} \mathsf{U}_s & : \mathsf{Type} & \forall s \in \mathcal{S}, \\ \mathsf{T}_s & : \mathsf{U}_s \to \mathsf{Type} & \forall s \in \mathcal{S}, \\ \mathsf{u}_{s_1} & : \mathsf{U}_{s_2} & \forall\,(s_1, s_2) \in \mathcal{A}, \\ \pi_{s_1,s_2} & : \Pi a : \mathsf{U}_{s_1} \,.\, (\mathsf{T}_{s_1}\, a \to \mathsf{U}_{s_2}) \to \mathsf{U}_{s_3} & \forall\,(s_1, s_2, s_3) \in \mathcal{R}, \end{array}$$

and the rewrite rules

$$\begin{array}{lll} \mathsf{T}_{s_2}\,\mathsf{u}_{s_1} & \longmapsto \mathsf{U}_{s_1} & \forall\,(s_1, s_2) \in \mathcal{A}, \\ \mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\, a\, b) & \longmapsto \Pi x : \mathsf{T}_{s_1}\, a \,.\, \mathsf{T}_{s_2}\,(b\, x) & \forall\,(s_1, s_2, s_3) \in \mathcal{R}. \end{array}$$

We write Σ instead of $\Sigma_{\mathcal{P}}$ when it is not ambiguous. This signature is well-formed.

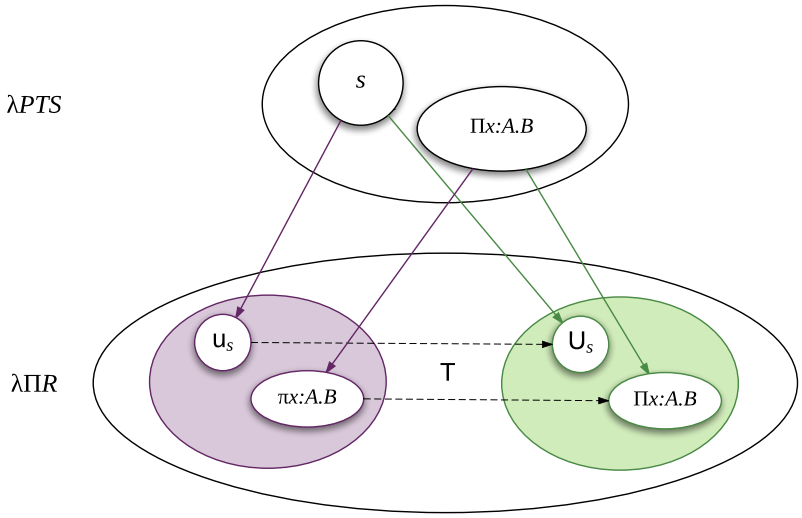

Figure 5.1 – Embedding of a pure type system in the λΠ-calculus modulo rewriting using universes à la Tarski. Each type has two representations: as a *term* (on the left, in purple) and as a *type* (on the right, in green), with the decoding function T making the link between the two.

Lemma 5.1.2 ([CD07])**.** *The relation* $\longrightarrow_\Sigma$ *is terminating.*

Lemma 5.1.3 (Confluence [CD07])**.** *The relation* $\longrightarrow_{\beta\Sigma}$ *is confluent.*

Lemma 5.1.4 (Well-typedness [CD07])**.** *The rewrite rules of* Σ *are well-typed.*

Definition 5.1.5 (Translation)**.** Let $\Gamma \vdash_{\lambda\mathcal{P}} M : A$. The translation of M *as a term* in Γ is defined by induction on M as

$$\begin{array}{lll}
[x]_\Gamma & = x & \\
[s_1]_\Gamma & = \mathsf{u}_{s_1} & \text{where } \Gamma \vdash s_1 : s_2 \\
[M\,N]_\Gamma & = [M]_\Gamma\ [N]_\Gamma & \\
[\lambda x : A\,.\,M]_\Gamma & = \lambda x : \mathsf{T}_{s_1}\,[A]_\Gamma\,.\,[M]_{\Gamma,x:A} & \text{where } \Gamma \vdash A : s_1 \\
[\Pi x : A\,.\,B]_\Gamma & = \pi^{s_3}_{s_1,s_2}\,[A]_\Gamma\,\left(\lambda x\,.\,[B]_{\Gamma,x:A}\right) & \text{where } \Gamma \vdash A : s_1 \\
 & & \text{and } \Gamma, x : A \vdash B : s_2.
\end{array}$$

By inversion (Lemma 4.3.5) and type uniqueness (Theorem 4.3.9), this function is well-defined. Indeed, if $\Gamma \vdash A : s$ and $\Gamma \vdash A : s'$ then $s \equiv s'$ which implies $s = s'$. We write $[M]$ instead of $[M]_\Gamma$ when it is not ambiguous.

Definition 5.1.6 (Type translation)**.** Let $\Gamma \vdash_{\lambda\mathcal{P}} A$ WF. The translation of A *as a type* in Γ is defined as

$$\begin{array}{lll}
[\![A]\!]_\Gamma & = \mathsf{T}_s\,[A]_\Gamma & \text{if } \Gamma \vdash A : s \\
[\![s]\!]_\Gamma & = \mathsf{U}_s & \text{if } s \in \mathcal{S}^\top.
\end{array}$$

By Remark 4.1.5, inversion (Lemma 4.3.5), and type uniqueness (Theorem 4.3.9), this function is well-defined. We write $[\![A]\!]$ instead of $[\![A]\!]_\Gamma$ when the context is not ambiguous.

Definition 5.1.7 (Context translation)**.** Let $\Gamma \vdash_{\lambda\mathcal{P}} \mathrm{WF}$. The translation of Γ is defined as

$$\begin{aligned} [\![\varnothing]\!] &= \varnothing \\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [\![A]\!]_\Gamma . \end{aligned}$$

Remark 5.1.8. If $\Gamma \vdash M : A$ then for any Γ' such that $\Gamma, \Gamma' \vdash \mathrm{WF}$, $[M]_\Gamma = [M]_{\Gamma'}$. Therefore, if $\Gamma \vdash \mathrm{WF}$ and $(x : A) \in \Gamma$ then $(x : [\![A]\!]_\Gamma) \in [\![\Gamma]\!]$.

Example 5.1.9. We show the example of the embedding of the calculus of constructions in full detail. Recall from Section 4.2 that the calculus of constructions is given by the specification

$$(C) \begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *, *), (*, \Box), (\Box, *), (\Box, \Box) \end{bmatrix}.$$

In $\lambda\Pi$, let Σ be the signature containing the declarations

$$\begin{aligned} \mathsf{U}_* &: \mathsf{Type}, \\ \mathsf{U}_\Box &: \mathsf{Type}, \\ \mathsf{T}_* &: \mathsf{U}_* \to \mathsf{Type}, \\ \mathsf{T}_\Box &: \mathsf{U}_\Box \to \mathsf{Type}, \\ \mathsf{u}_* &: \mathsf{U}_\Box, \\ \pi_{*,*} &: \Pi a : \mathsf{U}_* . (\mathsf{T}_* \, a \to \mathsf{U}_*) \to \mathsf{U}_*, \\ \pi_{*,\Box} &: \Pi a : \mathsf{U}_* . (\mathsf{T}_* \, a \to \mathsf{U}_\Box) \to \mathsf{U}_\Box, \\ \pi_{\Box,*} &: \Pi a : \mathsf{U}_\Box . (\mathsf{T}_\Box \, a \to \mathsf{U}_*) \to \mathsf{U}_*, \\ \pi_{\Box,\Box} &: \Pi a : \mathsf{U}_\Box . (\mathsf{T}_\Box \, a \to \mathsf{U}_\Box) \to \mathsf{U}_\Box, \end{aligned}$$

and the rewrite rules

$$\begin{aligned} \mathsf{T}_\Box \, \mathsf{u}_* &\longmapsto \mathsf{U}_*, \\ \mathsf{T}_* \, (\pi_{*,*} \, a \, b) &\longmapsto \Pi x : \mathsf{T}_* \, a . \mathsf{T}_* \, (b \, x), \\ \mathsf{T}_\Box \, (\pi_{*,\Box} \, a \, b) &\longmapsto \Pi x : \mathsf{T}_* \, a . \mathsf{T}_\Box \, (b \, x), \\ \mathsf{T}_* \, (\pi_{\Box,*} \, a \, b) &\longmapsto \Pi x : \mathsf{T}_\Box \, a . \mathsf{T}_* \, (b \, x), \\ \mathsf{T}_\Box \, (\pi_{\Box,\Box} \, a \, b) &\longmapsto \Pi x : \mathsf{T}_\Box \, a . \mathsf{T}_\Box \, (b \, x). \end{aligned}$$

Let Γ be the λC context $a : *, b : *, x : a, f : \Pi p : (a \to *) . p \, x \to b$. Then b is provable in Γ:

$$\Gamma \vdash_{\lambda C} f \, (\lambda y : a . a) \, x : b.$$

Notice that we need *both* higher-order quantification (in the type of p) *and* computation (to convert between $(\lambda y : a . a) \, x$ and a) for this term to be well typed.

The translation of Γ is $a : \mathsf{U}_*, b : \mathsf{U}_*, x : \mathsf{T}_* \, a, f : \Pi p : (\mathsf{T}_* \, a \to \mathsf{U}_*) \, . \, \mathsf{T}_* \, (p \, x) \to \mathsf{T}_* \, b$. The translation of the proof is well-typed in $\Sigma, [\![\Gamma]\!]$:

$$\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} f \, (\lambda y : \mathsf{T}_* \, a \, . \, a) \; x : \mathsf{T}_* \, b.$$

5.2 Completeness

A fundamental propery of the translation is that it preserves computation and typing. The original proof is due to Cousineau and Dowek [CD07]. We show the proof in full detail here.

5.2.1 Preservation of substitution

Because pure type systems have the conversion rule, it is necessary for the translation to also preserve term equivalence. We will see later that this property is crucial for the preservation of types. It follows from the preservation of reduction, which in turn depends on the preservation of substitution.

Lemma 5.2.1 (Preservation of substitution)**.** *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}} M : B$ *and* $\Gamma \vdash_{\lambda\mathcal{P}} N : A$ *then* $[M \, \{x \backslash N\}] = [M] \, \{x \backslash \, [N]\}$*. More precisely,*

$$[M \, \{x \backslash N\}]_{\Gamma, \Gamma' \{x \backslash N\}} \; = \; [M]_{\Gamma, x : A, \Gamma'} \, \{x \backslash \, [N]_\Gamma\} \, .$$

Proof. First note that the statement makes sense because $\Gamma, \Gamma' \, \{x \backslash N\} \vdash M \, \{x \backslash N\} : B \, \{x \backslash N\}$ by Lemma 4.3.3. The proof follows by induction on M. □

Corollary 5.2.2. *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}} B$ WF *and* $\Gamma \vdash_{\lambda\mathcal{P}} N : A$ *then* $[\![B \, \{x \backslash N\}]\!] = [\![B]\!] \, \{x \backslash \, [N]\}$*. More precisely,*

$$[\![B \, \{x \backslash N\}]\!]_{\Gamma, \Gamma' \{x \backslash N\}} = [\![B]\!]_{\Gamma, x : A, \Gamma'} \, \{x \backslash \, [N]_\Gamma\} \, .$$

Proof. If $B = s$ for some $s \in \mathcal{S}^\top$ then $[\![B \, \{x \backslash N\}]\!] = \mathsf{U}_s = [\![B]\!] \, \{x \backslash \, [N]\}$. Otherwise, $\Gamma \vdash B : s$ for some $s \in \mathcal{S}$. Then $[\![B \, \{x \backslash N\}]\!] = \mathsf{T}_s \, [B \, \{x \backslash N\}] = \mathsf{T}_s \, [B] \, \{x \backslash \, [N]\} = [\![B]\!] \, \{x \backslash \, [N]\}$. □

5.2.2 Preservation of equivalence

Lemma 5.2.3 (Preservation of single-step reduction)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *and* $M \longrightarrow_\beta M'$ *then* $[M]_\Gamma \longrightarrow^+_{\beta\Sigma} [M']_\Gamma$*.*

Proof. First note that the statement makes sense because $\Gamma \vdash M' : A$ by subject reduction (Theorem 4.3.7). The proof follows by induction on M, using Lemma 5.2.1 for the base case. □

Lemma 5.2.4 (Preservation of multi-step reduction)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *and* $M \longrightarrow^*_\beta M'$ *then* $[M]_\Gamma \longrightarrow^*_{\beta\Sigma} [M']_\Gamma$*.*

Proof. By induction on the number of steps. □

Lemma 5.2.5 (Preservation of equivalence)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *and* $\Gamma \vdash_{\lambda\mathcal{P}} M' : A'$ *and* $M \equiv_\beta M'$ *then* $[M]_\Gamma \equiv_{\beta\Sigma} [M']_\Gamma$.

Proof. By confluence (Theorem 4.3.1), there is a term M'' such that $M \longrightarrow^* M''$ and $M' \longrightarrow^*_\beta M''$. By Lemma 5.2.4, $[M] \longrightarrow^* [M'']$ and $[M'] \longrightarrow^* [M'']$. Therefore, $[M] \equiv [M']$. □

Corollary 5.2.6. *If* A *and* B *are well-formed types in* Γ *and* $A \equiv_\beta B$ *then* $[\![A]\!]_\Gamma \equiv_{\beta\Sigma} [\![B]\!]_\Gamma$.

Proof. By confluence, subject reduction, and type uniqueness, either $A = s = B$ for some $s \in \mathcal{S}^\top$ or $\Gamma \vdash A : s$ and $\Gamma \vdash B : s$ for some $s \in \mathcal{S}$. In the first case, we have $[\![A]\!] = \mathsf{U}_s = [\![B]\!]$. In the second case, we have $[\![A]\!] = \mathsf{T}_s\,[A] \equiv \mathsf{T}_s\,[B] = [\![B]\!]$. □

5.2.3 Preservation of typing

We start with the following technical lemmas before proving the main theorem.

Lemma 5.2.7. *If* $\Gamma \vdash_{\lambda\mathcal{P}} s$ WF *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![s]\!] : \mathsf{Type} \iff [\![\Gamma]\!] \vdash_{\lambda\Pi R} \mathsf{U}_s : \mathsf{Type}$.

Proof. Follows from correctness of typing and inversion. □

Corollary 5.2.8. *If* $\Gamma \vdash_{\lambda\mathcal{P}} A : s$ *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [A] : [\![s]\!] \iff [\![\Gamma]\!] \vdash_{\lambda\Pi R} [A] : \mathsf{U}_s$.

Proof. If $s \in \mathcal{S}^\top$ then $[\![s]\!] = \mathsf{U}_s$ so the statement is trivially true. Otherwise, we have $[\![s]\!] = \mathsf{T}_{s'}\,\mathsf{u}_s \equiv \mathsf{U}_s$. By Lemma 5.2.7, $[\![\Gamma]\!] \vdash [\![s]\!] : \mathsf{Type} \iff [\![\Gamma]\!] \vdash \mathsf{U}_s : \mathsf{Type}$. By conversion, both directions of the lemma are true. □

Lemma 5.2.9. *If* $\Gamma \vdash_{\lambda\mathcal{P}} \Pi x : A\,.\,B$ WF *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![\Pi x : A\,.\,B]\!] : \mathsf{Type} \iff [\![\Gamma]\!] \vdash_{\lambda\Pi R} \Pi x : [\![A]\!]\,.\,[\![B]\!] : \mathsf{Type}$.

Proof. Note that this statement makes sense because, by correctness of types $\Gamma \vdash \Pi x : A\,.\,B : s_3$ for some s_3 and, by inversion, $\Gamma \vdash A : s_1$ and $\Gamma \vdash B : s_2$ for some s_1, s_2 such that $(s_1, s_2, s_3) \in \mathcal{R}$. The statement follows by correctness of typing and inversion. □

Corollary 5.2.10. *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : \Pi x : A\,.\,B$ *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [M] : [\![\Pi x : A\,.\,B]\!] \iff [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M] : \Pi x : [\![A]\!]\,.\,[\![B]\!]$.

Proof. We have $[\![\Pi x : A\,.\,B]\!] = T_{s_3}\,(\pi_{s_1,s_2}\,[A]\ \lambda x\,.\,[B]) \equiv \Pi x : \mathsf{T}_{s_1}\,[A]\,.\,\mathsf{T}_{s_2}\,[B] = \Pi x : [\![A]\!]\,.\,[\![B]\!]$. By Lemma 5.2.9, we have $[\![\Gamma]\!] \vdash [\![\Pi x : A\,.\,B]\!] : \mathsf{Type} \iff [\![\Gamma]\!] \vdash \Pi x : [\![A]\!]\,.\,[\![B]\!] : \mathsf{Type}$. By conversion, both directions of the lemma are true. □

Theorem 5.2.11 (Preservation of typing)**.** *For all* Γ, M, A, *if* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_\Gamma : [\![A]\!]_\Gamma$. *For all* Γ, *if* $\Gamma \vdash_{\lambda\mathcal{P}}$ WF *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R}$ WF.

Proof. By simultaneous induction on the derivation. The details of the proof can be found in Appendix A.1. □

Remark 5.2.12. We could have defined the translation of sorts and products as

$$\begin{aligned} [\![s]\!] &= \mathsf{U}_s \\ [\![\Pi x : A \,.\, B]\!] &= \Pi x : [\![A]\!] \,.\, [\![B]\!] \end{aligned}$$

The two definitions are equivalent by Lemmas 5.2.8 and 5.2.10. This one has the advantage of producing simpler terms, but would have slightly complicated the metatheory. Now that we have proved the preservation of equivalence and the preservation of typing, we will use these two definitions interchangeably.

5.3 Alternative embeddings

In this section, we present two other embedings of pure type systems in the $\lambda\Pi$-calculus modulo rewriting. The novelty of these embeddings is that they use less rewriting *at the level of types*. A rewrite rule $(M \longmapsto N) \in \Sigma$ is at the level of types when the type of M and N is in Kind, i.e. when M and N are types. Having less rewriting at the level of types simplifies the metatheory, in particular to show soundness (see Chapter 6), but at the same time results in an embedding that is less efficient because of the quadratic blowup caused by the use of lam and app to represent terms (see Example 3.3.1). As such, they are less suited for practical use but we include them here for theoretical interest.

5.3.1 Embedding without Π in the rewriting at the level of types

The embedding is similar to that of Section 5.1 but instead of the rewrite rule

$$\mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\,a\,b) \longmapsto \Pi x : \mathsf{T}_{s_1}\,a \,.\, \mathsf{T}_{s_2}\,(b\,x)$$

we use constructors lam and app to represent terms, as in Section 3.3. To simulate β-reduction, we add the rewrite rule

$$\mathsf{app}\,a\,b\,(\mathsf{lam}\,a\,b\,f)\;x \longmapsto f\,x.$$

The signature becomes:

$$\begin{array}{lll}
\mathsf{U}_s & : \mathsf{Type} & \forall s \in \mathcal{S}, \\
\mathsf{T}_s & : \mathsf{U}_s \to \mathsf{Type} & \forall s \in \mathcal{S}, \\
\mathsf{u}_{s_1} & : \mathsf{U}_{s_2} & \forall\,(s_1, s_2) \in \mathcal{A}, \\
\pi_{s_1,s_2} & : \Pi a : \mathsf{U}_{s_1} \,.\, (\mathsf{T}_{s_1}\,a \to \mathsf{U}_{s_2}) \to \mathsf{U}_{s_3} & \forall\,(s_1, s_2, s_3) \in \mathcal{R}, \\
\mathsf{lam}_{s_1,s_2} & : \Pi a : \mathsf{U}_{s_1} \,.\, \Pi b : (\mathsf{T}_{s_1}\,a \to \mathsf{U}_{s_2}) \,. & \\
 & \qquad (\Pi x : \mathsf{T}_{s_1}\,a \,.\, \mathsf{T}_{s_2}\,(b\,x)) \to \mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\,a\,b) & \forall\,(s_1, s_2, s_3) \in \mathcal{R}, \\
\mathsf{app}_{s_1,s_2} & : \Pi a : \mathsf{U}_{s_1} \,.\, \Pi b : (\mathsf{T}_{s_1}\,a \to \mathsf{U}_{s_2}) \,. & \\
 & \qquad \mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\,a\,b) \to \Pi x : \mathsf{T}_{s_1}\,a \,.\, \mathsf{T}_{s_2}\,(b\,x) & \forall\,(s_1, s_2, s_3) \in \mathcal{R},
\end{array}$$

with the rewrite rules:

$$\begin{array}{lll} \mathsf{T}_{s_2}\,\mathsf{u}_{s_1} & \longmapsto \mathsf{U}_{s_1} & \forall\,(s_1, s_2) \in \mathcal{A}, \\ \mathsf{app}_{s_1,s_2}\,a\,b\,(\mathsf{lam}_{s_1,s_2}\,a\,b\,f)\;x & \longmapsto f\,x & \forall\,(s_1, s_2, s_3) \in \mathcal{R}. \end{array}$$

Notice that only the first rule is at the level of types. We do not give the definition of the translation as it should be straightforward. The main advantage of this embedding is that it does not use Π in the rewrite rules, and as a result automatically satisfies *product compatibility* (Definition 3.2.4), which is an essential property that is usually delicate to prove in practice. Moreover, conservativity is now much easier to prove, because we no longer mix essential redexes with administrative ones: since β-reduction is now simulated by $\longrightarrow_\Gamma$ instead of $\longrightarrow_\beta$, we can easily prove that β-reduction terminates in $\lambda\Pi R$ and show conservativity by relating β-normal forms to terms in $\lambda\mathcal{P}$, in the spirit of the traditional proofs of conservativity in $\lambda\Pi$. See Chapter 6 for a discussion on the interaction between rewriting, normalization, and conservativity.

5.3.2 Embedding without rewriting at the level of types

A natural follow-up question is: is it possible to completely do away with rewriting at the level of types? The answer is: yes, it is possible, but only for *finite acyclic systems*. The idea is to see U_{s_1} as syntactic sugar and replace it by $\mathsf{T}_{s_2}\,\mathsf{u}_{s_1}$ for all $(s_1, s_2) \in \mathcal{A}$ and only really declare u_{s_1} in the signature. Of course, since we need to declare u_{s_1} in U_{s_2}, we encounter a circularity problem. Nevertheless, it is still possible for finite acyclic systems: top-sorts are declared as U_s, and all the other sorts are declared sequentially as u_s, in topological order of the hierarchy from top to bottom. This solution excludes system $\lambda*$ (See Example 4.2.1) and systems with infinite hierarchies (see Chapter 8 and Chapter 10), but is possible for all the other systems of Example 4.2.1 including system F and the calculus of constructions.

This embedding has the additional advantage of having a slightly easier proof of termination of β-reduction than the previous one because there is no rewriting *at the level of types*. Note however that rewriting *inside types*, in particular in the conversion rule, is still necessary for the completeness of the translation.

6

Conservativity

Conservativity is the converse of the preservation of typing. It states that if a proposition is provable in the embedding then it is also provable in the original logic: if there is a term N such that $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} N : [\![A]\!]$ then there is a term M such that $\Gamma \vdash_{\lambda\mathcal{P}} M : A$. It is an important property for embeddings in a logical framework. Indeed, we know for example that the calculus of constructions or simple type theory are consistent, but how can we guarantee that we cannot prove $\bot$ in their embedding? We further motivate this question with the following example.

Example 6.0.1. Recall from Section 4.2 the PTS λHOL that corresponds to higher order logic:

$$(HOL) \begin{bmatrix} \mathcal{S} = *, \Box, \triangle \\ \mathcal{A} = (*, \Box), (\Box, \triangle) \\ \mathcal{R} = (*, *), (\Box, *), (\Box, \Box) \end{bmatrix}$$

This PTS is strongly normalizing, and therefore consistent. The PTS λU^- is a polymorphic extension of λHOL specified by $U^- = HOL + (\triangle, \Box, \Box)$. It turns out that λU^- is inconsistent: there is a term Ω such that $\varnothing \vdash_{\lambda U^-} \Omega : \Pi\alpha : *\,.\,\alpha$ and which is not normalizing [Bar92, Coq91, Hur95].

The polymorphic identity function $I = \lambda\alpha : \Box \,.\, \lambda x : \alpha \,.\, x$ is not well-typed in λHOL, but it is well-typed in λU^- and so is its type:

$$\varnothing \vdash_{\lambda U^-} I : \Pi\alpha : \Box \,.\, \alpha \to \alpha,$$

$$\varnothing \vdash_{\lambda U^-} \Pi\alpha : \Box \,.\, \alpha \to \alpha : \mathsf{Type}.$$

However, the translation $[I] = \lambda\alpha : \mathsf{U}_\Box \,.\, \lambda x : \mathsf{T}_\Box\, \alpha \,.\, x$ *is* well-typed in the embedding of λHOL (which we will refer to as $\lambda\Pi R_{HOL}$):

$$\Sigma_{HOL} \vdash_{\lambda\Pi R} [I] : \Pi\alpha : \mathsf{U}_\Box \,.\, \mathsf{T}_\Box\, \alpha \to \mathsf{T}_\Box\, \alpha,$$

$$\Sigma_{HOL} \vdash_{\lambda\Pi R} \Pi\alpha : \mathsf{U}_\Box \,.\, \mathsf{T}_\Box\, \alpha \to \mathsf{T}_\Box\, \alpha : \mathsf{Type}.$$

It seems that $\lambda\Pi R_{HOL}$, just like λU^-, allows more functions than λHOL, although the type of $[I]$ is *not* the translation of a λHOL type. Does that make $\lambda\Pi R_{HOL}$ inconsistent?

This question was an open problem for some time. Recently, Dowek [Dow15] proved conservativity for the embedding of some specific systems using models of strong normalization, based on previous work by Cousineau [Cou09]. In this chapter, we provide an original alternative proof that is generic and that works for all functional pure type systems. To achieve this, we use a novel approach that does not rely on strong normalization and that we call *relative normalization*. A similar proof has been the subject of a publication at TLCA 2015 [Ass15], although the proof we present here is more general and can be adapted to the translations we present later in this thesis.

6.1 Normalization and conservativity

Cousineau and Dowek [CD07] showed conservativity partially, by assuming that $\lambda\Pi R_{\mathcal{P}}$ is strongly normalizing (i.e. every well-typed term that is well-typed in $\Sigma_{\mathcal{P}}$ normalizes), but not much is known about normalization in $\lambda\Pi R$. The addition of rewriting can break normalization in a number of ways:

- The relation $\longrightarrow_\Gamma$ might not terminate on well-typed terms. For example, in:

$$\begin{aligned} &a : \mathsf{Type}, \\ &c : a, \\ &c \longmapsto c, \end{aligned}$$

  the term c does not terminate.

- The relation $\longrightarrow_{\beta\Gamma}$ might not terminate on well-typed terms. For example, in:

$$\begin{aligned} &\mathsf{term} : \mathsf{Type}, \\ &\mathsf{lam} \;\; : (\mathsf{term} \to \mathsf{term}) \to \mathsf{term}, \\ &\mathsf{app} \;\; : \mathsf{term} \to \mathsf{term} \to \mathsf{term}, \\ &\mathsf{app}\,(\mathsf{lam}\, f)\; x \longmapsto f\, x\,, \end{aligned}$$

  each of the relations $\longrightarrow_\beta$ and $\longrightarrow_\Gamma$ terminates but $\longrightarrow_{\beta\Gamma}$ does not in the term $\mathsf{app}\,(\mathsf{lam}\,(\lambda x \,.\, \mathsf{app}\, x\, x))\;(\mathsf{lam}\,(\lambda x \,.\, \mathsf{app}\, x\, x))\,.$

- The relation $\longrightarrow_\beta$ might not terminate on well-typed terms. For example, in:

$$\begin{aligned}&\mathsf{term} : \mathsf{Type},\\&\mathsf{term} \longmapsto \mathsf{term} \to \mathsf{term},\end{aligned}$$

  the term $(\lambda x : \mathsf{term} \,.\, x\, x)\ (\lambda x : \mathsf{term} \,.\, x\, x)$ is well-typed but does not terminate. One might think that this non-termination of β-reduction is caused by the non-termination of the rewrite system in the type of the term, but that is not necessary as is shown in the next point.

- The relation $\longrightarrow_\beta$ might not terminate on well-typed terms, <u>even though $\longrightarrow_\Gamma$ does terminate</u>. There are two known examples of this, the first being an adaptation of Russell's paradox by Dowek and Werner [DW00] and the second being the embeddings of system U and system U$^-$ shown in Chapter 5.

Lemma 5.2.3 shows that the embedding preserves reduction: if $M \longrightarrow M'$ then $[M] \longrightarrow^+ [M']$. As a consequence, if $\lambda\Pi R_\mathcal{P}$ is strongly normalizing then so is $\lambda\mathcal{P}$, but the converse might not be true *a priori*. Cousineau and Dowek's proof therefore relied on the unproven assumption that $\lambda\Pi R_\mathcal{P}$ is normalizing. This result is insufficient if one wants to consider the $\lambda\Pi$-calculus modulo rewriting as a general logical framework. Indeed, if the embedding turns out to be inconsistent then checking proofs in the framework has very little benefit.

One way to address the problem is to prove strong normalization of $\lambda\Pi R_\mathcal{P}$ by constructing a model, for example in the algebra of *reducibility candidates* [Gir72]. Dowek [Dow15] recently constructed such a model for the embedding of higher-order logic ($\lambda\Pi R_{HOL}$) and of the calculus of constructions ($\lambda\Pi R_C$). While correct, this technique is rather limited. Indeed, proving such a result is, by Lemma 5.2.3, at least as hard as proving the consistency of the original system. It requires specific knowledge of the PTS $\lambda\mathcal{P}$ and of its models, whose construction can be very involved, for example in the case of the calculus of constructions with an infinite universe hierarchy (λC^∞).

In our proof, we take a different approach and show that $\lambda\Pi R_\mathcal{P}$ is conservative in all cases, even when $\lambda\mathcal{P}$ is *not* normalizing. Instead of showing that $\lambda\Pi R_\mathcal{P}$ is strongly normalizing, we show that it is weakly normalizing *relative to* $\lambda\mathcal{P}$, meaning that proofs in the target language can be reduced to proofs in the source language. In the terminology of Chapter 2, we show that we can eliminate *administrative* β-redexes to obtain terms that contain only *essential* β-redexes. That way we prove only what is needed to show conservativity, without having to prove the consistency of $\lambda\mathcal{P}$ all over again. Together with the preservation of typing, this result shows that the embedding is sound and complete with respect to the original system.[1]

[1] Which direction is called *soundness* and which is called *completeness* is subjective and has been the subject of much debate by friendly colleagues and ruthless reviewers. In the original presentation [CD07], preservation of typing is called soundness and conservativity is called completeness. In our work, we prefer to use the other direction because we take the original system as a reference to which we compare the embedding. We prioritize using the terms *preservation of typing* and *conservativity* to avoid confusion.

6.2 Proof of conservativity

The exact statement is actually surprisingly tricky to get right. One could attempt to prove that if $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [M] : [\![A]\!]$ then $\Gamma \vdash_{\lambda\mathcal{P}} M : A$. However, that statement would be too weak because the translation $[M]$ is only defined for well-typed terms. A second attempt could be to define inverse translations $|M|$ and $\|A\|$ and prove that if $\Gamma \vdash_{\lambda\Pi R} M : A$ then $\|\Gamma\| \vdash_{\lambda\mathcal{P}} |M| : \|A\|$, but that would not work either because not all terms and types of $\lambda\Pi R_{\mathcal{P}}$ correspond to valid terms and types of $\lambda\mathcal{P}$, as was shown in Example 6.0.1. Therefore, the property that we want to prove is: if there is a term N such that $[\![\Gamma]\!] \vdash_{\lambda\Pi R} N : [\![A]\!]$ then there is a term M such that $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ and $[M] \equiv N$.

The main difficulty is that some of these *illegal* terms (with respect to $\lambda\mathcal{P}$) can be involved in witnessing *legal* types, as illustrated by the following example.

Example 6.2.1. Consider the context $nat : \Box$. Even though the polymorphic identity function I and its type are not well-typed in λHOL, they can be used in $\lambda\Pi R_{HOL}$ to construct a witness for $nat \to nat$:

$$\Sigma_{HOL}, nat : \mathsf{U}_\Box \vdash_{\lambda\Pi R} [I]\ nat : \mathsf{T}_\Box\, nat \to \mathsf{T}_\Box\, nat.$$

We can normalize the term $[I]\ nat$ to $\lambda x : \mathsf{T}_\Box\, nat\,.\, x$ which is a term that corresponds to a valid λHOL term: it is the translation of the term $\lambda x : nat\,.\, x$. However, as discussed previously, we cannot assume we can always normalize the terms because we do not know *a priori* if $\lambda\Pi R$ is normalizing.

To prove conservativity, we therefore need to address the following issues:

1. The system $\lambda\Pi R_{\mathcal{P}}$ can express more terms than $\lambda\mathcal{P}$.
2. These illegal terms can be used to construct proofs for the translation of $\lambda\mathcal{P}$ types.
3. The $\lambda\Pi R_{\mathcal{P}}$ terms that inhabit the translation of $\lambda\mathcal{P}$ types can be reduced to the translation of $\lambda\mathcal{P}$ terms.

To this end, we adapt Tait's reducibility method [Tai67] and define a relation

$$\Gamma \Vdash M : A$$

by induction on the type A, in such a way that the base case, when A is the translation of a $\lambda\mathcal{P}$ type, holds when M is (equivalent to) the translation of a $\lambda\mathcal{P}$ term. The main lemma that we prove is that typing implies reducibility, which we then instantiate to prove our theorem.

First we prove that the translation is sound with respect to equivalence:

$$[M] \equiv [M'] \implies M \equiv M'.$$

Together with Lemma 5.2.5, this shows that terms are equivalent if and only if their translations are equivalent. In order to do that, we define an inverse translation $|M|$ such

that $|[M]| \equiv M$, and use it to prove this theorem. Note that we will *not* use the inverse translation to express the conservativity theorem, as it has issues with η-equivalence. Indeed, some constants such as π_{s_1,s_2} can appear in η-reduced form in some well-typed terms, in which case it does not correspond to the translation of any valid term. This problem is well-known and can be hard to deal with, requiring either extending the $\lambda\Pi$-calculus with η-conversion [HHP93, GB99], or using complex arguments relying on terms in η-long form [CD07]. We avoid this problem by not using the inverse translation to state and prove conservativity of the translation. Instead, we define a weak form of η that we call η^- that is sufficient for our needs, and state the conservativity of typing using the forward translation only.

6.2.1 Conservativity of equivalence

Definition 6.2.2 (Inverse translation)**.** The *inverse term translation* $|M|$ is partially defined by induction on M:

$$\begin{aligned}
|\mathsf{u}_s| &= s \\
|\pi_{s_1,s_2}| &= \lambda\alpha : s_1 \,.\, \lambda\beta : \alpha \to s_2 \,.\, \Pi x : \alpha \,.\, \beta\, x \\
|x| &= x \\
|\lambda x : A \,.\, M| &= \lambda x : \|A\| \,.\, |M| \\
|M\, N| &= |M|\ |N|
\end{aligned}$$

The *inverse type translation* $\|A\|$ is partially defined by induction on A:

$$\begin{aligned}
\|\mathsf{U}_s\| &= s \\
\|\mathsf{T}_s\, M\| &= |M| \\
\|\Pi x : A \,.\, B\| &= \Pi x : \|A\| \,.\, \|B\| \\
\|\lambda x : A \,.\, B\| &= \lambda x : \|A\| \,.\, \|B\| \\
\|A\, N\| &= \|A\|\ |N|
\end{aligned}$$

The definition is partial but it is total on the image of the forward translation. Note that it is not an exact inverse. In particular, the definition of $|\pi_{s_1,s_2}|$ uses λ-abstractions that might not be legal in $\lambda\mathcal{P}$ because of the η expansion issue mentioned above. However, it is still an inverse *up to equivalence*.

Lemma 6.2.3 (Inverse)**.** *For all* Γ, M, A*, if* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *then* $|[M]_\Gamma| \equiv_\beta M$*.*

Proof. By induction on the structure of M. The details of the proof can be found in Appendix A.2. □

Corollary 6.2.4. *For all* Γ, A*, if* $\Gamma \vdash_{\lambda\mathcal{P}} A$ WF *then* $\|[\![A]\!]_\Gamma\| \equiv A$*.*

Proof. If $A = s$ for some $s \in \mathcal{S}^\top$ then $\|[\![A]\!]\| = \|\mathsf{U}_s\| = s$. Otherwise, $\Gamma \vdash A : s$ for some $s \in S$. By Lemma 6.2.3 $|[A]| \equiv A$. Therefore, $\|[\![A]\!]\| = \|\mathsf{T}_s\, [A]\| = |[A]| \equiv A$. □

Lemma 6.2.5 (Substitution)**.** *For all* x, M, N, $|M\{x \backslash N\}| = |M|\{x \backslash |N|\}$.

Proof. By induction on the structure of M. □

Corollary 6.2.6. *For all* x, B, N, $\|B\{x \backslash N\}\| = \|B\|\{x \backslash |N|\}$.

We now define *weak* η*-reduction*, written as η^-, and show that equivalence is preserved by the inverse translation. We then combine that with the previous results to deduce the conservativity of equivalence.

Definition 6.2.7 (Weak η-reduction)**.** We define weak η-reduction as the smallest relation $\longrightarrow_{\eta^-}$ closed by the subterm relation such that

$$\pi_{s_1,s_2}\, A\, B \longrightarrow_{\eta^-} \pi_{s_1,s_2}\, A\, (\lambda x\,.\, B\, x)\,.$$

Lemma 6.2.8. *If* $M \longrightarrow_{\beta\eta^-\Sigma} M'$ *then* $|M| \longrightarrow_{\beta}^{*} |M'|$.

Proof. By induction on the structure of M, using Lemma 6.2.5 for the base cases. □

Lemma 6.2.9. *If* $M \equiv_{\beta\eta^-\Sigma} M'$ *then* $|M| \equiv_{\beta} |M'|$.

Proof. By induction on the derivation of $\equiv_{\beta\eta^-\Sigma}$, using Lemma 6.2.8 for the base cases. □

Corollary 6.2.10. *If* $A \equiv_{\beta\eta^-\Sigma} A'$ *then* $\|A\| \equiv_{\beta} \|A'\|$.

Theorem 6.2.11 (Conservativity of equivalence)**.** *If* $[M]_\Gamma \equiv_{\beta\eta^-\Sigma} [M']_\Gamma$ *then* $M \equiv_{\beta} M'$.

Proof. By Lemma 6.2.9 and Lemma 6.2.3, $M \equiv |[M]| \equiv |[M']| \equiv M'$. □

Corollary 6.2.12. *If* $[\![M]\!]_\Gamma \equiv_{\beta\eta^-\Sigma} [\![M']\!]_\Gamma$ *then* $M \equiv_{\beta} M'$.

Before we move one to prove conservativity of typing, we prove the following variants involving sorts and Π types that will also be useful later.

Lemma 6.2.13. *If* $\Gamma \vdash_{\lambda\mathrm{WF}} A$ *and* $[\![A]\!]_\Gamma \equiv \mathsf{U}_s$ *then* $A \equiv s$.

Proof. By Corollary 6.2.10 and Corollary 6.2.4, $A \equiv \|[\![A]\!]_\Gamma\| \equiv s$. □

Corollary 6.2.14. *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *and* $[\![A]\!]_\Gamma \equiv \mathsf{U}_s$ *then* $\Gamma \vdash_{\lambda\mathcal{P}} M : s$.

Lemma 6.2.15. *If* $\Gamma \vdash_{\lambda\mathcal{P}} A$ WF *and* $[\![A]\!]_\Gamma \equiv \Pi x : A_1\,.\, B_1$ *then* $\exists A_1', B_1'$ *such that* $A \equiv \Pi x : A_1'\,.\, B_1'$.

Proof. By Corollary 6.2.10 and Corollary 6.2.4, $A \equiv \|[\![A]\!]_\Gamma\| \equiv \Pi x : \|A_1\|\,.\, \|B_1\|$. □

Corollary 6.2.16. *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *and* $[\![A]\!]_\Gamma \equiv \Pi x : A_1\,.\, B_1$ *then* $\exists A_1', B_1'$ *such that* $\Gamma \vdash_{\lambda\mathcal{P}} M : \Pi x : A_1'\,.\, B_1'$.

6.2.2 Conservativity of typing

To define the relation $\Gamma \Vdash M : A$ by induction, we first define the following measure.

Definition 6.2.17. For any term A, we define:

$$\begin{aligned}
\text{measure}\,(\mathsf{u}_s) &= 0 \\
\text{measure}\,(\mathsf{T}_s) &= 0 \\
\text{measure}\,(A\,N) &= \text{measure}\,(A) \\
\text{measure}\,(\lambda x : A\,.\,B) &= 1 + \text{measure}\,(B) \\
\text{measure}\,(\Pi x : A\,.\,B) &= 1 + \max\,(\text{measure}\,(A)\,, \text{measure}\,(B))
\end{aligned}$$

Note that this is a partial definition but it is total for the set of well-formed types.

Example 6.2.18. The measure of $\lambda\alpha : \mathsf{U}_s\,.\,\Pi f : (\mathsf{T}\,\alpha \to \mathsf{T}\,\alpha \to \mathsf{T}\,\alpha)\,.\,\mathsf{T}\,\alpha$ is 4.

Lemma 6.2.19. *For all* x, B, N, $\text{measure}\,(B\,\{x \backslash N\}) = \text{measure}\,(B)$.

Proof. By induction on B. □

Definition 6.2.20 (Reducibility)**.** Let $\Gamma \in \mathcal{G}_{\lambda\mathcal{P}}$ be a well-formed context of $\lambda\mathcal{P}$. For all terms $M, A \in \mathcal{T}_{\lambda\Pi R}$ of $\lambda\Pi R$, the relation $\Gamma \Vdash M : A$, meaning that the term M is *reducible* with type A in the context Γ, is defined by induction on the measure of A:

- if $A = \mathsf{U}_s$ or $A = \mathsf{T}_s\,A_1$ then $\exists M', A'$ such that

$$\begin{cases} \Gamma \vdash_{\lambda\mathcal{P}} M' : A', \\ [M']_\Gamma \equiv_{\beta\eta^-\Sigma} M, \\ [\![A']\!]_\Gamma \equiv_{\beta\eta^-\Sigma} A, \end{cases}$$

- if $A = (\lambda x : A_1\,.\,B_1)\;N_1 \cdots N_n$ then $\Gamma \Vdash M : B_1\,\{x \backslash N_1\}\;N_2 \cdots N_n$,
- if $A = \Pi x : A_1\,.\,B_1$ then $\forall \Gamma_N, N$ such that $\Gamma \subseteq \Gamma_N$, $\Gamma_N \Vdash M\,N : B_1\,\{x \backslash N\}$.

A context $\Delta \in \mathcal{G}_{\lambda\Pi R}$ of $\lambda\Pi R$ is an *object context* when $\Sigma_{\mathcal{P}}, \Delta \vdash_{\lambda\Pi R} A : \mathsf{Type}$ for all $(x : A) \in \Delta$. For all object context Δ and substitution $\sigma : \text{dom}\,(\Delta) \to \mathcal{T}_{\lambda\Pi R}$, the relation $\Gamma \Vdash \sigma : \Delta$ is defined as:

$$\Gamma \Vdash \sigma\,(x) : \sigma\,(A) \quad \forall\,(x : A) \in \Delta.$$

Lemma 6.2.21. *If* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *then* $\Gamma \Vdash [M]_\Gamma : [\![A]\!]_\Gamma$.

Proof. By definition, of $\Gamma \models [M] : \mathsf{T}_s\,[A]$. □

Corollary 6.2.22. *For all* $(x : A) \in \Gamma$, $\Gamma \Vdash x : [\![A]\!]$.

Lemma 6.2.23 (Weakening)**.** *If* $\Gamma \Vdash M : A$ *and* $\Gamma \subseteq \Gamma'$ *and* $\Gamma' \vdash_{\lambda\mathcal{P}} \text{WF}$ *then* $\Gamma' \Vdash M : A$.

Proof. By case analysis on the shape of A.

- Case $A = \mathsf{U}_s$ or $A = T_s\, B$. Then it follows from Lemma 4.3.4.
- Case $A = \Pi y : C \,.\, D$. Let Γ_N, N be such that $\Gamma' \subseteq \Gamma_N$ and $\Gamma_N \Vdash N : C$. Since $\Gamma \subseteq \Gamma'$, by transitivity we have $\Gamma \subseteq \Gamma_N$. By definition of $\Gamma \Vdash M : \Pi y : C \,.\, D$, we have $\Gamma_N \Vdash M\, N : D\,\{x \backslash N\}$. Therefore, $\Gamma' \Vdash M : \Pi y : C \,.\, D$.

□

Lemma 6.2.24. *If $M \equiv_{\beta\eta^-\Sigma} M'$ then $\Gamma \Vdash M : A$ iff $\Gamma \Vdash M' : A$.*

Proof. By induction on the measure of A. □

Lemma 6.2.25. *If $A \equiv_{\beta\eta^-\Sigma} B$ then $\Gamma \Vdash M : A$ iff $\Gamma \Vdash M : B$.*

Proof. By case analysis on the derivation of $A \equiv B$, using induction on the measure of A. The details of the proof can be found in Appendix A.2. □

Lemma 6.2.26. *For all Δ, M, A, if $\Sigma, \Delta \vdash_{\lambda\Pi R} M : A$ then for all Γ, σ such that $\Gamma \vdash_{\lambda\mathcal{P}}$ WF and $\Gamma \Vdash \sigma : \Delta$, $\Gamma \Vdash \sigma\,(M) : \sigma\,(A)$.*

Proof. By induction on the derivation. The details of the proof can be found in Appendix A.2. □

Theorem 6.2.27 (Conservativity of typing)**.** *If $\Gamma \vdash_{\lambda\mathcal{P}} A$ WF and $[\![\Gamma]\!] \vdash_{\lambda\Pi R} M : [\![A]\!]$ then there exists M' such that $\Gamma \vdash_{\lambda\mathcal{P}} M' : A$ and $[M']_\Gamma \equiv_{\beta\eta^-\Sigma} M$.*

Proof. Taking σ to be the identity substitution, by Corollary 6.2.22, we have $\Gamma \Vdash \sigma : [\![\Gamma]\!]$. By Lemma 6.2.26, $\Gamma \Vdash M : [\![A]\!]$. By definition, there exists M', A' such that $\Gamma \vdash M' : A'$ and $[M'] \equiv M$ and $[\![A']\!] \equiv [\![A]\!]$. By Corollary 6.2.12, $A' \equiv A$. Since $\Gamma \vdash A$ WF, by conversion we get $\Gamma \vdash M' : A$. □

Remark 6.2.28 (Conservativity of the embedding of complete systems)*.* When the PTS $\mathcal{P}$ is *complete* (Definition 4.1.8), there is an easier proof of conservativity: since all the products are allowed, a straightforward proof using the inverse translation is possible; there is no need for reducibility. This can be used for example for the embedding of the calculus of constructions with infinite hierarchy.

7

Application: translating HOL to Dedukti

Church's *simple type theory* (STT) [Chu40], also known as *higher-order logic* (HOL), is a natural deduction system where the objects of discourse are terms of the simply typed λ-calculus and propositions are terms of a special type. Since propositions are objects and can be universally quantified, the logic allows higher-order reasoning, hence the name. The system has been modernized and implemented in a number of theorem provers including HOL4, HOL Light, HOL Zero, ProofPower-HOL, and Isabelle/HOL, which are commonly referred to as the theorem provers of the *HOL family* [Ada10, Art04, Har09, NWP02, SN08]. These systems are fairly popular and many important mathematical results have been formalized in them, such as the Jordan curve theorem [Hal07] and the Kepler conjecture [HHM+10, Hal14].

We applied the ideas of the previous chapters to implement an automatic translation of HOL proofs to Dedukti. Although HOL is a relatively simple system that does not require computational embeddings, it still benefits from the more compact term representation that they provide. Our tool, called Holide, takes HOL proofs written in the OpenTheory format, which is a format for exchanging HOL proofs, and generates Dedukti files that can be checked in a specific HOL signature. We used it to successfully translate and check the OpenTheory standard library. In this chapter, we briefly present HOL and OpenTheory along with our translation to the $\lambda\Pi$-calculus modulo rewriting. We describe our implementation and compare it to other available translations based on experimental results. Part of this work was presented at PxTP 2015 [AB15].

7.1 HOL

There are many formulations of HOL. The minimal intuitionistic formulation is based on implication and universal quantification as primitive connectives, but the current systems generally use a formulation called Q_0 [And86] based on equality as a primitive connective. We take as reference the logical system used by OpenTheory [Hur11], which we now briefly present.

The terms of the logic are terms of the simply typed λ-calculus, with a base type bool representing the type of propositions and a type ind of individuals. The terms can contain constant symbols such as $(=)$, the symbol for equality, or select, the symbol of choice. The logic supports a restricted form of polymorphism, known as *ML-style polymorphism*, by allowing type variables, such as α or β, to appear in types. For example, the type of $(=)$ is $\alpha \to \alpha \to \mathsf{bool}$.

Types can be parameterized through type operators of the form $p(A_1, \ldots, A_n)$. For example, list is a type operator of arity 1, and $\mathsf{list}(\mathsf{bool})$ is the type of lists of booleans. Type variables and type operators are enough to describe all the types of HOL, because bool can be seen as a type operator of arity 0, and the arrow $\to$ as a type operator of arity 2. Hence, the type of $(=_\alpha)$ is in fact $\to(\alpha, \to(\alpha, \mathsf{bool}()))$. We still write $A \to B$ instead of $\to(A, B)$ for arrow types, p instead of $p()$ for type operators of arity 0, and $M = N$ instead of $(=)\, M\, N$ when it is more convenient. Types are implicitly assumed to be well-formed and terms are implicitly assumed to be well-typed.

Definition 7.1.1 (HOL)**.** The syntax of HOL is:

type variables	α, β
type operators	p
types	$A, B \ ::= \alpha \mid p\,(A_1, \ldots, A_n)$
term variables	x, y
term constants	c
terms	$M, N ::= x \mid c \mid \lambda x : A\,.\,M \mid M\,N$

The propositions of the logic are the terms of type bool and the predicates are the terms of type $A \to \mathsf{bool}$. We use letters such as ϕ or ψ to denote propositions. We write $\phi \Leftrightarrow \psi$ instead of $\phi = \psi$ when ϕ and ψ are propositions. The contexts, denoted by Γ or Δ, are sets of propositions, and the judgments of the logic are of the form $\Gamma \vdash \phi$. The derivation rules are presented in Figure 7.1.

Example 7.1.2. Here is a derivation of the transitivity of equality. If $\Gamma \vdash x = y$ and $\Delta \vdash y = z$, then $\Gamma \cup \Delta \vdash x = z$:

$$\dfrac{\dfrac{\dfrac{}{\vdash ((=)\ x) = ((=)\ x)}\ \textsc{Refl} \qquad \Delta \vdash y = z}{\vdash (x = y) = (x = z)}\ \textsc{AppThm} \qquad \vdash x = y}{x = z}\ \textsc{EqMp}$$

$$\frac{}{\vdash M = M}\ \textsc{Refl} \qquad \frac{}{\vdash (\lambda x : A \,.\, M)\ x = M}\ \textsc{Beta}$$

$$\frac{\Gamma \vdash M = N}{\Gamma \vdash \lambda x : A \,.\, M = \lambda x : A \,.\, N}\ \textsc{AbsThm} \qquad \frac{\Gamma \vdash F = G \qquad \Delta \vdash M = N}{\Gamma \cup \Delta \vdash F\,M = G\,N}\ \textsc{AppThm}$$

$$\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume} \qquad \frac{\Gamma \vdash \phi = \psi \qquad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\ \textsc{EqMp}$$

$$\frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{(\Gamma - \{\psi\}) \cup (\Delta - \{\phi\}) \vdash \phi = \psi}\ \textsc{DeductAntiSym}$$

$$\frac{\Gamma \vdash \phi}{\sigma\,(\Gamma) \vdash \sigma\,(\phi)}\ \textsc{Subst} \qquad \frac{}{\vdash c = M}\ \textsc{DefineConst}$$

$$\frac{\vdash P\,t}{\vdash abs\,(rep\,y) = y \qquad \vdash P\,x \Leftrightarrow (rep\,(abs\,x) = x)}\ \textsc{DefineTypeOp}$$

Figure 7.1 – Derivation rules of OpenTheory HOL

Type and constant definitions In addition to the primitive types and constants, HOL supports mechanisms for defining new types and constants in a conservative way. The rule DefineConst allows us to define a new constant c as equal to another closed term M. The constant c must be previously undefined. As an example, we can define the top logical connective as $\top = ((\lambda x : \mathsf{bool} \,.\, x) = (\lambda x : \mathsf{bool} \,.\, x))$.

Type definitions are a bit more complicated. Given a type A and a non-empty predicate P on A, the rule DefineTypeOp defines the type of elements of A that satisfy P. More precisely, it defines a new type B that is isomorphic to the subset $\{x : A \mid P\,x\}$. This mechanism also creates two functions, $abs : (A \to B)$ and $rep : (B \to A)$ that go back and forth between the two types. The two constants abs and rep must be previously undefined. The two conclusions of the rule express the isomorphism between $\{x : A \mid P\,x\}$ and B. All the new datatypes of HOL (inductive types, etc.) are constructed using this mechanism [Har95].

Axioms In addition to the core derivation rules, three axioms are assumed:

- *η-equality*, which states that $\lambda x : A \,.\, M\,x = M$,
- the *axiom of choice*, with a predeclared symbol of choice called select,
- the *axiom of infinity*, which states that the type ind is infinite.

It is important to note that, from η-equality and AbsThm, we can derive functional extensionality and, with the axiom of choice, we can derive the excluded middle [Bee85, Dia75], making this formulation of HOL a classical logic.

7.2 Translation

Several formalizations of HOL in LF have been proposed [App01, Rab10, SS06] using the principles of Chapter 2. As explained in Section 2.4, because the $\lambda\Pi$-calculus does not have polymorphism, we cannot translate propositions directly as types, as doing so would prevent us from quantifying over propositions, so we have to use a higher-order embedding. For each proposition ϕ, we have two translations: one translation $[\phi]$ *as a term*, and another $[\![\phi]\!] = \mathsf{proof}\,[\phi]$ *as a type*. The signature in $\lambda\Pi$ typically looks like:

$$
\begin{aligned}
&\mathsf{type} && : \mathsf{Type} \\
&\mathsf{prop} && : \mathsf{type} \\
&\mathsf{arrow} && : \mathsf{type} \to \mathsf{type} \to \mathsf{type} \\
&\mathsf{term} && : \mathsf{type} \to \mathsf{Type} \\
&\mathsf{lam} && : (\mathsf{term}\,\alpha \to \mathsf{term}\,\beta) \to \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \\
&\mathsf{app} && : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \to \mathsf{term}\,\alpha \to \mathsf{term}\,\beta \\
&\mathsf{proof} && : \mathsf{term}\,\mathsf{prop} \\
&\mathsf{rule_1} && : \ldots \\
&\mathsf{rule_2} && : \ldots
\end{aligned}
$$

The resulting embedding is not computational. This is not a big problem for HOL because the logic can be formulated without any notion of reduction, by taking the Q_0 formulation, which takes equality as a primitive connective, and stating β-equality as an axiom. Since the logic does not have dependent types, there is no issue of incompleteness. However, this approach has a couple of other issues:

- From a practical point of view, the encoding of terms is larger because of the quadratic blowup (see Example 3.3.1).
- From a constructive point of view, the proofs have no computational content because they have no notion of reduction.

In our embedding, we add the rewrite rule

$$\mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta) \to \mathsf{term}\,\alpha \to \mathsf{term}\,\beta\ ,$$

which allows us to identify the type $\mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)$ with the type $\mathsf{term}\,\alpha \to \mathsf{term}\,\beta$ and thus define an embedding that is computational like in Section 3.3. The encoding of the terms becomes more compact, as we represent λ-abstractions by λ-abstractions, applications by applications, etc. For example, the term $(\lambda x : \alpha\,.\,x)\,x$ is encoded as $(\lambda x : \mathsf{term}\,\alpha\,.\,x)\ x$. Moreover, we can add reductions to the proofs of HOL, in a way that is similar to the pure type system formulation of HOL mentioned in Section 4.2.

This might be beneficial for several reasons:

1. It gives a reduction semantics for the proofs of HOL.

2. It allows compressing the proofs further by replacing conversion proofs (that use Beta and DefineConst) with reflexivity.

3. Several other proof systems (Coq, Agda, etc.) are based on pure type systems, so expressing HOL as a PTS fits in the large scale of interoperability.

Other translations have been proposed to automatically extract the proofs of HOL to other systems such as Isabelle [KK13, OS06], Nuprl [NSM01], and Coq [KW10]. Except for Kalyszyk and Krauss' implementation [KK13], these tools suffer from scalability problems. Our translation is lightweight enough to be scalable and provides promising results. The implementation of Kalyszyk and Krauss [KK13] is the first efficient and scalable translation of HOL Light proofs, but its target is Isabelle/HOL, a system that, unlike Dedukti, is foundationally very close to HOL Light.

7.2.1 Translation of types

We declare a new type called type and three constructors bool, ind and arrow:

$$\begin{aligned}
\mathsf{type} &: \mathsf{Type}, \\
\mathsf{bool} &: \mathsf{type}, \\
\mathsf{ind} &: \mathsf{type}, \\
\mathsf{arrow} &: \mathsf{type} \to \mathsf{type} \to \mathsf{type}.
\end{aligned}$$

Definition 7.2.1 (Translation of a HOL type as a term)**.** For any HOL type A, we define $[A]$, the translation of A *as a term*, to be:

$$\begin{aligned}
[\alpha] &= \alpha \\
[\mathsf{bool}] &= \mathsf{bool} \\
[\mathsf{ind}] &= \mathsf{ind} \\
[A \to B] &= \mathsf{arrow}\,[A]\;[B]\,.
\end{aligned}$$

More generally, if we have an n-ary HOL type operator p, we declare a constant p of type $\mathsf{type}^n \to \mathsf{type}$, and we translate an instance $p\,(A_1, \ldots, A_n)$ of this type operator to $p\,[A_1] \cdots [A_n]$.

7.2.2 Translation of terms

We declare a new dependent type called term indexed by a type, and we identify the terms of type $\mathsf{term}(\mathsf{arrow}\,A\,B)$ with the functions of type $\mathsf{term}\,A \to \mathsf{term}\,B$ by adding a

rewrite rule. We also declare a constant eq for HOL equality and a constant select for the choice operator:

$$\begin{aligned}
\mathsf{term} &: \mathsf{type} \to \mathsf{Type}, \\
\mathsf{eq} &: \Pi\alpha : \mathsf{type} \,.\, \mathsf{term}\,(\mathsf{arrow}\,\alpha\,(\mathsf{arrow}\,\alpha\,\mathsf{bool}))\,, \\
\mathsf{select} &: \Pi\alpha : \mathsf{type} \,.\, \mathsf{term}\,(\mathsf{arrow}\,(\mathsf{arrow}\,\alpha\,\mathsf{bool})\;\alpha)\,,
\end{aligned}$$

$$\mathsf{term}\,(\mathsf{arrow}\,a\,b) \longmapsto \mathsf{term}\,a \to \mathsf{term}\,b.$$

Definition 7.2.2 (Translation of a HOL type as a type)**.** For any HOL type A, we define $[\![A]\!]$, the translation of A *as a type*, to be:

$$[\![A]\!] = \mathsf{term}\,[A]\,.$$

Definition 7.2.3 (Translation of a HOL term as a term)**.** For any HOL term M, we define $[M]$, the translation of M *as a term*, to be:

$$\begin{aligned}
[x] &= x \\
[M\,N] &= [M]\;[N] \\
[\lambda x : A \,.\, M] &= \lambda x : [\![A]\!] \,.\, [M] \\
[(=_A)] &= \mathsf{eq}\,[A] \\
[\mathsf{select}_A] &= \mathsf{select}\,[A]\;\;.
\end{aligned}$$

More generally, for every HOL constant c of type A, if $\alpha_1, \ldots, \alpha_n$ are the free type variables that appear in A, we declare a new constant c of type

$$\Pi\alpha_1 : \mathsf{type} \ldots . \Pi\alpha_n : \mathsf{type} \,.\, [\![A]\!]$$

and we translate an instance $c_{A_1,\ldots,A_n}$ of this constant to $c\,[A_1] \cdots [A_n]$.

7.2.3 Translation of proofs

We declare a new type proof indexed by a proposition:

$$\mathsf{proof} : \mathsf{term}\,\mathsf{bool} \to \mathsf{Type}.$$

Definition 7.2.4 (Translation of HOL propositions as types)**.** For any HOL proposition ϕ (i.e. a HOL term of type bool), we define $[\![\phi]\!]$, the translation of ϕ *as a type*, to be:

$$[\![\phi]\!] = \mathsf{proof}\,[\phi]\;.$$

For any HOL context $\Gamma = \phi_1, \ldots, \phi_n$, we define

$$[\![\Gamma]\!] = h_{\phi_1} : [\![\phi_1]\!]\,, \ldots, h_{\phi_n} : [\![\phi_n]\!]\;.$$

Equality proofs

We declare Refl, FunExt, and AppThm:

$$
\begin{aligned}
&\mathsf{Refl} && : \Pi\alpha : \mathsf{type} \,.\, \Pi x : \mathsf{term}\,\alpha \,.\, \mathsf{proof}\,(\mathsf{eq}\,\alpha\,x\,x)\,,\\
&\mathsf{FunExt} && : \Pi\alpha,\beta : \mathsf{type} \,.\, \Pi f,g : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)\,.\\
& && \quad (\Pi x : \mathsf{term}\,\alpha \,.\, \mathsf{proof}\,(\mathsf{eq}\,\beta\,(f\,x)\;(g\,x))) \to \mathsf{proof}\,(\mathsf{eq}\,(\mathsf{arrow}\,\alpha\,\beta)\;f\,g)\,,\\
&\mathsf{AppThm} && : \Pi\alpha,\beta : \mathsf{type} \,.\, \Pi f,g : \mathsf{term}\,(\mathsf{arrow}\,\alpha\,\beta)\,.\,\Pi x,y : \mathsf{term}\,\alpha\,.\\
& && \quad \mathsf{proof}\,(\mathsf{eq}\,(\mathsf{arrow}\,\alpha\,\beta)\;f\,g) \to \mathsf{proof}\,(\mathsf{eq}\,\alpha\,x\,y) \to \mathsf{proof}\,(\mathsf{eq}\,\beta\,(f\,x)\;(g\,y))\,.
\end{aligned}
$$

The constant FunExt expresses *functional extensionality*, which states that if two functions f and g of type $A \to B$ are equal on all values x of type A, then f and g are equal. We use it to translate both AbsThm and the axiom of η-equality. Since our encoding is computational, we prove β-equality by reflexivity.

Definition 7.2.5. The rules Refl, AbsThm, AppThm, and Beta are translated as:

$$\left[\frac{}{\vdash M = M}\ \textsc{Refl}\right] = \mathsf{Refl}\,[A]\;[M] \quad \text{(where } A \text{ is the type of } M\text{)}$$

$$\left[\frac{}{(\lambda x : A \,.\, M)\,x = M}\ \textsc{Beta}\right] = \mathsf{Refl}\,[B]\;[M] \quad \text{(where } B \text{ is the type of } M\text{)}$$

$$\left[\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma \cup \Delta \vdash F\,M = G\,N}\ \textsc{AppThm}\right] = \mathsf{AppThm}\,[A]\;[B]\;[F]\;[G]\;[M]\;[N]\;[\mathcal{D}_1]\;[\mathcal{D}_2]$$

$$\left[\frac{\mathcal{D}}{\Gamma \vdash \lambda x : A \,.\, M = \lambda x : A \,.\, N}\ \textsc{AbsThm}\right] =$$

$$\mathsf{FunExt}\,[A]\;[B]\;[\lambda x : A \,.\, M]\;[\lambda x : A \,.\, N]\;(\lambda x : [\![A]\!] \,.\, [\mathcal{D}])\,.$$

Boolean proofs

We declare the constants PropExt and EqMp:

$$
\begin{aligned}
&\mathsf{PropExt} && : \Pi p,q : \mathsf{term}\,\mathsf{bool}\,.\\
& && \quad (\mathsf{proof}\,q \to \mathsf{proof}\,p) \to (\mathsf{proof}\,q \to \mathsf{proof}\,p) \to \mathsf{proof}\,(\mathsf{eq}\,\mathsf{bool}\,p\,q)\,,\\
&\mathsf{EqMp} && : \Pi p,q : \mathsf{term}\,\mathsf{bool}\,.\,\mathsf{proof}\,(\mathsf{eq}\,\mathsf{bool}\,p\,q) \to \mathsf{proof}\,p \to \mathsf{proof}\,q.
\end{aligned}
$$

The constant PropExt expresses *propositional extensionality* and, together with EqMp, states that equality on booleans in HOL behaves like the connective"*if and only if*".

Definition 7.2.6. The rules ASSUME, DEDUCTANTISYM, and EQMP are translated as:

$$\left[\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume}\right] = h_\phi$$

$$\left[\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma \cup \Delta \vdash \psi}\ \textsc{EqMp}\right] = \mathsf{EqMp}\,[\phi]\ [\psi]\ [\mathcal{D}_1]\ [\mathcal{D}_2]$$

$$\left[\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{(\Gamma - \{\psi\}) \cup (\Delta - \{\phi\}) \vdash \phi = \psi}\ \textsc{DeductAntiSym}\right] =$$

$$\mathsf{PropExt}\,[\phi]\ [\psi]\ (\lambda h_\psi : [\![\psi]\!] \,.\, [\mathcal{D}_1])\ (\lambda h_\phi : [\![\phi]\!] \,.\, [\mathcal{D}_2])\ .$$

Substitution proofs

The HOL rule SUBST derives $\sigma(\Gamma) \vdash \sigma(\phi)$ from $\Gamma \vdash \phi$. The substitution can substitute for both term variables and type variables but type variables are instantiated first. For the sake of clarity, we split this rule in two steps: one for term substitution of the form $\sigma = x_1 \backslash M_1, \ldots, x_n \backslash M_n$, and one for type substitution of the form $\theta = \alpha_1 \backslash A_1, \ldots, \alpha_m \backslash A_m$. In Dedukti, we have to rely on β-reduction to express substitution. We can correctly translate a substitution $M\,\{x_1 \backslash M_1, \ldots, x_n \backslash M_n\}$ as

$$(\lambda x_1 : B_1 \ldots . \lambda x_n : B_n \,.\, M)\, M_1 \ldots\ M_n$$

where B_i is the type of M_i.

Definition 7.2.7. The ruleSUBST is translated to

$$\left[\frac{\mathcal{D}}{\theta(\Gamma) \vdash \theta(\phi)}\ \textsc{TypeSubst}\right] = (\lambda \alpha_1 : \mathsf{type} \ldots . \lambda \alpha_m : \mathsf{type} \,.\, [\mathcal{D}])\, [A_1]\ \ldots\ [A_m]$$

$$\left[\frac{\mathcal{D}}{\sigma(\Gamma) \vdash \sigma(\phi)}\ \textsc{TermSubst}\right] = (\lambda x_1 : [\![B_1]\!] \ldots . \lambda x_n : [\![B_n]\!] \,.\, [\mathcal{D}])\, [M_1]\ \ldots\ [M_n]\ .$$

Constant and type definitions For every constant definition $c = M$, we declare in the signature a new constant:

$$c : A := M$$

where A is the type of M. We can then derive the rule DEFINECONST using reflexivity. Similarly, for every definition of a new type operator p from a predicate P on the type A, we declare in the signature:

$$\begin{aligned} p &: \mathsf{type}^n \to \mathsf{type}, \\ abs &: \Pi \vec{\alpha} : \vec{\mathsf{type}} \,.\, [\![A]\!] \to [\![p\,(\alpha_1, \ldots, \alpha_n)]\!]\,, \\ rep &: \Pi \vec{\alpha} : \vec{\mathsf{type}} \,.\, [\![p\,(\alpha_1, \ldots, \alpha_n)]\!] \to [\![A]\!]\,, \end{aligned}$$

where $\alpha_1, \ldots, \alpha_n$ are the free type variables that occur in A. We derive the first conclusion of DefineTypeOp by adding the rule

$$abs\,(rep\,y) \longmapsto y\,.$$

We were not able to find suitable rewrite rules for deriving the second conclusion of DefineTypeOp so we declare it as an axiom.

7.3 Completeness

The translation preserves well-formation of types, typing of terms, and correctness of proofs. We do not give the proofs of the following results, as they are a straightforward adaptation of the results of Chapter 5.

Lemma 7.3.1 (Preservation of well-formed types)**.** *For any HOL type A,*

$$\Sigma, \vec{\alpha} : \vec{\mathsf{type}} \vdash_{\lambda\Pi R} [A] : \mathsf{type}$$

and

$$\Sigma, \vec{\alpha} : \vec{\mathsf{type}} \vdash_{\lambda\Pi R} [\![A]\!] : \mathsf{Type}$$

where $\vec{\alpha}$ are the free type variables appearing in A.

Lemma 7.3.2 (Preservation of well-typed terms)**.** *For any HOL term M of type A,*

$$\Sigma, \vec{\alpha} : \vec{\mathsf{type}}, \vec{x} : [\![\vec{A}]\!] \vdash_{\lambda\Pi R} [M] : [\![A]\!]$$

where $\vec{\alpha}$ are the free type variables and $\vec{x} : \vec{A}$ are the free term variables appearing in M. For any HOL proposition ϕ,

$$\Sigma, \vec{\alpha} : \vec{\mathsf{type}}, \vec{x} : [\![\vec{A}]\!] \vdash_{\lambda\Pi R} [\![\phi]\!] : \mathsf{Type}$$

where $\vec{\alpha}$ are the free type variables and $\vec{x} : \vec{A}$ are the free term variables appearing in ϕ.

Lemma 7.3.3 (Preservation of valid proofs)**.** *For any HOL proof $\mathcal{D}$ of $\Gamma \vdash \phi$,*

$$\Sigma, \vec{\alpha} : \vec{\mathsf{type}}, \vec{x} : [\![\vec{A}]\!], [\![\Gamma]\!] \vdash_{\lambda\Pi R} [\mathcal{D}] : [\![\phi]\!]$$

where $\vec{\alpha}$ are the free type variables and $\vec{x} : \vec{A}$ are the free term variables appearing in $\mathcal{D}$.

Notice that in the last lemma there can be free type variables and free term variables that appear in $\mathcal{D}$ but not in the statement $\Gamma \vdash \phi$. Typically, this is due to the left branch of the EqMp rule because ϕ does not appear in the conclusion. Thankfully, we can get rid of these using substitution. We replace every free type variable that is not in $\Gamma \vdash \phi$ by a closed type (e.g. bool) and every free term variable of type A that is not in $\Gamma \vdash \phi$ by a closed term of type A (e.g. $\mathsf{select}\,A\,(\lambda x\,.\,\top)$). This is possible because of the constant select, which implicitly assumes that every simple type is inhabited. Without it, we would need to still assume one inhabitant for every free variable in the context.

7.4 Implementation

7.4.1 Proof retrieval

The members of the HOL family have very similar implementations based on Milner's LCF, usually in OCaml or SML. The core idea behind that approach is to represent theorems as sequents that are members of an *abstract datatype* `thm`, i.e. a datatype that cannot be constructed outside of the kernel. As such, the sequents of type `thm` can only come from the kernel and so must be constructed using the derivation rules of the logic, thus guaranteeing safety without the need to remember the proofs. That approach reduces memory consumption but hinders the system's ability to share proofs. This is a recurrent issue when trying to retrieve proofs of HOL [Hur11, KW10, OS06].

Fortunately, several proposals have already been made to solve this problem [Ada15, Hur11, KK13, OS06]. Among them is the OpenTheory project, which we chose because it is the most mature and most well documented. It defines a standard format called the *article format* for recording and sharing HOL theorems. An article file contains a sequence of elementary commands for an abstract machine to reconstruct proofs. Importing a theorem thus only requires a mechanical execution of the commands. The format of OpenTheory is limited to the HOL family, and cannot be used to communicate the proofs of Coq for example. However, it is an excellent starting point for our translation. Choosing OpenTheory as a front-end has several advantages:

- We cover all the systems of the HOL family that can export their proofs to OpenTheory with a single implementation. As of today, this includes HOL Light, HOL4, and ProofPower.[1]
- The implementation of a theorem prover can change with time, sometimes in undocumented ways, so the existence of this standard, well-documented proof format is extremely helpful, if not necessary for a good translation.
- The OpenTheory project also defines a large standard theory library, covering the development of common datatypes and mathematical theories such as lists and natural numbers. This substantial body of theories was used as a benchmark for our implementation.

7.4.2 Holide: HOL to Dedukti

We implemented our translation in an automated tool called Holide[2]. It works as an OpenTheory virtual machine that additionally keeps track of the corresponding proof terms for theorems. The program reads a HOL proof written in the OpenTheory article format (`.art`) and outputs a Dedukti file (`.dk`) containing its translation. We can run Dedukti on the generated file to check it. All generated files are linked with a hand-written file `hol.dk` containing the signature of HOL. The translation process is illustrated in Figure 7.2.

[1] Isabelle/HOL can currently read from but not write to OpenTheory.
[2] https://www.rocq.inria.fr/deducteam/Holide/

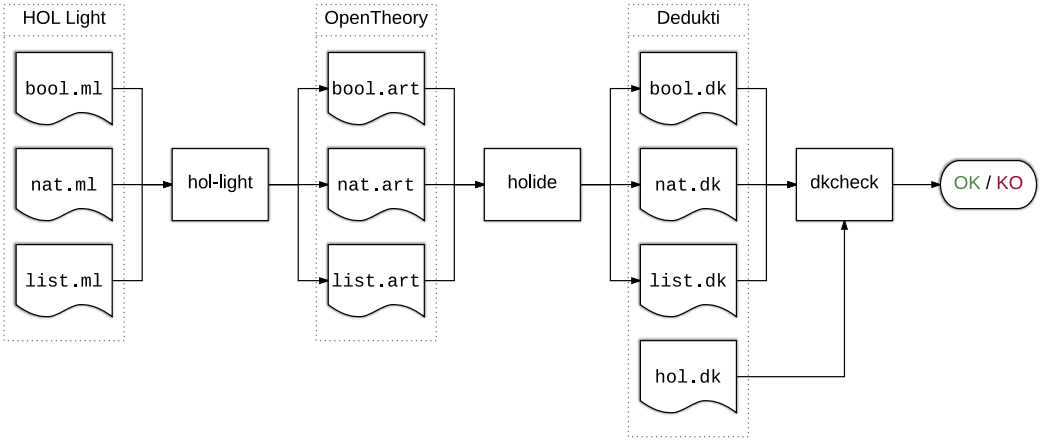

Figure 7.2 – Translation of HOL proofs to Dedukti using Holide

HOL proofs are known to be very large [KK13, KW10, OS06], and we needed to implement sharing of proofs, terms, and types in order to reduce them to a manageable size. OpenTheory already provides some form of proof sharing but we found that it was easier to completely factorize the derivations into individual intermediary steps.

Experimental results We used Holide to translate the OpenTheory standard library. The library is organized into individual packages, each corresponding to a theory such as lists or natural numbers. We were able to verify all the generated files. The results are summarized in Table 7.1. We list the size of both the source files and the files generated by the translation after compression using `gzip`. The reason we use the size of the compressed files for comparison is because it provides a more reasonable measure that is less affected by syntax formatting and whitespace. We also list the time it takes to translate and verify each package. These tests were done on a 64-bit Intel Xeon(R) CPU @ 2.67GHz $\times$ 4 machine with 4 GB of RAM.

Overall, the size of the generated files is about 3 to 4 times larger than the source files. Given that this is an encoding in a logical framework, an increase in the size is to be expected, and we find this factor reasonable. There are no similar translations to compare to except the one of Keller and Werner [KW10]. The comparison is difficult because they work with a slightly different form of input, but they produce several hundreds of megabytes of proofs. Similarly, an increase in verification time is to be expected compared to verifying OpenTheory directly, but our results are still reasonable given the nature of the translation. Our time is about 4 times larger than OpenTheory, which takes about 5 seconds to verify the standard library. It is in line with the scalable translation of Kalyszyk and Krauss to Isabelle/HOL, which takes around 30 seconds[KK13]. In comparison, Keller and Werner's translation takes several hours, although we should note that our work greatly benefited from their experience.

	Size (kB)		Time (s)	
Package	OpenTheory	Dedukti	Translation	Verification
`unit`	5	13	0.2	0
`function`	16	53	0.3	0.2
`pair`	38	121	0.8	0.5
`bool`	49	154	0.9	0.5
`sum`	84	296	2.1	1.1
`option`	93	320	2.2	1.2
`relation`	161	620	4.6	2.8
`list`	239	827	5.7	3.2
`real`	286	945	6.5	3.1
`natural`	343	1065	6.8	3.2
`set`	389	1462	10.2	5.8
Total	1702	4877	40.3	21.6

Table 7.1 – Translation of the OpenTheory standard theory library using Holide

7.4.3 Extensions

In this section we show some possible extensions that make use of the advantages of having a translation that preserves computation.

Compressing conversion proofs

One of the reasons why HOL proofs are so large is that conversion proofs have to traverse the terms using the congruence rules AbsThm and AppThm. Since we now prove β-equality and constant definitions computationally using reflexivity, large conversion proofs could be reduced to a single reflexivity step, therefore reducing the size of the proofs.

Example 7.4.1. The following proof of $f(g((\lambda x : A\,.\,x)\,x)) = f(g(x))$,

$$\dfrac{\dfrac{}{\vdash f = f}\,\textsc{Refl} \qquad \dfrac{\dfrac{}{\vdash g = g}\,\textsc{Refl} \qquad \dfrac{}{\vdash (\lambda x : A\,.\,x)\;x = x}\,\textsc{Beta}}{\vdash g\,((\lambda x : A\,.\,x)\;x) = g\,x}\,\textsc{AppThm}}{\vdash f\,(g\,((\lambda x : A\,.\,x)\;x)) = f\,(g\,x)}\,\textsc{AppThm}$$

can be translated simply as $\mathsf{Refl}\,B(f(g\,x))$.

HOL as a pure type system

In our implementation, we represented the rules of HOL using axioms. This means that proofs lack a computational behavior. As mentioned in Section 4.2, HOL can be seen as a pure type system called λHOL with three sorts. That formulation corresponds to minimal constructive higher-order logic, with only implication and universal quantification as primitive connectives. In particular, it has a very clear computational behavior. However, that structure is lost in the Q_0 formulation used by the actual HOL systems. Our translation can be adapted to recover that structure.

Instead of equality, we declare implication and universal quantification as primitive connectives, and we define what provability means through rewriting.

$$\begin{aligned}
&\mathsf{imp} \;\; : \mathsf{term}\,(\mathsf{arrow}\,\mathsf{bool}\,(\mathsf{arrow}\,\mathsf{bool}\,\mathsf{bool}))\\
&\mathsf{forall} : \Pi a : \mathsf{type}\,.\,\mathsf{term}\,(\mathsf{arrow}\,(\mathsf{arrow}\,a\,\mathsf{bool})\;\mathsf{bool})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{proof}\,(\mathsf{imp}\,p\,q) \;\;\longmapsto \mathsf{proof}\,p \to \mathsf{proof}\,q\\
&\mathsf{proof}\,(\mathsf{forall}\,a\,p) \longmapsto \Pi x : \mathsf{term}\,a\,.\,\mathsf{proof}\,(p\,x)
\end{aligned}$$

However, this time we do not even need to declare constants for the counterparts of rules like Refl and AppThm because they are *derivable*. For example, here is a derivation of the introduction and elimination rules for implication:

$$\begin{aligned}
\mathsf{imp_intro} \;&:\; \Pi p, q : \mathsf{term}\,\mathsf{bool}\,.\,(\mathsf{proof}\,p \to \mathsf{proof}\,q) \to \mathsf{proof}\,(\mathsf{imp}\,p\,q)\\
&=\; \lambda p, q : \mathsf{term}\,\mathsf{bool}\,.\,\lambda h : (\mathsf{proof}\,p \to \mathsf{proof}\,q)\,.\,h\\
\mathsf{imp_elim} \;&:\; \Pi p, q : \mathsf{term}\,\mathsf{bool}\,.\,\mathsf{proof}\,(\mathsf{imp}\,p\,q) \to \mathsf{proof}\,p \to \mathsf{proof}\,q\\
&=\; \lambda p, q : \mathsf{term}\,\mathsf{bool}\,.\,\lambda h : \mathsf{proof}\,(\mathsf{imp}\,p\,q)\,.\,\lambda x : \mathsf{proof}\,p\,.\,h\,x
\end{aligned}$$

By translating the introduction rules as λ-abstractions, and the elimination rules as applications, we recover the reduction of the proof terms, which corresponds to *cut elimination* in the original proofs.

As for equality, it is also possible to define it in terms of these connectives. For example, we could use the Leibniz definition of equality, which is the one used by Coq:

$$\begin{aligned}
\mathsf{eq} \;&:\; \Pi a : \mathsf{type}\,.\,\mathsf{term}\,(\mathsf{arrow}\,a\,(\mathsf{arrow}\,a\,\mathsf{bool}))\\
&=\; \lambda a : \mathsf{type}\,.\,\lambda x : \mathsf{term}\,a\,.\,\lambda y : \mathsf{term}\,a\,.\\
&\quad\; \mathsf{forall}\,(\mathsf{arrow}\,a\,\mathsf{bool})\;(\Pi p : \mathsf{term}\,(\mathsf{arrow}\,a\,\mathsf{bool})\,.\,\mathsf{imp}\,(p\,x)\;(p\,y))
\end{aligned}$$

We would still need to assume some axioms to prove all the rules, namely FunExt and PropExt [KW10], but this definition is closer to that of Coq. Since the PTS λHOL is a strict subset of the calculus of inductive constructions, we can use this to inject HOL directly into an embedding of Coq in Dedukti [AC15] (see Section 12.2.3). The implementation of this extension is the subject of currently ongoing work in the team.

III

Cumulative and infinite hierarchies

8

Cumulative type systems

Cumulativity is the internalization of the notion of *containment*. It allows us to view the members of a universe Type_i as members of a higher universe. If we think of the typing relation $M : A$ as set membership, then the rule

$$\frac{}{\vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}$$

expresses that each universe is *a member* of the next:

$$\mathsf{Type}_0 \in \mathsf{Type}_1 \in \mathsf{Type}_2 \in \cdots .$$

This stratification is required to avoid paradoxes related to $\mathsf{Type} \in \mathsf{Type}$ (such as in $\lambda*$ from Example 4.2.1). From a set-theoretical point of view, it is similar to the distinction between *sets* (or small sets) and *collections* (or large sets). In contrast, the cumulativity rule

$$\frac{\Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash A : \mathsf{Type}_{i+1}}$$

expresses that each universe is *contained* in the next:

$$\mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \mathsf{Type}_2 \subseteq \cdots .$$

This feature was first introduced in Martin-Löf's intuitionistic type theory [ML73, ML84]. In that theory, universes are built sequentially, one after the other. At each step, a new universe is introduced containing all the previously existing types, which by definition includes the members of all previous universes too. This is not surprising from a set-theoretical point of view since quantifying over all large sets usually includes small sets too. The feature was later introduced in the calculus of constructions by Coquand [Coq86] and by Luo in his *extended calculus of constructions* [Luo89, Luo90].

Cumulativity also has practical applications, as it allows us to write code that is more generic. Suppose for example we want to define the polymorphic type of lists but that we want to use it at different levels, so that we can talk about lists of natural numbers, lists of types, etc. Without cumulativity, we would need to define a type $\mathsf{list}_i : \forall \alpha : \mathsf{Type}_i.\mathsf{Type}_i$ for every i. By quantifying once over all types that are smaller than some fixed universe Type_j, we would only need to define $\mathsf{list} : \forall \alpha : \mathsf{Type}_j.\mathsf{Type}_j$ once at a high enough level j to be able to use it for all types $A : \mathsf{Type}_i$ with $i \leq j$. Of course, this approach is still limited as we would need to define another type, or "bump" the level of list once we want to use it at a level higher than j. Several other features have been developed to cope with this issue, including *floating universes* and *universe polymoprhism* [HP89, ST14], but cumulativity has become an integral part of type theory and is implemented in systems such as Coq and Matita. Since cumulativity is not a conservative extension, we have to deal with it if we hope to embed the proofs of those systems.

In this thesis, we generalize the embedding of pure type systems to systems with cumulativity. To this end, we first generalize the framework of pure type systems. Indeed, cumulativity is present in many systems in the literature, with myriads of variations in the typing rules that are not always equivalent. For example, some systems have the cumulativity relation $\mathsf{Prop} \subseteq \mathsf{Type}_0$, while others have $\mathsf{Prop} \subseteq \mathsf{Type}_1$, and some have neither, and these variations are not equivalent. Some systems have the rule $(\mathsf{Type}_i, \mathsf{Type}_i, \mathsf{Type}_i)$ while some $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max(i,j)})$ and others have $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_k)$ for all $i, j \leq k$, but all these variations are usually equivalent. We therefore need a common framework in which to describe and compare these systems.

Cumulative type systems were introduced by Barras [Bar99] as a generalization of pure type systems with cumulativity. They provide a general framework for studying the λ-calculus core of many systems with cumulativity, which allows us to give a homogeneous description and a common metatheory for a wide variety of systems. The metatheory was first explored by Barras and Gregoire [Bar99, BG05]. Cumulative type systems were later taken and reformulated by Lasson, who provides an extensive and meticulous study of their metatheory in his PhD thesis [Las12]. In this chapter, we briefly present cumulative type systems and the main results of Barras and Lasson on which we will build Chapters 9 and 10. Our goal is to eventually embed both intuitionistic type theory and the calculus of inductive constructions. We also present a new a bidirectional typing system for deriving *principal types*, which we will use for our embedding in the $\lambda\Pi$-calculus modulo rewriting, and prove that it is sound and complete.

8.1 Definition

8.1.1 Specification and syntax

Cumulative type systems share the same syntax as pure type systems, with β-equivalence as the main equivalence relation. Specifications are extended by a new relation $\mathcal{C}$ that describes cumulativity.

Definition 8.1.1 (Specification)**.** A *cumulative type system* (CTS) *specification* is a quadruple $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of constants called *sorts,*
- $\mathcal{A} \in \mathcal{S} \times \mathcal{S}$ is a relation called *axioms,*
- $\mathcal{R} \in \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a relation called *rules,*
- $\mathcal{C} \in \mathcal{S} \times \mathcal{S}$ is a relation called *cumulativity.*

We sometimes write (s_1, s_2) for the rule (s_1, s_2, s_2). The cumulative type system associated with a specification $\mathcal{P}$ is written $\lambda P_{\preceq}$. In the following, we assume a fixed specification $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$.

Definition 8.1.2 (Syntax)**.** The syntax of $\lambda\mathcal{P}_{\preceq}$ is given by the following grammar:

$$\begin{array}{lll} \text{sorts} & s & \in \mathcal{S} \\ \text{terms} & M, N, A, B & \in \mathcal{T} ::= x \mid s \mid M\,N \mid \lambda x : A\,.\,M \mid \Pi x : A\,.\,B \\ \text{contexts} & \Gamma, \Delta & \in \mathcal{G} ::= \varnothing \mid \Gamma, x : A \end{array}$$

8.1.2 Subtyping

We define a subtyping relation $\preceq$ that generalizes β-equivalence and includes the cumulativity relation $\mathcal{C}$.

Definition 8.1.3 (Subtyping)**.** The *subtyping relation* $\preceq$ is given by the rules:

$$\frac{A \equiv_\beta B}{A \preceq B} \qquad \frac{(s_1, s_2) \in \mathcal{C}^*}{s_1 \preceq s_2} \qquad \frac{B \preceq C}{\Pi x : A\,.\,B \preceq \Pi x : A\,.\,C} \qquad \frac{A \preceq B \qquad B \preceq C}{A \preceq C}$$

The relation $\prec$ is defined as the strict version of $\preceq$, i.e.

$$A \prec B \overset{\text{def}}{\iff} A \preceq B \wedge A \not\equiv B.$$

Since these relations depend on $\mathcal{C}$, we sometimes write $\preceq_{\mathcal{P}}$ and $\prec_{\mathcal{P}}$ to disambiguate.

Notice that we allow covariance in the co-domain of Π. Although this feature is not present originally in intuitionistic type theory, type inference would be more cumbersome without it. In particular, λ-abstractions could otherwise be typed in multiple ways that are not related. Moreover, a variable would not have the same types as its η-expansions. Notice also that we do not allow contravariance in the domain of Π. The main reason for that is that it would not behave well with respect to type inference and principal typing. This is a standard restriction [Luo90, Las12] and is found in Coq [Cdt12] and Matita [ARCT09] for example.

Lemma 8.1.4. *If $A \preceq B$ then either $A \equiv B$ or $\exists C_1, \ldots, C_n, s, s'$ such that*

$$\begin{cases} A \equiv \Pi x_1 : C_1 . \cdots \Pi x_n : C_n . s \\ B \equiv \Pi x_1 : C_1 . \cdots \Pi x_n : C_n . s' \end{cases}$$

and $s \prec s'$.

Proof. By induction on the derivation of $A \preceq B$. □

The subtyping relation behaves well with substitution.

Lemma 8.1.5. *If $A \preceq B$ then $A\{x \backslash N\} \preceq B\{x \backslash N\}$.*

Proof. By induction on the derivation of $A \preceq B$. □

8.1.3 Typing

Definition 8.1.6 (Typing)**.** The typing relations

- $\Gamma \vdash M : A$, meaning that the term M *has type* A in the context Γ,
- $\Gamma \vdash \mathrm{WF}$, meaning that the context Γ is *well-formed*,

are derived by the rules in Figure 8.1. A term A is a *well-formed type* in Γ when $\Gamma \vdash \mathrm{WF}$ and either $\Gamma \vdash A : s$ or $A = s$ for some $s \in \mathcal{S}$. We write this as $\Gamma \vdash A\ \mathrm{WF}$. It is *inhabited* in Γ when $\exists M, \Gamma \vdash M : A$. We write $\Gamma \vdash M : A : B$ as a shortcut for $\Gamma \vdash M : A$ and $\Gamma \vdash A : B$. We sometimes write $\lambda\mathcal{P}_{\preceq}$ instead of $\vdash$ to disambiguate.

Notice that the conversion rule Conv of pure type systems has been replaced by the subtyping rule Sub. The rule Sub-sort is required if we want to allow promoting the type of a term to a top-sort. It is redundant in complete systems such as the calculus of inductive constructions because they have no top-sorts. Finally, remark that pure type systems are a special instance of cumulative type systems with $\mathcal{C} = \varnothing$.

$$\frac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash \mathrm{WF} \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A \,.\, B : s_3}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \preceq B}{\Gamma \vdash M : B}\ \textsc{Sub}$$

$$\frac{\Gamma \vdash M : A \qquad A \preceq s}{\Gamma \vdash M : s}\ \textsc{Sub-sort}$$

Figure 8.1 – Typing rules of the system $\lambda\mathcal{P}_{\preceq}$

8.2 Basic properties

The properties of cumulative type systems are analogous to those of pure type systems. We state the following results with no proof. The complete proofs can be found in the works of Barras and Lasson [Bar99, Las12].

Lemma 8.2.1 (Confluence)**.** *If $M \equiv_\beta M'$ then $\exists M''$ such that $M \longrightarrow_\beta^* M''$ and $M' \longrightarrow_\beta^* M''$.*

Lemma 8.2.2 (Injectivity of syntax)**.** *The following holds:*

$$\begin{aligned} x \equiv_\beta x' &\implies x = x' \\ s \equiv_\beta s' &\implies s = s' \\ \Pi x : A \,.\, B \equiv_\beta \Pi x : A' \,.\, B' &\implies A \equiv_\beta A' \wedge B \equiv_\beta B' \\ \lambda x : A \,.\, M \equiv_\beta \lambda x : A' \,.\, M' &\implies A \equiv_\beta A' \wedge M \equiv_\beta M' \end{aligned}$$

Lemma 8.2.3 (Free variables)**.** *If $\Gamma \vdash M : A$ then $\mathrm{FV}(M) \cup \mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$. If $\Gamma, x : A, \Gamma \vdash \mathrm{WF}$ then $\mathrm{FV}(A) \subseteq \mathrm{dom}(\Gamma)$ and $x \notin \mathrm{dom}(\Gamma)$.*

Lemma 8.2.4 (Substitution)**.** *If $\Gamma, x : A, \Gamma' \vdash M : B$ and $\Gamma \vdash N : A$ then $\Gamma, \Gamma' \{x \backslash N\} \vdash M \{x \backslash N\} : B \{x \backslash N\}$.*

Lemma 8.2.5 (Weakening)**.** *If $\Gamma \vdash M : A$ and $\Gamma' \vdash \mathrm{WF}$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash M : A$.*

Lemma 8.2.6 (Inversion)**.** *The following holds:*

$$\begin{array}{lll} \Gamma \vdash x : C & \Rightarrow \exists A. & \Gamma \vdash \mathrm{WF} \wedge (x : A) \in \Gamma \wedge A \preceq C \\ \Gamma \vdash s_1 : C & \Rightarrow \exists s_2. & \Gamma \vdash \mathrm{WF} \wedge (s_1, s_2) \in \mathcal{A} \wedge s_2 \preceq C \\ \Gamma \vdash \Pi x : A \,.\, B : C & \Rightarrow \exists s_1, s_2, s_3. & \Gamma \vdash A : s_1 \wedge \Gamma, x : A \vdash B : s_2 \wedge (s_1, s_2, s_3) \in \mathcal{R} \wedge s_3 \preceq C \\ \Gamma \vdash \lambda x : A \,.\, M : C & \Rightarrow \exists B, s. & \Gamma, x : A \vdash M : B \wedge \Gamma \vdash \Pi x : A \,.\, B : s \wedge \Pi x : A \,.\, B \preceq C \\ \Gamma \vdash M\, N : C & \Rightarrow \exists A, B. & \Gamma \vdash M : \Pi x : A \,.\, B \wedge \Gamma \vdash N : A \wedge B \{x \backslash N\} \preceq C \end{array}$$

Lemma 8.2.7 (Correctness of typing)**.** *If $\Gamma \vdash M : A$ then $\Gamma \vdash A\ \mathrm{WF}$.*

Lemma 8.2.8 (Subject reduction)**.** *If $\Gamma \vdash M : A$ and $M \longrightarrow_\beta^* M'$ then $\Gamma \vdash M' : A$.*

8.3 Inter-system properties

8.3.1 Subsystems

Definition 8.3.1 (Subsystem)**.** A CTS $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$ is a *subsystem* of $\mathcal{P}' = (\mathcal{S}', \mathcal{A}', \mathcal{R}', \mathcal{C}')$ when $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{A} \subseteq \mathcal{A}'$, $\mathcal{R} \subseteq \mathcal{R}'$, and $\mathcal{C} \subseteq \mathcal{C}'$. We write $\mathcal{P} \subseteq \mathcal{P}'$.

Lemma 8.3.2. *If $\mathcal{P} \subseteq \mathcal{P}'$ then $\Gamma \vdash_{\lambda P_\preceq} M : A \implies \Gamma \vdash_{\lambda P'_\preceq} M : A$.*

Lemma 8.3.3 (Compactness)**.** *If $\Gamma \vdash_{\lambda P_\preceq} M : A$ then there is a finite sub-system $\mathcal{P}'$ such that $\Gamma \vdash_{\lambda P'_\preceq} M : A$.*

8.3.2 Morphisms

Definition 8.3.4 (Morphism)**.** Let $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$ and $\mathcal{P}' = (\mathcal{S}', \mathcal{A}', \mathcal{R}', \mathcal{C}')$ be two CTSs. A function $\varphi : \mathcal{S} \to \mathcal{S}'$ is a *CTS morphism* when:

1. $\forall (s_1, s_2) \in \mathcal{A}, (\varphi(s_1), \varphi(s_2)) \in \mathcal{A}'$,
2. $\forall (s_1, s_2, s_3) \in \mathcal{R}, (\varphi(s_1), \varphi(s_2)) \in \mathcal{R}'$,
3. $\forall (s_1, s_2) \in \mathcal{C}, (\varphi(s_1), \varphi(s_2)) \in \mathcal{C}'$.

The function φ can be extended to all terms and contexts of $\mathcal{P}$ as follows:

$$\begin{aligned} \varphi(x) &= x \\ \varphi(M\,N) &= \varphi(M)\ \varphi(N) \\ \varphi(\lambda x : A\,.\,M) &= \lambda x : \varphi(A)\,.\,\varphi(N) \\ \varphi(\Pi x : A\,.\,B) &= \Pi x : \varphi(A)\,.\,\varphi(B) \end{aligned}$$

We write $\varphi : \mathcal{P} \to \mathcal{P}'$ to say that φ is a morphism between $\mathcal{P}$ and $\mathcal{P}'$.

Lemma 8.3.5. *If $\mathcal{P} \subseteq \mathcal{P}'$ then the identity function is a morphism from $\mathcal{P}$ to $\mathcal{P}'$, i.e.* $\mathrm{id} : \mathcal{P} \to \mathcal{P}'$.

Morphisms preserve β-reductions, β-equivalence, cumulativity, and typing.

Lemma 8.3.6. *If $\varphi : \mathcal{P} \to \mathcal{P}'$ then*

1. $M \longrightarrow_\beta M' \iff \varphi(M) \longrightarrow_\beta \varphi(M')$,
2. $M \equiv_\beta M' \iff \varphi(M) \equiv_\beta \varphi(M')$,
3. $M \preceq_{\mathcal{P}} M' \implies \varphi(M) \preceq_{\mathcal{P}'} \varphi(M')$,
4. $\Gamma \vdash_{\lambda P_{\preceq}} M : A \implies \Gamma \vdash_{\lambda P'_{\preceq}} \varphi(M) : \varphi(A)$

Corollary 8.3.7. *If $\varphi : \mathcal{P} \to \mathcal{P}'$ and $\mathcal{P}'$ is weakly (resp. strongly) normalizing then $\mathcal{P}$ is weakly (resp. strongly) normalizing.*

8.3.3 Closures

In cumulative type systems, certain axioms and rules can be redundant because of cumulativity. For example, the calculus of constructions equipped with the cumulativity relation $(*, \Box)$:

$$(C_{\preceq}) \begin{bmatrix} \mathcal{S} = *, \Box \\ \mathcal{A} = (*, \Box) \\ \mathcal{R} = (*, *), (*, \Box), (\Box, *), (\Box, \Box) \\ \mathcal{C} = (*, \Box) \end{bmatrix}$$

is equivalent to the CTS:

$$\left(C'_{\preceq}\right)\begin{bmatrix} \mathcal{S}' = *, \Box \\ \mathcal{A}' = (*, \Box) \\ \mathcal{R}' = (*, *), (\Box, *), (\Box, \Box) \\ \mathcal{C}' = (*, \Box) \end{bmatrix}$$

because the rule $(*, \Box)$ can be derived from $(*, \Box) \in \mathcal{C}$ and $(\Box, \Box) \in \mathcal{R}$:

$$\dfrac{\dfrac{\Gamma \vdash A : * \qquad * \preceq' \Box}{\Gamma \vdash A : \Box} \qquad \Gamma, x : A \vdash B : \Box \qquad (\Box, \Box) \in \mathcal{R}'}{\Gamma \vdash \Pi x : A \,.\, B : \Box}\ .$$

For this reason, we define the following operations on axioms and rules.

Definition 8.3.8 (Cumulative closure)**.** The *cumulative closure* of a CTS $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$ is the CTS $\overline{\mathcal{P}} = \left(\overline{\mathcal{S}}, \overline{\mathcal{A}}, \overline{\mathcal{R}}, \overline{\mathcal{C}}\right)$ where:

$$\begin{aligned} \overline{\mathcal{S}} &= \mathcal{S} \\ \overline{\mathcal{A}} &= \left\{(s_1, s_2) \in \mathcal{S}^2 \mid \exists s'_2 \in \mathcal{S}, (s_1, s'_2) \in \mathcal{A} \wedge s'_2 \preceq s_2\right\} \\ \overline{\mathcal{R}} &= \left\{(s_1, s_2, s_3) \in \mathcal{S}^3 \mid \exists\, (s'_1, s'_2, s'_3) \in \mathcal{R}, s_1 \preceq s'_1 \wedge s_2 \preceq s'_2 \wedge s'_3 \preceq s_3\right\} \\ \overline{\mathcal{C}} &= \mathcal{C} \end{aligned}$$

Obviously, $\mathcal{P} \subseteq \overline{\mathcal{P}}$. Moreover, all the axioms and rules of $\overline{\mathcal{P}}$ are derivable from those of $\mathcal{P}$, which gives the following result.

Lemma 8.3.9. *For any CTS $\mathcal{P}$, $\Gamma \vdash_{\lambda P_{\preceq}} M : A \iff \Gamma \vdash_{\lambda \overline{P}_{\preceq}} M : A$.*

8.4 Principal typing

Not all CTSs are well-behaved with respect to cumulativity. Type uniqueness does not hold. In practice, we need *principal types*; that is, the property that all the types of a well-typed term are supertypes of a single type. Lasson [Las12] determined a criterion for CTSs to be well-behaved called the *local minimum* property. It is the equivalent of the *functionality* property in PTSs. It is a requirement, together with the well-foundedness of $\prec$, for the system to have principal types.

8.4.1 Definition

Definition 8.4.1 (Local minimum)**.** A CTS has the *local minimum* property if

$$\left.\begin{array}{l} (s_1, s_2) \in \mathcal{A} \\ (s'_1, s'_2) \in \mathcal{A} \\ r_1 \preceq s_1 \wedge r_1 \preceq s'_1 \end{array}\right\} \implies \exists r_2, (r_1, r_2) \in \mathcal{A} \wedge r_2 \preceq s_2 \wedge r_2 \preceq s'_2$$

and

$$\left.\begin{array}{l}(s_1, s_2, s_3) \in \mathcal{R} \\ (s'_1, s'_2, s'_3) \in \mathcal{R} \\ r_1 \preceq s_1 \wedge r_1 \preceq s'_1 \\ r_2 \preceq s_2 \wedge r_2 \preceq s'_2\end{array}\right\} \implies \exists r_3, (r_1, r_2, r_3) \in \mathcal{R} \wedge r_3 \preceq s_3 \wedge r_3 \preceq s'_3.$$

The following lemma shows that the local minimum property for CTSs is the equivalent of the functionality property for PTSs.

Lemma 8.4.2. *A pure type system is functional if and only if it has the local minimum property.*

Proof. In pure type systems, $A \preceq B \iff A \equiv B$, so $r \preceq s \iff r = s$. □

Definition 8.4.3 (Principal type)**.** A term M has *principal type* A in the context Γ when $\Gamma \vdash M : A$ and, for all B, if $\Gamma \vdash M : B$ then $A \preceq B$. We write this as $\Gamma \models M \Rightarrow A$, with the symbol $\models$ to emphasize that this is a semantic property. Note that the principal type of a term, if it exists, is unique up to β-equivalence.

Lemma 8.4.4. *If $\Gamma \models M \Rightarrow A$ then, for all B, $\Gamma \vdash M : B \iff (\Gamma \vdash B \text{ WF} \wedge A \preceq B)$.*

Proof. The first direction follows by definition of principal types and by correctness of typing (Lemma 8.2.7), the second by the rules Sub and Sub-sort. □

Principal types are *not* preserved by substitution and reduction. Indeed, the principal type of a term can decrease after substitution, as shown by the following example.

Example 8.4.5 (Principal types are not preserved)**.** We have $x : \mathsf{Type}_2 \models x \Rightarrow \mathsf{Type}_2$ and $\vdash \mathsf{Type}_0 : \mathsf{Type}_2$, however $x\{x \backslash \mathsf{Type}_0\} = \mathsf{Type}_0$ and $\mathsf{Type}_2\{x \backslash \mathsf{Type}_0\} = \mathsf{Type}_2$ and $\not\models \mathsf{Type}_0 \Rightarrow \mathsf{Type}_2$.

The following are sufficient conditions for the existence of principal types.

Lemma 8.4.6. *If $\mathcal{P}$ has the local minimum property and $\prec$ is a well-founded order then, for any Γ, M, if M is well-typed in Γ then M has a principal type in Γ.*

The well-foundedness of the strict subtyping relation $\prec$ follows from that of the cumulativity relation $\mathcal{C}$.

Lemma 8.4.7. *If $\mathcal{C}$ is a well-founded order then $\prec$ is a well-founded order.*

All cumulative type systems used in practice (pure type systems, intuitionistic type theory, calculus of inductive constructions) have the local minimum property and are well-founded. **In the rest of this thesis, we will only consider systems that satisfy these properties.**

8.4.2 Derivation rules

We now present a typing system for deriving principal types. We define a new bidirectional typing system with two new judgements, $\Gamma \vdash M \Rightarrow A$ ("M has minimal type A") and $\Gamma \vdash M \Leftarrow A$ ("M checks against type A"). We show that the first one is equivalent to $\Gamma \models M \Rightarrow A$ and that the second one is equivalent to $\Gamma \vdash M : A$. First, we define a minified version of $\mathcal{P}$ that is functional but still equivalent.

Definition 8.4.8 (Minification)**.** The *minification* of a CTS $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{C})$ is the CTS $\underline{\mathcal{P}} = (\underline{\mathcal{S}}, \underline{\mathcal{A}}, \underline{\mathcal{R}}, \underline{\mathcal{C}})$ where:

$$\begin{aligned}
\underline{\mathcal{S}} &= \mathcal{S} \\
\underline{\mathcal{A}} &= \{(s_1, s_2) \in \mathcal{A} \mid \forall s_2' \in \mathcal{S}, (s_1, s_2') \in \mathcal{A} \implies s_2 \preceq s_2'\} \\
\underline{\mathcal{R}} &= \{(s_1, s_2, s_3) \in \mathcal{R} \mid \forall s_3' \in \mathcal{S}, (s_1, s_2, s_3') \in \mathcal{R}, s_3 \preceq s_3'\} \\
\underline{\mathcal{C}} &= \mathcal{C}
\end{aligned}$$

This CTS is functional. Obviously, $\underline{\mathcal{P}} \subseteq \mathcal{P}$. Moreover, since $\mathcal{P}$ has principal types, all the axioms and rules of $\mathcal{P}$ are still derivable in $\mathcal{P}'$. In that sense, minification is the opposite of the closure operation.

Lemma 8.4.9. *The following holds:*

- $(s_1, s_2') \in \mathcal{A} \implies \exists s_2. (s_1, s_2) \in \underline{\mathcal{A}}$,
- $(s_1, s_2, s_3') \in \mathcal{R} \implies \exists s_3. (s_1, s_2, s_3) \in \underline{\mathcal{R}}$.

Proof.

- Suppose $(s_1, s_2') \in \mathcal{A}$. Consider the non-empty set $S = \{s_2 \mid (s_1, s_2) \in \mathcal{A}\}$. Since $\prec$ is well-founded, S has a minimal element s_2. Moreover, for any $s_2' \in S$, by the local minimum property, there exists r_2 such that $(s_1, r_2) \in \mathcal{A}$ and $r_2 \preceq s_2$ and $r_2 \preceq s_2'$. Since s_2 is minimal, we have $r_2 = s_2 \preceq s_2'$.
- Suppose $(s_1, s_2, s_3') \in \mathcal{R}$. Consider the non-empty set $S = \{s_3 \mid (s_1, s_2, s_3) \in \mathcal{R}\}$. Since $\prec$ is well-founded, S has a minimal element s_3. Moreover, for any $s_3' \in S$, by the local minimum property, there exists r_3 such that $(s_1, s_2, r_3) \in \mathcal{R}$ and $r_3 \preceq s_3$ and $r_3 \preceq s_3'$. Since s_3 is minimal, we have $r_3 = s_3 \preceq s_3'$.

□

Corollary 8.4.10. $\Gamma \vdash_{\lambda\mathcal{P}_\preceq} M : A \iff \Gamma \vdash_{\lambda\underline{\mathcal{P}}_\preceq} M : A$.

Proof. All the axioms and rules of $\lambda\mathcal{P}'$ are in $\lambda\mathcal{P}$ (more precisely, $\underline{\mathcal{P}} \subseteq \mathcal{P}$), and all the axioms and rules of $\lambda\mathcal{P}$ are derivable in $\lambda\mathcal{P}'$ by Lemma 8.4.9. □

We can now give the system for deriving principal types. We will first call these types *minimal types* and later show that they coincide with principal types.

Definition 8.4.11 (Minimal typing)**.** The typing relations

- $\Gamma \vdash M \Rightarrow A$, meaning the term M *has minimal type* A in the context Γ,
- $\Gamma \vdash M \Leftarrow A$, meaning the term M *checks against type* A,
- $\Gamma \vdash\Rightarrow \mathrm{WF}$, meaning the judgement Γ *is well-formed* with respect to minimal typing,

are derived by the rules in Figure 8.2. A term A is a well-formed type in Γ with respect to minimal typing when $\Gamma \vdash\Rightarrow \mathrm{WF}$ and either $A = s$ or $\Gamma \vdash A \Rightarrow s$ for some sort $s \in \mathcal{S}$, and we write this as $\Gamma \vdash A \Rightarrow \mathrm{WF}$. We sometimes write $\vdash_{\lambda\mathcal{P}_{\preceq}}$ instead of $\vdash$ to disambiguate.

Since principal typing is not preserved by substitution and reduction (Example 8.4.5), special care must be taken when formulating the substitution and subject reduction lemmas.

Lemma 8.4.12. *If* $\Gamma \vdash M \Leftarrow C$ *then* $\exists A$ *such that* $\Gamma \vdash M \Rightarrow A$ *and* $\Gamma \vdash C \Rightarrow \mathrm{WF}$ *and* $A \preceq C$.

Proof. By inspection of the typing rules, the last rule in the derivation of $\Gamma \vdash M \Leftarrow C$ must be either Check or Check-sort. □

Lemma 8.4.13. *If* $\Gamma, x : A, \Gamma' \vdash M \Rightarrow B$ *and* $\Gamma \vdash N \Leftarrow A$ *then* $\Gamma, \Gamma'\{x \backslash N\} \vdash M\{x \backslash N\} \Leftarrow B\{x \backslash N\}$.

Proof. By induction on the typing derivation. □

Lemma 8.4.14 (Inversion)**.** *The following holds:*

$$
\begin{array}{lll}
\Gamma \vdash x \Rightarrow C & \Rightarrow \exists A. & \Gamma \vdash\Rightarrow \mathrm{WF} \wedge (x : A) \in \Gamma \wedge A \equiv C \\
\Gamma \vdash s_1 \Rightarrow C & \Rightarrow \exists s_2. & \Gamma \vdash\Rightarrow \mathrm{WF} \wedge (s_1, s_2) \in \underline{\mathcal{A}} \wedge s_2 \equiv C \\
\Gamma \vdash \Pi x : A \,.\, B \Rightarrow C & \Rightarrow \exists s_1, s_2, s_3. & \Gamma \vdash A \Rightarrow s_1 \wedge \Gamma, x : A \vdash B \Rightarrow s_2 \wedge (s_1, s_2, s_3) \in \underline{\mathcal{R}} \wedge s_3 \equiv C \\
\Gamma \vdash \lambda x : A \,.\, M \Rightarrow C & \Rightarrow \exists B, s. & \Gamma, x : A \vdash M \Rightarrow B \wedge \Gamma \vdash \Pi x : A \,.\, B \Rightarrow s \wedge \Pi x : A \,.\, B \equiv C \\
\Gamma \vdash M\,N \Rightarrow C & \Rightarrow \exists A, B. & \Gamma \vdash M \Rightarrow \Pi x : A \,.\, B \wedge \Gamma \vdash N \Leftarrow A \wedge B\{x \backslash N\} \equiv C
\end{array}
$$

Proof. By induction on the typing derivation. □

Lemma 8.4.15. *If* $\Gamma \vdash M \Rightarrow A$ *then* $\Gamma \vdash A \Rightarrow \mathrm{WF}$.

Proof. By induction on the typing derivation. □

Lemma 8.4.16. *If* $\Gamma \vdash M \Rightarrow A$ *and* $M \longrightarrow_\beta^* M'$ *then* $\Gamma \vdash M' \Leftarrow A$.

Proof. By induction on the number of steps. For a single step, by induction on M, using Lemma 8.4.13 for the base case. □

$$\frac{}{\varnothing \vdash \Rightarrow \mathrm{WF}}\ \textsc{Empty} \qquad \frac{\Gamma \vdash A \Rightarrow s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \Rightarrow \mathrm{WF}}\ \textsc{Decl}$$

$$\frac{\Gamma \vdash \Rightarrow \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash \Rightarrow \mathrm{WF} \qquad (s_1 : s_2) \in \underline{\mathcal{A}}}{\Gamma \vdash s_1 \Rightarrow s_2}\ \textsc{Sort}$$

$$\frac{\Gamma \vdash A \Rightarrow s_1 \qquad \Gamma, x : A \vdash B \Rightarrow s_2 \qquad (s_1, s_2, s_3) \in \underline{\mathcal{R}}}{\Gamma \vdash \Pi x : A \,.\, B \Rightarrow s_3}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M \Rightarrow B \qquad \Gamma \vdash \Pi x : A \,.\, B \Rightarrow s}{\Gamma \vdash \lambda x : A \,.\, M \Rightarrow \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M \Rightarrow \Pi x : A \,.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash M\, N \Rightarrow B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M \Rightarrow A \qquad \Gamma \vdash B \Rightarrow s \qquad A \equiv B}{\Gamma \vdash M \Rightarrow B}\ \textsc{Conv}$$

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \equiv s}{\Gamma \vdash M \Rightarrow s}\ \textsc{Conv-sort}$$

$$\frac{\Gamma \vdash M \Rightarrow A \qquad \Gamma \vdash B \Rightarrow s \qquad A \preceq B}{\Gamma \vdash M \Leftarrow B}\ \textsc{Check}$$

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq s}{\Gamma \vdash M \Leftarrow s}\ \textsc{Check-sort}$$

Figure 8.2 – Minimal-typing rules of the system $\lambda\mathcal{P}_{\preceq}$

8.4.3 Soundness

We now show the soundness of the typing rules with respect to principal typing, meaning that if we can derive $\Gamma \vdash M \Rightarrow A$ then A is the principal type of M. First we need a technical lemma that shows that each individual rule is sound.

Lemma 8.4.17. *The following holds:*

1. $\Gamma \vdash \mathrm{WF} \wedge (x : A) \in \Gamma \implies \Gamma \models x \Rightarrow A$
2. $\Gamma \vdash \mathrm{WF} \wedge (s_1, s_2) \in \underline{\mathcal{A}} \implies \Gamma \models s_1 \Rightarrow s_2$
3. $\Gamma \models A \Rightarrow s_1 \wedge \Gamma, x : A \models B \Rightarrow s_2 \wedge (s_1, s_2, s_3) \in \underline{\mathcal{R}} \implies \Gamma \models \Pi x : A \,.\, B \Rightarrow s_3$
4. $\Gamma, x : A \models M \Rightarrow B \wedge \Gamma \vdash \Pi x : A \,.\, B : s \implies \Gamma \models \lambda x : A \,.\, M \Rightarrow \Pi x : A \,.\, B$
5. $\Gamma \models M \Rightarrow \Pi x : A \,.\, B \wedge \Gamma \vdash N : A \implies \Gamma \models M\, N \Rightarrow B\,\{x \backslash N\}$
6. $\Gamma \models M \Rightarrow A \wedge \Gamma \vdash B : s \wedge A \equiv B \implies \Gamma \models M \Rightarrow B$
7. $\Gamma \models M \Rightarrow A \wedge A \equiv s \implies \Gamma \models A \Rightarrow s$

Proof. The details of the proof can be found in Appendix A.3. □

Lemma 8.4.18 (Soundness of minimal typing)**.** *If* $\Gamma \vdash M \Rightarrow A$ *then* $\Gamma \models M \Rightarrow A$*. If* $\Gamma \vdash M \Leftarrow A$ *then* $\Gamma \vdash M : A$*. If* $\Gamma \vdash \Rightarrow \mathrm{WF}$ *then* $\Gamma \vdash \mathrm{WF}$*.*

Proof. By induction on the typing derivation, using Lemma 8.4.17. □

8.4.4 Completeness

We now show the converse, namely that if a term M has principal type A, then we can derive $\Gamma \vdash M \Rightarrow A$. We start with the following technical lemmas.

Lemma 8.4.19. *If* $\Gamma \vdash M \Rightarrow A$ *and* $A \preceq s$ *then* $\exists s'$ *such that* $\Gamma \vdash M \Rightarrow s'$ *and* $s' \preceq s$*.*

Proof. By Lemma 8.1.4 and confluence, $\exists s'$ such that $A \longrightarrow^* s'$ and $s' \preceq s$. Therefore, $\Gamma \vdash M \Rightarrow s'$ by the Conv-sort rule. □

Lemma 8.4.20. *If* $\Gamma \vdash M \Rightarrow A$ *and* $A \preceq \Pi x : C \,.\, D$ *then* $\exists C', D'$ *such that* $\Gamma \vdash M \Rightarrow \Pi x : C' \,.\, D'$ *and* $C' \equiv C$ *and* $D' \equiv D$*.*

Proof. By Lemma 8.1.4 and confluence, $\exists C', D'$ such that $A \longrightarrow^* \Pi x : C' \,.\, D'$ and $C' \equiv C$ and $D' \preceq D$. Therefore, $\Gamma \vdash M \Rightarrow \Pi x : C' \,.\, D'$ by the Conv rule. □

We also need the following technical lemma.[1]

[1] This technical lemma is needed for the Lam case where the Π premise is not present in the conclusion. The lemma is similar to Lemma 1.2.32, p. 71 of Lasson's thesis [Las12].

Lemma 8.4.21. *If* $\Gamma \vdash A \Rightarrow s$ *and* $\Gamma \vdash A' \Rightarrow s'$ *and* $A' \preceq A$ *then* $\exists A'', s''$ *such that* $A' \longrightarrow^* A''$ *and* $\Gamma \vdash A'' \Rightarrow s''$ *and* $s'' \preceq s$.

Proof. By Lemma 8.1.4, there are two cases to consider. The details of the proof can be found in Appendix A.3. □

We can finally prove completeness using the following lemma.

Lemma 8.4.22. *If* $\Gamma \vdash M : C$ *then* $\exists A$ *such that* $\Gamma \vdash M \Rightarrow A$ *and* $A \preceq C$. *If* $\Gamma \vdash \mathrm{WF}$ *then* $\Gamma \vdash\Rightarrow \mathrm{WF}$.

Proof. By induction on the typing derivation. The details of the proof can be found in Appendix A.3. □

Corollary 8.4.23 (Completeness)**.** *If* $\Gamma \models M \Rightarrow A$ *then* $\Gamma \vdash M \Rightarrow A$. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash M \Leftarrow A$.

Proof. If $\Gamma \models M \Rightarrow A$ then by Lemma 8.4.22, $\exists A'$ such that $\Gamma \vdash M \Rightarrow A'$ and $A' \preceq A$. By soundness (Lemma 8.4.18), we have $\Gamma \vdash M : A'$. By principality, we have $A \preceq A'$ and therefore $A' \equiv A$. If $A = s$ for some $s \in \mathcal{S}$ then $\Gamma \vdash M \Rightarrow A$ by the Conv-sort rule. Otherwise, $\Gamma \vdash A : s$ for some $s \in \mathcal{S}$. By Lemma 8.4.22 and Lemma 8.4.19, $\exists s'$ such that $\Gamma \vdash A \Rightarrow s'$. Therefore, $\Gamma \vdash M \Rightarrow A$ by the Conv rule. The proof of the second statement is similar. □

Theorem 8.4.24 (Soundness and completeness of minimal typing)**.** *We have:*

- $\Gamma \vdash M : A \iff \Gamma \vdash M \Leftarrow A$,
- $\Gamma \models M \Rightarrow A \iff \Gamma \vdash M \Rightarrow A$,
- $\Gamma \vdash \mathrm{WF} \iff \Gamma \vdash\Rightarrow \mathrm{WF}$.

Proof. By combining Lemma 8.4.18 and Corollary 8.4.23. □

8.5 Examples of CTSs with principal typing

The following important systems can be described as cumulative type systems [Las12] .

- Given an ordinal α, the CTS $\lambda P^{\alpha}_{\preceq}$ is given by the specification:

$$\left(P^{\alpha}_{\preceq}\right) \begin{bmatrix} \mathcal{S} &= \{i \mid i < \alpha\} \\ \mathcal{A} &= \{(i, i+1) \mid i+1 < \alpha\} \\ \mathcal{R} &= \{(i, i, i) \mid i < \alpha\} \\ \mathcal{C} &= \{(i, i+1) \mid i+1 < \alpha\} \end{bmatrix}$$

It satisfies the following properties:

- principal typing,
- fullness,
- strengthening,
- predicativity,
- strong normalization,
- decidability of type-checking.

The core of **intuitionistic type theory** (ITT) corresponds to the instance $\lambda P^{\omega}_{\preceq}$ where $\alpha = \omega$, which is moreover complete. Lasson [Las12] showed that, for any *predicative* CTS $\lambda\mathcal{P}_{\preceq}$, there is an ordinal α and a morphism from $\lambda\mathcal{P}_{\preceq}$ to $\lambda P^{\alpha}_{\preceq}$. Since morphisms preserve divergence (Corollary 8.3.7), this shows that any such $\lambda\mathcal{P}_{\preceq}$ is also strongly normalizing.

- Given an ordinal α, the CTS $\lambda C^{\alpha}_{\preceq}$ is given by the specification:

$$\lambda C^{\alpha} \begin{bmatrix} \mathcal{S} &= \{*\} \cup \{i \mid i < \alpha\} \\ \mathcal{A} &= (*, 0) \cup \{(i, i+1) \mid i+1 < \alpha\} \\ \mathcal{R} &= \{(*, *, *)\} \cup \\ & \quad \{(i, *, *) \mid i < \alpha\} \cup \\ & \quad \{(i, i, i) \mid i < \alpha\} \\ \mathcal{C} &= \{(*, 0)\} \cup \{(i, i+1) \mid i+1 < \alpha\} \end{bmatrix}$$

It satisfies the following properties:

- principal typing,
- fullness,
- strengthening,
- weak impredicativity,
- strong normalization,
- decidability of type-checking.

The core of the **calculus of inductive constructions** (CIC) corresponds to the instance $\lambda C^{\omega}_{\preceq}$ where $\alpha = \omega$, which is moreover complete. Lasson [Las12] showed that, for any *weakly impredicative* CTS $\lambda\mathcal{P}_{\preceq}$, there is an ordinal α and a morphism from $\lambda P_{\preceq}$ to $\lambda C^{\alpha}_{\preceq}$. Since morphisms preserve divergence (Corollary 8.3.7), this shows that any such $\lambda\mathcal{P}_{\preceq}$ is also strongly normalizing.

9

Embedding cumulativity

In this chapter, we generalize the embedding of Cousineau and Dowek [CD07] presented in Chapter 5 to cumulative pure type systems. This turns out to be surprisingly challenging. As we saw in the previous chapter, adding cumulativity breaks several good properties such as uniqueness of types. While we were able to formalize a notion of minimal types, they are not preserved by reduction (see Example 8.4.5). Since the $\lambda\Pi$-calculus modulo rewriting satisfies both uniqueness of types (Theorem 3.2.10) and subject reduction (Theorem 3.2.9), we must find a way to represent the same term at different types. A trivial solution is to rewrite Type_{i+1} to Type_i, however that would collapse the universe hierarchy and lead to an inconsistent system (see $\lambda*$ in Section 4.2).

We treat cumulativity using explicit coercions $\uparrow$ that transport terms from one universe to a larger one. For each $s_1 \preceq s_2$, we introduce a coercion $\uparrow_{s_1}^{s_2}$ of type $\mathsf{U}_{s_1} \to \mathsf{U}_{s_2}$. These coercions are very similar to the t operators in Martin-Löf's intuitionistic type theory [ML73, ML84, Pal98] with universes *à la Tarski*:

$$\frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash \mathsf{t}_i\,(A) : \mathsf{U}_{i+1}}\ ,$$

except that we represent them as constants in curried form. Adding the equations

$$\mathsf{T}_{s_2}\,(\uparrow_{s_1}^{s_2}\ a) \equiv \mathsf{T}_{s_1}\,a$$

then allows us to correctly interpret coerced terms as types.

The main difficulty lies in trying to maintain the same expressivity as universes *à la Russell*. It is commonly believed that the two styles are equivalent. However, this belief is not well-founded, as we show that naive formulations of universes à la Tarski are not equivalent to formulations à la Russell in the presence of cumulativity. The reason is that introducing explicit coercions in a dependently-typed setting can interfere with convertibility checks, as we illustrate in the following example (for conciseness, we sometimes abbreviate Type_i as i).

Example 9.0.1 (Naive formulations of universes in the Tarski style are not equivalent to the Russell style)**.** In the context

$$\begin{aligned}\Gamma = {} & p : \mathsf{Type}_1 \to \mathsf{Type}_1, \\ & q : \mathsf{Type}_1 \to \mathsf{Type}_1, \\ & f : \Pi c : \mathsf{Type}_0 \,.\, p\, c \to q\, c, \\ & g : \Pi a : \mathsf{Type}_1 \,.\, \Pi b : \mathsf{Type}_1 \,.\, p\,(\Pi x : a \,.\, b) \\ & a : \mathsf{Type}_0, \\ & b : \mathsf{Type}_0,\end{aligned}$$

the term $f\,(\Pi x : a \,.\, b)\ (g\, a\, b)$ has type $q\,(\Pi x : a \,.\, b)$:

$$\Gamma \vdash f\,(\Pi x : a \,.\, b)\ (g\, a\, b) : q\,(\Pi x : a \,.\, b)$$

but the corresponding Tarski-style translation

$$f\,(\pi_{0,0}\, a\,(\lambda x \,.\, b))\ \left(g\left(\uparrow^1_0\ a\right)\ \left(\uparrow^1_0\ b\right)\right)$$

is ill-typed because $\mathsf{T}_1\,(p\,(\pi_{1,1}\,(\uparrow^1_0\ a)\ (\lambda x \,.\, \uparrow^1_0\ b))) \not\equiv \mathsf{T}_1\,(p\,(\uparrow^1_0\ (\pi_{0,0}\, a\,(\lambda x \,.\, b))))$. The type corresponding to $q\,(\Pi x : a \,.\, b)$ is not provable in the Tarski style without adding further equations!

This result, previously unkown, shows that we need to be careful when adding coercions if we want to preserve the expressivity of the systems we are embedding. After some investigation, it turns out that the issue arises when types, because of cumulativity, have multiple non-equivalent representations as terms in the same universe. This leads us to add equations for guaranteeing the unique representation of types as terms, a property also known as *full reflection* [ML84, Pal98] because it reflects the equality of types in their term representation. While the equations needed for the predicative universes Type_i have been known for some time [Luo94, Pal98], the equations needed for impredicative universes such as Prop are less obvious and have not been studied before. In this chapter, we present a general embedding of cumulative type systems with principal types, and show precisely which equations are needed to ensure full reflection. The ideas of this chapter were the subject of a publication at TYPES 2014 [Ass14] although under a different context, that of a *calculus of constructions with explicit subtyping*.

(s_1, s_2, s_3)	(s'_1, s'_2, s'_3)	Conditions
$(\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop})$	$\left(\mathsf{Type}_j, \mathsf{Type}_j, \mathsf{Type}_j\right)$	
$(\mathsf{Type}_i, \mathsf{Prop}, \mathsf{Prop})$	$\left(\mathsf{Type}_j, \mathsf{Prop}, \mathsf{Prop}\right)$	$i \leq j$
$(\mathsf{Type}_i, \mathsf{Prop}, \mathsf{Prop})$	$\left(\mathsf{Type}_j, \mathsf{Type}_j, \mathsf{Type}_j\right)$	$i \leq j$
$(\mathsf{Type}_i, \mathsf{Type}_i, \mathsf{Type}_i)$	$\left(\mathsf{Type}_j, \mathsf{Type}_j, \mathsf{Type}_j\right)$	$i \leq j$

Table 9.1 – Rule inclusions for the system $\lambda C^{\omega}_{\preceq}$

9.1 Translation

As usual, for a given CTS $\mathcal{P}$, we define a signature $\Sigma_{\mathcal{P}}$ containing the constants and rewrite rules needed to represent $\mathcal{P}$, and then define a translation of the terms and types into this signature. As mentioned above, we need the equivalence relation $\equiv_{\beta\Sigma}$ to ensure that terms representing the same type are unique in each universe. This is not always evident or practical to achieve. In this chapter, we only give the necessary equations and we defer their presentation as rewrite rules to the next chapter. First, we define the following relation on CTS rules that are useful for defining these equations.

Definition 9.1.1. The *rule subtyping relation* is the extension of the subtyping relation defined as:

$$(s_1, s_2, s_3) \preceq (s'_1, s'_2, s'_3) \overset{\text{def}}{\iff} s_1 \preceq s'_1 \wedge s_2 \preceq s'_2 \wedge s_3 \preceq s'_3.$$

As an example, the (strict) rule subtyping pairs for $\lambda C^{\omega}_{\preceq}$ are summarized in Table 9.1.

Definition 9.1.2 (Signature)**.** Let $\Sigma_{\mathcal{P}}$ be the context containing

$$\begin{array}{lll}
\mathsf{U}_s & : \mathsf{Type} & \forall s \in \mathcal{S} \\
\mathsf{T}_s & : \mathsf{U}_s \to \mathsf{Type} & \forall s \in \mathcal{S} \\
\mathsf{u}_{s_1} & : \mathsf{U}_{s_2} & \forall\, (s_1, s_2) \in \underline{\mathcal{A}} \\
\uparrow^{s_2}_{s_1} & : \mathsf{U}_{s_1} \to \mathsf{U}_{s_2} & \forall\, (s_1, s_2) \in \mathcal{C}^* \\
\pi_{s_1,s_2} & : \Pi\alpha : \mathsf{U}_{s_1} \,.\, (\mathsf{T}_{s_1}\, \alpha \to \mathsf{U}_{s_2}) \to \mathsf{U}_{s_3} & \forall\, (s_1, s_2, s_3) \in \underline{\mathcal{R}}
\end{array}$$

We assume $\Sigma_{\mathcal{P}}$ also contains rewrite rules verifying the equations

$$\begin{array}{lll}
\mathsf{T}_{s_2}\, \mathsf{u}_{s_1} & \equiv \mathsf{U}_{s_1} & \forall\, (s_1, s_2) \in \underline{\mathcal{A}} \\
\mathsf{T}_{s_2}\, (\uparrow^{s_2}_{s_1}\ a) & \equiv \mathsf{T}_{s_1}\, a & \forall\, (s_1, s_2) \in \mathcal{C}^* \\
\mathsf{T}_{s_3}\, (\pi_{s_1,s_2}\, a\, b) & \equiv \Pi x : \mathsf{T}_{s_1}\, a \,.\, \mathsf{T}_{s_2}\, (b\, x) & \forall\, (s_1, s_2, s_3) \in \underline{\mathcal{R}}
\end{array}$$

and the following *full reflection* equations

$$
\begin{aligned}
\uparrow^{s}_{s}\ a &\equiv a \\
\uparrow^{s_3}_{s_2}\ (\uparrow^{s_2}_{s_1}\ a) &\equiv \uparrow^{s_3}_{s_1}\ a \\
\uparrow^{s'_3}_{s_3}\ (\pi_{s_1,s_2}\, a\, b) &\equiv \pi_{s'_1,s'_2}\left(\uparrow^{s'_1}_{s_1}\ a\right)\left(\lambda x\,.\,\uparrow^{s'_2}_{s_2}\ (b\, x)\right) \\
&\qquad \forall\,(s_1,s_2,s_3)\in\mathcal{R},(s'_1,s'_2,s'_3)\in\mathcal{R} \\
&\qquad\quad (s_1,s_2,s_3)\preceq(s'_1,s'_2,s'_3)\,.
\end{aligned}
\tag{9.1}
$$

We sometimes write Σ instead of $\Sigma_{\mathcal{P}}$ when it is not ambiguous.

Because a term can have multiple types, we need a way to translate it in multiple ways. For this reason, we mutually define two translation functions: a *term translation* that relies on minimal typing and translates a term according to its minimal type, and a *cast translation* that lifts that to the appropriate type using coercions.

Definition 9.1.3 (Translation functions)**.** For any Γ, M, A such that $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow A$, the *term translation* $[M]_\Gamma$ is defined by induction on M as:

$$
\begin{aligned}
[x]_\Gamma &= x \\
[s_1]_\Gamma &= \mathsf{u}_{s_1} && \text{where } (s_1,s_2)\in\underline{\mathcal{A}} \\
[M\,N]_\Gamma &= [M]_\Gamma\ [N]_{\Gamma\vdash A} && \text{where } \Gamma\vdash M\Rightarrow \Pi x:A\,.\,B \\
[\lambda x:A\,.\,M]_\Gamma &= \lambda x:[\![A]\!]\,.\,[M]_{\Gamma,x:A} \\
[\Pi x:A\,.\,B]_\Gamma &= \pi_{s_1,s_2}\,[A]_\Gamma\left(\lambda x\,.\,[B]_{\Gamma,x:A}\right) && \text{where } \Gamma\vdash A\Rightarrow s_1 \\
& && \text{and } \Gamma\vdash B\Rightarrow s_2 \\
& && \text{and } (s_1,s_2,s_3)\in\underline{\mathcal{R}}.
\end{aligned}
$$

For any Γ, M, C such that $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M : C$, the *cast translation* $[M]_{\Gamma\vdash C}$ is defined as:

$$
\begin{aligned}
[M]_{\Gamma\vdash C} &= [M]_\Gamma && \text{if } \Gamma\vdash M\Rightarrow A \text{ and } A\equiv C \\
[M]_{\Gamma\vdash C} &= \uparrow^{s_2}_{s_1}\ [M]_s && \text{if } \Gamma\vdash M\Rightarrow A \text{ and } A\equiv s_1 \\
& && \text{and } C\equiv s_2 \text{ and } s_1\prec s_2 \\
[M]_{\Gamma\vdash C} &= \lambda x:[\![A]\!]\,.\,[M\,x]_{\Gamma,x:A\vdash B} && \text{if } \Gamma\vdash M\Rightarrow A \text{ and } A\equiv \Pi x:A_1\,.\,B_1 \\
& && \text{and } C\equiv \Pi x:A_1\,.\,B'_1 \text{ and } B_1\prec B'_1.
\end{aligned}
$$

For any Γ, A such that $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} A$ WF, the *type translation* $[\![A]\!]_\Gamma$ is defined as:

$$
\begin{aligned}
[\![A]\!]_\Gamma &= \mathsf{T}_s\,[A]_\Gamma && \text{if } \Gamma\vdash A:s \\
[\![s]\!]_\Gamma &= \mathsf{U}_s && \text{if } s\in\mathcal{S}^{\top}.
\end{aligned}
$$

For any Γ such that $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}}$ WF, the *context translation* $[\![\Gamma]\!]$ is defined by induction on Γ as:

$$
\begin{aligned}
[\![\varnothing]\!] &= \varnothing \\
[\![\Gamma, x:A]\!] &= [\![\Gamma]\!]\,, x:[\![A]\!]_\Gamma\,.
\end{aligned}
$$

9.2 Completeness

Once again, we prove that the translation is complete, meaning that it preserves typing. This property depends on the preservation of equivalence, which in turn depends on the preservation of substitution.

9.2.1 Preservation of substitution

Since minimal types are not preserved by substitution (see Example 8.4.5), we must be careful in the formulation of the substitution lemma. In particular, we must use the cast translation $[M]_{\vdash A}$ instead of the term translation $[M]$ when the type can decrease. We first relate substitution in the cast translation and in the type translation to substitution in the term translation using the following two lemmas.

Lemma 9.2.1. *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} B \Rightarrow s$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N : A$ *then*

$$[B]_{\Gamma, x:A, \Gamma'} \left\{ x \backslash [N]_{\Gamma \vdash A} \right\} \equiv_{\beta\Sigma} [B \{x \backslash N\}]_{\Gamma, \Gamma' \{x \backslash N\} \vdash s}$$

implies

$$[\![B]\!]_{\Gamma, x:A, \Gamma'} \left\{ x \backslash [N]_{\Gamma \vdash A} \right\} \equiv_{\beta\Sigma} [\![B \{x \backslash N\}]\!]_{\Gamma, \Gamma' \{x \backslash N\}} .$$

Proof. We have $[\![B]\!] \left\{ x \backslash [N]_{\vdash A} \right\} = \mathsf{T}_s [B] \left\{ x \backslash [N]_{\vdash A} \right\} \equiv \mathsf{T}_s [B \{x \backslash N\}]_{\vdash s}$. If $\Gamma \vdash B \{x \backslash N\} \Rightarrow s$ then so $\mathsf{T}_s [B \{x \backslash N\}]_{\vdash s} = \mathsf{T}_s [B \{x \backslash N\}] = [\![B \{x \backslash N\}]\!]$. Otherwise, $\Gamma \vdash B \{x \backslash N\} \Rightarrow s'$ for some s' such that $s' \prec s$. Then $\mathsf{T}_s [B \{x \backslash N\}]_{\vdash s} = \mathsf{T}_s (\uparrow^s_{s'} [B \{x \backslash N\}]) \equiv \mathsf{T}_{s'} [B \{x \backslash N\}]$. □

Lemma 9.2.2. *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow B$ *and* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} M \Leftarrow C$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N \Leftarrow A$ *then*

$$[M]_{\Gamma, x:A, \Gamma'} \left\{ x \backslash [N]_{\Gamma \vdash A} \right\} \equiv_{\beta\Sigma} [M \{x \backslash N\}]_{\Gamma, \Gamma' \{x \backslash N\} \vdash B\{x \backslash N\}}$$

implies

$$[M]_{\Gamma, x:A, \Gamma' \vdash C} \left\{ x \backslash [N]_{\Gamma \vdash A} \right\} \equiv_{\beta\Sigma} [M \{x \backslash N\}]_{\Gamma, \Gamma' \{x \backslash N\} \vdash C\{x \backslash N\}} .$$

Proof. Since B is the minimal type of M, we have $B \preceq C$. If $B \equiv C$ then

$$\begin{aligned} [M]_{\vdash C} \left\{ x \backslash [N]_{\vdash A} \right\} &= [M] \left\{ x \backslash [N]_{\Gamma \vdash A} \right\} \\ &\equiv [M \{x \backslash N\}]_{\vdash B\{x \backslash N\}} \\ &\equiv [M \{x \backslash N\}]_{\vdash C\{x \backslash N\}} . \end{aligned}$$

Otherwise, $\exists D_1, \ldots, D_n, s, s'$ such that $B \equiv \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . s$ and $C \equiv \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . s'$ and $s \prec s'$. Then $B \{x \backslash N\} \equiv \Pi x_1 : D'_1 . \cdots \Pi x_n : D'_n . s$ and

$C\{x\backslash N\} \equiv \Pi x_1 : D'_1 . \cdots \Pi x_n : D'_n . s'$ where $D'_i = D_i\{x\backslash N\}$. Therefore,

$$\begin{aligned}
[M]_{\vdash C}\left\{x\backslash [N]_{\vdash A}\right\} &= \left(\lambda x_1 : [\![D_1]\!] . \cdots \lambda x_n : [\![D_n]\!] . \uparrow_s^{s'} \ ([M]\ x_1 \ldots x_n)\right)\left\{x\backslash [N]_{\vdash A}\right\} \\
&= \lambda x_1 : [\![D'_1]\!] . \cdots \lambda x_n : [\![D'_n]\!] . \uparrow_s^{s'} \ \left([M]\left\{x\backslash [N]_{\vdash A}\right\}\ x_1 \ldots x_n\right) \\
&\equiv \lambda x_1 : [\![D'_1]\!] . \cdots \lambda x_n : [\![D'_n]\!] . \uparrow_s^{s'} \ \left([M\{x\backslash N\}]_{\vdash B\{x\backslash N\}}\ x_1 \ldots x_n\right) \\
&\equiv [M\{x\backslash N\}]_{C\{x\backslash N\}} .
\end{aligned}$$

□

The following key lemma shows that the order in which casts are performed does not matter. It is crucial for the preservation of substitution. It depends directly on the uniqueness of names equations.

Lemma 9.2.3. *We have:*

- *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} A \Rightarrow s_1$ *and* $\Gamma, x : A \vdash_{\lambda\mathcal{P}_{\preceq}} B \Rightarrow s_2$ *and* $(s_1, s_2, s_3) \in \underline{\mathcal{R}}$ *then*

$$\pi_{s_1,s_2}\,[A]_{\Gamma\vdash s_1}\left(\lambda x . [B]_{\Gamma,x:A\vdash s_2}\right) \equiv_{\beta\Sigma} [\Pi x : A . B]_{\Gamma\vdash s_3} .$$

- *If* $\Gamma, x : A \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow B$ *and* $\Gamma \vdash \Pi x : A . B \Rightarrow s$ *then*

$$\lambda y : [\![A]\!] . [M]_{\Gamma,x:A\vdash B} \equiv_{\beta\Sigma} [\lambda y : A . M]_{\Gamma\vdash \Pi y:A.B} .$$

- *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow \Pi x : A . B$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N \Leftarrow A$ *then*

$$[M]_{\vdash \Pi x:A.B}\ [N]_{\Gamma\vdash A} \equiv [M\,N]_{\Gamma\vdash B\{x\backslash N\}} .$$

Proof. By definition of the cast translation $[M]_{\Gamma\vdash A}$ and using the equations in Σ. Note that this proposition would *not* be true if $\equiv_{\beta\Sigma}$ did not satisfy full reflection. In particular, the first statement depends directly on Equation (9.1). □

We can now prove the preservation of substitution by the translation.

Lemma 9.2.4 (Substitution)**.** *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow B$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N \Leftarrow A$ *then* $[M]\left\{x\backslash [N]_{\vdash A}\right\} \equiv_{\beta\Sigma} [M\{x\backslash N\}]_{\vdash B\{x\backslash N\}}$. *More precisely,*

$$[M]_{\Gamma,x:A,\Gamma'}\left\{x\backslash [N]_{\Gamma\vdash A}\right\} \equiv_{\beta\Sigma} [M\{x\backslash N\}]_{\Gamma,\Gamma'\{x\backslash N\}\vdash B\{x\backslash N\}} .$$

Proof. First note that the statement makes sense because $\Gamma, \Gamma'\{x\backslash N\} \vdash M\{x\backslash N\} : B\{x\backslash N\}$ by 8.2.4. The proof follows by induction on M. The details of the proof can be found in Appendix A.4. □

Corollary 9.2.5. *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} B$ WF *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N \Leftarrow A$ *then* $[\![B\{x\backslash N\}]\!] \equiv [\![B]\!]\left\{x\backslash [N]_{\vdash A}\right\}$. *More precisely,*

$$[\![B]\!]_{\Gamma,x:A,\Gamma'}\left\{x\backslash [N]_{\Gamma\vdash A}\right\} \equiv [\![B\{x\backslash N\}]\!]_{\Gamma,\Gamma'\{x\backslash N\}}.$$

Proof. If $B = s$ for some $s \in \mathcal{S}^{\top}$ then $[\![B\{x\backslash N\}]\!] = [\![s]\!] = s = s\{x\backslash [N]\} = [\![B]\!]\{x\backslash [N]\}$. Otherwise, $\Gamma \vdash B \Rightarrow s$ for some $s \in \mathcal{S}$. Then $[\![B\{x\backslash N\}]\!] = \mathsf{T}_s[B\{x\backslash N\}] = \mathsf{T}_s[B]\{x\backslash [N]\} = [\![B]\!]\{x\backslash [N]\}$. □

Example 9.2.6. For Example 8.4.5, we have

$$\begin{aligned}
[x]_{x:\mathsf{Type}_2}\left\{x\backslash [\mathsf{Type}_0]_{\vdash \mathsf{Type}_2}\right\} &= x\left\{x\backslash \uparrow^{\mathsf{Type}_2}_{\mathsf{Type}_1} \mathsf{u}_{\mathsf{Type}_0}\right\} \\
&= \uparrow^{\mathsf{Type}_2}_{\mathsf{Type}_1} \mathsf{u}_{\mathsf{Type}_0} \\
&= [\mathsf{Type}_0]_{\vdash \mathsf{Type}_2} \\
&= [x\{x\backslash \mathsf{Type}_0\}]_{\vdash \mathsf{Type}_2\{x\backslash \mathsf{Type}_0\}}.
\end{aligned}$$

9.2.2 Preservation of equivalence

Lemma 9.2.7 (Preservation of single-step reduction)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow A$ *and* $M \longrightarrow_{\beta} M'$ *then* $[M]_{\Gamma} \equiv_{\beta\Sigma} [M']_{\Gamma\vdash A}$.

Proof. First note that the statement makes sense because $\Gamma \vdash M' \Leftarrow A$ by subject reduction (8.2.8). The proof follows by induction on M, using Lemma 9.2.4 for the base case. □

Lemma 9.2.8 (Preservation of multi-step reduction)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow A$ *and* $M \longrightarrow_{\beta}^{*} M'$ *then* $[M]_{\Gamma} \equiv_{\beta\Sigma} [M']_{\Gamma\vdash A}$.

Proof. By induction on the number of steps, using Lemma 9.2.7. □

Lemma 9.2.9 (Preservation of equivalence)**.** *If* A *and* B *are well-formed types in* Γ *and* $A \equiv_{\beta} B$ *then* $[\![A]\!]_{\Gamma} \equiv_{\beta\Sigma} [\![B]\!]_{\Gamma}$.

Proof. If $A = s$ for some $\mathcal{S}^{\top}$ then we must have $B = s$, and $[\![A]\!] = \mathsf{U}_s = [\![B]\!]$. Similarly, if $B = s$ for some $s \in \mathcal{S}^{\top}$ then $A = s$ and $[\![A]\!] = \mathsf{U}_s = [\![B]\!]$. Otherwise, we have $\Gamma \vdash A \Rightarrow s$ and $\Gamma \vdash B \Rightarrow s'$ for some $s, s' \in \mathcal{S}$. By confluence, there exists C such that $A \longrightarrow^{*} C$ and $B \longrightarrow^{*} C$. By subject reduction, $\Gamma \vdash C \Leftarrow s$ and $\Gamma \vdash C \Leftarrow s'$. By the existence of principal types, $\exists s''$ such that $\Gamma \vdash C \Rightarrow s''$ and $s'' \preceq s$ and $s'' \preceq s'$. By Lemma 9.2.8, $[A] \equiv [C]_{\vdash s}$. Therefore,

$$\begin{aligned}
[\![A]\!] &= \mathsf{T}_s[A] \\
&\equiv \mathsf{T}_s[C]_{\vdash s} \\
&\equiv \mathsf{T}_s(\uparrow^{s}_{s''} [C]) \\
&\equiv \mathsf{T}_{s''}[C] \\
&= [\![C]\!].
\end{aligned}$$

Similarly, $[B] \equiv [C]_{\vdash s'}$, so $[\![B]\!] \equiv [\![C]\!]$. Therefore, $[\![A]\!] \equiv [\![B]\!]$. □

9.2.3 Preservation of typing

Once again we need the following technical lemmas. The proofs are similar to those of Lemmas 5.2.7 and 5.2.9.

Lemma 9.2.10. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} s \Rightarrow \mathrm{WF}$ *then* $[\![s]\!]_\Gamma \equiv_{\beta\Sigma} \mathsf{U}_s$ *and* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![s]\!] : \mathsf{Type} \iff \Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} \mathsf{U}_s : \mathsf{Type}$.

Lemma 9.2.11. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} \Pi x : A \,.\, B \Rightarrow \mathrm{WF}$ *then* $[\![\Pi x : A \,.\, B]\!] \equiv \Pi x : [\![A]\!] \,.\, [\![B]\!]$ *and* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![\Pi x : A \,.\, B]\!] : \mathsf{Type} \iff \Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} \Pi x : [\![A]\!] \,.\, [\![B]\!] : \mathsf{Type}$.

Lemma 9.2.12. *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} B \Rightarrow \mathrm{WF}$ *and* $\Gamma \vdash_{\lambda N_{\preceq}} \Leftarrow A$ *then*

$$\Sigma, [\![\Gamma]\!], x : [\![A]\!]_\Gamma, [\![\Gamma']\!]_{\Gamma, x:A} \vdash_{\lambda\Pi R} [\![B]\!]_{\Gamma, x:A, \Gamma'} \left\{x \backslash [N]_{\Gamma \vdash A}\right\} : \mathsf{Type}$$

if and only if

$$\Sigma, [\![\Gamma]\!], x : [\![A]\!]_\Gamma, [\![\Gamma']\!]_{\Gamma, x:A} \vdash_{\lambda\Pi R} [\![B\,\{x \backslash N\}]\!]_{\Gamma, \Gamma'\{x \backslash N\}} : \mathsf{Type}.$$

The following lemmas link the term translation to the cast translation and type translation respectively.

Lemma 9.2.13. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow A$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} B \Rightarrow \mathrm{WF}$ *and* $A \preceq B$ *and* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_\Gamma : [\![A]\!]_\Gamma$ *and* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![B]\!]_\Gamma : \mathsf{Type}$ *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_{\Gamma \vdash B} : [\![B]\!]_\Gamma$.

Proof. If $A \equiv B$ then $[M]_{\vdash B} = [M]$. By Lemma 9.2.9, $[\![A]\!] \equiv [\![B]\!]$, and by conversion, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![B]\!]$. Otherwise, $\exists C_1, \ldots C_n, s, s'$ such that $A \equiv \Pi x_1 : C_1 \ldots . \Pi x_n : C_n \,.\, s$ and $B \equiv \Pi x_1 : C_1 \ldots . \Pi x_n : C_n \,.\, s'$ and $s \prec s'$ and $\Sigma, [\![\Gamma]\!] \vdash [M] : \Pi x_1 : [\![C_1]\!] \ldots . \Pi x_n : [\![C_n]\!] \,.\, \mathsf{U}_s$. We have $[M]_{\vdash B} = \lambda x_1 : [\![C_1]\!] \ldots . \lambda x_n : [\![C_n]\!] \,.\, \uparrow_s^{s'} ([M]\ x_1 \cdots x_n)$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash [M]_{\vdash B} : \Pi x_1 : [\![C_1]\!] \ldots . \Pi x_n : [\![C_n]\!] \,.\, \mathsf{U}_{s'}$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash [M]_{\vdash B} : [\![B]\!]$. □

Lemma 9.2.14. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} A \Rightarrow s$ *and* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [A]_\Gamma : [\![s]\!]$ *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} [\![A]\!]_\Gamma : \mathsf{Type}$.

Proof. By Lemma 9.2.10, $[\![\Gamma]\!] \vdash [A]_\Gamma : \mathsf{U}_S$. Therefore, $[\![\Gamma]\!] \vdash \mathsf{T}_s\, [A]_\Gamma : \mathsf{Type}$. □

We can now prove the main lemma for the preservation of typing.

Lemma 9.2.15. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_\Gamma : [\![A]\!]_\Gamma$. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M \Leftarrow A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_{\Gamma \vdash A} : [\![A]\!]$. *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} \Rightarrow \mathrm{WF}$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} \mathrm{WF}$.

Proof. By induction on the typing derivations. The details of the proofs can be found in Appendix A.4. □

Theorem 9.2.16 (Preservation of typing)**.** *If* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} M : C$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_{\Gamma \vdash C} : [\![C]\!]_\Gamma$.

Proof. By Corollary 8.4.23, $\Gamma \vdash M \Leftarrow C$. By Lemma 9.2.15, $\Sigma, [\![\Gamma]\!] \vdash [M]_{\vdash C} : [\![C]\!]$. □

10

Infinite universe hierarchies

In practice, the systems we consider have an *infinite* hierarchy of universes, like systems $\lambda P^{\omega}_{\preceq}$ and $\lambda C^{\omega}_{\preceq}$ from Section 8.5. To embed infinite hierarchies in $\lambda\Pi R$, we cannot rely on the embeddings of chapters 5 and 9, because that would lead to an infinite signature Σ. To deal with this, we could examine which universes are needed in a proof—by Lemma 8.3.3 (Compactness), there must be a finite number of them—and content ourselves with just those universes. For example, it is common lore that most of the library of Coq needs only 2 universes [ARCT09]. However, this solution is not very modular because we would need a different signature for each proof. Another solution is to encode this infinite hierarchy, which is what we will do here.

While encoding an infinite hierarchy in a finite signature is relatively straightforward, it is much more difficult to do so in the presence of cumulativity. In particular, equations for guaranteeing *full reflection* (Definition 9.1.2) are surprisingly difficult to express in a finite, confluent, and terminating rewrite system (and usual completion techniques such as the Knuth–Bendix method fail here). It is not entirely impossible though and there are various ways to achieve this, in a more or less satisfying way. We propose a solution that works in practice. We first consider the non-cumulative systems λP^{∞} and λC^{∞} to give an idea on how to encode the infinite hierarchy. We then show how to incorporate cumulativity and give the signature that we used in our implementations.

We think this chapter is necessary for someone who wants to replicate our efforts. We will only focus on giving the finite signatures. The definitions of translations should be fairly obvious. We do not prove the soundness of the associated embeddings. We conjecture that the proof of soundness of Chapter 6 and the proof of completeness of Chapter 9 can be adapted without much trouble.

10.1 Predicative universes

Consider the PTS λP^∞ given by the specification

$$(P^\infty)\begin{bmatrix}\mathcal{S} &= \mathbb{N} \\ \mathcal{A} &= \{(i, i+1) \mid i \in \mathbb{N}\} \\ \mathcal{R} &= \{(i, j, \max(i, j)) \mid i, j \in \mathbb{N}\}\end{bmatrix}$$

which is a subset of intuitionistic type theory. It is functional but has an infinite number of sorts. First, we declare the type of natural numbers, that we represent using constants for zero and the successor function, and the maximum function:

$$\begin{aligned}
&\mathsf{nat} &&: \mathsf{Type}, \\
&\mathsf{zero} &&: \mathsf{nat}, \\
&\mathsf{succ} &&: \mathsf{nat} \to \mathsf{nat}, \\
& && \\
&\mathsf{max} &&: \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}, \\
&\mathsf{max}\, i\, \mathsf{zero} &&\longmapsto i, \\
&\mathsf{max}\, \mathsf{zero}\, j &&\longmapsto j, \\
&\mathsf{max}\, (\mathsf{succ}\, i)\ (\mathsf{succ}\, j) &&\longmapsto \mathsf{succ}\, (\mathsf{max}\, i\, j)\,.
\end{aligned}$$

Instead of indexing the constants $\mathsf{U}, \mathsf{T}, \mathsf{u}, \pi$ by natural numbers *externally* as we did before, we index them *internally* by terms of type nat:

$$\begin{aligned}
&\mathsf{U} &&: \mathsf{nat} \to \mathsf{Type}, \\
&\mathsf{T} &&: \Pi i : \mathsf{nat}\,.\, \mathsf{U}\, i \to \mathsf{Type}, \\
&\mathsf{u} &&: \Pi i : \mathsf{nat}\,.\, \mathsf{U}\, (\mathsf{succ}\, i)\,, \\
&\pi &&: \Pi i : \mathsf{nat}\,.\, \Pi j : \mathsf{nat}\,.\, \Pi a : \mathsf{U}\, i\,.\, (\mathsf{T}\, i\, a \to \mathsf{U}\, j) \to \mathsf{U}\, (\mathsf{max}\, i\, j)\,.
\end{aligned}$$

We need rewrite rules so that

$$\begin{aligned}
&\mathsf{T}\, (\mathsf{succ}\, i)\ (\mathsf{u}\, i) &&\equiv \mathsf{U}\, i, \\
&\mathsf{T}\, (\mathsf{max}\, i\, j)\ (\pi\, i\, j\, a\, b) &&\equiv \Pi x : \mathsf{T}\, i\, a\,.\, \mathsf{T}\, j\, (b\, x)\,.
\end{aligned}$$

However, the rewrite rules

$$\begin{aligned}
&\mathsf{T}\, (\mathsf{succ}\, i)\ (\mathsf{u}\, i) &&\longmapsto \mathsf{U}\, i \\
&\mathsf{T}\, (\mathsf{max}\, i\, j)\ (\pi\, i\, j\, a\, b) &&\longmapsto \Pi x : \mathsf{T}\, i\, a\,.\, \mathsf{T}\, j\, (b\, x)
\end{aligned}$$

are not confluent, because $\mathsf{max}\, i\, j$ is not a normal form on closed terms. In fact, we don't *care* about the first argument of T; we can replace it by a fresh variable:

$$\begin{aligned} \mathsf{T}\, k\, (\mathsf{u}\, i) &\longmapsto \mathsf{U}\, i, \\ \mathsf{T}\, k\, (\pi\, i\, j\, a\, b) &\longmapsto \Pi x : \mathsf{T}\, i\, a\, .\, \mathsf{T}\, j\, (b\, x)\,, \end{aligned}$$

which we will also write as

$$\begin{aligned} \mathsf{T}\, _\, (\mathsf{u}\, i) &\longmapsto \mathsf{U}\, i, \\ \mathsf{T}\, _\, (\pi\, i\, j\, a\, b) &\longmapsto \Pi x : \mathsf{T}\, i\, a\, .\, \mathsf{T}\, j\, (b\, x)\,. \end{aligned} \tag{10.1}$$

This system is now confluent. On the other hand, we cannot use Theorem 3.2.15 to show that it is well-typed because the left-hand side is not well-typed anymore. We therefore need to use another argument to convince ourselves that the rules are well-typed.

Lemma 10.1.1. *The rewrite rules in Equation* (10.1) *are well-typed.*

Proof. We prove that each of the two rules is well-typed according to Definition 3.2.5. Let Γ be a context such that $\Sigma \subseteq \Gamma$ and σ be a substitution of variables free in Γ such that:

1. $\sigma\,(\mathsf{T}\, k\, (\mathsf{u}\, i))$ is well-typed. Then we must have $\sigma\,(i) : \mathsf{nat}$, so that $\mathsf{u}\,(\sigma\,(i)) : \mathsf{U}\,(\mathsf{succ}\,(\sigma\,(i)))$. This implies $\mathsf{succ}\,(\sigma\,(i)) \equiv \sigma\,(k)$ so that $\mathsf{T}\,(\sigma\,(k))\ (\mathsf{u}\,(\sigma\,(i))) : \mathsf{Type}$. Then we also have $\mathsf{U}\,(\sigma\,(i)) : \mathsf{Type}$.

2. $\sigma\,(\mathsf{T}\, k\, (\pi\, i\, j\, a\, b))$ is well-typed. Then we must have $\sigma\,(i) : \mathsf{nat}$, $\sigma\,(j) : \mathsf{nat}$, $\sigma\,(a) : \mathsf{U}\,(\sigma\,(i))$, and $\sigma\,(b) : (\mathsf{T}\,(\sigma\,(i))\ (\sigma\,(a)) \to \mathsf{U}\,(\sigma\,(j)))$ so that

$$\pi\,(\sigma\,(i))\ (\sigma\,(j))\ (\sigma\,(a))\ (\sigma\,(b)) : \mathsf{U}\,(\mathsf{max}\,(\sigma\,(i))\ (\sigma\,(j)))\ .$$

   This implies $\mathsf{max}\,(\sigma\,(i))\ (\sigma\,(j)) \equiv \sigma\,(k)$ so that

$$\mathsf{T}\,(\sigma\,(k))\ (\pi\,(\sigma\,(i))\ (\sigma\,(j))\ (\sigma\,(a))\ (\sigma\,(b))) : \mathsf{Type}\,.$$

   Then we also have $\Pi x : \mathsf{T}\,(\sigma\,(i))\ (\sigma\,(a))\,.\,\mathsf{T}\,(\sigma\,(j))\ (\sigma\,(b)\ x) : \mathsf{Type}$.

□

Moreoever, this system behaves well with β-reduction. We sketch the proofs for this particular system to give an idea on how to prove such results.

Lemma 10.1.2. *The relation* $\longrightarrow_\Sigma$ *is terminating.*

Proof. The relation $\longrightarrow_\Sigma$ strictly decreases the number of u and π constants in the term. Therefore, it must be terminating. □

Lemma 10.1.3. *The relation* $\longrightarrow_{\beta\Sigma}$ *is confluent.*

Proof. The rewrite rules are left-linear and do not have critical pairs. Therefore, $\longrightarrow_\Sigma$ is locally confluent. By Lemma 10.1.2, it is terminating and hence confluent. Therefore, its union with $\longrightarrow_\beta$ is confluent [vO94]. □

10.2 Impredicative universe

We now add an impredicative universe $*$. Consider the PTS λC^∞ given by the specification

$$(C^\infty)\quad \begin{bmatrix} \mathcal{S} &= \{*\} \cup \mathbb{N} \\ \mathcal{A} &= \{(*,0)\} \cup \\ & \quad \{(i,i+1) \mid i \in \mathbb{N}\} \\ \mathcal{R} &= \{(*,*,*)\} \cup \\ & \quad \{(i,*,*) \mid i \in \mathbb{N}\} \cup \\ & \quad \{(*,j,j) \mid j \in \mathbb{N}\} \cup \\ & \quad \{(i,j,\max(i,j)) \mid (i,j) \in \mathbb{N}\} \end{bmatrix}$$

which is a subset of the calculus of inductive constructions. Instead of parameterizing the constants U and T by natural numbers, we now parameterize them directly by sorts.

$$\begin{aligned} \mathsf{sort} &: \mathsf{Type}, \\ \mathsf{prop} &: \mathsf{sort}, \\ \mathsf{type} &: \mathsf{nat} \to \mathsf{sort}, \\ \\ \mathsf{U} &: \mathsf{sort} \to \mathsf{Type}, \\ \mathsf{T} &: \Pi s : \mathsf{sort} \,.\, \mathsf{U}\, s \to \mathsf{Type}. \end{aligned}$$

For the constants u and π, we need to find a way to restrict them internally to the axioms $\mathcal{A}$ and rules $\mathcal{R}$ of the PTS. One way to do that would be to internalize these relations:

$$\begin{aligned} \mathsf{axiom} &: \mathsf{sort} \to \mathsf{sort} \to \mathsf{Type}, \\ \mathsf{axiom_prop} &: \mathsf{axiom}\,\mathsf{prop}\,(\mathsf{type}\,\mathsf{zero})\,, \\ \mathsf{axiom_type} &: \Pi i : \mathsf{nat} \,.\, \mathsf{axiom}\,(\mathsf{type}\,i)\ (\mathsf{type}\,(\mathsf{succ}\,i))\,, \\ \\ \mathsf{rule} &: \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort} \to \mathsf{Type}, \\ \mathsf{rule_prop_prop} &: \mathsf{rule}\,\mathsf{prop}\,\mathsf{prop}\,\mathsf{prop}, \\ \mathsf{rule_type_prop} &: \Pi i : \mathsf{nat} \,.\, \mathsf{rule}\,(\mathsf{type}\,i)\ \mathsf{prop}\,\mathsf{prop}, \\ \mathsf{rule_prop_type} &: \Pi j : \mathsf{nat} \,.\, \mathsf{rule}\,\mathsf{prop}\,(\mathsf{type}\,j)\ (\mathsf{type}\,j)\,, \\ \mathsf{rule_type_type} &: \Pi i : \mathsf{nat} \,.\, \Pi j : \mathsf{nat} \,.\, \mathsf{rule}\,(\mathsf{type}\,i)\ (\mathsf{type}\,j)\ (\mathsf{type}\,(\mathsf{max}\,i\,j))\,, \end{aligned}$$

and then use them to index u and π by inhabitants of axiom and rule:

$$\begin{aligned} \mathsf{u} &: \Pi s_1, s_2 : \mathsf{sort} \,.\, \mathsf{axiom}\, s_1\, s_2 \to \mathsf{U}\, s_2, \\ \pi &: \Pi s_1, s_2, s_3 : \mathsf{sort} \,.\, \mathsf{rule}\, s_1\, s_2\, s_3 \to \Pi a : \mathsf{U}\, s_1 \,.\, (\mathsf{T}\, s_1\, a \to \mathsf{U}\, s_2) \to \mathsf{U}\, s_3. \end{aligned}$$

Alternatively, we can take advantage of the PTS's completeness (Definition 4.1.1) to define axiom and rule as functions instead of relations, which gives a cleaner and more compact representation: [1]

$$
\begin{aligned}
&\mathsf{axiom} : \mathsf{sort} \to \mathsf{sort}, \\
&\mathsf{axiom}\,\mathsf{prop} \longmapsto \mathsf{type}\,\mathsf{zero}, \\
&\mathsf{axiom}\,(\mathsf{type}\,i) \longmapsto \mathsf{type}\,(\mathsf{succ}\,i)\,, \\
\\
&\mathsf{rule} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort}, \\
&\mathsf{rule}\,\mathsf{prop}\,\mathsf{prop} \longmapsto \mathsf{prop}, \\
&\mathsf{rule}\,(\mathsf{type}\,i)\;\mathsf{prop} \longmapsto \mathsf{prop}, \\
&\mathsf{rule}\,\mathsf{prop}\,(\mathsf{type}\,j) \longmapsto \mathsf{type}\,j, \\
&\mathsf{rule}\,(\mathsf{type}\,i)\;(\mathsf{type}\,j) \longmapsto \mathsf{type}\,(\mathsf{max}\,i\,j)\,, \\
\\
&\mathsf{u} : \Pi s_1 : \mathsf{sort}\,.\,\mathsf{U}\,(\mathsf{axiom}\,s_1)\,, \\
&\pi : \Pi s_1, s_2 : \mathsf{sort}\,.\,\Pi a : \mathsf{U}\,s_1\,.\,(\mathsf{T}\,s_1\,a \to \mathsf{U}\,s_2) \to \mathsf{U}\,(\mathsf{rule}\,s_1\,s_2)\,.
\end{aligned}
$$

The rules for T are straightforward:

$$
\begin{aligned}
&\mathsf{T}\,_\,(\mathsf{u}\,s_1) \longmapsto \mathsf{U}\,s_1, \\
&\mathsf{T}\,_\,(\pi\,s_1\,s_2\,a\,b) \longmapsto \Pi x : \mathsf{T}\,s_1\,a\,.\,\mathsf{T}\,s_2\,(b\,x)\,.
\end{aligned}
$$

10.3 Cumulativity

Finally, we consider the unholy trinity: an infinite hierarchy of predicative universes, with an impredicative universe, and cumulativity. This is embodied by the CTS $\lambda C^{\omega}_{\preceq}$, given by the specification:

$$
\left(C^{\omega}_{\preceq}\right)
\begin{bmatrix}
\mathcal{S} &= \{*\} \cup \mathbb{N} \\
\mathcal{A} &= \{(*, 0)\} \cup \\
&\quad \{(i, i+1) \mid i \in \mathbb{N}\} \\
\mathcal{R} &= \{(*, *, *)\} \cup \\
&\quad \{(i, *, *) \mid i \in \mathbb{N}\} \cup \\
&\quad \{(*, j, j) \mid j \in \mathbb{N}\} \cup \\
&\quad \{(i, j, \max(i, j)) \mid (i, j) \in \mathbb{N}\} \\
\mathcal{C} &= \{(*, 0)\} \cup \\
&\quad \{(i, i+1) \mid i \in \mathbb{N}\}
\end{bmatrix}
$$

[1] Moreover, this allows us to avoid some *proof irrelevance* issues that can arise when there are multiple derivations of the same axiom or rule.

Note that there are several variations for this system, for example with $(*, 1) \in \mathcal{A}$ instead of $(*, 0)$ (such as in the Coq system [Cdt12]), or without $(*, 0) \in \mathcal{C}$ (such as in CC^ω [HP89, HP91]), which is why we went to great lengths to give a generic presentation.

The first things we need to add are explicit coercion constants $\uparrow_{s_1}^{s_2}$, also called *lifts*, that transport terms from a lower universe U_{s_1} to a higher one U_{s_2}. However, in order to give a finite rewrite system that guarantees the *uniqueness of names* property later, we found it necessary that these coercions be able to lift several levels at once. In other words, we must have $\uparrow_{s_1}^{s_2}$ for each (s_1, s_2) in the transitive closure $\mathcal{C}^*$ instead of just $\mathcal{C}$. At the same time, we cannot allow any $(s_1, s_2) \in \mathcal{S}^2$. How can we do this with a functional approach that does not rely on an encoding of $\mathcal{C}^*$ as a relation?

In our experience, we found it easiest to define a supremum function sup that computes the maximum of s_1 and s_2 with respect to $\mathcal{C}^*$ and let $\uparrow_{s_1}^{s_2}$ transport terms from $\mathsf{U}\, s_1$ to $\mathsf{U}\,(\mathsf{sup}\, s_1\, s_2)$ (which is the same as $\mathsf{U}\, s_2$ when $s_1 \preceq s_2$). That way, we ensure that the coercion is sound for *any* pair $(s_1, s_2) \in \mathcal{S}$, and moreover it simplifies the rewriting system for full reflection. We therefore add to our signature:

$$
\begin{aligned}
&\mathsf{sup} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort}, \\
&\mathsf{sup}\,\mathsf{prop}\,\mathsf{prop} && \longmapsto \mathsf{prop} \\
&\mathsf{sup}\,(\mathsf{type}\, i)\ \mathsf{prop} && \longmapsto \mathsf{type}\, i, \\
&\mathsf{sup}\,\mathsf{prop}\,(\mathsf{type}\, j) && \longmapsto \mathsf{type}\, j, \\
&\mathsf{sup}\,(\mathsf{type}\, i)\ (\mathsf{type}\, j) && \longmapsto \mathsf{type}\,(\mathsf{max}\, i\, j)\,,
\end{aligned}
$$

$$
\uparrow : \Pi s_1, s_2 : \mathsf{sort}\,.\, \mathsf{U}\, s_1 \to \mathsf{U}\,(\mathsf{sup}\, s_1\, s_2)\,,
$$

and the rewrite rule for T:

$$
\mathsf{T}\ _\,(\uparrow\ s_1\, s_2\, a) \longmapsto \mathsf{T}\, s_1\, a.
$$

We then add rewrite rules to ensure full reflection, by orienting the equations in Definition 9.1.2. Reflexivity and transitivity are straightforward:

$$
\begin{aligned}
&\uparrow\ s\, s\, a && \longmapsto a, \\
&\uparrow\ _\, s_3\,(\uparrow\ s_1\, s_2\, a) && \longmapsto \uparrow\ s_1\,(\mathsf{sup}\, s_2\, s_3)\ a.
\end{aligned}
$$

After much trial and error, we found that the best way to orient Equation (9.1):

$$
\uparrow_{s_3}\ (\pi_{s_1,s_2}\, a\, b) \equiv \pi_{s_1',s_2'}\,(\uparrow_{s_1}\ a)\ (\lambda x\,.\uparrow_{s_2}\ (b\, x))
$$

is from right to left instead of left to right, and for that reason, to split it in two rewrite rules (notice the need for higher-order matching and Miller patterns, which can be avoided in the first rule but are necessary in the second):

$$
\begin{aligned}
&\pi\ _\, s_2\,(\uparrow\ s_1\, s_1'\, a)\ (\lambda x\,.\, b\, x) \longmapsto \uparrow\ (\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1'\, s_2)\ (\pi\, s_1\, s_2\, a\, b)\,, \\
&\pi\, s_1\, _\, a\,(\lambda x\,.\uparrow\ s_2\, s_2'\,(b\, x)) \longmapsto \uparrow\ (\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1\, s_2')\ (\pi\, s_1\, s_2\, a\, b)\,.
\end{aligned}
$$

This might seem surprising at first. The other direction *seems* more natural, however it does not give a confluent rewrite system, and we believe it is simply not possible to make it so with a finite rewrite system. It is not completely unintuitive however, as it agrees with minimal typing: the coercions $\uparrow$ propagate towards the root of the term. This behavior matches the idea that, when computing minimal types, the cumulativity rule should be delayed as much as possible.

Finally, we also need to add the following rewrite rules to ensure that the previous rules are well-typed. These are derivable for closed terms but they are still needed to ensure well-typedness on open terms too:

$$\begin{aligned}
&\mathsf{sup}\, s\, s && \longmapsto s, \\
&\mathsf{sup}\,(\mathsf{sup}\, s_1\, s_2)\ s_3 && \longmapsto \mathsf{sup}\, s_1\,(\mathsf{sup}\, s_2\, s_3)\,, \\
&\mathsf{rule}\,(\mathsf{sup}\, s_1\, s_1')\ s_2 && \longmapsto \mathsf{sup}\,(\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1'\, s_2)\,, \\
&\mathsf{rule}\, s_1\,(\mathsf{sup}\, s_2\, s_2') && \longmapsto \mathsf{sup}\,(\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1\, s_2')\,.
\end{aligned}$$

Remark 10.3.1. The left-hand side of the first rule above is *non-linear* (the variable s appears twice) and *cannot* not be linearized without changing its meaning. Non-linear rules are less efficient to execute and complicate the proofs of confluence.

Remark 10.3.2. This rewrite system is *not* confluent on terms containing free universe variables because the critical pair $\mathsf{rule}\,(\mathsf{sup}\, s_1\, s_1')\ (\mathsf{sup}\, s_2\, s_2')$ is not joinable. However, it is not a problem in practice because the sorts produced by the translation are always closed. This critical pair needs a form of associativity and commutativity in order to close, which cannot be expressed as a traditional terminating rewrite system. A better notion of term rewriting is needed, for example *rewriting modulo associativity and commutativity* (AC) [JK86].

Remark 10.3.3. Even though full reflection is required for the completeness of the translation, this rewrite system does *not* satisfy it for *all* terms. In particular, if $s_2 \prec s_1$ then $\uparrow^{s_2}_{s_1} a$ is another name for a—they have the same type and their image by T is the same. However, such terms are degenerate and are neither generated by the translation, nor by reduction. Therefore, they do not affect the completeness of the translation.

As a summary, we give a slightly-optimized complete definition of the signature in Figures 10.1 and 10.2. We have verified that it is well-typed in Dedukti and used it in our implementation of the translations of Coq and Matita.

Natural numbers

$$
\begin{aligned}
&\mathsf{nat} : \mathsf{Type},\\
&\mathsf{zero} : \mathsf{nat},\\
&\mathsf{succ} : \mathsf{nat} \to \mathsf{nat},\\
&\\
&\mathsf{max} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat},\\
&\mathsf{max}\, i\, \mathsf{zero} \longmapsto i,\\
&\mathsf{max}\, \mathsf{zero}\, j \longmapsto j,\\
&\mathsf{max}\, (\mathsf{succ}\, i)\ (\mathsf{succ}\, j) \longmapsto \mathsf{succ}\, (\mathsf{max}\, i\, j)\,.
\end{aligned}
$$

Sorts

$$
\begin{aligned}
&\mathsf{sort} : \mathsf{Type},\\
&\mathsf{prop} : \mathsf{sort},\\
&\mathsf{type} : \mathsf{nat} \to \mathsf{sort},\\
&\\
&\mathsf{axiom} : \mathsf{sort} \to \mathsf{sort},\\
&\mathsf{axiom}\, \mathsf{prop} \longmapsto \mathsf{type}\, \mathsf{zero},\\
&\mathsf{axiom}\, (\mathsf{type}\, i) \longmapsto \mathsf{type}\, (\mathsf{succ}\, i)\,,\\
&\\
&\mathsf{sup} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort},\\
&\mathsf{sup}\, s_1\, \mathsf{prop} \longmapsto s_1,\\
&\mathsf{sup}\, \mathsf{prop}\, s_2 \longmapsto s_2,\\
&\mathsf{sup}\, (\mathsf{type}\, i)\ (\mathsf{type}\, j) \longmapsto \mathsf{type}\, (\mathsf{max}\, i\, j)\,,\\
&\\
&\mathsf{rule} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{sort},\\
&\mathsf{rule}\, s_1\, \mathsf{prop} \longmapsto \mathsf{prop},\\
&\mathsf{rule}\, \mathsf{prop}\, s_2 \longmapsto s_2,\\
&\mathsf{rule}\, (\mathsf{type}\, i)\ (\mathsf{type}\, j) \longmapsto \mathsf{type}\, (\mathsf{max}\, i\, j)\,,
\end{aligned}
$$

Figure 10.1 – Finite signature for $\lambda C^{\omega}_{\preceq}$ (part I)

Universes

$$
\begin{aligned}
&\mathsf{U} : \mathsf{sort} \to \mathsf{sort},\\
&\mathsf{T} : \Pi s : \mathsf{sort} \,.\, \mathsf{U}\, s \to \mathsf{Type}.\\
&\\
&\mathsf{u} : \Pi s_1 : \mathsf{sort} \,.\, \to \mathsf{U}\, s_1,\\
&\uparrow : \Pi s_1, s_2 : \mathsf{sort} \,.\, \mathsf{U}\, s_1 \to \mathsf{U}\,(\mathsf{sup}\, s_1\, s_2)\,,\\
&\pi : \Pi s_1, s_2 : \mathsf{sort} \,.\, \Pi a : \mathsf{U}\, s_1 \,.\, (\mathsf{T}\, s_1\, a \to \mathsf{U}\, s_2) \to \mathsf{U}\,(\mathsf{rule}\, s_1\, s_2)\,.\\
&\\
&\mathsf{T}\, _\, (\mathsf{u}\, s_1) \longmapsto \mathsf{U}\, s_1,\\
&\mathsf{T}\, _\, (\uparrow\ s_1\, s_2\, a) \longmapsto \mathsf{T}\, s_1\, a.\\
&\mathsf{T}\, _\, (\pi\, s_1\, s_2\, a\, b) \longmapsto \Pi x : \mathsf{T}\, s_1\, a \,.\, \mathsf{T}\, s_2\, (b\, x)\,.
\end{aligned}
$$

Full reflection

$$
\begin{aligned}
&\mathsf{sup}\, s\, s \longmapsto s,\\
&\mathsf{sup}\,(\mathsf{sup}\, s_1\, s_2)\ s_3 \longmapsto \mathsf{sup}\, s_1\,(\mathsf{sup}\, s_2\, s_3)\,,\\
&\mathsf{rule}\,(\mathsf{sup}\, s_1\, s_1')\ s_2 \longmapsto \mathsf{sup}\,(\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1'\, s_2)\,,\\
&\mathsf{rule}\, s_1\,(\mathsf{sup}\, s_2\, s_2') \longmapsto \mathsf{sup}\,(\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1\, s_2')\,.\\
&\\
&\uparrow\ s\, s\, a \longmapsto a,\\
&\uparrow\ _\, s_3\,(\uparrow\ s_1\, s_2\, a) \longmapsto \uparrow\ s_1\,(\mathsf{sup}\, s_2\, s_3)\ a.\\
&\pi\, _\, s_2\,(\uparrow\ s_1\, s_1'\, a)\ (\lambda x \,.\, b\, x) \longmapsto \uparrow\ (\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1'\, s_2)\ (\pi\, s_1\, s_2\, a\, b)\,,\\
&\pi\, s_1\, _\, a\,(\lambda x \,.\, \uparrow\ s_2\, s_2'\, (b\, x)) \longmapsto \uparrow\ (\mathsf{rule}\, s_1\, s_2)\ (\mathsf{rule}\, s_1\, s_2')\ (\pi\, s_1\, s_2\, a\, b)\,.
\end{aligned}
$$

Figure 10.2 – Finite signature for $\lambda C^{\omega}_{\preceq}$ (part II)

11

Application: translating CIC to Dedukti

One of the main goals of this thesis is the translation of the *calculus of inductive constructions* (CIC). This system is an extension of the calculus of constructions with universes, cumulativity, and inductive types, used as a basis in the famous theorem prover Coq[1] [Cdt12] and in Matita[2] [ARCT11]. We have implemented the translation of CIC in two automated tools: Coqine, which translates the proofs of Coq to Dedukti, and Krajono, which translates the proofs of Matita to Dedukti.

A first prototype implementation of Coqine was already written by Boespflug and Burel [BB12], however that version lacked support for the universe hierarchy and cumulativity. Our implementation is a new version written from scratch that translates these features correctly. Krajono is a completely new implementation effort. While the main theoretical basis has been covered in the previous chapters and in Boespflug and Burel's work, we found that in practice there are a lot of additions and subtle deviations that make the implementations difficult:

- Modules (Coq)
- Local let definitions (Coq and Matita) and fix definitions (Coq)
- Floating universes (Coq and Matita)
- Universe polymorphism (Coq)
- Proof irrelevance (Matita)

[1] https://coq.inria.fr/
[2] http://matita.cs.unibo.it/

We tried to accomodate these features as best as possible. However, we found that some of them can be very difficult, if not downright impossible, to translate in a satisfactory way. For these reasons, we think it would be good to share the lessons we learned from our implementation effort.

In this chapter, we will go over the remaining details of the translation and briefly discuss the implementations and our experimental results. We assume the reader is familiar with at least one of the two systems, which are fairly similar. The Coq reference manual [Cdt12] contains a good exposition to CIC. For Matita, we used the detailed description of its kernel by Asperti et al. [ARCT09]. In addition to those reference documents, we also studied the source code of the kernel of each system. Indeed, we found it hard to decouple the formal system from its implemantation, which can change throughout the versions and can deviate from published descriptions (in the case of Coq, sometimes significantly) but which at the same time is the de facto standard.

11.1 Inductive types

One of the most important features in CIC compared to the calculus of constructions is *inductive types* [CP90], which allows the definition of datatypes and of functions over these datatypes. In their paper on Coqine [BB12], Boespflug and Burel show how to embed inductive types in the $\lambda\Pi$-calculus modulo rewriting. Our own implementations of the translations of Coq and Matita are based on that embedding, so we will briefly discuss it here. We will not present the embedding formally and in detail. Stating the rules for inductive definitions in their general and complete form is very tedious, because we have to include mutual inductive types, parameters, real arguments, etc. We instead take concrete examples to illustrate the embedding and compare it to other alternatives.

Remark 11.1.1 (Coinductive types). We have purposely not treated *coinductive types* as they break subject reduction [Gim96, Our08]. Since we only work with systems that satisfy subject reduction in the $\lambda\Pi$-calculus modulo rewriting (Theorem 3.2.9), it is not clear how to encode the coinductive types of CIC in a way that is sound and complete.

11.1.1 Inductive types and eliminators

We take as example natural numbers and, for simplicity, place ourselves at first in the calculus of constructions.[3] First we need constants for the type and the constructors:

$$\begin{aligned} \mathsf{nat} &: \mathsf{Type}, \\ \mathsf{zero} &: \mathsf{nat}, \\ \mathsf{succ} &: \mathsf{nat} \to \mathsf{nat}, \end{aligned}$$

We also need an elimination mechanism, i.e. a way to define functions, predicates, and proofs by induction. This is much less straightforward to formalize and there are several

[3] This will allow us to omit the U and T operators, and use types directly as terms. Since we know we can embed the calculus of constructions (see Chapter 5), this is not a problem.

approaches for it. The first (I) is to use primitive eliminators, i.e. combinators for defining primitive recursive functions as in Gödel's system T:

$$\begin{aligned}
&\mathsf{nat_elim} : \Pi p : (\mathsf{nat} \to \mathsf{U}) \,.\, p\,\mathsf{zero} \to \\
&\qquad\qquad (\Pi n : \mathsf{nat} \,.\, p\,n \to p\,(\mathsf{succ}\,\mathsf{nat})) \to \\
&\qquad\qquad \Pi n : \mathsf{nat} \,.\, p\,n.
\end{aligned}$$

$$\begin{aligned}
&\mathsf{nat_elim}\,p\,x\,f\,\mathsf{zero} &&\longmapsto x, \\
&\mathsf{nat_elim}\,p\,x\,f\,(\mathsf{succ}\,n) &&\longmapsto f\,n\,(\mathsf{nat_elim}\,p\,x\,f\,n)\,.
\end{aligned}$$

There is a primitive eliminator for each inductively defined dataype. We can use the eliminator to define functions, predicates, and proofs:

$$\begin{aligned}
\mathsf{plus} \coloneqq\ &\mathsf{nat_elim}\,(\lambda x : \mathsf{nat} \,.\, \mathsf{nat} \to \mathsf{nat})\ (\lambda y : \mathsf{nat} \,.\, y) \\
&\quad (\lambda x : \mathsf{nat} \,.\, \lambda f : (\mathsf{nat} \to \mathsf{nat}) \,.\, \lambda y : \mathsf{nat} \,.\, \mathsf{succ}\,(f\,y))\,.
\end{aligned}$$

In CIC, instead of using primitive eliminators, the elimination mechanism is decomposed into two constructs, match and fix, similar to what is found in modern functional programming languages such as OCaml. The first construct allows pattern matching and case analysis while the second allows the construction of fixpoints. In that setting, the definition of plus would look like:

$$\begin{aligned}
\mathsf{plus} \coloneqq\ &\mathsf{fix}\,(\lambda plus : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \,.\, \lambda x : \mathsf{nat} \,. \\
&\qquad \mathsf{match}\,x\,(\mathsf{nat} \to \mathsf{nat}) \\
&\qquad (\lambda y : \mathsf{nat} \,.\, y) \\
&\qquad (\lambda x' : \mathsf{nat} \,.\, \lambda y : \mathsf{nat} \,.\, \mathsf{succ}\,(plus\,x'\,y)))\,.
\end{aligned}$$

Syntactic conditions, called *well-guardedness*, ensure that the recursive calls are well-founded so that the resulting functions are well-defined. This second approach (II) is more intuitive and easier to use than primitive eliminators, which in particular do not handle deep matching (pattern matching several levels at once) very well [Cdt12]. Gimenez [Gim95] proved that the two approaches are equivalent in the calculus of constructions, and indeed, in Coq, primitive eliminators are derived from match and fix for every inductive type. However, the syntactic conditions for guardedness are very tricky to get right and interact poorly with other features (e.g. impredicativity). They are often sources of soundness bugs in Coq [Dén13].

Yet another way (III), more natural when we have rewriting, is to define functions directly by rewriting:

$$\mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.$$

$$\begin{aligned}
&\mathsf{plus}\,\mathsf{zero}\,y &&\longmapsto y \\
&\mathsf{plus}\,(\mathsf{succ}\,x)\ y &&\longmapsto \mathsf{succ}\,(\mathsf{plus}\,x\,y)
\end{aligned}$$

Notice how the rewrite rules allow both pattern matching (plus is applied to constructors on the left-hand side) *and* recursion (plus appears on the right-hand side). Conditions for ensuring coverage, confluence, and normalization must be used to ensure the soundness of the calculus. Some techniques have been developed for this purpose [Bla03, Bla05, JLA15] but in practice this approach is less used than the other two.

11.1.2 Eliminators in the $\lambda\Pi$-calculus modulo rewriting

In our implementation, we use a mix of approaches II and III: we declare a constant match_nat that allows pattern matching on natural numbers by providing the branches for the different cases as arguments. An appropriate rewrite system for that constant gives it the proper reduction behavior. However, we cannot do the same for fix_nat because the resulting system would be unsound. Indeed, the termination of fix in CIC is ensured by purely syntactic conditions that we cannot express using types in the $\lambda\Pi$-calculus modulo rewriting. When we declare fix_nat, we have no choice but to give it a general rewrite rule such as:

$$\mathsf{fix}\, a\, f \longmapsto f\,(\mathsf{fix}\, a\, f)\,,$$

which would be unsound because we can use it to make non-decreasing recursive calls. Instead, we defer the declaration of rewrite rules to each fixpoint: for each fixpoint in the source program, we declare a constant and rewrite rules that simulate its behavior in the $\lambda\Pi$-calculus modulo rewriting. For plus, it would give something like:

$$\mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.$$

$$\begin{aligned}\mathsf{plus}\, x \longmapsto\ & \mathsf{match}\, x\, (\mathsf{nat} \to \mathsf{nat}) \\ & (\lambda y : \mathsf{nat}\,.\, y) \\ & (\lambda x' : \mathsf{nat}\,.\, \lambda y : \mathsf{nat}\,.\, \mathsf{succ}\,(\mathit{plus}\, x'\, y))\,.\end{aligned}$$

If the original fixpoint is well-guarded, then this rewrite system is sound. Note however that, using this approach, we do *not* check for the well-guardedness of the function in Dedukti, so we must still trust Coq or Coqine on this one.

There is still a problem with the definition we gave above: the rewrite system is not terminating, since the left-hand side $\mathsf{plus}\, x$ also appears on the right-hand side. This is a common problem; in CIC, it is solved by restricting the rule to the case where the term that instantiates x starts with a constructor. We cannot express this using term rewriting, so we must resort to something else. There are several solutions to this. One is to create a copy of the inductive type that acts as a wrapper around the constructors:

$$\begin{aligned}&\mathsf{nat} && : \mathsf{Type}, \\ &\mathsf{pre_nat} && : \mathsf{Type}, \\ &\mathsf{zero} && : \mathsf{pre_nat}, \\ &\mathsf{succ} && : \mathsf{nat} \to \mathsf{pre_nat}, \\ &\mathsf{nat_constr} && : \mathsf{pre_nat} \to \mathsf{nat}.\end{aligned}$$

We can then restrict the rewrite rule to the case where x begins with a wrapper:

$$\begin{aligned}\mathsf{plus}\,(\mathsf{nat_constr}\,x) \longmapsto\ & \mathsf{match}\,(\mathsf{nat_constr}\,x)\ (\mathsf{nat} \to \mathsf{nat}) \\ & (\lambda y : \mathsf{nat}\,.\,y) \\ & (\lambda x' : \mathsf{nat}\,.\,\lambda y : \mathsf{nat}\,.\,\mathsf{succ}\,(\mathit{plus}\,x'\,y))\,.\end{aligned}$$

Another solution is to achieve this restriction by making the constructors zero and succ appear on the left-hand side of the rule (as in approach III). However, as noted by Boespflug and Burel [BB12], because of dependent elimination, this requires higher-order unification which is undecidable. A workaround is to duplicate the arguments, the first copy being used for typing and the second for pattern matching:

$$\begin{aligned}\mathsf{plus}\,x &\longmapsto \mathsf{plus_filter}\,x\,x \\ \mathsf{plus_filter}\,x\,\mathsf{zero} &\longmapsto \ldots \\ \mathsf{plus_filter}\,x\,(\mathsf{succ}\,x') &\longmapsto \ldots\end{aligned}$$

To avoid duplication, we can factorize this mechanism in what we call a *filter function*. For each inductive type, we declare a filter function whose sole purpose is to freeze computation and only resume when its argument begins with a constructor:

$$\mathsf{filter_nat} : \Pi p : (\mathsf{nat} \to \mathsf{Type})\,.\,(\Pi x : \mathsf{nat}\,.\,p\,x) \to (\Pi x : \mathsf{nat}\,.\,p\,x)\,.$$

$$\begin{aligned}\mathsf{filter_nat}\,p\,f\,\mathsf{zero} &\longmapsto f\,\mathsf{zero}, \\ \mathsf{filter_nat}\,p\,f\,(\mathsf{succ}\,x) &\longmapsto f\,(\mathsf{succ}\,x)\,.\end{aligned}$$

We can then define the plus function by calling the filter function:

$$\mathsf{plus}\,x \longmapsto \mathsf{filter_nat}\,(\lambda x : \mathsf{nat}\,.\,\mathsf{nat} \to \mathsf{nat})\ (\lambda x : \mathsf{nat}\,.\,\mathsf{match_nat}\,x \cdots)\ x$$

It is not clear which solution is the best. The first duplicates constructors, increasing the size of the terms, while the other duplicates function arguments. Our implementation currently uses the filters solution. In our opinion, the most satisfying approach would be to transform functions defined using match and fix into primitive eliminators using the transformation of Gimenez [Gim95]. However, this transformation is very heavy and further work is required to see if it is feasible and/or practical. We also note that there is yet another approach, which is to derive inductive types using other constructs such as *well-ordered types* ($\mathcal{W}$-types) [ML84] or as done by Chapman et al. [CDMM10]. The advantage of that method is that it defines inductive types once and for all—we do not need to add rewrite rules for each inductive type—but it is also the heaviest.

11.2 Other features

11.2.1 Modules

Coq has an extensive module system, that includes nested modules, functors, aliasing, subtyping, etc. This feature allows the programming of theories in a modular way that encourages abstraction and code reuse. Although it is a conservative extension of the core CIC, it is directly handled by the kernel, so our translation needs to take care of it.

The translation of modules has already been covered in Boespflug and Burel's work [BB12], so we will not dwell on it here. The most straightforward solution is to eliminate modules, by flattening nested modules, parameterizing the entries of functors, and instantiating them in functor applications. This approach works in theory but is not ideal because it leads to a lot of duplication. Another approach would be to encode modules in a more faithful way using records and Σ-types, which we know how to represent in the $\lambda\Pi$-calculus modulo rewriting: they can be represented as inductive types, but there are also more direct and efficient representations that can handle subtyping as well [CD15]. This approach requires further investigation to see if it works well for Coq modules. At the time of writing, modules are not yet completely supported in our re-implementation of Coqine. The Matita system does not have modules.

11.2.2 Local let definitions and local fixpoints

Coq and Matita both support local `let...in...` definitions. We implement these using globally defined constants. For every expression $\mathsf{let}\, x : A = N \,\mathsf{in}\, M$ in the context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we lift the definition to the top-level and we add a constant declaration and a rewrite rule to the global context:

$$x : [\![\Gamma]\!] \to [\![A]\!]_\Gamma$$
$$x\,[\Gamma] \longmapsto [N]_\Gamma$$

We then translate each occurrence of x in M by $x\,[\Gamma]$. It is straightforward to see that these occurrences are equivalent to $[N]_\Gamma$ and that the translation is therefore correct.

Local fixpoint definitions are more subtle and are only supported by Coq. For every expression $\mathsf{fix}\, f : A = N \,\mathsf{in}\, M$, we can combine the technique of lifting of let definitions with the translation of fixpoints of Section 11.1.2. However, it can cause problems of convertibility: if two equivalent terms containing the same fixpoint definition appear in two different places in the source program, they should be considered equivalent after translation, but if we are not careful and lift the fixpoint twice using two different global constants, the resulting terms will not be equivalent, because the constants can appear in their definition on the right-hand side. Therefore, we need to cache the definition of fixpoints and reuse the same global constants for equivalent local fixpoint definitions. This then introduces another problem, because the same local fixpoint definition can appear in different contexts Γ_1 and Γ_2, so we must know how to generalize and instantiate them using a single context Γ.

We have not yet solved this problem, and we have found that it *does* occur in practice, for example in the standard library of Coq, where two definitions of the plus function should be considered equivalent but are not in our translation. Local fixpoint definitions are specific to Coq, and are one of the features that have been described as problematic in the description of the Matita kernel [ARCT09]. In that document, a global program transformation that lifts and eliminates local fixpoint definitions based on caching heuristics is briefly described, which seems like a good starting point for further work.

11.2.3 Floating universes

Coq has a feature which allows users to omit the level of the Type_i universe and just write Type instead. The kernel infers appropriate levels while maintaining consistency. It does so by representing universes as abstract universe expressions while maintaining and solving a set of universe level constraints.

In our implementation, we ask the kernel for the constraint graph and assign to each universe variable a concrete level that satisfies the constraints. This has the downside that we need to translate the whole graph every time a new constraint is added that changes the relationship between the universes (e.g. when a new constraint enforces that a universe variable is now strictly higher than another one). Unfortunately, there is no solution that can solve this lack of modularity completely. Indeed, universe floating breaks weakening: there exists $\Gamma_1, \Gamma_2, \Gamma_3$ such that $\Gamma_1, \Gamma_2 \vdash \mathrm{WF}$ and $\Gamma_1, \Gamma_3 \vdash \mathrm{WF}$ but $\Gamma_1, \Gamma_2, \Gamma_3 \nvdash \mathrm{WF}$.

Example 11.2.1 (Floating universes break weakening)**.** Suppose module `A` contains:

```
Definition MyType1 := Type.
Definition MyType2 := Type.
```

and module `B` contains:

```
Require Import A.
Definition b : MyType1 := MyType2.
```

and the file `C` contains:

```
Require Import A.
Definition c : MyType2 := MyType1.
```

Then modules `A` and `B` can be checked:

```
$ coqchk A B
...
Modules were successfully checked
```

and modules `A` and `C` can be checked:

```
$ coqchk A C
...
Modules were successfully checked
```

but the three modules `A`, `B`, and `C` cannot be checked at the same time:

```
$ coqchk A B C
...
Error: Universe inconsistency.
```

The entailment relation $\Gamma \vdash M : A$ in Coq must be understood as "there exists an instantiation of universe variables such that the constraints graph is acyclic and M has type A in the context Γ", and these side-conditions may be incompatible between different contexts. Since the $\lambda\Pi$-calculus *does* satisfy weakening, this example shows that there is no solution that can translate universe floating in a modular way while maintaining consistency.

Surprisingly, this problem also affects Matita, even though that system uses *explicit universes*: the user has to explicitly specify the universe variables and the constraints between the universes variables that are used, in the style of Courant [Cou02]. However, nothing prevents a user from adding constraints at any point in the program, thus changing the constraint graph. For these reasons, we advise the users of Coqine and Krajono to load all the modules that need to be translated *before* translating them, so that the constraints are stabilized.

11.2.4 Universe polymorphism

The Coq system recently started to support *universe polymorphism*, which is a feature that allows reusing the same constant and inductive type definitions at multiple universe levels. To avoid inconsistencies, only a restricted form of polymorphism, called *prenex polymorphism*, is allowed: universe variables can be abstracted at the beginning of the type of globally defined and inductive constants, and their use must be fully instantiated. One example that was previously impossible and that is now allowed is the self-application of the polymorphic identity function:

```
Coq < Definition id (a : Type) : a -> a := fun x => x.
id is defined

Coq < Check (id (forall a : Type, a -> a) id).
id (forall a : Type, a -> a) (fun a : Type => id a)
     : forall a : Type, a -> a
```

This is because the type of id can be seen as the universe-polymorphic type

$$\forall i.\Pi a : \mathsf{Type}_i \,.\, a \to a$$

and the term $\mathsf{id}\,(\Pi a : \mathsf{Type} \,.\, a \to a)\ \mathsf{id}$ can be seen as the term

$$\mathsf{id}\,(j+1)\ \Big(\Pi a : \mathsf{Type}_j \,.\, a \to a\Big)\ (\mathsf{id}\,j)\,.$$

As we can see, the two occurrences of id are instantiated with two different levels $(j+1)$ and j, which maintains consistency.

This then gives an intuitive translation of universe polymorphism. Indeed, in the light of the encoding of Chapter 10, we get universe polymorphism "for free", by using the abstraction and application mechanisms of the $\lambda\Pi$-calculus modulo rewriting:

$$\begin{aligned}\mathsf{id} : \quad & \Pi i : \mathsf{nat} \,.\, \Pi a : \mathsf{U}\,(\mathsf{type}\, i) \,.\, \mathsf{T}\,(\mathsf{type}\, i)\, a \to \mathsf{T}\,(\mathsf{type}\, i)\, a \\ := \; & \lambda i : \mathsf{nat} \,.\, \lambda a : \mathsf{U}\,(\mathsf{type}\, i) \,.\, \lambda x : \mathsf{T}\,(\mathsf{type}\, i)\, a \,.\, x.\end{aligned}$$

Then, if j is a term of type nat, we have

$$\mathsf{id}\,(j+1)\; A\,(\mathsf{id}\, j) : \Pi a : \mathsf{U}\,(\mathsf{type}\, j) \,.\, \mathsf{T}\,(\mathsf{type}\, j)\, a \to \mathsf{T}\,(\mathsf{type}\, j)\, a$$

where $A = \pi\,(\mathsf{type}\,(j+1))\;(\mathsf{type}\, j)\;(\mathsf{u}\,(\mathsf{type}\, j))\;(\lambda a \,.\, \cdots)$ is the term representation of the type of $\mathsf{id}\, j$. Just as in Chapter 6, there is no term representing the product type $\Pi i : \mathsf{nat} \,.\, \cdots$ in any universe, which ensures that this representation is adequate.

However, the terms we are manipulating now contain free variables for universes and/or universe levels. As we mentioned in Section 10.3, our rewrite system is not confluent on open universe terms. For this reason, we do not support universe polymorphism in our implementation. We note that there is also the solution that duplicates the definitions of constants and inductive types for each fully instantiated occurence, which is less desirable but works in theory.

11.3 Implementation

11.3.1 Coqine: Coq to Dedukti

Coqine[4] (Coq in Dedukti) is a tool for the automatic translation of Coq proofs to Dedukti. A primary version was previously developed by Bospflug and Burel [BB12] as a fork of the Coq kernel. However, this original version suffered from a number of problems:

- It supported neither the infinite universe hierarchy nor universe cumulativity. There were only 2 universes Prop and Type, with no inclusion between the two. To make matters worse, the implementation included a rule Type : Type to collapse the universe hierarchy. Therefore, it was able to translate most of the proofs of Coq but the embedding was inconsistent.

- Being a fork of the Coq kernel was an obstacle for the maintainability of the code, as changes brought to the kernel between versions had to be manually integrated in the source code. For example, the first version of Coqine was written for Coq version 8.3, and it was very difficult to port it to version 8.4 because the kernel code was duplicated and modified without restriction.

[4] https://gforge.inria.fr/projects/coqine/

We re-implemented Coqine completely from scratch as a Coq plugin. The plugin architecture is easier to maintain and allows for a better abstraction, by restricting itself to the kernel API. Overall, we found the API to be mostly sufficient. Some limitations had to be circumvented using clever tricks—for example, printing universes to examine their structure—but eventually we were able to write what we needed without modifying the kernel. It is our belief that this new version is better suited for long-term maintainability. More importantly, the new implementation integrates our results on universe hierarchies and cumulativity from chapters 9 and 10, therefore yielding an embedding that is consistent.

The plugin takes a list of Coq libraries (`.vo`) as arguments. It traverses the structure of the modules associated to the libraries and translates their content Dedukti files (`.dk`). Because several features are currently not supported well by the plugin (namely module functors, anonymous fixpoints, universe polymorphism, etc.) we were not able to translate a significant part of the standard library. Nonetheless, we were able to translate the `Init` library and parts of the `Logic` library, as well as hand-crafted examples. We also used Coqine as part of a small interoperability case study [AC15] (See Section 12.2.3).

11.3.2 Krajono: Matita to Dedukti

Krajono[5] (meaning "pencil" in Esperanto) is a tool for the automatic translation of the proofs of Matita to Dedukti. Because Matita does not have a rich plugin system like that of Coq, we implemented this tool as a fork of the Matita compiler. To alleviate the resulting lack of maintainability mentioned above, we modified the code of Matita as least as possible, to install our hooks and to uncover the features that are necessary for our translation. In the end, we only needed to modify the kernel in one place, to make accessible the function that checks equivalence, since only the function that checks subtyping was available to the public.[6]

Since Matita makes it a point to not have a lot of the features of Coq that were problematic to us (namely modules, anonymous fixpoints, universe polymorphism), we were able to translate a much more significant part of the standard library. We did encounter one obstacle though, and that is *proof irrelevance*. This feature is a modification of the conversion check that considers all terms inhabiting certain propositions to be equal:

$$\frac{\forall i \in \{1, \ldots, n\} \,.\, (M_i \equiv M_i' \vee M_i : A_i : \mathsf{Prop})}{c\, M_1 \cdots M_n \equiv c\, M_1' \cdots M_n'}$$

It is non-conservative and, in particular, allows proving Streicher's *axiom K*. Unfortunately, it does not seem that proof irrelevance can be encoded in the $\lambda\Pi$-calculus modulo rewriting in a way that is sound and complete. For these reasons, we deactivated the proof of Lemma K in the standard library and turned it back into an axiom. Fortunately, no further proof depended on it in our benchmark.

[5] https://www.rocq.inria.fr/deducteam/Krajono/

[6] We cannot test equivalence by checking $A \preceq B \wedge B \preceq A$ because the kernel does not support checking subtyping of arbitrary terms, for simplicity and efficiency reasons [ARCT09].

We tested our implementation on the `arithmetic` library which we used as our main benchmark. We were able to successfully translate and check each file in that library. The results are presented in Figure 11.1. As we can see, when compiled, the size of the generated code is around 4 times larger than the original code, which is fairly respectable for such a translation. Unfortunately, there is no other work to compare to. Similarly, we see that the verification time in Dedukti is around 3 times longer than the original verification time in Matita.

Upon closer inspection though, we notice that most of the time is spent on the file `factorial.ma`. After some investigation, it turned out that all the time is spent verifying a single theorem, `le_fact_10`, which states a numerical property about the factorial of the number 10. We are currently investigating this discrepancy and, at the time of writing, we still do not know its origin exactly. We suspect it comes from some clever heuristics in the Matita kernel that allows it to avoid normalizing such large terms in some conversion checks. If we remove that theorem, the total verification time drops to 127.8 seconds. That is about one third of the verification time of Matita, although to be fair the verification time of Matita includes the processing of proof scripts and tactics, i.e. proof *search* instead of just proof *checking* (Matita does not offer any way of decoupling the two). These experiments were performed on a 64-bit Intel Xeon(R) CPU @ 2.67GHz $\times$ 4 machine with 4 GB of RAM.

File	Compiled size (B)		Verification time (s)	
	Matita	Dedukti	Matita	Dedukti
arithmetics/bigops.ma	357886	3093447	13.1	1.1
arithmetics/binomial.ma	107207	321836	8.9	0.2
arithmetics/bounded_quantifiers.ma	16833	34302	0.7	0
arithmetics/chebyshev/bertrand.ma	136049	391600	15.1	4.6
arithmetics/chebyshev/bertrand256.ma	60464	151251	11.4	100.7
arithmetics/chebyshev/cheb..._psi.ma	40142	87933	3	0.1
arithmetics/chebyshev/cheb..._theta.ma	68763	180347	3.6	0.2
arithmetics/chebyshev/factorization.ma	145476	404735	8.1	1
arithmetics/chebyshev/psi_bounds.ma	139793	491258	18.9	0.9
arithmetics/chinese_reminder.ma	68050	297282	70.9	0.2
arithmetics/congruence.ma	49880	114767	1.3	0.1
arithmetics/div_and_mod.ma	142484	407073	38.4	0.1
arithmetics/exp.ma	35500	63873	10.2	0
arithmetics/factorial.ma	120377	527390	11.4	1285
arithmetics/fermat_little_theorem.ma	40948	101381	21.1	0.2
arithmetics/gcd.ma	124201	391963	59	0.2
arithmetics/iteration.ma	9402	9599	0.7	0
arithmetics/log.ma	52920	91783	5.7	0.1
arithmetics/lstar.ma	37167	112547	0.8	0
arithmetics/minimization.ma	103919	339775	12.6	0.1
arithmetics/nat.ma	246845	562027	58.3	0.2
arithmetics/ord.ma	109807	293024	6.8	0.2
arithmetics/permutation.ma	72502	216218	1.4	0.1
arithmetics/pidgeon_hole.ma	26013	55472	6.2	0.1
arithmetics/primes.ma	121816	249941	26.5	15.9
arithmetics/sigma_pi.ma	77985	220514	16.6	0.1
arithmetics/sqrt.ma	64365	149516	2.1	0.1
basics/bool.ma	35895	85276	0.3	0
basics/lists/list.ma	279000	1877314	3.6	0.6
basics/logic.ma	103974	316733	0.4	0.1
basics/relations.ma	33602	49277	0.2	0
basics/types.ma	166035	552920	0.5	0.1
Total	3195300	12242374	437.7	1412.3

Figure 11.1 – Translation of the `arithmetic` library using Krajono

Conclusion

12

Conclusion

With this thesis, we showed how to embed computational higher-order logics in the $\lambda\Pi$-calculus modulo rewriting. We presented a translation of pure type systems in this framework and proved that it is sound and complete. We then showed how to embed universe cumulativity, by generalizing the previous translation to cumulative type systems, and presented ways to deal with infinite universe hierarchies. Finally, we combined the previous features to give an embedding of the calculus of inductive constructions. We implemented these ideas in automated tools for the translation of the proofs of HOL, Coq, and Matita.

12.1 Related work

12.1.1 Logical embeddings

Our focus in this thesis was on the theoretical aspects of logical embeddings, and in particular to exhibit connections between typed lambda calculi. Our approach follows the LF tradition [HHP93] of using a relatively weak logical framework based on type theory for expressing logics, and draws heavily from the works of Barendregt and Geuvers [Bar92, Geu93, GB99] on defining sound and complete translations between calculi, of which we see this thesis as a continuation. As far as we are aware, this is the first time such work has been done for the calculus of inductive constructions. The idea of using explicit subtyping for cumulativity is inspired from intuitionistic type theory

[ML84, Pal98]. A similar idea for the calculus of constructions was hinted at by Herbelin and Spiwack [HS14], but this is the first time it is treated formally and completely for the full hierarchy of universes.

12.1.2 Proof interoperability

The topics of interoperability and translations of proofs to a common framework are not new. Several projects have been proposed with goals more or less similar to those of Dedukti but using different approaches.

Logosphere The Logosphere project[1] aims at building a large library of formal proofs coming from different provers. It uses Twelf as a logical framework. The approach is slightly different from ours in that the connections between the different logics are formalized in LF as *theory morphisms*, and the logic programming engine of Twelf is leveraged to execute them and translate the proofs. The main connection that was achieved this way is between HOL and Nuprl [SS06].

ProofCert The ProofCert project[2] aims at developing a universal proof checking framework. It uses a formalism based on *focused sequent calculus* that can support various forms of reasoning, including classical, intuitionistic, linear, modal, temporal, etc. Its focus is on checking proof *certificates* [CM15, Chi15, CMR13a, CMR13b] and as such allows proof search and proof reconstruction, unlike Dedukti. Various forms of proof formats can be handled by this framework, including sequent calculus, natural deduction, resolution, etc. A checker called Checkers[3] is currently being developed in λProlog. At the time of writing, it supports proofs coming from the E theorem prover.

LATIN, MMT, HETS The Logic Atlas and Integrator (LATIN) project[4] [CHK+11] aims at developing tools and techniques for the interfacing of different systems. It uses two tools: MMT [Rab13] and HETS [MML07]. The first tool is a module system for mathematical theories that focuses on connecting mathematical libraries, including ones coming from Mizar, HOL Light, and PVS. It formalizes logics as *theories* and translations as *theory morphisms*. The second tool focuses on connecting systems, including including HOL and LF, but also programming languages such as Haskell and Maude, as well as model checkers, SAT solvers, and automated theorem provers. It is based on model theory, in which it formalizes logics as *institutions* and translations as *institution morphisms*. Recent work aimed at unifying the two approaches [CHK+12].

[1] http://www.logosphere.org/
[2] https://team.inria.fr/parsifal/proofcert/
[3] https://github.com/proofcert/checkers
[4] https://latin.omdoc.org/

12.2 Future work

12.2.1 Improving the translations

HOL and Holide We already mentioned in Chapter 7 taking into account the *theory files* (.thy) of OpenTheory to obtain a modular translation of theories. Other than that, we could modify Holide to accept other forms of input. OpenTheory is a good tool but it still not universally adopted, and the export of proofs is not fully automated; manual tinkering is required to control which theorems are exported and how the proofs are factorized. This means that it is still not usable for large developments such as Flyspeck. Holide was written for OpenTheory, but its ideas are not limited to it. It would be worthwhile to adapt it to read the proof format used by Kalyzik and Krauss [KK13] to translate Flyspeck from HOL Light to Isabelle, or the newer Common HOL proof format [Ada15].

CIC and Coqine/Krajono There is still room for a lot of work in this area. A first improvement would be to add support for Coq modules in our new version of Coqine. We mentioned in Chapter 11 two possible ways of doing that, by flattening them or by translating their structure using Σ-types.

The translation of local fixpoint definitions must also be corrected, along the lines of Asperti et al. [ARCT09]. Ideally though, we would like to see the transformation into primitive eliminators of Gimenez [Gim95] implemented for both tools, as the thorny issue of inductive elimination has been the subject of much debate in the community, and there is general interest in seeing if this approach is at least possible. If the transformation is not too inefficient and if clever features are implemented to hide it from the user, we could envision an alternative implementation of the checkers of Coq or Matita where the kernel uses only primitive eliminators, for simplicity and safety reasons, while the user writes his code freely as usual.

Along with universe polymorphism, these missing features make Coqine hard to use, although we were able to successfully use it for practical applications (see Section 12.2.3). On the other hand, Krajono is much more usable in its current form, but handling proof irrelevance is still an open problem. Finally, we note that the use of explicit coercions for cumulativity is relevant outside of the direct scope of this thesis, and that similar ideas are currently in consideration for the next version of Coq.

12.2.2 Designing new embeddings

Limits of rewriting Adding rewriting to the $\lambda\Pi$-calculus turned out to be very powerful, but the work in this thesis already pushes against its limits. Ideally, what we want to express are equational theories, and it is not always possible to express them as a rewrite system in a satisfactory way. Richer forms of rewriting could be explored to alleviate this problem. As an example, rewriting modulo AC [JK86] could be useful for universe polymorphism (see Remark 10.3.2) and for intersection types (see below).

Intersection types Intersection types [CDC78] are usually presented in the Curry style and they are notoriously difficult to express in the Church style. Recent developments [HS15, Sta13] indicate that it is possible to represent something very similar using first-order logic and rewriting, but they use rewriting modulo AC, and further work is needed to see if they can lead to a well-behaved and practical embedding.

Linear logic The contexts of the $\lambda\Pi$-calculus modulo rewriting are not linear, so it is difficult to design embeddings of linear logic [Gir87] that are shallow, i.e. where variables are represented by variables. It is easy to design embeddings that are complete but unsound or vice versa. Further work is needed to see if the framework needs to be modified in order to accommodate linear types, or if it is somehow already possible to embed them in the framework, as it is, in a way that is sound and complete.

12.2.3 Interoperability in Dedukti

We motivated this thesis by the problem of interoperability, but a lot of work still needs to be done before that can be completely achieved. Indeed, what we have done so far is present translations of different systems in separate embeddings but there are still many challenges:

1. We do not provide backward translations from Dedukti to the original systems, so we cannot take a proof in one system and translate it to another one by going through Dedukti.

2. The translation of different systems use different signatures for the embeddings, which express the different logics that are used. We need a way to combine these signatures, either by merging them into unifying theories or by writing functions for interfacing between them. However, this might not always be possible without introducing inconsistencies, as the different logics might be incompatible.

3. Each system formalizes and represents data-types differently (e.g. real numbers as limits of converging sequences vs. axiomatic real numbers) and states theorems differently from the others. How can we relate the real numbers of one system with the real numbers of another one? How can we know if the equivalent of a theorem we are interested in is available in the libraries of a system so that we can reuse it? There are no tools that help us in doing so easily.

Coincidentally, we believe that these obstacles are listed in order of increasing importance. We do not consider the first point to be much of a problem. Theoretically, back translations are possible, thanks to the conservativity property. However, we don't think they are necessary, as we see the logical framework as a low-level system to which we compile proofs and in which we link proofs together, similar to how some modern programming languages are compiled to and linked in low-level frameworks (assembly, .NET, etc.).

The second point is more important, and requires a careful analysis of the logics in question in order to avoid inconsistencies. For example, we mentioned in Chapter 7 that we can modify the translation of HOL to make it closer to its PTS formulation and to make it compatible with Coq. The implementation of such transformations is the subject of a currently ongoing internship in the team. Moreover, even if some logical theories are incompatible as a whole, theorems seldom use all the features of the logic. We can therefore analyze the portions that are used and determine if they are compatible. This is the realm of *reverse mathematics*.

Finally, the third point requires long-term involvement in the development of systems and libraries, and falls outside the scope of this thesis. However, we strongly believe it is the most important obstacle to the interoperability of proof systems. In a separate work [AC15], we have experimented with linking the translation of HOL and Coq in Dedukti[5]. After analyzing and unifying the two theories, we used Holide to translate the theory of natural numbers from OpenTheory to Dedukti, and Coqine to translate lists and a sorting algorithm from Coq to Dedukti. We then linked the results together within Dedukti to obtain a sorting algorithm for Coq lists of HOL natural numbers, together with a proof that it respects its specification. All the files were successfully checked by Dedukti. While this small experiment was successful, we found that a lot of manual work was required to interface the two developments together and to relate the datatypes and the theorems of each embedding to the other. We therefore need more tools that help with interfacing the translations of different systems in order for this approach to scale well. Nonetheless, the experiment shows that the logical framework approach is indeed possible, and serves as a stepping stone for future work on interoperability.

[5]Our experiment is available online at: http://dedukti-interop.gforge.inria.fr/.

Bibliography

[AB15] Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. In *Proceedings of the Fourth Workshop on Proof Exchange for Theorem Proving (PxTP 2015)*, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 74–88, Berlin, 2015. Open Publishing Association. doi:10.4204/EPTCS.186.8.

[AC15] Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti (Extended Abstract). In *Proceedings of the Fourth Workshop on Proof Exchange for Theorem Proving (PxTP 2015)*, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 89–96, Berlin, 2015. Open Publishing Association. doi:10.4204/EPTCS.186.9.

[Ada06] Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(02):219–246, 2006. doi:10.1017/S0956796805005770.

[Ada10] Mark Adams. Introducing HOL Zero. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-15582-6_25.

[Ada15] Mark Adams. The Common HOL Platform. In *Electronic Proceedings in Theoretical Computer Science*, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 42–56, Berlin, July 2015. Open Publishing Association. arXiv: 1507.08718. doi:10.4204/EPTCS.186.6.

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof.* Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science, 2001. Proceedings*, pages 247–256, 2001. doi:10.1109/LICS.2001.932501.

[ARCT09] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Sadhana*, 34(1):71–144, February 2009. doi:10.1007/s12046-009-0003-3.

[ARCT11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-22438-6_7.

[Art04] Rob D. Arthan. Some mathematical Case Studies in ProofPower-HOL. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *Lecture Notes in Computer Science*, pages 1–16, Park City, Utah, USA, 2004. Springer.

[Ass14] Ali Assaf. A calculus of constructions with explicit subtyping. In *Post-proceedings of the 20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27–46, Paris, 2014. Schloss Dagstuhl. doi:10.4230/LIPIcs.TYPES.2014.27.

[Ass15] Ali Assaf. Conservativity of embeddings in the lambda-Pi calculus modulo rewriting. In Thorsten Altenkirch, editor, *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31–44, Warsaw, 2015. Schloss Dagstuhl. doi:10.4230/LIPIcs.TLCA.2015.31.

[Bar92] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

[BB12] Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the lambda-Pi-calculus modulo. In *Proof Exchange for Theorem Proving—Second International Workshop, PxTP*, page 44, 2012.

[BC04] Yves Bertot and Pierre Castéran. Proof by reflection. In *Interactive Theorem Proving and Program Development*, Texts in Theoretical Computer Science An EATCS Series, pages 433–448. Springer Berlin Heidelberg, 2004.

[BCH12] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lambda-Pi-calculus modulo as a universal proof language. In *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, pages 28–43, 2012.

[Bee85] Michael Beeson. *Foundations of constructive mathematics*. Springer-Verlag, 1985.

[Ber89] Stefano Berardi. *Type dependence and constructive mathematics*. PhD thesis, University of Torino, Italy, 1989.

[BG05] Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In Luke Ong, editor, *Computer Science Logic*, number 3634 in Lecture Notes in Computer Science, pages 151–166. Springer Berlin Heidelberg, 2005.

[Bla03] Frédéric Blanqui. Inductive Types in the Calculus of Algebraic Constructions. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 46–59. Springer Berlin Heidelberg, 2003. doi:10.1007/3-540-44904-3_4.

[Bla05] Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(01):37–92, February 2005. doi:10.1017/S0960129504004426.

[Bur13] Guillaume Burel. A shallow embedding of resolution and superposition proofs into the lambda-Pi-Calculus Modulo. In *Proceedings of the Third International Workshop on Proof Exchange for Theorem Proving (PxTP)*, volume 14, pages 43–57. EasyChair, 2013.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-73228-0_9.

[CD15] Raphaël Cauderlier and Catherine Dubois. Objects and subtyping in the lambda-Pi-calculus modulo. In *Postproceedings of the 20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics*, pages 47–71, Paris, 2015. Dagstuhl Publishing, Germany. doi:10.4230/LIPIcs.TYPES.2014.47.

[CDC78] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978. doi:10.1007/BF02011875.

[CDMM10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. doi:10.1145/1863543.1863547.

[Cdt12] The Coq development team. *The Coq reference manual, version 8.4*. 2012. URL: http://coq.inria.fr/doc.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[CH15] Raphaël Cauderlier and Pierre Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In *Proceedings of the Fourth Workshop on Proof Exchange for Theorem Proving (PxTP 2015)*, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 57–73, Berlin, 2015. Open Publishing Association. doi:10.4204/EPTCS.186.7.

[Chi15] Zakaria Chihani. *Proof certificates for first-order classical and intuitionistic logics*. PhD thesis, École Polytechnique, Palaiseau, 2015.

[CHK+11] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, and Florian Rabe. Project abstract: Logic Atlas and Integrator (LATIN). In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics*, number 6824 in Lecture Notes in Computer Science, pages 289–291. Springer Berlin Heidelberg, 2011.

[CHK+12] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe, and Kristina Sojakova. Towards logical frameworks in the Heterogeneous Tool Set HETS. In Till Mossakowski and Hans-Jörg Kreowski, editors, *Recent Trends in Algebraic Development Techniques*, number 7137 in Lecture Notes in Computer Science, pages 139–159. Springer Berlin Heidelberg, 2012.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(02):56–68, 1940. doi:10.2307/2266170.

[CM15] Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. Manuscript, 2015. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/rew-fpc.pdf.

[CMR13a] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38574-2_11.

[CMR13b] Zakaria Chihani, Dale Miller, and Fabien Renaud. A semantics for proof evidence. In *Collected abstracts of Theory and Application of Formal Proofs (LIX Colloquium 2013)*, pages 36–37, Palaiseau, 2013. URL: http://www.lix.polytechnique.fr/colloquium2013/abstracts/book.pdf.

[Coq86] T. Coquand. An analysis of Girard's paradox. Technical report 00076023, INRIA, 1986. URL: https://hal.inria.fr/inria-00076023/document.

[Coq91] Thierry Coquand. A new paradox in type theory. In *Proceedings 9th Int. Congress of Logic, Methodology and Philosophy of Science*, pages 555–570, 1991.

[Cou02] Judicaël Courant. Explicit universes for the calculus of constructions. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-45685-6_9.

[Cou09] Denis Cousineau. *Models and proof normalization*. PhD thesis, École Polytechnique, Palaiseau, 2009.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer Berlin Heidelberg, 1990. doi:10.1007/3-540-52335-9_47.

[Cur34] Haskell B. Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, November 1934. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076489/.

[dB68] Nicolaas G. de Bruijn. AUTOMATH, a language for mathematics. Technical report 68-WSK-05, Eindhoven University of Technology, 1968.

[Dén13] Maxime Dénès. Propositional extensionality is inconsistent in Coq, December 2013. E-mail communication. URL: https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html.

[DF92] Oliver Danvy and Andrzex Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(04):361–391, December 1992. doi:10.1017/S0960129500001535.

[DHK03] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003. doi:10.1023/A:1027357912519.

[Dia75] Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975. doi:10.1090/S0002-9939-1975-0373893-X.

[Dow15] Gilles Dowek. Models and termination of proof-reduction in the lambda-Pi-calculus modulo theory. Technical report abs/1501.06522, Inria, Paris, 2015. URL: http://arxiv.org/abs/1501.06522.

[DW00] Gilles Dowek and Benjamin Werner. An inconsistent theory modulo defined by a confluent and terminating rewrite system. Manuscript, 2000. URL: https://who.rocq.inria.fr/Gilles.Dowek/Publi/counterexample.ps.gz.

[DW03] Gilles Dowek and Benjamin Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(04):1289–1316, 2003. doi:10.2178/jsl/1067620188.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39634-2_14.

[GB99] Herman Geuvers and Erik Barendsen. Some logical and syntactical observations concerning the first-order dependent type system lambda-P. *Mathematical Structures in Computer Science*, 9(04):335–359, 1999.

[Geu93] Jan Herman Geuvers. *Logics and type systems*. PhD thesis, University of Nijmegen, 1993.

[Gim95] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin Heidelberg, 1995. doi:10.1007/3-540-60579-7_3.

[Gim96] Eduardo Giménez. *Un calcul de constructions infinies et son application a la vérification de systèmes communicants*. PhD thesis, École normale supérieure de Lyon, 1996.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de Doctorat, Université Paris VII, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GLR09] Georges Gonthier and Stéphane Le Roux. An Ssreflect tutorial. Technical report 00407778, Microsoft Research Inria Joint Centre, 2009. URL: https://hal.inria.fr/inria-00407778/.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[GN91] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[Gon05] Georges Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research, Cambridge, 2005.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Hal07] Thomas C. Hales. The Jordan curve theorem, formally and informally. *American Mathematical Monthly*, 114(10):882–894, 2007.

[Hal14] Thomas C. Hales. Announcement of completion, August 2014. URL: https://code.google.com/archive/p/flyspeck/wikis/AnnouncingCompletion.wiki.

[Har95] John Harrison. Inductive definitions: Automation and application. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213. Springer Berlin Heidelberg, 1995. doi:10.1007/3-540-60275-5_66.

[Har09] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_4.

[HHM+10] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010. doi:10.1007/s00454-009-9148-4.

[HHP93] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

[How80] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Boston, MA, 1980.

[HP89] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions (Draft). In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 241–256. Springer Berlin Heidelberg, 1989. doi:10.1007/3-540-50940-2_39.

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, October 1991. doi:10.1016/0304-3975(90)90108-T.

[HS14] Hugo Herbelin and Arnaud Spiwack. The rooster and the syntactic bracket. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 169–187. Schloss Dagstuhl, 2014. doi:10.4230/LIPIcs.TYPES.2013.169.

[HS15] Olivier Hermant and Ronan Saillard. A rewrite system for strongly normalizable terms. 2015. Manuscrit. URL: http://www.cri.ensmp.fr/classement/doc/A-599.pdf.

[Hur95] Antonius J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer Berlin Heidelberg, 1995. doi:10.1007/BFb0014058.

[Hur11] Joe Hurd. The OpenTheory standard theory library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-20398-5_14.

[JK86] Jean-Pierre Jouannaud and Helene Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, 1986. doi:10.1137/0215084.

[JLA15] Jean-Pierre Jouannaud, Jianqi Li, and Thorsten Altenkirch. Termination of dependently typed rewrite rules. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 257–272, Berlin, 2015. Schloss Dagstuhl. doi:10.4230/LIPIcs.TLCA.2015.257.

[Jut93] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993. doi:10.1006/inco.1993.1038.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4:

Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. doi:10.1145/1629575.1629596.

[KK13] Cezary Kaliszyk and Alexander Krauss. Scalable LCF-style proof translation. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 51–66. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-39634-2_7.

[KW10] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14052-5_22.

[Las12] Marc Lasson. *Réalisabilité et paramétricité dans les systèmes de types purs*. PhD thesis, École normale supérieure de Lyon, 2012.

[Ler15] Xavier Leroy. The CompCert C verified compiler version 2.5. Manual, Inria, June 2015. URL: http://compcert.inria.fr/man/manual.pdf.

[Luo89] Zhaohui Luo. ECC, an extended calculus of constructions. In *Fourth Annual Symposium on Logic in Computer Science, 1989. LICS '89, Proceedings*, pages 386–395, 1989. doi:10.1109/LICS.1989.39193.

[Luo90] Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, 1990.

[Luo94] Zhaohui Luo. *Computation and reasoning: A type theory for computer science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, Inc., New York, NY, USA, 1994.

[MCDB03] César Muñoz, Víctor Carreño, Gilles Dowek, and Ricky Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003. doi:10.1007/s10009-002-0084-3.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, number 475 in Lecture Notes in Computer Science, pages 253–281. Springer Berlin Heidelberg, 1991.

[ML73] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings of the logic colloquium*, volume 80 of *Studies in logic and the foundations of mathematics*, pages 73–118, Bristol, 1973. North-Holland. doi:10.1016/S0049-237X(08)71945-1.

[ML84] Per Martin-Löf. *Intuitionistic type theory*. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Bibliopolis Naples, 1984.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory*, volume 36, pages 127–172. Oxford University Press, 1998.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set, HETS. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 4424 in Lecture Notes in Computer Science, pages 519–522. Springer Berlin Heidelberg, 2007.

[MN98] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998. doi:10.1016/S0304-3975(97)00143-6.

[Muñ03] César Muñoz. Rapid prototyping in PVS. *NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report*, (2003-03), 2003. URL: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040046914.pdf.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science, 1991 (LICS '91)*, pages 342–349, 1991. doi:10.1109/LICS.1991.151658.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory*. Number 7 in International Series of Monographs on Computer Science. Oxford University Press Oxford, 1990.

[NSM01] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-44755-5_23.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[OS06] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, number 4130 in Lecture Notes in Computer Science, pages 298–302. Springer Berlin Heidelberg, 2006.

[Our08] Nicolas Oury. Coinductive types and type preservation, June 2008. E-mail communication. URL: https://sympa.inria.fr/sympa/arc/coq-club/2008-06/msg00022.html.

[Pal98] Erik Palmgren. On universes in type theory. In *Twenty-five years of constructive type theory*, pages 191–204. Oxford University Press, October 1998.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975. doi:10.1016/0304-3975(75)90017-1.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48660-7_14.

[Rab10] Florian Rabe. Representing Isabelle in LF. *Electronic Proceedings in Theoretical Computer Science*, 34:85–99, 2010. arXiv: 1009.2794. doi:10.4204/EPTCS.34.8.

[Rab13] Florian Rabe. The MMT API: a generic MKM system. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Intelligent Computer Mathematics*, number 7961 in Lecture Notes in Computer Science, pages 339–343. Springer Berlin Heidelberg, 2013.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908. doi:10.2307/2369948.

[Sai13] Ronan Saillard. Towards explicit rewrite rules in the lambda-Pi-calculus modulo. In *IWIL - 10th International Workshop on the Implementation of Logics*, 2013. URL: https://hal.inria.fr/hal-00921340/.

[Sai15] Ronan Saillard. *Type checking in the lambda-Pi-calculus modulo: theory and practice*. PhD thesis, École Nationale Supérieure des Mines, Paris, 2015.

[SH12] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(02):153–180, 2012. doi:10.1017/S0956796812000044.

[Smi88] Jan M. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(03):840–845, 1988. doi:10.2307/2274575.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-71067-7_6.

[SS06] Carsten Schürmann and Mark-Oliver Stehr. An executable formalization of the HOL/Nuprl connection in the metalogical framework twelf. In Miki Hermann and Andrei Voronkov, editors, *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, volume 4246 of *Lecture Notes in Computer Science*, pages 150–166. Springer Berlin Heidelberg, 2006. doi:10.1007/11916277_11.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *5th International Conference on Interactive Theorem Proving (ITP 2014)*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer International Publishing, 2014. doi:10.1007/978-3-319-08970-6_32.

[Sta13] Rick Statman. A new type assignment for strongly normalizable terms. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 634–652, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2013.634.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967. doi:10.2307/2271658.

[Ter89] Jan Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, The Netherlands, 1989.

[vO94] Vincent van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.

"How do I look?"
"You look ready."

Appendix

A

Proof details

A.1 Proofs of Chapter 5

Theorem (5.2.11)**.** *For all* Γ*,* M*,* A*, if* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda\Pi R} [M]_\Gamma : [\![A]\!]_\Gamma$*. For all* Γ*, if* $\Gamma \vdash_{\lambda\mathcal{P}} \mathrm{WF}$ *then* $[\![\Gamma]\!] \vdash_{\lambda\Pi R} \mathrm{WF}$*.*

Proof. By simultaneous induction on the derivation.

1. $\dfrac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$.
   Then $[\![\varnothing]\!] = \varnothing$ and $\varnothing \vdash \mathrm{WF}$.

2. $\dfrac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$.
   Then $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$. By induction hypothesis, $[\![\Gamma]\!] \vdash [A] : [\![s]\!]$. By Corollary 5.2.8, $[\![\Gamma]\!] \vdash [A] : \mathsf{U}_s$. Therefore, $[\![\Gamma]\!] \vdash [\![A]\!] : \mathsf{Type}$. Therefore, $[\![\Gamma, x : A]\!] \vdash \mathrm{WF}$.

3. $\dfrac{\Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$.
   Then $[x] = x$. By induction hypothesis, $[\![\Gamma]\!] \vdash \mathrm{WF}$. Since $(x : A) \in \Gamma$, we have $(x : [\![A]\!]) \in \Gamma$. Therefore, $[\![\Gamma]\!] \vdash x : [\![A]\!]$.

4. $$\frac{\Gamma \vdash \mathrm{WF} \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}.$$
Then $[s_1] = \mathsf{u}_{s_1}$. By induction hypothesis, $[\![\Gamma]\!] \vdash \mathrm{WF}$. Since $(s_1, s_2) \in \mathcal{A}$, we have $(\mathsf{u}_{s_1} : \mathsf{U}_{s_2}) \in \Sigma$. Therefore, $[\![\Gamma]\!] \vdash \mathsf{u}_{s_1} : \mathsf{U}_{s_2}$. By Corollary 5.2.8, $[\![\Gamma]\!] \vdash \mathsf{u}_{s_1} : [\![s_2]\!]$.

5. $$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A \,.\, B : s_3}\ \textsc{Prod}.$$
Then $[\Pi x : A \,.\, B] = \pi_{s_1,s_2}\,[A]\,(\lambda x \,.\, [B])$. By induction hypothesis, $[\![\Gamma]\!] \vdash [A] : [\![s_1]\!]$ and $[\![\Gamma]\!], x : [\![A]\!] \vdash [B] : [\![s_2]\!]$. By Corollary 5.2.8, $[\![\Gamma]\!] \vdash [A] : \mathsf{U}_{s_1}$ and $[\![\Gamma]\!], x : [\![A]\!] \vdash [B] : \mathsf{U}_{s_2}$. Therefore, $[\![\Gamma]\!] \vdash \pi_{s_1,s_2}\,[A]\,(\lambda x \,.\, [B]) : \mathsf{U}_{s_3}$. By Corollary 5.2.8, $[\![\Gamma]\!] \vdash \pi_{s_1,s_2}\,[A]\,(\lambda x \,.\, [B]) : [\![s_3]\!]$.

6. $$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}.$$
Then $[\lambda x : A \,.\, M] = \lambda x : [\![A]\!] \,.\, [M]$. By induction hypothesis, $[\![\Gamma]\!], x : [\![A]\!] \vdash [M] : [\![B]\!]$. Therefore, $[\![\Gamma]\!] \vdash \lambda x : [\![A]\!] \,.\, [M] : \Pi x : [\![A]\!] \,.\, [\![B]\!]$. By Corollary 5.2.10, $[\![\Gamma]\!] \vdash \lambda x : [\![A]\!] \,.\, [M] : [\![\Pi x : A \,.\, B]\!]$.

7. $$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B\,\{x \backslash N\}}\ \textsc{App}.$$
Then $[M\,N] = [M]\,[N]$. By induction hypothesis, $[\![\Gamma]\!] \vdash [M] : [\![\Pi x : A \,.\, B]\!]$ and $[\![\Gamma]\!] \vdash [N] : [\![A]\!]$. By Corollary 5.2.10, $[\![\Gamma]\!] \vdash [M] : \Pi x : [\![A]\!] \,.\, [\![B]\!]$. Therefore, $[\![\Gamma]\!] \vdash [M]\,[N] : [\![B]\!]\,\{x \backslash [N]\}$. By Lemma 5.2.1, $[\![B]\!]\,\{x \backslash [N]\} = [\![B\,\{x \backslash N\}]\!]$. Therefore, $[\![\Gamma]\!] \vdash [M]\,[N] : [\![B\,\{x \backslash N\}]\!]$.

8. $$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}\ \textsc{Conv}.$$
By induction hypothesis, $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ and $[\![\Gamma]\!] \vdash [\![B]\!] : \mathsf{Type}$. By Corollary 5.2.6, $[\![A]\!] \equiv [\![B]\!]$. By conversion, $[\![\Gamma]\!] \vdash [M] : [\![B]\!]$.

□

A.2 Proofs of Chapter 6

Lemma (6.2.3)**.** *For all* Γ, M, A*, if* $\Gamma \vdash_{\lambda\mathcal{P}} M : A$ *then* $|[M]_\Gamma| \equiv_\beta M$*.*

Proof. By induction on the structure of M.

- Case $M = x$. Then$|[x]| = |x| = x$.
- Case $M = s$. Then $|[s]| = \left|\mathsf{u}_s^{s'}\right| = s$.

- Case $M = \Pi x : A_1 \,.\, B_1$. By induction hypothesis, $|[A_1]| \equiv A_1$ and $|[B_1]| \equiv B_1$. Therefore,

$$\begin{aligned} |[\Pi x : A_1 \,.\, B_1]| &= |\pi_{s_1,s_2}\,[A_1]\;(\lambda x \,.\, [B_1])| \\ &= (\lambda \alpha \,.\, \lambda \beta \,.\, \Pi x : \alpha \,.\, \beta\, x)\;|[A_1]|\;(\lambda x \,.\, |[B_1]|) \\ &\equiv \Pi x : |[A_1]| \,.\, |[B_1]| \\ &\equiv \Pi x : A_1 \,.\, B_1. \end{aligned}$$

- Case $M = \lambda x : A_1 \,.\, M_1$. By induction hypothesis, $|[A_1]| \equiv A_1$ and $|[M_1]| \equiv M_1$. Therefore,

$$\begin{aligned} |[\lambda x : A_1 \,.\, M_1]| &= |\lambda x : \mathsf{T}_s\,[A_1] \,.\, [M_1]| \\ &= \lambda x : |[A_1]| \,.\, |[M_1]| \\ &\equiv \lambda x : A_1 \,.\, M_1. \end{aligned}$$

- Case $M = M_1\, N_1$. By induction hypothesis, $|[M_1]| \equiv M_1$ and $|[N_1]| \equiv N_1$. Therefore,

$$\begin{aligned} |[M_1\, N_1]| &= |[M_1]\;[N_1]| \\ &= |[M_1]|\;|[N_1]| \\ &\equiv M_1\, N_1. \end{aligned}$$

□

Lemma (6.2.25)**.** *If $A \equiv_{\beta\eta^-\Sigma} B$ then $\Gamma \Vdash M : A$ iff $\Gamma \Vdash M : B$.*

Proof. By case analysis on the derivation of $A \equiv B$, using induction on the measure of A. In each case, we show $\Gamma \Vdash M : A \implies \Gamma \Vdash M : B$, the other direction being similar.

- Base cases:
  - Case $\mathsf{T}_{s_2}\,\mathsf{u}_{s_1} \longrightarrow \mathsf{U}_{s_1}$. By definition, there exists M', A' such that $\Gamma \vdash M' : A'$ and $[M'] \equiv M$ and $[\![A']\!] \equiv \mathsf{T}_{s_2}\,\mathsf{u}_{s_1}$. Then $[\![A']\!] \equiv \mathsf{U}_{s_1}$. Therefore, $\Gamma \Vdash M : \mathsf{U}_{s_1}$
  - Case $\mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\,A_1\,B_1) \longrightarrow \Pi x : \mathsf{T}_{s_1}\,A_1 \,.\, \mathsf{T}_{s_2}\,(B_1\,x)$. By definition, there exists M', A' such that $\Gamma \vdash M' : A'$ and $[M'] \equiv M$ and $[\![A']\!] \equiv \mathsf{T}_{s_3}\,(\pi_{s_1,s_2}\,A_1\,B_1)$. Then $[\![A']\!] \equiv \Pi x : \mathsf{T}_{s_1}\,A_1 \,.\, \mathsf{T}_{s_2}\,(B_1\,x)$. By Corollary 6.2.16, there exists A_1' and B_1' such that $A' \equiv \Pi x : A_1' \,.\, B_1'$ and $\Gamma \vdash M' : \Pi x : A_1' \,.\, B_1'$. Therefore, $[\![A']\!] \equiv [\![\Pi x : A_1' \,.\, B_1']\!]$. By product compatibility, this means that $[\![A_1']\!] \equiv \mathsf{T}_{s_1}\,A_1$ and $[\![B_1']\!] \equiv \mathsf{T}_{s_2}\,(B_1\,x)$. Let Γ_N, N be such that $\Gamma \subseteq \Gamma_N$ and $\Gamma_N \Vdash N : \mathsf{T}_{s_1}\,A_1$. By definition, there exists N', A_1'' such that $\Gamma_N \vdash N' : A_1''$ and $[N'] \equiv N$ and $[\![A_1'']\!] \equiv \mathsf{T}_{s_1}\,A_1$. Then $[\![A_1'']\!] \equiv [\![A_1']\!]$. By Corollary 6.2.12 and conversion, $A_1'' \equiv A_1'$ and $\Gamma \vdash N : A_1'$. Therefore, $\Gamma \vdash M'\,N' : B_1'\,\{x \backslash N'\}$. By Corollary 5.2.2, $[\![B_1'\,\{x \backslash N'\}]\!] = [\![B_1']\!]\,\{x \backslash [N_1]\}$. Then $[\![B_1'\,\{x \backslash N'\}]\!] \equiv \mathsf{T}_{s_2}\,(B_1\,x)\,\{x \backslash N\}$. Therefore, $\Gamma' \Vdash M\,N : \mathsf{T}_{s_2}\,(B_1\,x)\,\{x \backslash N\}$. Therefore, $\Gamma \Vdash M : \Pi x : \mathsf{T}_{s_1}\,A_1 \,.\, \mathsf{T}_{s_2}\,(B_1\,x)$.

- Case $(\lambda x : A_1 \,.\, B_1)\ N_1 \cdots N_n \longrightarrow B_1\,\{x \backslash N_1\}\ N_2 \cdots N_n$. Follows from the definition of $\Gamma \Vdash M : (\lambda x : A_1 \,.\, B_1)\ N_1 \cdots N_n$.

- Structural cases:

  - Case $\Pi x : A_1 \,.\, B_1 \equiv \Pi x : A_1' \,.\, B_1$ with $A_1 \equiv A_1'$ and $B_1 \equiv B_2'$. Let Γ_N, N such that $\Gamma \subseteq \Gamma_N$, and $\Gamma \Vdash N : A_1'$. By induction, $\Gamma \Vdash N : A_1$. By definition, $\Gamma \Vdash M\,N : B_1\,\{x \backslash N\}$. By induction $\Gamma \Vdash M\,N : B_1'\,\{x \backslash N\}$. Therefore, $\Gamma \Vdash M : \Pi x : A_1' \,.\, B_1'$.
  - Case $\mathsf{T}_s\,A_1 \equiv \mathsf{T}_s\,A_1'$ with $A_1 \equiv A_1'$. By definition, there is M' and A_1'' such that $\Gamma \vdash M' : A_1''$ and $[M'] \equiv M$ and $[\![A_1'']\!] \equiv A_1$. Therefore, $[\![A_1'']\!] \equiv A_1'$. Therefore, $\Gamma \Vdash M : \mathsf{T}_s\,A_1'$.
  - Case $(\lambda x : A_1 \,.\, B_1)\ N_1 \cdots N_n \equiv (\lambda x : A_1' \,.\, B_1')\ N_1' \cdots N_n'$ with $A_1 \equiv A_1'$, $B_1 \equiv B_1'$, and $N_i \equiv N_i'$. By definition, $\Gamma \Vdash M : B_1\,\{x \backslash N_1\}\ N_2 \cdots N_n$. By induction, $\Gamma \Vdash M : B_1'\,\{x \backslash N_1'\}\ N_2' \cdots N_n'$. Therefore, $\Gamma \Vdash (\lambda x : A_1' \,.\, B_1')\ N_1' \cdots N_n'$.

- Equivalence cases:

  - Case $A \equiv A$ (reflexivity) is straightforward.
  - Case $B \equiv A$ with $A \equiv B$ (symmetry) is straightforward.
  - Case $A \equiv C$ with $A \equiv B$ and $B \equiv V$ (transitivity) is straightforward.

□

Lemma (6.2.26)**.** *For all Δ, M, A, if $\Sigma, \Delta \vdash_{\lambda\Pi R} M : A$ then for all Γ, σ such that $\Gamma \vdash_{\lambda\mathcal{P}}$ WF and $\Gamma \Vdash \sigma : \Delta$, $\Gamma \Vdash \sigma\,(M) : \sigma\,(A)$.*

Proof. By induction on the derivation.

- $$\frac{\Delta \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Delta \vdash x : A}\ \mathrm{VAR}$$
  Then $\Gamma \Vdash \sigma\,(x) : \sigma\,(A)$ by definition of $\Gamma \Vdash \sigma : \Delta$.

- $$\frac{\Delta \vdash \mathrm{WF} \qquad (\mathsf{u}_{s_1} : \mathsf{U}_{s_2}) \in \Sigma}{\Delta \vdash \mathsf{u}_{s_1} : \mathsf{U}_{s_2}}\ \mathrm{VAR}$$
  Then $\mathsf{u}_{s_1} \equiv [s_1]$ and $\mathsf{U}_{s_2} \equiv [\![s_2]\!]$ and $(s_1, s_2) \in \mathcal{A}$. Since $\Gamma \vdash \mathrm{WF}$, we have $\Gamma \vdash s_1 : s_2$.

- $$\frac{\Delta \vdash \mathrm{WF} \qquad (\pi_{s_1,s_2} : \Pi\alpha : \mathsf{U}_{s_1} \,.\, \cdots) \in \Sigma}{\Delta \vdash \pi_{s_1,s_2} : \Pi\alpha : \mathsf{U}_{s_1} \,.\, (\Pi x : \mathsf{T}_{s_1}\,\alpha \,.\, \mathsf{U}_{s_2}) \to \mathsf{U}_{s_3}}\ \mathrm{VAR}$$
  Let Γ_A, A be such that $\Gamma \subseteq \Gamma_A$ and $\Gamma_A \Vdash A : \mathsf{U}_{s_1}$. By definition and Corollary 6.2.14, there is A' such that $\Gamma_A \vdash A' : s_1$ and $[A'] \equiv A$. We need to show that $\Gamma_A \Vdash \pi_{s_1,s_2}\,A : (\Pi x : \mathsf{T}_{s_1}\,A \,.\, \mathsf{U}_{s_2}) \to \mathsf{U}_{s_3}$. Let Γ_B, B be such that $\Gamma_A \subseteq \Gamma_B$ and $\Gamma_B \Vdash B : \Pi x : \mathsf{T}_{s_1}\,A \,.\, \mathsf{U}_{s_2}$. By Lemma 4.3.4, $\Gamma_B \vdash A' : s_1$. Therefore, $\Gamma_B, x : A' \vdash \mathrm{WF}$. By Lemma 6.2.23, $\Gamma_B, x : A' \Vdash B : \Pi x : \mathsf{T}_{s_1}\,A \,.\, \mathsf{U}_{s_2}$. By Corollary 6.2.22, we

have $\Gamma_B, x : A' \Vdash x : [\![A']\!]$. Since $A \equiv [A']$, we have $T_s\, A \equiv [\![A']\!]$. Therefore, $\Gamma_B, x : A' \Vdash x : \mathsf{T}_{s_1}\, A$. Therefore, $\Gamma_B, x : |A'| \Vdash B\, x : \mathsf{U}_{s_2}$. By definition and Corollary 6.2.14, there is B' such that $\Gamma_B, x : A' \vdash B' : s_2$ and $[B'] \equiv B\, x$. Therefore, $\Gamma_B \vdash \Pi x : A' \,.\, B' : s_3$ and

$$\begin{aligned}[\Pi x : A' \,.\, B'] &\equiv \pi_{s_1,s_2}\, [A']\ (\lambda x : \mathsf{T}_{s_1}\, [A'] \,.\, [B']) \\ &\equiv \pi_{s_1,s_2}\, A\, (\lambda x : \mathsf{T}_{s_1}\, A \,.\, B\, x) \\ &\equiv \pi_{s_1,s_2}\, A\, B.\end{aligned}$$

Therefore, $\Gamma_B \Vdash \pi_{s_1,s_2}\, A\, B : \mathsf{U}_{s_3}$.

- $$\frac{\Delta \vdash A_1 : \mathsf{Type} \qquad \Delta, x : A_1 \vdash M_1 : B_1}{\Delta \vdash \lambda x : A_1 \,.\, M_1 : \Pi x : A_1 \,.\, B_1}\ \textsc{lam}$$
  Without loss of generality, we can assume x is fresh. We have $\sigma\,(\lambda x : A_1 \,.\, M_1) = \lambda x : \sigma\,(A_1) \,.\, \sigma\,(M_1)$ and $\sigma\,(\Pi x : A_1 \,.\, B_1) = \Pi x : \sigma\,(A_1) \,.\, \sigma\,(B_1)$. Let Γ_N, N be such that $\Gamma \subseteq \Gamma_N$ and $\Gamma_N \Vdash N : \sigma\,(A_1)$. Define $\Delta_x = \Delta, x : A_1$ and $\sigma_x = \sigma\,[x := N]$. Then $\Gamma_N \Vdash \sigma_x : \Delta_x$. By induction hypothesis, $\Gamma_N \Vdash \sigma_x\,(M_1) : \sigma_x\,(B_1)$. Since $(\lambda x : \sigma\,(A_1) \,.\, \sigma\,(M_1))\ N \longrightarrow_\beta \sigma\,(M_1)\,\{x \backslash N\} = \sigma_x\,(M_1)$, by Lemma 6.2.24, $\Gamma_N \Vdash (\lambda x : \sigma\,(A_1) \,.\, \sigma\,(M_1))\ N : \sigma\,(B_1)\,\{x \backslash N\}$.

- $$\frac{\Delta \vdash M_1 : \Pi x : A_1 \,.\, B_1 \qquad \Delta \vdash N_1 : A_1}{\Gamma \vdash M_1\, N_1 : B_1\,\{x \backslash N_1\}}\ \textsc{app}$$
  Without loss of generality, we can assume x is fresh. We have $\sigma\,(M) = \sigma\,(M_1)\ \sigma\,(N_1)$ and $\sigma\,(B_1\,\{x \backslash N_1\}) = \sigma\,(B_1)\,\{x \backslash \sigma\,(N_1)\}$. By induction hypothesis, $\Gamma \Vdash \sigma\,(M_1) : \Pi x : \sigma\,(A_1) \,.\, \sigma\,(B_1)$ and $\Gamma \Vdash \sigma\,(N_1) : \sigma\,(A_1)$. By definition, $\Gamma \Vdash \sigma\,(M_1)\ \sigma\,(N_1) : \sigma\,(B_1)\,\{x \backslash \sigma\,(N)\}$.

- $$\frac{\Delta \vdash M : B \qquad \Delta \vdash A : \mathsf{Type} \qquad B \equiv A}{\Delta \vdash M : A}\ \textsc{conv}$$
  By induction hypothesis, $\Gamma \Vdash \sigma\,(M) : \sigma\,(B)$. Since $A \equiv B$, we have $\sigma\,(A) \equiv \sigma\,(B)$. By lemma Lemma 6.2.25, $\Gamma \Vdash \sigma\,(M) : \sigma\,(A)$.

□

A.3 Proofs of Chapter 8

Lemma (8.4.17)**.** *The following holds:*

1. $\Gamma \vdash \mathrm{WF} \wedge (x : A) \in \Gamma \implies \Gamma \models x \Rightarrow A$
2. $\Gamma \vdash \mathrm{WF} \wedge (s_1, s_2) \in \underline{\mathcal{A}} \implies \Gamma \models s_1 \Rightarrow s_2$
3. $\Gamma \models A \Rightarrow s_1 \wedge \Gamma, x : A \models B \Rightarrow s_2 \wedge (s_1, s_2, s_3) \in \underline{\mathcal{R}} \implies \Gamma \models \Pi x : A \,.\, B \Rightarrow s_3$
4. $\Gamma, x : A \models M \Rightarrow B \wedge \Gamma \vdash \Pi x : A \,.\, B : s \implies \Gamma \models \lambda x : A \,.\, M \Rightarrow \Pi x : A \,.\, B$

5. $\Gamma \models M \Rightarrow \Pi x : A \,.\, B \wedge \Gamma \vdash N : A \implies \Gamma \models M\,N \Rightarrow B\,\{x \backslash N\}$

6. $\Gamma \models M \Rightarrow A \wedge \Gamma \vdash B : s \wedge A \equiv B \implies \Gamma \models M \Rightarrow B$

7. $\Gamma \models M \Rightarrow A \wedge A \equiv s \implies \Gamma \models A \Rightarrow s$

Proof.

1. We have $\Gamma \vdash x : A$ by the Var rule. Suppose $\Gamma \vdash x : C$. By inversion, there exists A' such that $(x : A') \in \Gamma$ and $A' \preceq C$. By injectivity of Γ (Lemma 8.2.3), we must have $A' = A$. Therefore, $A \preceq C$.

2. We have $\Gamma \vdash s_1 : s_2$ by the Sort rule. Suppose $\Gamma \vdash s_1 : C$. By inversion, there exists s_2' such that $(s_1, s_2') \in \mathcal{A}$ and $s_2' \preceq C$. By definition of $\underline{\mathcal{A}}$ (Definition 8.4.8), $s_2 \preceq s_2'$. By transitivity, $s_2 \preceq C$.

3. We have $\Gamma \vdash \Pi x : A \,.\, B : s_3$ by the Prod rule. Suppose $\Gamma \vdash \Pi x : A \,.\, B : C$. By inversion, there exist s_1', s_2', s_3' such that $\Gamma \vdash A : s_1'$ and $\Gamma, x : A \vdash B : s_2'$ and $(s_1', s_2', s_3') \in \mathcal{R}$ and $s_3' \preceq C$. By principality (Definition 8.4.3), we have $s_1 \preceq s_1'$ and $s_2 \preceq s_2'$. By the local minimum property (Definition 8.4.1), there exists s_3'' such that $(s_1, s_2, s_3'') \in \mathcal{R}$ and $s_3'' \preceq s_3'$. By definition of $\underline{\mathcal{R}}$ (Definition 8.4.8), $s_3 \preceq s_3''$. By transitivity, $s_3 \preceq C$.

4. We have $\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B$ by the Prod rule. Suppose $\Gamma \vdash \lambda x : A \,.\, M : C$. By inversion, there exists B' such that $\Gamma, x : A \vdash M : B'$ and $\Gamma \vdash \Pi x : A \,.\, B' : s'$ and $\Pi x : A \,.\, B' \preceq C$. By principality (Definition 8.4.3), $B \preceq B'$ and so $\Pi x : A \,.\, B \preceq \Pi x : A \,.\, B'$. By transitivity, $\Pi x : A \,.\, B \preceq C$.

5. We have $\Gamma \vdash M\,N : B\,\{x \backslash N\}$ by the App rule. Suppose $\Gamma \vdash M\,N : C$. By inversion, there exist A', B' such that $\Gamma \vdash M : \Pi x : A' \,.\, B'$ and $\Gamma \vdash N : A'$ and $B'\,\{x \backslash N\} \preceq C$. By principality (Definition 8.4.3), we have $\Pi x : A \,.\, B \preceq \Pi x : A' \,.\, B'$ and so $A \equiv A'$ and $B \preceq B'$. By substitution (Lemma 8.1.5), $B\,\{x \backslash N\} \preceq B'\,\{x \backslash N\}$. By transitivity, $B\,\{x \backslash N\} \preceq C$.

6. We have $\Gamma \vdash M : B$ by the Conv rule. Suppose $\Gamma \vdash M : C$. By principality, $A \preceq C$. By transitivity, $B \preceq C$.

7. If $A = s'$ for some $s' \in \mathcal{S}^{\top}$ then $s' = s$ and we are done. Otherwise, $\exists s'$ such that $\Gamma \vdash A : s'$. By confluence, $A \longrightarrow^* s$. By subject reduction, $\Gamma \vdash s : s'$. Therefore, $\Gamma \vdash M : s$ by the Conv rule. Suppose $\Gamma \vdash M : C$. By principality, $A \preceq C$. By transitivity, $s \preceq C$.

□

Lemma (8.4.21)**.** *If $\Gamma \vdash A \Rightarrow s$ and $\Gamma \vdash A' \Rightarrow s'$ and $A' \preceq A$ then $\exists A'', s''$ such that $A' \longrightarrow^* A''$ and $\Gamma \vdash A'' \Rightarrow s''$ and $s'' \preceq s$.*

Proof. By Lemma 8.1.4, there are two cases to consider.

- If $A \equiv A'$ then by confluence $\exists A''$ such that $A \longrightarrow^* A''$ and $A' \longrightarrow^* A''$. By subject reduction (Lemma 8.2.8), $\Gamma \vdash A'' \Leftarrow s$. By Lemma 8.4.12, $\exists s''$ such that $\Gamma \vdash A'' \Rightarrow s''$ and $s'' \preceq s$.

- Otherwise, $\exists C_1, \ldots, C_n, r, r'$ such that $A \equiv \Pi x_1 : C_1 . \cdots \Pi x_n : C_n . r$ and $A' \equiv \Pi x_1 : C_1 . \cdots \Pi x_n : C_n . r'$ and $r' \preceq r$. By confluence, $\exists D_1, \ldots, D_n$ such that $A \longrightarrow^* \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . r$ and $A' \longrightarrow^* \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . r'$. By subject reduction (Lemma 8.2.8), $\Gamma \vdash \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . r \Leftarrow s$ and $\Gamma \vdash \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . r' \Leftarrow s'$. By inversion and the local minimum property, we can show that $\exists s''$ such that $\Gamma \vdash \Pi x_1 : D_1 . \cdots \Pi x_n : D_n . r' \Rightarrow s''$ and $s'' \preceq s$.

□

Lemma (8.4.22)**.** *If* $\Gamma \vdash M : C$ *then* $\exists A$ *such that* $\Gamma \vdash M \Rightarrow A$ *and* $A \preceq C$*. If* $\Gamma \vdash \mathrm{WF}$ *then* $\Gamma \vdash \Rightarrow \mathrm{WF}$*.*

Proof. By induction on the typing derivation.

- Case Empty. Then $\varnothing \vdash \Rightarrow \mathrm{WF}$.

- Case Decl. By induction hypothesis and Lemma 8.4.19, $\Gamma \vdash \Rightarrow \mathrm{WF}$ and $\exists s'$ such that $\Gamma \vdash A \Rightarrow s'$. Therefore, $\Gamma, x : A \vdash \Rightarrow \mathrm{WF}$.

- Case Var. By induction hypothesis, $\Gamma \vdash \Rightarrow \mathrm{WF}$. Therefore, $\Gamma \vdash x \Rightarrow A$.

- Case Sort. By induction hypothesis, $\Gamma \vdash \Rightarrow \mathrm{WF}$. By Lemma 8.4.9, $\exists s_2'$ such that $(s_1, s_2') \in \underline{\mathcal{A}}$. Therefore, $\Gamma \vdash s_1 \Rightarrow s_2'$ and $s_2' \preceq s_2$.

- Case Prod. By induction hypothesis and Lemma 8.4.19, $\exists s_1', s_2'$ such that $\Gamma \vdash A \Rightarrow s_1'$ and $\Gamma, x : A \vdash B \Rightarrow s_2'$ and $s_1' \preceq s_1$ and $s_2' \preceq s_2$. By the local minimum property, $\exists r_3$ such that $(s_1', s_2', r_3) \in \mathcal{R}$ and $r_3 \preceq s_3$. By Lemma 8.4.9, $\exists s_3'$ such that $(s_1', s_2', s_3') \in \underline{\mathcal{R}}$. Therefore, $\Gamma \vdash \Pi x : A . B \Rightarrow s_3'$ and $s_3' \preceq r_3 \preceq s_3$.

- Case Lam. By induction hypothesis and Lemma 8.4.19, $\exists B', s'$ and such that $\Gamma, x : A \vdash M \Rightarrow B'$ and $\Gamma \vdash \Pi x : A . B \Rightarrow s'$ and $B' \preceq B$ and $s' \preceq s$. By inversion of minimal types (Lemma 8.4.14), $\exists s_1', s_2'$ such that $\Gamma \vdash A \Rightarrow s_1'$ and $\Gamma, x : A \vdash B \Rightarrow s_2'$ and $(s_1', s_2', s') \in \underline{\mathcal{R}}$. By correctness of minimal types (Lemma 8.4.15) and Lemma 8.4.21, $\exists B''$ and s_2'' such that $B' \longrightarrow^* B''$ and $\Gamma, x : A \vdash B'' \Rightarrow s_2''$. By conversion, $\Gamma, x : A \vdash M \Rightarrow B''$. By the local minimum property and Lemma 8.4.9, $\exists s''$ such that $(s_1', s_2'', s'') \in \underline{\mathcal{R}}$. Therefore, $\Gamma \vdash \Pi x : A . B'' \Rightarrow s''$. Therefore, $\Gamma \vdash \lambda x : A . M : \Pi x : A . B''$ and $\Pi x : A . B'' \equiv \Pi x : A . B' \preceq \Pi x : A . B$.

- Case App. By induction hypothesis and Lemma 8.4.20, $\exists A', B'$ such that $\Gamma \vdash M \Rightarrow \Pi x : A'' . B'$ and $\Gamma \vdash N \Rightarrow A'$ and $A' \preceq A$ and $A'' \equiv A$ and $B' \preceq B$. By correctness of minimal types (Lemma 8.4.15) and inversion of minimal types (Lemma 8.4.14), we have $\Gamma \vdash A'' \Rightarrow s_1$ for some s_1. Therefore, $\Gamma \vdash N \Leftarrow A''$ by the Check rule. Therefore, $\Gamma \vdash M\, N \Rightarrow B' \{x \backslash N\}$. By substitution (Lemma 8.1.5), $B' \{x \backslash N\} \preceq B \{x \backslash N\}$.

- Case Sub. By induction hypothesis, $\exists A'$ such that $\Gamma \vdash M \Rightarrow A'$ and $A' \preceq A$. By transitivity, $A' \preceq B$.

- Case Sub-sort. By induction hypothesis, $\exists A'$ such that $\Gamma \vdash M \Rightarrow A'$ and $A' \preceq A$. By transitivity, $A' \preceq s$.

□

A.4 Proofs of Chapter 9

Lemma (9.2.4)**.** *If* $\Gamma, x : A, \Gamma' \vdash_{\lambda\mathcal{P}_{\preceq}} M \Rightarrow B$ *and* $\Gamma \vdash_{\lambda\mathcal{P}_{\preceq}} N \Leftarrow A$ *then* $[M]\left\{x\backslash [N]_{\vdash A}\right\} \equiv_{\beta\Sigma} [M\{x\backslash N\}]_{\vdash B\{x\backslash N\}}$*. More precisely,*

$$[M]_{\Gamma,x:A,\Gamma'}\left\{x\backslash [N]_{\Gamma\vdash A}\right\} \equiv_{\beta\Sigma} [M\{x\backslash N\}]_{\Gamma,\Gamma'\{x\backslash N\}\vdash B\{x\backslash N\}}.$$

Proof. First note that the statement makes sense because $\Gamma, \Gamma'\{x\backslash N\} \vdash M\{x\backslash N\} : B\{x\backslash N\}$ by 8.2.4. The proof follows by induction on M.

- Case $M = x$. Then $B \equiv A$. We have $A\{x\backslash N\} = A$ because $x \notin A$. Therefore,

$$\begin{aligned}[x]\left\{x\backslash [N]_{\vdash A}\right\} &= x\left\{x\backslash [N]_{\vdash A}\right\} \\ &= [N]_{\vdash A} \\ &= [x\{x\backslash N\}]_{\vdash A\{x\backslash N\}}.\end{aligned}$$

- Case $M = y \neq x$. Then we have $(y : B\{x\backslash N\}) \in \Gamma, \Gamma'\{x\backslash N\}$. Therefore,

$$\begin{aligned}[y]\left\{x\backslash [N]_{\vdash A}\right\} &= y\left\{x\backslash [N]_{\vdash A}\right\} \\ &= y \\ &= [y]_{\vdash B\{x\backslash N\}} \\ &= [y\{x\backslash N\}]_{\vdash B\{x\backslash N\}}.\end{aligned}$$

- Case $M = s_1$. Then $B \equiv s_2$ where $(s_1 : s_2) \in \underline{\mathcal{A}}$. Therefore,

$$\begin{aligned}[s_1]\left\{x\backslash [N]_{\vdash A}\right\} &= \mathsf{u}_{s_1}\left\{x\backslash [N]_{\vdash A}\right\} \\ &= \mathsf{u}_{s_1} \\ &= [s_1]_{\vdash B} \\ &= [s_1\{x\backslash N\}]_{\vdash B\{x\backslash N\}}.\end{aligned}$$

- Case $M = \Pi y : C \,.\, D$. Then $B \equiv s_3$ where $\Gamma \vdash C \Rightarrow s_1$ and $\Gamma, y : C \vdash D \Rightarrow s_2$ and $(s_1, s_2, s_3) \in \underline{\mathcal{R}}$. Therefore,

$$\begin{aligned}
[\Pi y : C \,.\, D] \left\{x \backslash [N]_{\vdash A}\right\} &= (\pi_{s_1,s_2} [C] \ (\lambda y \,.\, [D])) \left\{x \backslash [N]_{\vdash A}\right\} \\
&= \pi_{s_1,s_2} \left([C] \left\{x \backslash [N]_{\vdash A}\right\}\right) \left(\lambda y \,.\, [D] \left\{x \backslash [N]_{\vdash A}\right\}\right) \\
&\equiv \pi_{s_1,s_2} [C \{x \backslash N\}]_{\vdash s_1} \left(\lambda y \,.\, [D \{x \backslash N\}]_{\vdash s_2}\right) \\
&\quad \text{by induction hypothesis} \\
&\equiv [\Pi y : C \{x \backslash N\} \,.\, D \{x \backslash N\}]_{\vdash s_3} \\
&\quad \text{by lemma 9.2.3} \\
&= [(\Pi y : C \,.\, D) \{x \backslash N\}]_{\vdash s_3 \{x \backslash N\}} \,.
\end{aligned}$$

- Case $M = \lambda y : C \,.\, M_1$. Then $B \equiv \Pi y : C \,.\, D$ where $\Gamma, y : C \vdash M_1 \Rightarrow D$. Therefore,

$$\begin{aligned}
[\lambda y : C \,.\, M_1] \left\{x \backslash [N]_{\vdash A}\right\} &= (\lambda y : [\![C]\!] \,.\, [M_1]) \left\{x \backslash [N]_{\vdash A}\right\} \\
&= \lambda y : [\![C]\!] \left\{x \backslash [N]_{\vdash A}\right\} \,.\, [M_1] \left\{x \backslash [N]_{\vdash A}\right\} \\
&\equiv \lambda y : [\![C \{x \backslash N\}]\!] \,.\, [M_1 \{x \backslash N\}]_{\vdash D\{x \backslash N\}} \\
&\quad \text{by induction hypothesis and lemma 9.2.1} \\
&\equiv [\lambda y : C \{x \backslash N\} \,.\, M_1 \{x \backslash N\}]_{\vdash \Pi y : C\{x \backslash N\} . D\{x \backslash N\}} \\
&\quad \text{by lemma 9.2.3} \\
&= [(\lambda y : C \,.\, M_1) \{x \backslash N\}]_{\vdash (\Pi y : C . D)\{x \backslash N\}} \,.
\end{aligned}$$

- Case $M = M_1 \, M_2$. Then $B \equiv D \{y \backslash M_2\}$ where $\Gamma \vdash M_1 \Rightarrow \Pi y : C \,.\, D$ and $\Gamma \vdash M_2 \Leftarrow C$. Therefore,

$$\begin{aligned}
[M_1 \, M_2] \left\{x \backslash [N]_{\vdash A}\right\} &= \left([M_1] \ [M_2]_{\vdash C}\right) \left\{x \backslash [N]_{\vdash A}\right\} \\
&= [M_1] \left\{x \backslash [N]_{\vdash A}\right\} \ [M_2]_{\vdash C} \left\{x \backslash [N]_{\vdash A}\right\} \\
&\equiv [M_1 \{x \backslash N\}]_{\vdash (\Pi y : C . D)\{x \backslash N\}} \ [M_2 \{x \backslash N\}]_{\vdash C\{x \backslash N\}} \\
&\quad \text{by induction hypothesis and lemma 9.2.2} \\
&= [M_1 \{x \backslash N\} \ M_2 \{x \backslash N\}]_{\vdash D\{x \backslash N\}\{y \backslash M_2\{x \backslash N\}\}} \\
&\quad \text{by lemma 9.2.3} \\
&= [(M_1 \, M_2) \{x \backslash N\}]_{\vdash D\{y \backslash M_2\}\{x \backslash N\}} \,.
\end{aligned}$$

□

Lemma (9.2.15)**.** *If* $\Gamma \vdash_{\lambda \mathcal{P}_{\preceq}} M \Rightarrow A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda \Pi R} [M]_{\Gamma} : [\![A]\!]_{\Gamma}$. *If* $\Gamma \vdash_{\lambda \mathcal{P}_{\preceq}} M \Leftarrow A$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda \Pi R} [M]_{\Gamma \vdash A} : [\![A]\!]$. *If* $\Gamma \vdash_{\lambda \mathcal{P}_{\preceq}} \Rightarrow \mathrm{WF}$ *then* $\Sigma, [\![\Gamma]\!] \vdash_{\lambda \Pi R} \mathrm{WF}$.

Proof. By induction on the typing derivations.

1. $\dfrac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Empty}$.
Then $[\![\varnothing]\!] = \varnothing$ and $\Sigma \vdash \mathrm{WF}$.

2. $\dfrac{\Gamma \vdash A \Rightarrow s \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Decl}$.
Then $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$. By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [A] : [\![s]\!]$. By Lemma 9.2.14 $[\![\Gamma]\!] \vdash [\![A]\!] : \mathsf{Type}$. Therefore, $\Sigma, [\![\Gamma]\!], x : [\![A]\!] \vdash \mathrm{WF}$.

3. $\dfrac{\Gamma \vdash\Rightarrow \mathrm{WF} \qquad (x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Var}$.
Then $[x] = x$. By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash \mathrm{WF}$. Since $(x : A) \in \Gamma$, we have $(x : [\![A]\!]) \in [\![\Gamma]\!]$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash x : [\![A]\!]$.

4. $\dfrac{\Gamma \vdash\Rightarrow \mathrm{WF} \qquad (s_1 : s_2) \in \underline{\mathcal{A}}}{\Gamma \vdash s_1 \Rightarrow s_2}\ \textsc{Sort}$.
Then $[s_1] = \mathsf{u}_{s_1}$. By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash \mathrm{WF}$. Since $(s_1, s_2) \in \underline{\mathcal{A}}$, we have $(\mathsf{u}_{s_1} : \mathsf{U}_{s_2}) \in \Sigma$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash \mathsf{u}_{s_1} : \mathsf{U}_{s_2}$. By Lemma 9.2.10, $\Sigma, [\![\Gamma]\!] \vdash \mathsf{u}_{s_1} : [\![s_2]\!]$.

5. $\dfrac{\Gamma \vdash A \Rightarrow s_1 \qquad \Gamma, x : A \vdash B \Rightarrow s_2 \qquad (s_1, s_2, s_3) \in \underline{\mathcal{R}}}{\Gamma \vdash \Pi x : A \,.\, B \Rightarrow s_3}\ \textsc{Prod}$.
Then $[\Pi x : A \,.\, B] = \pi^{s_3}_{s_1,s_2}\,[A]\ (\lambda x \,.\, [B])$. By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [A] : [\![s_1]\!]$ and $\Sigma, [\![\Gamma]\!], x : [\![A]\!] \vdash [B] : [\![s_2]\!]$. By Lemma 9.2.10, $\Sigma, [\![\Gamma]\!] \vdash [A] : \mathsf{U}_{s_1}$ and $\Sigma, [\![\Gamma]\!], x : [\![A]\!] \vdash [B] : \mathsf{U}_{s_2}$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash \pi_{s_1,s_2}\,[A]\ (\lambda x \,.\, [B]) : \mathsf{U}_{s_3}$. By Lemma 9.2.10, $\Sigma, [\![\Gamma]\!] \vdash \pi_{s_1,s_2}\,[A]\ (\lambda x \,.\, [B]) : [\![s_3]\!]$.

6. $\dfrac{\Gamma, x : A \vdash M \Rightarrow B \qquad \Gamma \vdash \Pi x : A \,.\, B \Rightarrow s}{\Gamma \vdash \lambda x : A \,.\, M \Rightarrow \Pi x : A \,.\, B}\ \textsc{Lam}$.
Then $[\lambda x : A \,.\, M] = \lambda x : [\![A]\!] \,.\, [M]$. By induction hypothesis, $\Sigma, [\![\Gamma]\!], x : [\![A]\!] \vdash [M] : [\![B]\!]$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash \lambda x : [\![A]\!] \,.\, [M] : \Pi x : [\![A]\!] \,.\, [\![B]\!]$. By Lemma 9.2.11, $\Sigma, [\![\Gamma]\!] \vdash \lambda x : [\![A]\!] \,.\, [M] : [\![\Pi x : A \,.\, B]\!]$.

7. $\dfrac{\Gamma \vdash M \Rightarrow \Pi x : A \,.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash M\,N \Rightarrow B\,\{x \backslash N\}}\ \textsc{App}$.
Then $[M\,N] = [M]\ [N]_{\vdash A}$. By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![\Pi x : A \,.\, B]\!]$ and $\Sigma, [\![\Gamma]\!] \vdash [N]_{\vdash A} : [\![A]\!]$. By Lemma 9.2.11, $\Sigma, [\![\Gamma]\!] \vdash [M] : \Pi x : [\![A]\!] \,.\, [\![B]\!]$. Therefore, $\Sigma, [\![\Gamma]\!] \vdash [M]\ [N]_{\vdash A} : [\![B]\!]\left\{x \backslash [N]_{\vdash A}\right\}$. By Corollary 9.2.5 and Lemma 9.2.12, $\Sigma, [\![\Gamma]\!] \vdash [M]\ [N]_{\vdash A} : [\![B\,\{x \backslash N\}]\!]$.

8. $\dfrac{\Gamma \vdash M \Rightarrow A \qquad \Gamma \vdash B \Rightarrow s \qquad A \equiv_\beta B}{\Gamma \vdash M \Rightarrow B}\ \textsc{Conv}$.
By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![A]\!]$ and $\Sigma, [\![\Gamma]\!] \vdash [B] : [\![s]\!]$. By Lemma 9.2.14, $\Sigma, [\![\Gamma]\!] \vdash [\![B]\!] : \mathsf{Type}$. By Lemma 9.2.9, $[\![A]\!] \equiv [\![B]\!]$. By conversion, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![B]\!]$.

9. $$\frac{\Gamma \vdash M \Rightarrow A \qquad A \equiv_\beta s}{\Gamma \vdash M \Rightarrow s}\ \textsc{Conv-sort}.$$
By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![A]\!]$. By Lemma 9.2.9, $[\![A]\!] \equiv [\![s]\!]$. By conversion, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![B]\!]$.

10. $$\frac{\Gamma \vdash M \Rightarrow A \qquad \Gamma \vdash B \Rightarrow s \qquad A \preceq B}{\Gamma \vdash M \Leftarrow B}\ \textsc{Check}.$$
By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![A]\!]$ and $\Sigma, [\![\Gamma]\!] \vdash [B] : [\![s]\!]$. By Lemma 9.2.14, $\Sigma, [\![\Gamma]\!] \vdash [\![B]\!] : \mathsf{Type}$. By Lemma 9.2.13, $\Sigma, [\![\Gamma]\!] \vdash [M]_{\vdash B} : [\![B]\!]$.

11. $$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq s}{\Gamma \vdash M \Leftarrow s}\ \textsc{Check-sort}.$$
By induction hypothesis, $\Sigma, [\![\Gamma]\!] \vdash [M] : [\![A]\!]$. By Lemma 9.2.13, $\Sigma, [\![\Gamma]\!] \vdash [M]_{\vdash B} : [\![s]\!]$.

$\square$

B

Original presentations

B.1 The $\lambda\Pi$-calculus

Syntax

type constants	a, p	
term constants	c, f	
term variables	x, y	
kinds	K	$::= \mathsf{Type} \mid \Pi x : A \,.\, K$
types	A, B	$::= a \mid \Pi x : A \,.\, B \mid \lambda x : A \,.\, B \mid A\, N$
objects	M, N	$::= x \mid c \mid \lambda x : A \,.\, M \mid M\, N$
contexts	Γ, Δ	$::= \varnothing \mid \Gamma, x : A$
signatures	Σ	$::= \varnothing \mid \Sigma, a : K \mid \Sigma, c : A$

Typing

The different judgments are:

- $\Sigma \vdash \mathrm{WF}$, the signature Σ is well-formed
- $\Sigma \mid \Gamma \vdash \mathrm{WF}$, the context Γ is well-formed in the signature Σ

$$\frac{}{\varnothing \vdash \mathrm{WF}}\ \textsc{Sig-Empty}$$

$$\frac{\Sigma \mid \varnothing \vdash K\ \mathrm{WF} \qquad a \notin \Sigma}{\Sigma, a : K \vdash \mathrm{WF}}\ \textsc{Sig-Kind}$$

$$\frac{\Sigma \mid \varnothing \vdash A : \mathsf{Type} \qquad c \notin \Sigma}{\Sigma, c : A \vdash \mathrm{WF}}\ \textsc{Sig-Type}$$

$$\frac{\Sigma \vdash \mathrm{WF}}{\Sigma \mid \varnothing \vdash \mathrm{WF}}\ \textsc{Ctx-Empty}$$

$$\frac{\Sigma \mid \Gamma \vdash A : \mathsf{Type} \qquad x \notin \Gamma}{\Sigma \mid \Gamma, x : A \vdash \mathrm{WF}}\ \textsc{Ctx-Type}$$

$$\frac{\Sigma \mid \Gamma \vdash \mathrm{WF}}{\Sigma \mid \Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{Kind-Type}$$

$$\frac{\Sigma \mid \Gamma \vdash A : \mathsf{Type} \qquad \Sigma \mid \Gamma, x : A \vdash K\ \mathrm{WF}}{\Sigma \mid \Gamma \vdash \Pi x : A \,.\, K\ \mathrm{WF}}\ \textsc{Kind-Prod}$$

Figure B.1 – Original typing rules of the system $\lambda\Pi$ (part I)

- $\Sigma \mid \Gamma \vdash K$ WF, the kind K is well-formed in the context Γ and signature Σ
- $\Sigma \mid \Gamma \vdash A : K$, type A has kind K in the context Γ and signature Σ
- $\Sigma \mid \Gamma \vdash M : A$, object M has type A in the context Γ and signature Σ

The typing rules are shown in Figures B.1 and B.2.

Notes

This presentation suffers from a lot of duplication in the syntax and typing rules. An upside is that we do not need to check the sort of the product type in the λ rule because we already know the sorts: they are given by the syntactic category. Another advantage is that definitions and proofs by induction on the structure of the syntax are easier, since we know at all times what is an object, what is a type, and what is a kind.

$$\frac{\Sigma \mid \Gamma \vdash \mathrm{WF} \qquad (a : K) \in \Sigma}{\Sigma \mid \Gamma \vdash a : K}\ \textsc{Type-Const}$$

$$\frac{\Sigma \mid \Gamma \vdash A : \mathsf{Type} \qquad \Sigma \mid \Gamma, x : A \vdash B : \mathsf{Type}}{\Sigma \mid \Gamma \vdash \Pi x : A \,.\, B : \mathsf{Type}}\ \textsc{Type-Prod}$$

$$\frac{\Sigma \mid \Gamma, x : A \vdash B : K}{\Sigma \mid \Gamma \vdash \lambda x : A \,.\, B : \Pi x : A \,.\, K}\ \textsc{Type-Lam}$$

$$\frac{\Sigma \mid \Gamma \vdash B : \Pi x : A \,.\, K \qquad \Sigma \mid \Gamma \vdash N : A}{\Sigma \mid \Gamma \vdash B\, N : K\,\{x \backslash N\}}\ \textsc{Type-App}$$

$$\frac{\Sigma \mid \Gamma \vdash A : K \qquad \Sigma \mid \Gamma \vdash K' \ \mathrm{WF} \qquad K \equiv_\beta K'}{\Sigma \mid \Gamma \vdash A : K'}\ \textsc{Type-Conv}$$

$$\frac{\Sigma \mid \Gamma \vdash \mathrm{WF} \qquad (c : A) \in \Sigma}{\Sigma \mid \Gamma \vdash c : A}\ \textsc{Const}$$

$$\frac{\Sigma \mid \Gamma \vdash \mathrm{WF} \qquad (x : A) \in \Gamma}{\Sigma \mid \Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Sigma \mid \Gamma, x : A \vdash M : B}{\Sigma \mid \Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \Pi x : A \,.\, B \qquad \Sigma \mid \Gamma \vdash N : A}{\Sigma \mid \Gamma \vdash M\, N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Sigma \mid \Gamma \vdash M : A \qquad \Sigma \mid \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Sigma \mid \Gamma \vdash M : B}\ \textsc{Conv}$$

Figure B.2 – Original typing rules of the system $\lambda\Pi$ (part II)

B.2 Pure type systems

Syntax

$$
\begin{array}{lll}
\text{constants } c & & \in \mathcal{O} \\
\text{terms} & M, N, A, B & \in \mathcal{T} ::= x \mid c \mid M\,N \mid \lambda x : A \,.\, M \mid \Pi x : A \,.\, B \\
\text{contexts} & \Gamma, \Delta & \in \mathcal{G} ::= \varnothing \mid \Gamma, x : A
\end{array}
$$

Specifications

- $\mathcal{S} \subseteq \mathcal{O}$
- $\mathcal{A} \subseteq \mathcal{O} \times \mathcal{S}$
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$

Typing

There is only the typing judgment $\Gamma \vdash M : A$ and there is no context formation judgment. The typing rules are shown in Figure B.3.

Notes

This presentation allows special constants which are not sorts but which have a predefined sort type. This allows the definition of the simply typed λ calculus with only one sort $*$ instead of two sorts $*, \Box$ by using special constants for base types, but that disallows the addition of additional type variables in the context. For example, the simply typed λ-calculus with one base type ι corresponds to the PTS $\lambda \to_\iota$:

$$
(\to_\iota) \begin{bmatrix} \mathcal{S} = * \\ \mathcal{A} = (\iota, *) \\ \mathcal{R} = (*, *) \end{bmatrix}
$$

Another difference is that this presentation does not use a separate context formation judgment, relying instead on a weakening rule.

$$\frac{(s_1 : s_2) \in \mathcal{A}}{\varnothing \vdash s_1 : s_2}\ \textsc{Start}$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash M : B}\ \textsc{Weak}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A \,.\, B : s_3}\ \textsc{Prod}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A \,.\, B : s}{\Gamma \vdash \lambda x : A \,.\, M : \Pi x : A \,.\, B}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash M : \Pi x : A \,.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B\,\{x \backslash N\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}\ \textsc{Conv}$$

Figure B.3 – Original typing rules of the system $\lambda\mathcal{P}$