Formal Verification of the Equivalence of System F and the Pure Type System L2

Jonas Kaiser

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

UNIVERSITÄT
DES
SAARLANDES

Saarbrücken, 2019

Berichterstatter:
Prof. Dr. Gert Smolka, Universität des Saarlandes
Prof. Brigitte Pientka, Ph.D., McGill University

Dekan:
Univ.-Prof. Dr. Sebastian Hack

Prüfungsausschuss:
Prof. Dr. Markus Bläser (Vorsitz)
Prof. Dr. Gert Smolka
Prof. Brigitte Pientka, Ph.D.
Dr. Richard Membarth (Beisitzer)

Tag des Kolloquiums:
11. Juli 2019

Textfassung vom 11. August 2019

Abstract

We develop a formal proof of the equivalence of two different variants of System F. The first is close to the original presentation where expressions are separated into distinct syntactic classes of types and terms. The second, L2 (also written as $\lambda 2$), is a particular pure type system (PTS) where the notions of types and terms, and the associated expressions are unified in a single syntactic class. The employed notion of equivalence is a bidirectional reduction of the respective typing relations. A machine-verified proof of this result turns out to be surprisingly intricate, since the two variants noticeably differ in their expression languages, their type systems and the binding of local variables.

Most of this work is executed in the Coq theorem prover and encompasses a general development of the PTS metatheory, an equivalence result for a stratified and a PTS variant of the simply typed λ-calculus as well as the subsequent extension to the full equivalence result for System F. We utilise nameless de Bruijn syntax with parallel substitutions for the representation of variable binding and develop an extended notion of context morphism lemmas as a structured proof method for this setting.

We also provide two developments of the equivalence result in the proof systems Abella and Beluga, where we rely on higher-order abstract syntax (HOAS). This allows us to compare the three proof systems, as well as HOAS and de Bruijn for the purpose of developing formal metatheory.

Kurzzusammenfassung

Wir präsentieren einen maschinell verifizierten Beweis der Äquivalenz zweier Darstellungen des Lambda-Kalküls System F. Die erste unterscheidet syntaktisch zwischen Termen und Typen und entspricht somit der geläufigen Form. Die zweite, L2 bzw. $\lambda 2$, ist ein sog. Pure Type System (PTS), bei welchem alle Ausdrücke in einer syntaktischen Klasse zusammen fallen. Unser Äquivalenzbegriff ist eine bidirektionale Reduktion der jeweiligen Typrelationen. Ein formaler Beweis dieser Eigenschaft ist aufgrund der Unterschiede der Ausdruckssprachen, der Typrelationen und der Bindung lokaler Variablen überraschend anspruchsvoll.

Der Hauptteil dieser Arbeit wurde in dem Beweisassistenten Coq entwickelt und umfasst eine Abhandlung der PTS Metatheorie, sowie einen Äquivalenzbeweis für das einfach getypte Lambda-Kalkül, welcher dann zu dem vollen Ergebnis für System F skaliert wird. Für die Darstellung lokaler Variablenbindung verwenden wir de Bruijn Syntax, gepaart mit parallelen Substitutionen. Außerdem entwickeln wir eine generalisierte Form von Kontext-Morphismen Lemmas, welche eine strukturierte Beweismethodik in diesem Umfeld liefern.

Darüber hinaus betrachten wir zwei weitere Formalisierungen des Äquivalenzresultats in den Beweissystemen Abella und Beluga, welche beide höherstufige abstrakte Syntax (HOAS) zur Darstellung lokaler Bindung verwenden. Dies ermöglicht es uns, sowohl die drei Beweissysteme, als auch den HOAS und den de Bruijn Ansatz mit Hinblick auf die Entwicklung formaler Metatheorie zu vergleichen.

Acknowledgements

I am indebted to a number of people without whom this work would not have been possible and I would like to use this space to express my sincerest gratitude.

First and foremost I thank Gert for his supervision and for opening my eyes to the beauty of a well-engineered, formalised proof, as well as the attention to detail that goes with it. Meanwhile, it was Chad who taught me that, despite their apparent power, proof assistants are not a substitute for but rather a complement to rigorous mathematical thinking.

This project benefitted from countless discussions with my collaborators and colleagues at the Programming Systems Lab. I want to thank, in particular, Tobias for introducing me to the intricacies of de Bruijn syntax, Steven for being a fountain of ideas that would never run dry, and Kathrin for her dedication to the Autosubst 2 project. My gratitude also extends to the rest of the lab which provided a relaxed, pleasant and inspiring research environment. Here Ute deserves a special mention for keeping everything running as smoothly as it did and helping with all kinds of organisational concerns.

Beyond the lab it was Dale, who approached me in Porto to propose Abella as a candidate proof system for my equivalence proof, and later Brigitte, who in Paris introduced me to Beluga. Without those two, the project would likely have remained stuck in the walled-in perspective of a single proof system.

Contrary to a common adage in academia, there is more to life than research. Hence I would like to thank my family and friends for their moral support and just being there these past years. A big thank you goes out to Klaas, Christian and Roland for many splendid game nights and the much needed levity, as well as to Linus and Fabian for always offering sound advice and an outside perspective.

Finally, and most important of all, I want to say thank you to my wife Verena for her unwavering love, encouragement and patience through the many ups and downs of my research and dissertation phase. The lines in this text are mine, but it is her presence that carries them.

Contents

1 Introduction

This thesis formally verifies the claim that System F, also known as the polymorphic λ-calculus (PLC), and its formulation as the pure type system $\lambda 2$ (pronounced "lambda-two")[1] are equivalent. Here, and throughout the remainder of this text, **formal** means machine-verified with a proof assistant. Both System F and pure type systems (PTSs) are formalisms that consist of a syntactic expression language, a notion of computation and a typing discipline. Accordingly, both systems support the notion of typing judgements $\Gamma \vdash s : t$, which assign types t to terms s under a given typing context Γ. We consider two formalisms as **equivalent** when they are **co-typeable**, that is we have a bidirectional reduction of typing between the two formalisms. The complementary notion of co-reducibility is not covered in any detail in this text.

As a precursor to the full result for System F we will first tackle a similar result for the simply typed λ-calculus (STLC) and the corresponding PTS $\lambda_{\to}$ (pronounced "lambda-base"). This allows us to demonstrate some of the necessary proof constructions and definitions in a simplified setting. We also present a general development of the PTS metatheory since a lot of the required properties can be uniformly established for the two PTSs for which we consider the correspondence result.

Viewed from a slightly different angle, our work illustrates how we can establish that a certain PTS, like $\lambda 2$, faithfully captures a given traditional formalism like PLC.

1.1 Motivation

System F is well-established and its metatheory has been discussed at length in the literature. Gentle introductions can be found in [Pie02, Har13], while an excellent in-depth discussion is given in [SU06]. It is both interesting from a logical point of view, since it extends intuitionistic logic with universal quantification, as well as a prototypical programming language with support for polymorphic types. Girard introduced the system with a focus on the first aspect in [Gir72], while Reynolds independently invented it based on the latter motivation [Rey74].

We found that since its inception, quite a number of different presentations of the formalism have been proposed. Some of the differences are negligible, others not so much. While not problematic per se, it becomes an issue as soon as we want to transfer metatheoretic results from one presentation to another. Such results include properties like the correctness of types or the compatibility of the typing judgement

[1] As in the title of this thesis, we also denote this system as L2, where the usage of Greek letters could be undesirable.

with β-substitutions. The soundness of such transfers clearly depends on a sufficiently strong notion of system equivalence.

As it turns out, the need for such a correspondence result is rarely acknowledged, and when it is, its proof is considered technical but obvious [Geu93]. We agree with the "technical" part of this assessment, but a proof is certainly not straightforward. The two variants for which we are going to formally establish an equivalence result do not even share the same expression language and their type systems exhibit noticeably different structures. In addition, both variants involve the local binding of variables, which immediately puts the issue of α-equivalence on the table. Add to this different, but partially overlapping, scopes of free variables and it should become clear that the required result is not as obvious as it might appear at first.

Carefully engineered definitions and suitably generalised statements are paramount to tackling the posed equivalence claim in a formal development, where all the above issues are much more pronounced than in a traditional on-paper proof. This issue was nicely summarised by Berardi in the context of a formal normalisation proof in the proof system LEGO.

> *We may think of [the] proof as an iceberg. In the top of it, we find what we usually consider the real proof; underwater, the most of the matter, consisting of all mathematical preliminaries a reader must know in order to understand what is going on.* — Berardi [Ber90a]

An illustration of this analogy is shown in Figure 1.1, where all the formal technicalities that have to be handled sit below the proverbial water line. The image is due to D. Schlimm and B. Pientka and was taken from [Pie15].

We can add more voices to this issue. Let us first cite the mathematical perspective where the exact challenge of our thesis is discussed, that is the correspondence of well known systems and their PTS variants.

> *[Barendregt's λ-cube] includes well-known systems like the simply typed and polymorphically typed lambda calculus. To show that the two representations of these systems are in fact the same requires some technical but not difficult work.* — Geuvers [Geu93]

Let us now contrast this with the formal perspective of machine-verified proof developments. Consider for example the following abstract from a publication in 2008.

> *Machine-checked proofs of properties of programming languages have become a critical need, both for increased confidence in large and complex designs and as a foundation for technologies such as proof-carrying code. However, constructing these proofs remains a black art, involving many choices in the formulation of definitions and theorems that make a huge cumulative difference in the difficulty of carrying out large formal developments. The representation and manipulation of terms with variable binding is a key issue.* — Aydemir et al. [ACP+08]

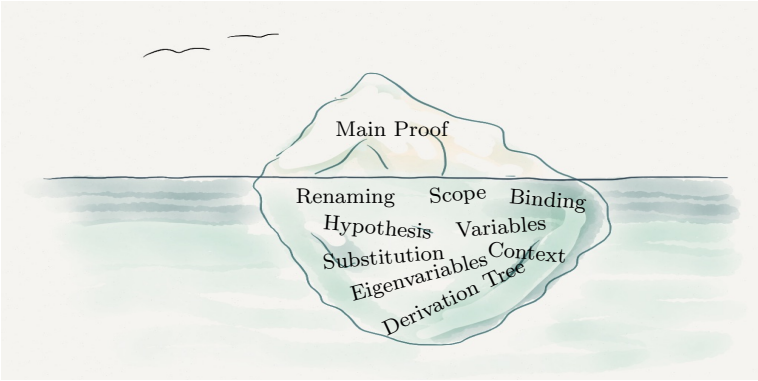


Figure 1.1: Proofs: the tip of the iceberg.

When we fast-forward several years to the FSCD conference in 2016 we were able to witness the following bon mot of A. Ahmed during her invited talk on the challenges of establishing compositional compiler correctness.

> *Binders should have been a solved problem 10 years ago . . .*
>
> — Ahmed, Porto, Portugal, 2016, paraphrased

The subsequent presentation then made it very clear that the problem of handling binders is not at all "solved", despite numerous suggested approaches.

What all of this emphasises, is that problems which look seemingly innocent from a mathematical point of view, may have tremendous hidden complications when all implicit assumptions have to be taken at face value. The area of variable binding in programming languages and logics is particularly prone to this deceptive simplicity, and our work demonstrates what it takes to formally establish the "not difficult" problem posed in [Geu93]. The fact that a formal proof forces us to explicitly consider all these added concerns should not, in principle, be considered a burden, but an opportunity to obtain a really rigorous result. In addition, it allows us to uncover complexities, and also potential fallacies, that might otherwise slip by unnoticed in a conventional on-paper proof. As we will see, there are, in practice, noticeable differences in the management of all the added concerns among different proof systems.

1.2 Related Work

The main topics of this thesis are pure type systems, correspondence proofs of typed λ-calculi, and a comparison of three proof assistants and their native syntax representations. All three are well-established topics and we briefly survey the most relevant literature.

1.2.1 Pure Type Systems

Pure type systems (PTSs) are an abstract formalism which captures a large family of typed λ-calculi. The framework was introduced by Barendregt [Bar91] for the purpose of analysing the fine-structure of the Calculus of Constructions (CC) [CH88], though Barendregt himself attributes the essential notion of a generalised type system to Berardi and Terlouw.[2] An early form of *pure systems* which constitutes a subset of the full PTS formalism appears in Berardi's doctoral dissertation [Ber90b], though he in turn attributes the ideas to Barendregt and also an unpublished manuscript by Terlouw from 1989, titled "A possibly more perspicuous proof of strong normalization in Coquand's system"[3].

In his seminal work, Barendregt shows that several well-known systems, including the simply typed λ-calculus [Chu40], System F [Gir72, Rey74], F_ω [Gir72] and LF [HHP87], appear as subsystems of CC by controlling which forms of abstraction and (dependent) function type formation are allowed. The inclusion hierarchy of the systems forms a cube-like structure, known as the **λ-cube**, which constitutes a particular subclass of the PTS framework. He then proceeds to show how several other formalisms like Girard's inconsistent System U [Gir72], various systems of the Automath family [vD80] and also various propositional and predicate logics can be presented in terms of the more general framework.

The PTS formalism is clearly constructive, though classical variants have been proposed. Take for example [BHS97], where a primitive double-negation operator is added to the system to facilitate classical reasoning.

The main advantage of PTSs arises from the fact that metatheoretic results can be obtained uniformly for all representable systems. This notion of abstract metatheory has been extensively explored in [GN91, vBJ93], including notions like the Church-Rosser property, strong normalisation or strengthening/context contraction.

Not surprisingly, various efforts have been made to obtain machine-verified versions of these results. An extensive body of work in this area is the Coq formalisation of Adams [Ada04], which even includes the rather involved strengthening proof of van Benthem-Jutting [vBJ93] for arbitrary PTSs.

Our formalisation of the PTS metatheory is close to the one of Adams, but it differs in a number of places. First of all, we reformulate the PTS notion in such a way that typing and context validity can be decoupled, while Adams works with the traditional variant where validity is built into the type system. The main benefit of our approach is that certain results can be established independent of context validity and our proofs further avoid auxiliary lemmas which establish validity at every context lookup. For a comparison of the two approaches to context validity see [Luo90]. A further advantage of our formulation is that the weakening rule is not required to be primitive; its admissibility is provable. Like Adams, we work with a de Bruijn encoding, though we resort to plain inductive types to represent our object languages, while Adams

[2] He claims to have learned the ideas from Berardi and Terlouw via private communication in 1988.
[3] We were, unfortunately, unable to obtain a copy of this text.

uses dependent types to ensure well-scopedness of terms. Both the plain encoding as well as Adams' dependent, well-scoped term encoding have advantages and drawbacks and we will discuss these later. We also do not adopt the first of his two "maxims for formalization" [Ada04, §2.1], that is the preference of recursive over inductive definitions. We make heavy use of inductive characterisations and found this to work rather well, in particular in settings where we would otherwise have to deal with partial functions. Furthermore, as far as we can tell, our notions of partial context renamings, type uniqueness *modulo partial context renaming* and the partial context renaming lemma appear to be novel constructions. The latter admits strengthening for functional PTSs as a special case and can be considered as the formal de Bruijn adaptation of the strengthening proof of [GN91]. Note that Adams instead directly formalises the rather technical general strengthening result of [vBJ93].

Another early formalisation of PTS metatheory in the proof assistant LEGO was published by McKinna and Pollack in [MP99]. They make the conscious decision to work with named variables rather than with the much simpler-to-formalise de Bruijn indices. Their justification for this choice is the use of the PTS formalism as a foundation for the construction of proof assistants where the user-facing portion of the system should present theorems and proof terms in a named fashion. While we do agree with the usability point of view, we still believe that the concerns of user presentation and metatheory development should be decoupled. The scope of their work is similar to that of Adams in [Ada04].

In addition to, and because of, its solid theoretical foundation, the PTS formalism has also been considered as a basis for designing functional programming languages. Consider for example *Henk* [JM97] which is based on the λ-cube or the work of Roorda and Jeuring [Roo00], which extends the full PTS setup with algebraic datatypes, casing and a type checking algorithm, among other things, to obtain a workable language. Note that both works are mostly intended as powerful intermediate language representations of a compiler rather than as usable source languages.

1.2.2 Correspondence Proofs

The correspondence of two typed λ-calculi essentially amounts to an isomorphism proof. That is, we need a structure-preserving mapping between the two systems. The mapping can come in the form of translation functions or as a relation on syntactic expressions. The crucial question however is, what kinds of structures should be preserved? For typed λ-calculi the natural choices are the typing discipline and the reduction behaviour.

This notion of system equivalence as co-typeability and co-reducibility is well established and can for example be found in [BDS13], where the equivalence of three different variants of STLC (varying in the degree of type annotations on terms) is established. In this particular setting, the different syntactic languages are related via type annotation erasure functions and proofs that annotations can be recovered.

The first mention of correspondence results of the form we consider in this text dates back to [Geu93] where Geuvers justifies the need for both the PTS and the two-sorted variants of typed λ-calculi. As it turns out, however, his formulation of the correspondence result does not fully capture the complexity of the problem, mostly due to the fact that he does not explicitly spell out the term syntax of the two-sorted setting. As a consequence he does not recognise the complications that arise from the differing syntaxes and therefore claims that the correspondence proof "requires some technical but not difficult work" [Geu93, p. 80]. He does not provide proofs of his claimed correspondence results.

Further examples of correspondence proofs can be found in [BD01], where alternative, PTS-like type systems are defined. The correspondence statements take the usual form of co-typeability, while the proofs are somewhat simpler due to a shared syntactic language. The authors introduce a notion of set-modified PTSs (SPTSs) and it is interesting to note that the corresponding typing rules are fairly close to our formulation (that is without a built-in weakening rule and validity). The match is however not exact since the employed typing contexts are rather different from ours: SPTS contexts may contain typings for arbitrary terms rather than just variables.[4]

1.2.3 Benchmarking Proof Systems

In 2005, the PoplMark challenge [ABF+05] was posed as a benchmark of how well certain systems cope with formalised metatheory of programming languages. The benchmark mostly revolves around proving metatheoretical properties of $F_{<:}$, with a focus on the handling of variable binders and inductions over open terms and statements. There have been numerous solutions to the challenge, e.g. [AW10, Vou12, LdSOCY12, STS15, KWS16, SSK19]. As a follow-up, a second challenge was posed in [AMP17], which asks for a formalised Kripke-style strong normalisation proof of STLC. The latter shifts the focus from the binders themselves to the handling of contextual information as a consequence of open expressions.

The treatment of contextual information as a core benchmark also appears in various of the ORBI problems [FMP15a] which focus on higher-order abstract syntax systems for metatheoretical reasoning. Solutions to the ORBI problems in various systems and frameworks and a comparison of these are presented in [FMP15b]. Most of the ORBI problems are, however, only dealing with small-scale contextual reasoning.

Our work can be seen as a complement to the aforementioned benchmarks. On the one hand we extend the underlying idea of the PoplMark challenge to the multi-system setting. That is, we do not simply care how binders are treated in general, but how the binding disciplines of different systems can be brought into correspondence. The resulting need to deal with complex contextual information can then be seen as a complement to the ORBI results at a somewhat larger scale. In addition, our comparison bridges the gap between first-order and higher-order approaches.

[4] Interestingly, something similar happens with the contexts of the logical embedding that we will encounter in Abella. There we have to take explicit measures to enforce the variables-only aspect.

1.3 Formalisation

All results of this thesis are machine-verified, unless explicitly noted otherwise.

For our main development we use the general-purpose Coq proof assistant [COQ], which is based on constructive type theory, or more precisely the Calculus of (co)inductive Constructions [PPM89]. In Coq, lemmas and theorems are formulated as types, and proofs take the form of terms inhabiting these types. That is, the framework utilises the Curry-Howard correspondence [How80]. For our work we make heavy use of inductive types, and in particular inductively defined predicates. We also crucially rely on the Autosubst library [STS15] which provides a notion of instantiation for our syntactic languages, that is based on parallel substitutions. Note that Autosubst adds the *axiom of functional extensionality* to our constructive metatheory.

Apart from the main formalisation, which is based on first-order de Bruijn representations of the syntactic languages, we also consider the concept of higher-order abstract syntax (HOAS) [PE88]. Since the underlying theory of Coq is, in a sense, too powerful to natively support HOAS definitions[5], we replay the correspondence proof for System F in two other proof assistants, namely Abella [BCG+14] and Beluga [PC15]. Both can handle HOAS definitions and were designed with support for metatheoretical reasoning. Abella is a two-level system structured around a λProlog-based specification layer (for an introduction on λProlog, see [MN12]) and a reasoning layer which is a version of Church's simple type theory. Beluga, on the other hand, is an implementation of contextual modal type theory [NPP08], with, as the name suggests, dedicated support for reasoning about contextual structures and objects within contexts.

We thus have a total of three different, machine-verified proofs of the equivalence of the two variants of System F. This threefold approach provides deep insights into the problem domain and allows us to isolate prover-specific complexities. It also enables us to provide a brief discussion of the various advantages and disadvantages of the three proof assistants.

The accompanying formalisations are all available online at:

`http://www.ps.uni-saarland.de/static/kaiser-diss/index.php`

Some of the minor proof details are omitted in the present text for the sake of a cleaner presentation. We direct the interested reader to the above resource, where all formal proofs are laid out in full detail.

For the Coq development we have generated a browsable version of the sources using the `coqdoc` tool and all definitions and results of Chapters 2 through 5 of the digital version of this document are hyperlinked to their respective counterpart in the formal development. All developments are additionally available as source archives.

[5] There are various attempts to bring HOAS reasoning to Coq, including parametric HOAS (PHOAS) [Chl08] and the HYBRID framework [FM12]. The former is problematic since Coq does not appear to exhibit parametricity in general, while the latter appears to not (yet) be powerful enough for the correspondence problem presented here.

1.4 Contributions

The main contributions of this work are as follows.

- We present a detailed and modular first-order de Bruijn formalisation of generalised pure type systems (PTSs) in the proof assistant Coq. We treat context validity and typeability separately, in contrast to similar existing developments. This allows us to study the two aspects in isolation. It also leads to simpler proofs, since validity is in fact not needed as a premise for several results. The formalisation of the PTS metatheory includes confluence, the compatibility of typing with renaming and instantiation, subject and predicate reduction, as well as uniqueness of typing and a proof of strengthening for the subclass of functional PTSs.
- We present first-order de Bruijn formalisations of the traditional, two-sorted variants of the simply typed λ-calculus and the polymorphic λ-calculus (System F) and prove that these are equivalent to their respective PTS instances, that is, the corresponding corners of Barendregt's λ-cube. The employed notion of equivalence is in each case a bidirectional reduction of type formation and typeability, which supports the transfer of properties like propagation and β-substitutivity from one variant to the other.
- To facilitate the equivalence proofs we develop a proof strategy which decomposes the various proof obligations into structured invariants, context extension lemmas and inductive proofs which tie these components together. The main technical challenge is the alignment of several forms of local variable binding.
- We identify our strategy as a relational, multi-system generalisation of the notion of context morphism lemmas (CMLs) [GM97]. CMLs were originally designed as an elegant generalisation of the substitution lemma, which is amenable to formal treatment. As such, a CML connects two typings under a controlled change of the involved typing contexts. Our generalisation similarly connects inductive judgements from different systems where the differences in the respective syntactic languages are bridged with an inductively defined correspondence relation. We demonstrate that the main CML techniques scale accordingly.
- For the case of System F we adapt our correspondence proof to frameworks based on higher-order abstract syntax (HOAS) and execute comparable developments in the proof assistants Abella and Beluga. This allows us to compare and contrast the first-order and the higher-order approach in general and the two higher-order implementations in particular. The discussion highlights the organisation and tracking of contextual information as the central topic when it comes to formally dealing with syntax and local variable binding.
- We also briefly discuss in how far the regular structures exposed in our work could be generated automatically and what form of tool support could be useful for future formalisation endeavours in this field.

1.4.1 Supporting Publications

Preliminary results of this work have been presented in the following publications.

1. [KTS17]: *Equivalence of System F and λ2 in Coq based on Context Morphism Lemmas.*

   The main focus of this paper was the generalisation of context morphism lemmas to the multi-system setting and the first fully formalised correspondence proof for the two variants of System F, based on syntactic translation functions.

2. [KPS17]: *Relating System F and λ2: A Case Study in Coq, Abella and Beluga.*

   Here we generalised the main approach from syntactic translation functions to inductively defined syntactic correspondence relations, which noticeably simplified the original proof. We further extended the discussion to the HOAS setting as exemplified in the proof assistants Abella and Beluga, with a focus on the handling of contextual information.

The presentation of the PTS λ2 in the two preceding publications was not equipped with a conversion rule, which allowed us to completely ignore reduction. The omission can be justified since well-typed System F types cannot contain applications and therefore no β-redices. In contrast to this we now include a treatment of reduction and add the conversion rule for a full treatment of the PTS formalism.

Note that the inclusion of the conversion rule was a non-trivial extension for a number of reasons. Firstly we had to formally define β-reduction, parallel reduction and conversion, and then establish substitutivity of these relations, as well as confluence and Church-Rosser. Secondly, the PTS conversion rule is not structural and therefore often applicable. As a consequence, many results have to be formulated modulo conversion, which complicates inductive invariants. The strengthening proof turned out to be particularly problematic in this sense and required the introduction of yet another technical generalisation from context renamings to *partial context renamings*.

Our work involves a significant amount of constructions that follow very regular patterns, and should hence be amenable to extensive automation. Preliminary results of our investigation into how such automation could be facilitated resulted in the following texts:

3. [KSS17]: *Autosubst 2: Towards Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions.*

   Here we propose a HOAS-style input specification language for multi-sorted, possibly mutually recursive syntactic expression languages. We compile these to first-order de Bruijn Coq definitions where variable instantiation is based on the novel notion of vectors of parallel substitutions and then explore the resulting equational theory. The main goal of this work is an extension of the Autosubst Coq library which admits more complex syntactic systems and enables a coherent notion of instantiation for such involved object languages.

4. [KSS18]: *Binder Aware Recursion over Well-Scoped de Bruijn Syntax.*

   In this work we present a recursor for definitions over syntactic systems which ensures the compatibility with instantiation. Library implementations based on this work could for example enable the automatic generation of properties like β-substitutivity for a given type system.

We briefly discuss this latter line of work in Section 8.2.5. This brings us to our final publication, where preceding directions of work are beginning to coalesce.

5. [SSK19]: *Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions.*

   While [KSS17] mostly reports on work-in-progress developments, this text can be seen as the cleaned-up, matured and therefore for now definitive presentation of version 2 of the Autosubst framework. Some of the technical realisations of syntax traversals were guided by the findings in [KSS18], though there is still scope for future improvement.

1.5 Thesis Overview

The remainder of this thesis is structured as follows.

In Chapter 2 we cover the necessary preliminaries, including notational conventions, the core principle of variable binding, the basics of de Bruijn syntax with parallel substitution and a brief tutorial on abstract reduction systems, where the treatment of confluence and the Church-Rosser property is the main goal.

In Chapter 3 we then give a detailed presentation of our PTS formalisation. In this setting we also introduce context morphism lemmas in their original form, that is as a means to obtain the substitution lemma for a typing judgement, and highlight those aspects which are crucial to a formal treatment.

Chapter 4 concerns the STLC equivalence result. We present the formalisation of the usual, two-sorted STLC variant and also discuss the specifics of the corresponding concrete PTS $\lambda_{\to}$, including custom induction principles. The chapter also introduces the relational treatment of free variables from different systems, which is in fact applicable to multiple settings, and then carefully extends the CML techniques to obtain the desired equivalence result.

Chapter 5 then scales the results of the STLC chapter to the formal equivalence proof of System F and its corresponding PTS $\lambda 2$.

The alternate HOAS approach is covered in Chapter 6, with a primary focus on the treatment of contextual information. The chapter covers the two HOAS formalisations in Abella and, respectively, Beluga of the System F equivalence result.

Finally in Chapter 7 we summarise and discuss some of the more technical aspects of the various formalisations and then conclude in Chapter 8.

2 Technical Preliminaries

Before we can take on the various systems and correspondence proofs that form the core of this work we have to lay some foundations. Apart from a brief overview of our notational conventions we need to discuss a few topics of interest in this chapter.

First of all we are going to take a careful look at the concept of local variable binding, including the core notion of α-equivalence of syntactic expressions. To illustrate the ideas, we will use the untyped λ-calculus (ULC). The problems of binding and α-equivalence then naturally lead to a first-order, nameless (and therefore canonical) representation of syntax which is due to de Bruijn [dB72]. We will introduce the notion of **de Bruijn syntax** in detail, together with the associated notion of instantiation with parallel substitutions. A significant portion of this work is based on this form of language representation and it intimately affects the way in which we phrase our various lemmas and theorems.

The other important topic is reduction, which we handle in the context of **abstract reduction systems** (ARSs) [Hue80, BN98]. Here we recall an ARS framework to establish the various properties of the notion of β-reduction for our PTS development in Chapter 3. While we only use the results for a single reduction system, it is still a useful abstraction to disentangle the generic features of reduction, like confluence and Church-Rosser, from the peculiarities of the PTS formalism. In Section 2.4 we provide an overview of a formal ARS development, mostly adapted from [Smo15a, Smo15b], with a focus on the aforementioned properties.

2.1 Notational Conventions

All mathematical results in this thesis are developed upon the foundation of constructive type theory. While the exact nature of the employed type theory is determined by the respective proof assistant, they all share common constructions for which we introduce uniform notation.

First of all, let us consider universal quantification and implication, which we write

$$\forall x{:}A.\ B \qquad \text{and} \qquad A \implies B$$

respectively. Note that the latter is simply a shorthand for the former, when the bound identifier x does not occur freely in B. Both are primarily used to express propositions (of type $\mathbb{P}$) but it should be understood that they can be used to form general (dependent) function types. Non-propositional, non-dependent functions are introduced as $f : A \to B$. With respect to propositions, we also use the following common logical connectives.

$\exists x{:}A.\ P$	(existential quantification)
$P \wedge Q$	(conjunction)
$P \vee Q$	(disjunction)
$P \iff Q$	(equivalence)
$\top$	(true proposition)
$\bot$	(false proposition)

Note that for both quantifications, we sometimes incorporate additional information into the quantor. We do for example write

$$\exists(x, y, z) \in \mathcal{R}.\ \ldots \qquad \text{and} \qquad \forall(x, y, z) \in \mathcal{R}.\ \ldots$$

which should respectively be understood as

$$\exists xyz.\ \mathcal{R}\, x\, y\, z \wedge \ldots \qquad \text{and} \qquad \forall xyz.\ \mathcal{R}\, x\, y\, z \implies \ldots$$

The examples highlight that we sometimes prefer set notation to express that a predicate (here $\mathcal{R}$) holds on some arguments, provided that the meaning is clear from the context. We also occasionally adopt set notation for types with decidable equality, where this leads to a clearer or more familiar presentation (see for example the treatment of reduction in Section 2.4). Also note that we often drop type annotations on quantified variables where these are easily inferrable.

Since we have functional types in our metatheory we do occasionally need to express meta-level functions. We write these as follows.

$$\lambda x{:}A \Rightarrow t$$

Note that we will later introduce various object-level abstractions, which all use a dot instead of an arrow ($\Rightarrow$) to separate the bound identifier and its type from the body of the abstraction. This allows us to disambiguate object-level and meta-level abstractions.

We make extensive use of the Backus-Naur-Form (BNF) to introduce the various syntactic languages, which then translate to the definition of inductive types in the type theories of our respective formalisations. The name of the defined type is given in a box to the left of the BNF, and any required side conditions, like the scope of certain parameters, are expressed on the right. Definitions of this form look as follows.

$$\boxed{\mathsf{T}} \qquad s, t ::= x \mid s\, t \mid \ldots \qquad P\, x$$

Here T is the newly introduced type, s, t are the associated metavariables and to the right of the $::=$ we find the productions, including variables x for which some condition P should hold.

We will encounter several inductively defined predicates, like for example type systems. Such predicates are usually defined in terms of inference rules of the following form.

$$\textsc{Rule}\ \frac{P \qquad Q \qquad R}{T}$$

The idea is that the set of valid predicate instances is the least collection of instances closed under the given set of rules. Note that this associates the notion of derivation trees (which are build up from inference rules) with valid predicate instances. The plain, type theoretic formulation of the above rule is

$$\textsc{Rule} : P \Longrightarrow Q \Longrightarrow R \Longrightarrow T.$$

We exploit this dual representation to phrase certain lemmas with similar implicational structures also as rules. The motivation for this is that the respective lemmas are usually either obtained as derivation tree fragments from the original set of rules, or via an inductive argument over potential derivation trees for the posed conclusion. Such lemmas are said to be **admissible** for the original set of rules and can be used to construct new derivation trees. Presenting them as rules highlights this intended use.

Many of the inductive predicates we define throughout this work establish properties of syntactic expressions, such as types and terms, under some contextual information. We refer to such predicates as **judgements** and uniformly annotate them with a turnstile ($\vdash$) which separates the contextual information (on the left) from the remaining parameters (on the right). A simple typing judgement would thus be expressed as

$$\Gamma \vdash a : b,$$

which states that a has type b under the contextual assumptions in Γ. We place suitable subscripts on the $\vdash$ to disambiguate the various judgements.

We have a large collection of proofs about various judgements and we will make extensive use of a certain class of inductive invariants which establish mappings between different contexts. We denote such invariants uniformly as

$$\sigma \Vdash \Gamma \to \Delta,$$

where σ is a mapping from the contextual information in Γ to properties under Δ. We again use subscripts and similar annotations to clarify which systems are involved for each concrete invariant. Also be aware that in some cases each of the three quantities σ, Γ and Δ can appear as complex compound constructions. Consider for example the following.

$$\langle (\rho, \sigma) \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1; \Sigma_1 \to \Theta_2; \Sigma_2$$

We will encounter this invariant at a later stage and thus not define it here, but it should be possible to identify the three main constituents. That is, $\langle (\rho, \sigma) \sim \tau \rangle$ is some kind of mapping that translates contextual information in $\Theta_1; \Sigma_1$ to properties under $\Theta_2; \Sigma_2$. The subscript P indicates that this invariant is used in the context of the polymorphic λ-calculus (PLC).

2.2 Variables, Binding and α-Equivalence

Let us recall the syntax of the untyped λ-calculus (ULC). It captures the essence of functions, that is their construction via abstraction over free variables and their elimination via application.

$$\boxed{\mathsf{Tm}_{\mathsf{U}}^{\mathsf{nmd}}} \qquad u, v ::= x \mid u\, v \mid \lambda x.\, v \qquad x \in \mathcal{V}$$

Here we have assumed some countably infinite supply of variables $\mathcal{V}$. For a given term we distinguish the two classes of free and bound variables. Consider for example the term $\lambda x.\, x\, y$, where y occurs freely, while x is bound in the abstraction. We usually state that a bound occurrence refers to its binder and can visualise it like this:

We have indicated the intended binding structure of this term with an arrow, pointing from the occurrence of the bound variable to its binder. In fact, this arrow is the only information that is relevant to determine the meaning of the term. In other words, we could express **the same** term also as

or even as

where we have dropped the bound name altogether. Thus the inherent equality of syntactic expressions that involve binders and bound variables is not literal equality, but equality up to renaming of bound variables, known as **α-equivalence**, and expressed as follows.

$$\lambda x.\, x\, y \equiv_\alpha \lambda z.\, z\, y$$

Another situation where this concept comes into play is the instantiation of a free variable x in a term u with a term v, written

$$u[v/x].$$

We are, in particular, concerned about the unintentional capture of free variables of expressions that are placed below binders. Consider the following instantiation instance where we substitute the term $(x\, z)$ with both x and z occurring freely for the free variable y in $(\lambda x.\, x\, y)$. A naive, but faulty, implementation would evaluate the instantiation to the expression shown on the right.

$$(\lambda x.\, x\, y)[(x\, z)/y] \neq \lambda x.\, x\, (x\, z)$$

In this hypothetical, faulty instantiation, the x which occurs freely in $(x\,z)$ was captured by the binder for x. The reason, why this is problematic, is that it does not preserve α-equivalence. That is, the same instantiation applied to two α-equivalent expressions could possibly result in expressions that are no longer α-equivalent. In other words, the names on binders would suddenly matter to determine the behaviour of instantiation, contrary to our previous exposition. Thus any sensible notion of instantiation should be **capture-avoiding**.

The solution, of course, is to use α-equivalence to rename all those bound variables that are also used as free variables to something fresh. This is always possible since we have infinitely many variable names. Formulated in a slightly different fashion we obtain the (in)famous Barendregt convention:

> *2.1.13. Variable Convention. If $u_1, \ldots, u_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.* — Barendregt [Bar84, p. 26].

As long as we are only considering on-paper reasoning, this is usually (but not always [UBN07]) a reasonable global invariant. The preservation of this invariant is then normally left implicit throughout the proofs.

When we, however, consider **mechanised** metatheory, that is proofs about such syntactic systems developed in a proof assistant like Coq, we cannot afford this level of laxness. We quickly notice that we have to explicitly spell out the convention as a formal invariant and then make sure that each and every definition, recursive operation and inductive proof preserves it. Simply put, this is not feasible.

The main culprit in this mess is that the usual on-paper presentations pretend to work directly on the definition of the syntax, while in fact they really deal with α-equivalence classes of terms, and any form of formalisation effort has to take this seriously.

The problem is of course not new and several techniques have been developed to deal with it, including nominal syntax [Pit03], higher-order abstract syntax (HOAS) [PE88], locally nameless syntax (LN) [ACP+08] and of course fully nameless (or pure) de Bruijn syntax [dB72]. It is our opinion, for a number of reasons discussed later, that the latter is the most robust and flexible approach and also rather elegant if care is taken in some places, like the definition of instantiation. We therefore adopt a pure de Bruijn approach for our main developments and now introduce its basics. We defer the introduction of basic HOAS principles to Section 6.1.

2.3 De Bruijn Syntax

The basic idea is to construct a representation which reduces α-equivalence to basic syntactic equality and thereby avoid the need to reason modulo said equivalence. De Bruijn's key insight in [dB72] was the idea to use numerical indices in place of variable names. A variable represented by an index n is then understood to reference

the nth enclosing binder, starting from 0 and counting upwards in the syntax tree. Indices that are greater than the total number of enclosing binders are interpreted as free variables. Since we now reference a corresponding binder by position, rather than by name, there is no longer any need to attach a name to the binder itself. We effectively obtain an index-based encoding of the binding (or pointer) structure which we illustrated above with arrows from occurrence to binder.

We define the de Bruijn presentation of the ULC syntax as follows.

$$\boxed{\mathsf{Tm}_{\mathsf{U}}} \qquad u, v ::= \mathsf{v}_n \mid u\, v \mid \lambda.\, u \qquad n : \mathbb{N}$$

We illustrate the correspondence to the named term representation, $\mathsf{Tm}_{\mathsf{U}}^{\mathsf{nmd}}$, with an example in Figure 2.1. We have two α-equivalent, named, expressions on the left and give the single corresponding de Bruijn variant on the right, where we represent the free variables x and z at the top level as v_0 and v_1 respectively. The example demonstrates a number of key considerations.

First, the interpretation of a given index *depends upon its position in the syntax tree*. That is, the same variable may be encoded using different indices (e.g. both v_1 and v_3 encode the free variable z) while at the same time a single index may refer to different variables (e.g. v_1 encodes three different entities). More generally a variable encoded as v_k at some point will be encoded as as v_{k+n} when we descend below n further binders.

Second, α-equivalent named terms are encoded as the same de Bruijn term. Hence each de Bruijn term canonically captures a full α-equivalence class of named terms. This is the key feature which makes the encoding tractable for mechanised proof developments.

Lastly, the meaning of an open de Bruijn term depends on the mapping of free variables to indices. This will have implications once we introduce contexts for the free variables. Since we cannot reference context entries by variable name (as there are none) we instead reference by position. The meaning of a de Bruijn-encoded term then depends on the ordering of such a context and is crucially *not* invariant under modifications of this ordering. We are going to discuss this slight drawback further in the next chapter, where we have a typed system with typing contexts.

We chose to preserve the dot on all object-level binders throughout this work, even for those where no expression appears to the left of it, as for example in $\lambda.\, u$. The reason for this is three-fold. Firstly, the dot serves as a clear and uniform indicator of the presence of a binder (we will later see other binders beside λ). Secondly, it clearly marks the position where the set of variables in scope changes. Finally, as mentioned above, it is a convenient way to disambiguate meta-level and object-level abstractions.

Note that in the following we also drop the explicit mention of the variable constructor and simply write k instead of v_k. There will be a small number of places where the distinction between an index, that is a number, and a variable, that is a term, is crucial. In these cases we will highlight the transition, whenever it is not clear from the context.

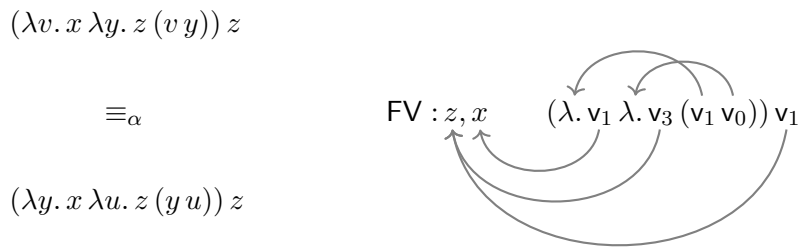


Figure 2.1: De Bruijn terms canonically encode α-equivalence classes of terms.

2.3.1 Instantiation, Parallel Substitutions and Autosubst

In the named setting, it is common to work with so-called **single-point substitutions** when defining **instantiation**, that is the operation that takes a substitution and applies it to a term. Instantiation is written as $u[v/x]$, where the free occurrences of the variable x in u are instantiated with the term v. While this process is intuitive, conceptually, it turns out to be ill-suited for the nameless de Bruijn encoding. Already in [dB72], de Bruijn proposed an instantiation operation that instead takes a parallel substitution as the natural notion of instantiation for the nameless setting.

A **parallel substitution** is a function $\sigma : \mathbb{N} \to \mathcal{T}$, where $\mathcal{T}$ is some syntactic class. For the purpose of this chapter, let $\mathcal{T}$ be Tm_U. In the following, substitutions, denoted by σ, τ or ρ, should always be understood as parallel substitutions. We also consider a particular subclass of substitutions, called **renamings** and denoted by ξ or ζ, which map indices to indices (i.e. they are functions of type $\mathbb{N} \to \mathbb{N}$). The lifting of a renaming to a substitution via composition with the variable constructor of the underlying syntactic class is left implicit. Instantiation of a term u with a substitution σ, written $u[\sigma]$, acts on all free variables v_k of u *simultaneously* by replacing them with the value of σ at k. Since σ is just a plain function, the latter is plain application, i.e. $\sigma\, k$.

In our Coq development we use the **Autosubst** library [STS15] to implement de Bruijn encodings of our various term languages.[1] All work in this thesis is done with Autosubst 1, though we will later borrow some notational ideas from the ongoing development of Autosubst 2 [SSK19]. The library operates as follows. The user provides his syntactic classes as inductive types, augmented with some binder annotations. For ULC, such a definition looks like this (where *var* is just an alias for $\mathbb{N}$):

$$
\begin{aligned}
&\mathsf{Inductive}\ tm : \mathsf{Type} :=\\
&\quad |\ \mathsf{Var}\ (x : var)\\
&\quad |\ \mathsf{App}\ (u\ v : tm)\\
&\quad |\ \mathsf{Lam}\ (v : \{\mathsf{bind}\ tm\}).
\end{aligned}
$$

[1] This is where the axiom of functional extensionality is introduced into our metatheory.

Next we ask Coq to instantiate a number of type classes of the Autosubst framework, which in turn triggers the inference of the correct definition of capture-avoiding instantiation for the provided de Bruijn syntax (see below), together with an equational theory that governs the interplay of terms, instantiation, renamings and substitutions. Expressions involving any subset of these constructions are known as **substitution expressions**. The theory constitutes a convergent rewriting system which forms the basis for a decision procedure for equations over such substitution expressions. Autosubst ships with a custom proof tactic that uses the decision procedure to automatically solve such equational goals. Since we make extensive use of the generated definitions, properties and automation tactics, it is useful to briefly introduce the key concepts.

Substitutions are functions but it is often instructive to view them as infinite sequences of terms: $\sigma = u_0, u_1, u_2, \ldots$. This motivates the introduction of a **cons** operation, written $u \cdot \sigma$, as a primitive operation on substitutions (as well as renamings). Similar to a list cons, this operation prepends a single term u to the front of a given substitution σ. Viewed as a sequence, we obtain the following equality.

$$u \cdot \sigma = u, u_0, u_1, \ldots$$

Meanwhile, with respect to indexing (implemented as plain function application) we also obtain the following equations.

$$(u \cdot \sigma)\, 0 = u$$
$$(u \cdot \sigma)\, (n+1) = \sigma\, n$$

The second primitive is the **shift** renaming, denoted by $\uparrow$, which represents the following sequence of indices.

$$\uparrow = 1, 2, 3, \ldots$$

With these two primitives it is possible to construct all possible substitutions. As a simple example, we define the **identity** renaming id in terms of shift and cons.

$$\mathsf{id} := 0 \cdot \uparrow = 0, 1, 2, \ldots$$

We can now define **instantiation** for ULC mutually recursive with the **forward composition** of substitutions, written $\sigma \circ \sigma'$.

$$\begin{aligned} \mathsf{v}_n[\sigma] &:= \sigma\, n & (\sigma \circ \sigma')\, n &:= (\sigma\, n)[\sigma'] \\ (u\, v)[\sigma] &:= u[\sigma]\, v[\sigma] & & \\ (\lambda.\, u)[\sigma] &:= \lambda.\, u[\Uparrow \sigma] & \Uparrow \sigma &:= \mathsf{v}_0 \cdot \sigma \circ \uparrow \end{aligned}$$

We adopt the convention that composition binds stronger than cons, hence the expression $\mathsf{v}_0 \cdot \sigma \circ \uparrow$ should be read as $\mathsf{v}_0 \cdot (\sigma \circ \uparrow)$. We further observe that when we move a substitution underneath a binder we have to adjust it to accommodate for the change of variables in scope. Recall that the interpretation of de Bruijn indices

changes when we traverse a binder. The requisite operation to adjust for this change, called **up** or **lift** and written as $\Uparrow\sigma$, performs two things. First it ensures that the newly bound variable is preserved by consing v_0 onto the substitution, so that we get $(\Uparrow\sigma)\,0 = \mathsf{v}_0$. This also takes care of the fact that variables encoded as n above the binder are encoded as $n+1$ below it. The second necessary adjustment is the post-composition of $\uparrow$ to all results of the original substitution σ to correctly bypass the binder and thus avoid a form of capturing. We hence obtain $(\Uparrow\sigma)\,(n+1) = (\sigma\,n)[\uparrow]$.

Due to the mutual recursion we have to argue why this definition is well-founded, that is, why the recursion terminates. The trick to breaking the cycle is to observe that we can give a direct, structurally recursive definition for instantiation with renamings, written $u[\xi]$, where the occurrence of $\xi \circ \uparrow$ in $\Uparrow\xi \coloneqq 0 \cdot \xi \circ \uparrow$ is just ordinary forward function composition.[2] We can then define composition of a substitution and a renaming, $(\sigma \circ \xi)\,n \coloneqq (\sigma\,n)[\xi]$, which is sufficient to obtain $\Uparrow\sigma$. Once we have $\Uparrow\sigma$, it is possible to define full instantiation and finally full forward composition of two substitutions.

When working with de Bruijn syntax and parallel substitutions, we often find similar situations where a definition or proof about statements involving general substitutions often relies on having the same definition or proof already available for the special case of renamings, which in turn can usually be established directly. This phenomenon is due to the fact that structural recursion on de Bruijn syntax is a priori not binder-aware. The staged approach works around this issue.

We have mentioned above that β-reduction is the essential operation of any given λ-calculus, so it is worth considering how this is implemented in our present setting. Let us consider the following β-redex.

$$(\lambda.\,u)\,v$$

The idea is to replace this term with the body of the abstraction, u, where all occurrences of the bound variable are replaced by v. Note that the bound variable in question is encoded as v_0 directly below the binder. We also observe that the operation removes a binder, which forces us to lower all non-zero indices by 1 to accommodate for the change of variable scope. Our present framework admits a very concise definition of this β-reduction step.

$$(\lambda.\,u)\,v \succ u[v \cdot \mathsf{id}]$$

We refer to substitutions of the form $(v \cdot \mathsf{id})$ as **β-substitutions** due to their role in the contraction of β-redices. We give their sequence notation and the indexing equations to illustrate how the lowering of non-zero indices is facilitated.

$$\begin{aligned} v \cdot \mathsf{id} &= v, \mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2, \ldots \\ (v \cdot \mathsf{id})\,0 &= v \\ (v \cdot \mathsf{id})\,(n+1) &= \mathsf{v}_n \end{aligned}$$

[2] We crucially exploit the fact that renamings have type $\mathbb{N} \to \mathbb{N}$ here.

Similar to our convention in regard to the term constructor for variables, we will in the following refer to de Bruijn indices simply as variables, unless the distinction is of crucial relevance.

2.3.2 σ-Calculus

The setup we have outlined thus far is intimately connected to the work of Abadi, Cardelli, Curien and Levy. In [ACCL91], they introduce the σ-calculus as a calculus of explicit substitutions for the fine-grained study of implementations of reduction. As such it can express all substitutions necessary to describe reductions in a λ-calculus. In addition, it yields the convergent rewriting system which is shown in Figure 2.2.

The definitions of de Bruijn terms with parallel substitutions which are generated by the Autosubst library form a model of the σ-calculus [STS15]. The model is complete [SST15], which entails that the automation tactics of the Autosubst system, which are based on the rewriting system of the σ-calculus, are in fact a decision procedure. Further details are outlined in [STS15, SST15].

2.4 Abstract Reduction Systems

Set-theoretically, an abstract reduction system (ARS) [Hue80, BN98] consists of a set X with a binary relation $R \subseteq X \times X$. It is customary to write $x\,R\,y$ for $(x, y) \in R$ and we denote the inverse relation of R as R^{-1}. Our type-theoretic formalisation refines these notions to a type X and a binary predicate $R : X \to X \to \mathbb{P}$. We implement relation containment as

$$S \subseteq R \quad := \quad \forall xy \in X.\ x\,S\,y \implies x\,R\,y$$

and define equality of relations via mutual containment,

$$R = S \quad := \quad R \subseteq S \land S \subseteq R.$$

Definition 2.4.1 Let R be a binary relation over X, then we define the notions of **reflexivity**, **symmetry** and **transitivity**, as well as the combined notion of an **equivalence** as usual.

$$\begin{aligned} R\ \mathsf{refl} &:= \forall x \in X.\ x\,R\,x \\ R\ \mathsf{sym} &:= \forall xy \in X.\ x\,R\,y \implies y\,R\,x \\ R\ \mathsf{trans} &:= \forall xyz \in X.\ x\,R\,y \implies y\,R\,z \implies x\,R\,z \\ R\ \mathsf{equiv} &:= R\ \mathsf{refl} \land R\ \mathsf{sym} \land R\ \mathsf{trans} \end{aligned}$$

Definition 2.4.2 (Reflexive, Transitive Closure) Let R be a binary relation on X, then its reflexive, transitive closure, R^* is inductively defined as follows.

$$\frac{}{x\,R^*\,x} \qquad \frac{x\,R\,y \qquad y\,R^*\,z}{x\,R^*\,z}$$

$$
\begin{aligned}
(u\,v)[\sigma] &= u[\sigma]\,v[\sigma] & \mathsf{id} \circ \sigma &= \sigma \\
(\lambda.\,u)[\sigma] &= \lambda.\,u[0 \cdot \sigma \circ \uparrow] & \sigma \circ \mathsf{id} &= \sigma \\
0[u \cdot \sigma] &= u & (\sigma_1 \circ \sigma_2) \circ \sigma_3 &= \sigma_1 \circ (\sigma_2 \circ \sigma_3) \\
\uparrow \circ (u \cdot \sigma) &= \sigma & (u \cdot \sigma_1) \circ \sigma_2 &= u[\sigma_2] \cdot (\sigma_1 \circ \sigma_2) \\
u[\mathsf{id}] &= u & u[\sigma_1][\sigma_2] &= u[\sigma_1 \circ \sigma_2] \\
0[\sigma] \cdot \uparrow \circ \sigma &= \sigma & 0 \cdot \uparrow &= \mathsf{id}
\end{aligned}
$$

Figure 2.2: The convergent rewriting system of the σ-calculus.

Fact 2.4.3 (Properties of R^*) Let R be a binary relation on X, then all the following hold with respect to its reflexive, transitive closure.

$$R \subseteq R^* \qquad\qquad R^*\ \mathsf{trans}$$

Definition 2.4.4 (Equivalence Closure) Let R be a binary relation on X, then its equivalence closure, $R^{\equiv}$ is inductively defined as follows.

$$\frac{}{x\,R^{\equiv}\,x} \qquad \frac{x\,R^{\equiv}\,y \qquad y\,R\,z}{x\,R^{\equiv}\,z} \qquad \frac{x\,R^{\equiv}\,y \qquad z\,R\,y}{x\,R^{\equiv}\,z}$$

Fact 2.4.5 (Properties of $R^{\equiv}$) Let R be a binary relation on X, then all the following hold with respect to its equivalence closure.

$$R \subseteq R^{\equiv} \qquad R^{-1} \subseteq R^{\equiv} \qquad R^{\equiv}\ \mathsf{trans} \qquad R^{\equiv}\ \mathsf{equiv} \qquad R^{\equiv}\ \mathsf{sym}$$

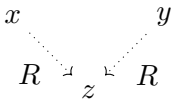
Fact 2.4.6 $R^* \subseteq R^{\equiv}$

In addition to the concepts introduced so far we also require the notion of terms which are joinable with a given relation. The idea is a key ingredient in the statements and proofs of confluence and the Church-Rosser property. Normally we are considering two values $x, y \in X$ and want to know if there is a third value $z \in X$ to which both are related via some binary relation R on X. Since the relation R tends to represent some form of reduction it is often useful to consider the property diagrammatically, so we include this notion in the formal definition.

Definition 2.4.7 (Joinability) Let R be a binary relation on X, then $x, y \in X$ are **joinable** by R, written $x \triangledown_R y$, whenever the following holds.

$$x \triangledown_R y \quad :\Leftrightarrow \quad \exists z \in X.\ x\,R\,z \wedge y\,R\,z$$

Joinability of x and y via R is diagrammatically expressed as follows.



Fact 2.4.8 Let R be a relation on X, then joinability is symmetric: ∇_R sym.

Fact 2.4.9 Two values $x, y \in X$ are R-equivalent, whenever they can be joined in multiple R-steps.

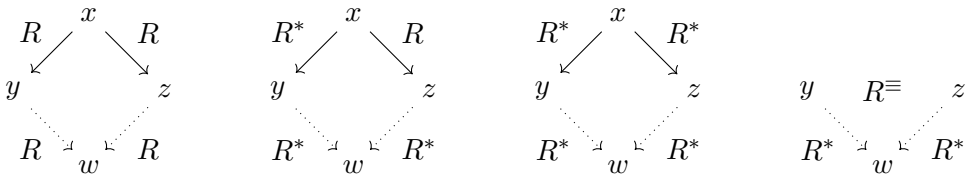
$$\nabla_{R^*} \subseteq R^{\equiv}$$

Now that we have a notion of joinability, we can introduce some further properties, that a given relation R may exhibit, including confluence and Church-Rosser. All of them are concerned with the joinability of divergent reduction sequences.

Definition 2.4.10 Let R be a binary relation on X. Then we can define the following properties of R.

$$\begin{aligned}
R \text{ diamond} &:= \forall xyz \in X.\ x\,R\,y \implies x\,R\,z \implies y \nabla_R z \\
R \text{ semiconf} &:= \forall xyz \in X.\ x\,R^*\,y \implies x\,R\,z \implies y \nabla_{R^*} z \\
R \text{ confluent} &:= \forall xyz \in X.\ x\,R^*\,y \implies x\,R^*\,z \implies y \nabla_{R^*} z \\
R \text{ CR} &:= \forall yz \in X.\ y\,R^{\equiv}\,z \implies y \nabla_{R^*} z
\end{aligned}$$

Each of the properties can also be diagrammatically expressed, which respectively yields the following.



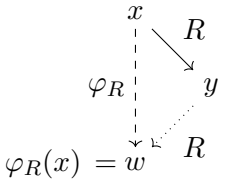
Fact 2.4.11 R confluent $= R^*$ diamond

In addition, we need the notion of a **triangle function**, which is due to Takahashi [Tak95]. The idea is to decompose a diamond into two triangles in such a way that a common reduct w can be computed.

Definition 2.4.12 Let R be a binary relation on X, then we say that φ_R is a triangle function for R, whenever the following is satisfied for all x and y.

$$x\,R\,y \implies y\,R\,\varphi_R(x)$$

The diagrammatic representation of this property justifies the name.



The existence of a triangle function φ_R is sufficient for R to be confluent. The construction is standard so we only briefly recap the main steps.

Lemma 2.4.13 Let φ_R be a triangle function for R, then we have R diamond.

Proof Assume $x\,R\,y$ and $x\,R\,z$, then we get $y \triangledown_R z$ with $\varphi_R(x)$ as the witness. ∎

Lemma 2.4.14 Let R be a relation with R diamond, then also R semiconf.

Proof Assume $x\,R^*\,y$ and $x\,R\,z$. The proof is basic diagram chasing with an induction on $x\,R^*\,y$. We use R diamond to close the diamond in the inductive step. ∎

Lemma 2.4.15 Let R be a relation with R semiconf, then also R confluent.

Proof Assume $x\,R^*\,y$ and $x\,R^*\,z$. The proof is basic diagram chasing with an induction on $x\,R^*\,z$. We use R semiconf to close the initial slice in the inductive step. ∎

Theorem 2.4.16 Let R be a relation with R diamond, then also R confluent.

Proof Combine Lemmas 2.4.14 and 2.4.15. ∎

Theorem 2.4.17 Let φ_R be a triangle function for R, then we have R confluent.

Proof Combine Lemma 2.4.13 with Theorem 2.4.16. ∎

Now that we have a way of obtaining confluence we can also consider the equivalent Church-Rosser property. The equivalence argument is indirect and goes through semi-confluence.

Lemma 2.4.18 Let R be a semi-confluent relation, then we also have R CR.

Proof By induction on $y\,R^{\equiv}\,z$. ∎

Lemma 2.4.19 Let R be a relation with the Church-Rosser property, then we also have R confluent.

Proof Assume $xR^*\,y$ and $xR^*\,z$, then with R CR it is sufficient to show $y\,R^{\equiv}\,z$. We clearly have $xR^{\equiv}\,y$ and $xR^{\equiv}\,z$ by containment and therefore $yR^{\equiv}\,x$ by symmetry. The result is then by transitivity of $R^{\equiv}$. ∎

Fact 2.4.20 Any confluent relation R is also semi-confluent.

Lemma 2.4.21 Let R be a confluent relation, then we also have R CR.

Proof Combine Fact 2.4.20 and Lemma 2.4.18. ∎

As we have seen, it is easy to lift the existence of a triangle function to confluence, but there are certainly cases of confluent reduction systems that do not possess such a function, and neither do they exhibit the single-step diamond property. In these cases it is sufficient to find an auxiliary relation S which satisfies $R \subseteq S \subseteq R^*$ and also is equipped with a triangle function φ_S. The idea to use a parallel reduction relation for S when R is β-reduction of the λ-calculus was independently suggested by Tait (1965) and Martin-Löf (1971) (see [HS08, Appendix A2]). The whole approach is therefore commonly referred to as the Takahashi/Tait/Martin-Löf confluence proof, or TTML for short. We give a brief overview of the construction.

First of all, we have to consider further properties of the reflexive, transitive closure operation, namely monotonicity and idempotency.

Lemma 2.4.22 The reflexive, transitive closure is monotone with respect to relation containment.

$$R \subseteq S \implies R^* \subseteq S^*$$

Proof Let $R \subseteq S$ and $x\, R^*\, y$, then $x\, S^*\, y$ holds by induction on $x\, R^*\, y$. ∎

Lemma 2.4.23 The reflexive, transitive closure operator is idempotent.

$$R^{**} \subseteq R^*$$

Proof By induction on $x\, R^{**}\, y$, using transitivity of R^* for the inductive step. ∎

Lemma 2.4.24 When a relation S interpolates between R and R^*, then the reflexive, transitive closures of R and S coincide.

$$R \subseteq S \subseteq R^* \implies R^* = S^*$$

Proof Assume $R \subseteq S$ and $S \subseteq R^*$. Now by monotonicity we have $R^* \subseteq S^*$ and $S^* \subseteq R^{**}$ respectively. By idempotency we have $R^{**} \subseteq R^*$ and thus by transitivity of containment $S^* \subseteq R^*$. The equality follows. ∎

The last missing ingredient is the fact that, as one would expect, equality of relations preserves the diamond property.

Fact 2.4.25 R diamond $\implies R = S \implies S$ diamond

Theorem 2.4.26 (TTML Confluence Proof) Let R be a binary relation on X and S a binary relation that interpolates between R and R^*. Moreover, let S have a triangle function φ_S. Then R is confluent.

Proof We have to show R confluent, that is R^* diamond. Since S interpolates between R and R^*, we know by Lemma 2.4.24 that $R^* = S^*$. Since S also has a triangle function we clearly have S confluent by Theorem 2.4.17, or alternatively S^* diamond. Fact 2.4.25 closes the proof. ∎

While Theorem 2.4.26 is the main result of this section, there are further properties of the two closure operators that we will use repeatedly in the subsequent development. Since they can also be established abstractly, it makes sense to cover them here.

First of all, it should be clear that if an R-step preserves a given predicate P, then this should also be the case for R^*, though not necessarily for $R^{\equiv}$.

Lemma 2.4.27 Let R be a binary relation on X and $P : X \to \mathbb{P}$ be predicate on X. Then R^* preserves P whenever R preserves P.

$$(\forall xy \in X.\ x\,R\,y \implies P\,x \implies P\,y) \implies \forall xy \in X.\ x\,R^*\,y \implies P\,x \implies P\,y$$

Proof By induction on $x\,R^*\,y$. ∎

We also need to consider how the two closure operators interact with functions on the carrier type X. We are in particular concerned with unary $(f : X \to X)$ and binary $(g : X \to X \to X)$ functions, due to the fact that we are going to instantiate X with inductive syntax definitions, where the unary and binary functions appear as syntactic constructors. In this setting we often have to establish that whenever a given base relation is a congruence on its carrier type then this also extends to the respective closures. We now give the respective lifting constructions for the unary and binary case.

Lemma 2.4.28 (Unary Congruence Lifting for R^*) Let R be a binary relation on X. Let R further be a congruence with respect to $f : X \to X$, that is we have

$$\forall xy \in X.\ x\,R\,y \implies (f\,x)\,R\,(f\,y).$$

Then R^* is also a congruence with respect to f.

Proof By induction on $x\,R^*\,y$. ∎

Lemma 2.4.29 (Binary Congruence Lifting for R^*) Let R be a binary relation on X. Let R further be a componentwise congruence with respect to $g : X \to X \to X$, that is we have

$$\begin{aligned}\forall xyz \in X.\ x\,R\,y &\implies (g\,x\,z)\,R\,(g\,y\,z) \text{ and}\\ \forall xyz \in X.\ y\,R\,z &\implies (g\,x\,y)\,R\,(g\,x\,z).\end{aligned}$$

Then R^* is a congruence with respect to g.

$$\forall xx'yy' \in X.\ x\,R^*\,x' \implies y\,R^*\,y' \implies (g\,x\,y)\,R^*\,(g\,x'\,y')$$

Proof By transitivity of R^* it is sufficient to establish $(g\,x\,y)\,R^*\,(g\,x\,y')$ and $(g\,x\,y')\,R^*\,(g\,x'\,y')$. Both easily follow from Lemma 2.4.28, as one of the parameters is fixed. For the first we set $f := g\,x$ and for the second we set $f := (\lambda w \Rightarrow g\,w\,y')$. ∎

Lemma 2.4.30 (Unary Congruence Lifting for $R^{\equiv}$) Let R be a binary relation on X. Let R further be a congruence with respect to $f : X \to X$, that is we have

$$\forall xy \in X.\ x\,R\,y \implies (f\,x)\,R\,(f\,y).$$

Then $R^{\equiv}$ is also a congruence with respect to f.

Proof By induction on $x\,R^{\equiv}\,y$. ∎

Lemma 2.4.31 (Binary Congruence Lifting for $R^{\equiv}$) Let R be a binary relation on X. Let R further be a componentwise congruence with respect to $g : X \to X \to X$, that is we have

$$\forall xyz \in X.\ x\,R\,y \implies (g\,x\,z)\,R\,(g\,y\,z) \text{ and}$$
$$\forall xyz \in X.\ y\,R\,z \implies (g\,x\,y)\,R\,(g\,x\,z).$$

Then $R^{\equiv}$ is a congruence with respect to g.

$$\forall xx'yy' \in X.\ x\,R^{\equiv}\,x' \implies y\,R^{\equiv}\,y' \implies (g\,x\,y)\,R^{\equiv}\,(g\,x'\,y')$$

Proof By transitivity of $R^{\equiv}$ it is sufficient to establish $(g\,x\,y)\,R^{\equiv}\,(g\,x\,y')$ and $(g\,x\,y')\,R^{\equiv}\,(g\,x'\,y')$. Both easily follow from Lemma 2.4.30, as one of the parameters is fixed. For the first we set $f \coloneqq g\,x$ and for the second we set $f \coloneqq (\lambda w \Rightarrow g\,w\,y')$. ∎

In the context of congruence, we can also obtain a useful inversion principle for the reflexive, transitive closure R^*. Informally it states that whenever a relation R is guaranteed to preserve a function in head-position (usually a syntactic term constructor), meaning that the expression with the function application can only R-step, if *one of its constituent arguments* R-steps, then the only way in which the application can R^*-step is when *all its arguments* can R^*-step. Observe how we switch from the stepping of one argument to the potential stepping of all arguments when we lift the result from R to R^*. This is possible thanks to the reflexivity of R^*, and necessary since we do not know which argument R-stepped in each of the steps in a given R^* chain.

Lemma 2.4.32 (Binary Congruence Inversion for R^*) Let R be a binary relation on X and g be a binary function on X. Then R^* satisfies the following inversion principle.

$$(\forall x_1y_1z.\ (g\,x_1\,y_1)\,R\,z \implies (\exists x_2.\ z = g\,x_2\,y_1 \wedge x_1\,R\,x_2) \vee (\exists y_2.\ z = g\,x_1\,y_2 \wedge y_1\,R\,y_2))$$
$$\implies \forall x_1y_1z.\ (g\,x_1\,y_1)\,R^*\,z \implies \exists x_1x_2.\ z = g\,x_2\,y_2 \wedge x_1\,R^*\,x_2 \wedge y_1\,R^*\,y_2$$

Proof By induction on $(g\,x_1\,y_1)\,R^*\,z$. In the base case we have $z = g\,x_1\,y_1$ and can use x_1 and y_1 as witnesses since R^* is reflexive. Otherwise we know that $(g\,x_1\,y_1)\,R\,w$ and $w\,R^*\,z$. From our assumption we know that either $x_1\,R\,x_2$ and $w = g\,x_2\,y_1$ or $y_1\,R\,y_2$ and $w = g\,x_1\,y_2$. In either case we can use the inductive hypothesis. From the two required R^*-steps, one is immediate, while the other is easy to compose from the single step and the respective component of the inductive hypothesis. ∎

3 Pure Type Systems

The notion of a pure type system (PTS) was introduced by Barendregt [Bar91] as an abstract representation of typed λ-calculi. Its most important aspect is the ability to establish a significant portion of metatheory uniformly for a large family of systems, including, among others, the simply typed λ-calculus and the calculus of constructions (CC) [CH88]. Consider for example properties like confluence and Church-Rosser for β-reduction, compatibility of typing with instantiation, correctness of typing (also known as propagation) or subject reduction.

As mentioned in the introduction, the key question we consider in this thesis is how we can formally justify the claim that a certain PTS faithfully captures a given system. To approach this question we first need a suitable presentation of the PTS formalism, which is the purpose of this chapter. The formal development which accompanies the present exposition mostly adapts the named, on-paper development of Geuvers and Nederhof [GN91] to the de Bruijn setting.

The notion of a PTS and its metatheory constitute a focal point of our equivalence proofs in the following chapters. It is therefore crucial to have a firm understanding of the main principles as well as the technical subtleties which arise in the context of a formal development. To this end we present a detailed and self-contained introduction to the PTS framework as observed through the lens of de Bruijn. This also provides us with an opportunity to introduce the proof method based on context morphism lemmas, which is essential for our later work.

3.1 A Class of Systems

The PTS framework is parametric in three arguments, $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$.

The first, $\mathcal{S}$, is a set of constants, called **sorts**, which serve as universes. While the whole PTS theory can technically be developed for $\mathcal{S} = \emptyset$, at least one universe is necessary to obtain a non-empty set of typeable expressions. The second, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$, is the **axiom relation** that controls relative inhabitance of the various sorts, where $(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{A}$ enforces that s_1 is a term in the universe s_2. Finally, the **rule relation** $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ controls which function types – and subsequently which abstractions – are typeable and which universe they inhabit. We postpone further discussion of $\mathcal{A}$ and $\mathcal{R}$ to Section 3.5, where the type system is introduced.

Formally we represent $\mathcal{S}$ as a type with decidable equality, $\mathcal{A}$ and $\mathcal{R}$ as binary and ternary predicates over $\mathcal{S}$, and package all parameters in a module. The PTS theory

is then developed within a functor[1] which takes a concrete PTS specification as input. This later allows us to quickly generate different concrete PTS instances.

For the remainder of this chapter, assume a fixed specification $\mathcal{P} := (\mathcal{S}, \mathcal{A}, \mathcal{R})$ without any further properties apart from those mentioned here, unless otherwise noted.

3.2 Uniform Syntax

One of the most striking features of the PTS framework is the use of uniform syntax. All expressions, like terms, types, kinds and so on, are taken from a single syntactic class of **terms**. This is in stark contrast to more traditional system presentations, where such concepts are distinguished via separate syntactic classes. As a consequence, the class of PTS terms admits the construction of certain expressions which are devoid of any semantic meaning and which would be syntactically malformed in the more traditional presentations. For this reason the class of syntactic PTS expressions has occasionally been referred to as **pseudoterms** [Bar91, Geu93]. For the remainder of this work we will however refer to syntactic PTS expressions as terms, both for simplicity and since they already exhibit interesting structures, even without any notion of typing.

We define the syntactic class of PTS terms, Tm_λ, with the following grammar.

$$\boxed{\mathsf{Tm}_\lambda} \qquad a, b, c, d ::= \mathsf{s} \mid n \mid a\, b \mid \lambda a.\, b \mid \Pi a.\, b \qquad \mathsf{s} \in \mathcal{S} \qquad n : \mathbb{N}$$

For reference we provide the named syntax, $\mathsf{Tm}_\lambda^{\mathsf{nmd}}$, which, in addition to the specification $\mathcal{P}$, also requires a countably infinite supply of variables $\mathcal{V}$.

$$\boxed{\mathsf{Tm}_\lambda^{\mathsf{nmd}}} \qquad a, b, c, d ::= \mathsf{s} \mid x \mid a\, b \mid \lambda x{:}a.\, b \mid \Pi x{:}a.\, b \qquad \mathsf{s} \in \mathcal{S} \qquad x \in \mathcal{V}$$

When we compare these to the definitions of the syntactic classes $\mathsf{Tm}_\mathsf{U}^{\mathsf{nmd}}$ and Tm_U of ULC introduced in Sections 2.2 and 2.3, we observe two interesting differences, apart from the fact that certain constants (i.e. the sorts) are now part of the language.

First, we have an additional binder, namely the **dependent function type** constructor $\Pi a.\, b$. We will refrain from referring to this construction as dependent product or just product, as often done throughout the relevant literature, to avoid confusing it with the structurally quite different cartesian product. We also intentionally do not adopt the common practice to "abbreviate $\Pi a.\, b$ as $a \to b$ whenever the bound variable does not occur freely in the body b" (at the object level), as the formal treatment of this particular case turns out to be technically non-trivial. We will further discuss this issue in Chapters 4 and 5, where we encounter arrow types (i.e. non-dependent function types) as separate constructions in the context of the STLC and System F correspondence proofs.

[1] Here "functor" should be understood in the OCaml/SML sense, i.e. as a higher-order module.

Second, note that both binders, $\lambda a.\, b$ and $\Pi a.\, b$, carry a type annotation a on the bound variable. We are therefore working with Church-style PTSs, as is common, rather than the somewhat unusual Curry-style variants of PTSs.[2] For a detailed discussion of the differences of the two approaches in the context of PTSs we refer the interested reader to [KSW16].

When dealing with these binders in their de Bruijn presentation, it is crucial to keep in mind that the a in $\lambda a.\, b$ and $\Pi a.\, b$ is the type of the bound variable, not the bound variable itself. At this point it is also worthwhile to reiterate the twofold significance of the dot in the binder notation. Firstly it serves as a reminder that there is a binder at all, and secondly it precisely marks the place where the set of variables in scope changes, i.e. where in the de Bruijn setting shifts are necessary to preserve the semantic meaning of a term. More precisely, the set of variables in scope remains stable if we descend into the subexpression a but changes when we descend into b. To consistently maintain this intuition throughout this thesis we also wrote ULC abstractions as $\lambda.\, b$, with an explicit dot, in Chapter 2.

The astute reader may have noticed that we have referred to certain syntactic expressions as types. We want to emphasise that in the context of PTSs, this is a *purely semantic notion* which depends on the derivability of certain judgements. That is, we can only state that a given PTS expression **acts as a type** if it appears either as the domain of a binder, as an element in a typing context or in the type position of a typing judgement. The latter two of course rely on a notion of typing which we will introduce shortly. *Note that the very same expression may appear or act as a type in one situation, while not in another.* Meanwhile, all these distinctions do not exist from a purely syntactic point of view. We therefore describe our PTS syntax as **uniform**.

3.3 Reduction and Conversion

A PTS captures the essence of typed λ-calculi and as such also includes the key operational notion of β-reduction. In the following we define both regular one-step β-reduction[3], full reduction and conversion, as well as a parallel notion of reduction. The latter serves to establish that β-reduction is confluent, or equivalently, satisfies the Church-Rosser property (CR). We also present an inductive characterisation of normal and neutral terms which is commonly used in *normalisation by evaluation* proofs (see for example [Abe08]). We establish that terms which are normal in this sense do not β-reduce.

Note that in this section we heavily rely on our results about abstract reduction systems introduced in Section 2.4. So without further ado, let us consider basic one-step β-reduction.

[2] To be pedantic: the practice of only ascribing types at the binding sites of variables is technically known as de Bruijn-style [BDS13, Sec. 1.1], but we avoid this terminology to prevent obvious confusion.

[3] Terminology taken from [GN91, vBJ93].

$$\frac{}{(\lambda a.\, b)\, c \succ b[c \cdot \mathsf{id}]} \qquad \frac{a \succ a'}{a\, b \succ a'\, b} \qquad \frac{b \succ b'}{a\, b \succ a\, b'}$$

$$\frac{b \succ b'}{\lambda a.\, b \succ \lambda a.\, b'} \qquad \frac{a \succ a'}{\lambda a.\, b \succ \lambda a'.\, b} \qquad \frac{a \succ a'}{\Pi a.\, b \succ \Pi a'.\, b} \qquad \frac{b \succ b'}{\Pi a.\, b \succ \Pi a.\, b'}$$

Figure 3.1: Definition of PTS one-step β-reduction.

Definition 3.3.1 (One-Step β-Reduction) The relation $\succ \subseteq \mathsf{Tm}_\lambda \times \mathsf{Tm}_\lambda$ is the least congruence which includes the β-rule:

$$(\lambda a.\, b)\, c \succ b[c \cdot \mathsf{id}].$$

Recall that $c \cdot \mathsf{id}$ is a parallel β-substitution, which substitutes c for the bound variable at index 0 and accurately lowers all other variables to reflect the removal of the binder. The full inductive definition of $\succ$ is given in Figure 3.1. When there is no term b such that $a \succ b$ holds, we write $a \not\succ$.

It should be immediately clear from the definition of $\succ$, that sorts s and variables n are irreducible, that is $\mathsf{s} \not\succ$ and $n \not\succ$ for all s and n. It is also worth pointing out that, due to the uniform nature of the PTS syntax, β-redices can very well appear in type positions. As a consequence, we need the last three rules of Figure 3.1 to capture reductions like the one-step β-reduction defined in [GN91]. This will also affect our definition of normality and neutrality below. As we will see later, all (semantic) types of the concrete PTSs which we consider in this thesis happen to be normal, but this does not hold for all PTSs. The PTS formulation of CC is a good counter-example here.

When it comes to the notion of normal terms we now have two options. We could either simply identify the irreducible and the normal terms definitionally (i.e. a is normal exactly when $a \not\succ$ holds), or we could give a separate inductive definition of normality and then establish that the two concepts coincide. Here we opt for the second approach. While this initially entails some extra work to establish the coincidence, it will later pay off when we have to exploit the normality of certain terms throughout our proofs.

One complication of our direct definition of normality is the necessity of an auxiliary notion of neutral terms, which are not only normal themselves but also preserve normality when inserted into a one-hole normal expression. Loosely speaking, inserting a neutral term is guaranteed to not create a new β-redex, which practically means that it should not be an abstraction. In conjunction, the two concepts allow us to capture all irreducible terms, including abstractions and applications. The following definition is adapted from [Abe08].

Definition 3.3.2 (Normal and Neutral Terms) We define the characterisation of normal terms, written $\lfloor a \rfloor$, and neutral terms, written $\lceil a \rceil$, mutually inductive according to the following rules.

$$\frac{\lceil a \rceil}{\lfloor a \rfloor} \qquad \frac{\lfloor a \rfloor \qquad \lfloor b \rfloor}{\lfloor \lambda a.\, b \rfloor} \qquad \frac{}{\lceil n \rceil} \qquad \frac{}{\lceil \mathsf{s} \rceil} \qquad \frac{\lfloor a \rfloor \qquad \lfloor b \rfloor}{\lceil \Pi a.\, b \rceil} \qquad \frac{\lceil a \rceil \qquad \lfloor b \rfloor}{\lceil a\, b \rceil}$$

Theorem 3.3.3 The inductive characterisation of normal and neutral terms captures the notion of irreducibility.

$$\lfloor a \rfloor \implies a \not\succ \tag{i}$$
$$\lceil a \rceil \implies a \not\succ \tag{ii}$$
$$\lceil a \rceil \implies \forall cd.\ a \neq \lambda c.\, d \tag{iii}$$

Proof Simultaneously by mutual induction on $\lfloor a \rfloor$ and $\lceil a \rceil$. The fact that a neutral term cannot be an abstraction (iii) is required to ensure that neutral applications are irreducible. ∎

Definition 3.3.4 (β-Reduction and β-Conversion) We define β-reduction, written $a \succ^* b$, as the reflexive, transitive closure of one-step β-reduction using Definition 2.4.2. We similarly define β-conversion, written $c \equiv d$, as the equivalence closure using Definition 2.4.4. For the latter, note that we write $\equiv$ for $\succ^{\equiv}$.

With respect to reduction and normality of terms we observe two facts that we often use tacitly. The second is an immediate corollary of the first since $\lfloor \mathsf{s} \rfloor$ holds for all $\mathsf{s} \in \mathcal{S}$.

Fact 3.3.5 $\lfloor a \rfloor \implies a \succ^* b \implies a = b$

Fact 3.3.6 $\mathsf{s} \succ^* b \implies \mathsf{s} = b$

We also observe that dependent function types are structurally preserved under β-reduction.

Lemma 3.3.7

$$\Pi a.\, b \succ^* c \implies \exists a' b'.\ c = \Pi a'.\, b' \wedge a \succ^* a' \wedge b \succ^* b'$$

Proof Instance of Lemma 2.4.32. We obtain the required premise

$$\Pi a.\, b \succ c \implies (\exists a'.\ c = \Pi a'.\, b \wedge a \succ a') \vee (\exists b'.\ c = \Pi a.\, b' \wedge b \succ b')$$

directly from the definition of $\succ$ and inversion on $\Pi a.\, b \succ c$. ∎

One of the key results we will need in the subsequent development is that β-reduction is confluent, or equivalently, has the Church-Rosser (CR) property. Recall the usual reduction diagrams of confluence (left) and Church-Rosser (right):

We are going to use our abstract TTML proof developed in Section 2.4 to obtain confluence for $\succ$, and therefore also CR. In order to achieve this, recall that we require two ingredients, namely an auxiliary reduction relation $\triangleright$ satisfying

$$\succ \quad \subseteq \quad \triangleright \quad \subseteq \quad \succ^*$$

and a suitable triangle function $\varphi_\triangleright$. The auxiliary reduction $\triangleright$ that we need here is known as **parallel one-step β-reduction**.

Definition 3.3.8 (Parallel One-Step β-Reduction) We define a reduction relation $a \triangleright b$ that can reduce all immediately visible β-redices of a in a single reduction step. The full inductive definition of $\triangleright$ is given in Figure 3.2. We also define its reflexive, transitive closure $\triangleright^*$ with Definition 2.4.2.

We observe that $\triangleright$ is reflexive by construction. This is needed to later show that $\triangleright$ interpolates between $\succ$ and $\succ^*$, as well as for a number of substitutivity results.

Fact 3.3.9 $a \triangleright a$

To proceed, we now have to consider a number of instantiation compatibility results for our various relations on PTS terms. As a starting point, we consider the situation where both sides of a reduction are instantiated with the same parallel substitution.

Lemma 3.3.10 Let a, b be PTS terms and σ be a parallel substitution, then the following hold.

$$a \succ b \implies a[\sigma] \succ b[\sigma] \tag{i}$$
$$a \triangleright b \implies a[\sigma] \triangleright b[\sigma] \tag{ii}$$
$$a \succ^* b \implies a[\sigma] \succ^* b[\sigma] \tag{iii}$$
$$a \equiv b \implies a[\sigma] \equiv b[\sigma] \tag{iv}$$

Proof Statements (i) and (ii) are by induction on $a \succ b$ and $a \triangleright b$ respectively. Most cases are trivial, apart from the variable case of (ii), where the reflexivity of $\triangleright$ is essential, and of course the actual reduction step. For the latter, we have to solve the following equation.

$$b[c[\sigma] \cdot \sigma] = b[\Uparrow\sigma][c[\sigma] \cdot \mathsf{id}]$$

Here we can rely on the decision procedure of the Autosubst framework, which can easily establish this equality and thus close the proof. To see why, expand the definition of $\Uparrow\sigma$ and then apply the rewriting rules of Figure 2.2 from left to right. This converts

$$\frac{}{n \rhd n} \qquad \frac{}{\mathsf{s} \rhd \mathsf{s}} \qquad \frac{b \rhd b' \qquad c \rhd c'}{(\lambda a.\, b)\, c \rhd b'[c' \cdot \mathsf{id}]}$$

$$\frac{a \rhd a' \qquad b \rhd b'}{a\, b \rhd a'\, b'} \qquad \frac{a \rhd a' \qquad b \rhd b'}{\lambda a.\, b \rhd \lambda a'.\, b'} \qquad \frac{a \rhd a' \qquad b \rhd b'}{\Pi a.\, b \rhd \Pi a'.\, b'}$$

Figure 3.2: Definition of PTS parallel one-step β-reduction.

the LHS to the already normal RHS of the equation. Statements (iii) and (iv) are trivial corollaries of (i) and Lemma 2.4.28 and, respectively, Lemma 2.4.30, with f in each case instantiated to $\lambda a \Rightarrow a[\sigma]$. ∎

When we consider the β-rule of $\rhd$ in Figure 3.2, namely

$$\frac{b \rhd b' \qquad c \rhd c'}{(\lambda a.\, b)\, c \rhd b'[c' \cdot \mathsf{id}]}$$

it will become apparent that we will eventually have to cope with reductions within substitutions as well. In particular, when it comes to the construction and correctness proof of the triangle function $\varphi_{\rhd}$. We therefore require a notion of reduction for substitutions.

Definition 3.3.11 We define $\tau \rhd \sigma$ as the pointwise extension of $\rhd$ to parallel substitutions:

$$\tau \rhd \sigma := \forall n.\ \tau n \rhd \sigma n.$$

Lemma 3.3.12 Reduction of parallel substitutions is preserved under lifting.

$$\tau \rhd \sigma \implies \Uparrow \tau \rhd \Uparrow \sigma$$

Proof By case analysis on the quantified variable. For $n = 0$ we know that $0 \rhd 0$ by the definition of $\Uparrow$. Otherwise we have $n + 1$ and need to show $(\tau n)[\uparrow] \rhd (\sigma n)[\uparrow]$, which holds due to part (ii) of Lemma 3.3.10. ∎

We can now prove the stronger instantiation compatibility result which also incorporates reductions within substitutions.

Lemma 3.3.13 (Instantiation Compatibility for $\rhd$) Let a and b be terms and τ and σ be substitutions, then the following holds.

$$a \rhd b \implies \tau \rhd \sigma \implies a[\tau] \rhd b[\sigma]$$

Proof By induction on $a \rhd b$, using Lemma 3.3.12 to handle the congruence cases involving binders. We again rely on the Autosubst decision procedure to establish the required equality for the β-reduction case. ∎

Theorem 3.3.14 (Compatibility of $\triangleright$ with β-Substitutions) Let a, a', b and b' be terms, then the following holds.

$$a \triangleright a' \implies b \triangleright b' \implies a[b \cdot \mathsf{id}] \triangleright a'[b' \cdot \mathsf{id}]$$

Proof This is an instance of Lemma 3.3.13. We obtain $b \cdot \mathsf{id} \triangleright b' \cdot \mathsf{id}$ with a simple case analysis on the quantified variable. ∎

Theorem 3.3.15 The parallel one-step reduction relation $\triangleright$ interpolates between one-step β-reduction and full β-reduction.

$$\succ \quad \subseteq \quad \triangleright \quad \subseteq \quad \succ^*$$

Proof The first inclusion is a simple induction on $a \succ b$, where the reflexivity of $\triangleright$ is necessary to handle the non-stepping subexpressions. For the second inclusion we perform an induction on $a \triangleright b$. Here we crucially rely on the fact that the componentwise congruence of $\succ$ lifts to the full congruence of $\succ^*$ (Lemma 2.4.29). The slightly involved case is, as so often, the β-redex, where we have to unfold $(\lambda a.\, b)\, c \succ^* b'[c' \cdot \mathsf{id}]$ to $(\lambda a.\, b)\, c \succ^* (\lambda a.\, b')\, c' \succ b'[c' \cdot \mathsf{id}]$. The former holds again thanks to Lemma 2.4.29 and the latter is the β-rule of $\succ$. ∎

Corollary 3.3.16 Parallel stepping entails convertibility.

$$\triangleright \quad \subseteq \quad \equiv$$

Proof Consequence of Theorem 3.3.15 and the fact that $\succ^* \subseteq \equiv$. ∎

Theorem 3.3.17 (Reduction Coincidence) The transitive closure of parallel one-step reduction coincides with regular β-reduction.

$$a \triangleright^* b \iff a \succ^* b$$

Proof Since we know that $\triangleright$ interpolates between $\succ$ and $\succ^*$ (Theorem 3.3.15), the desired result is a consequence of the general interpolation law for reduction relations (Lemma 2.4.24). ∎

With this result, we are now able to establish that not only parallel stepping, but also regular β-reduction is compatible with β-substitutions.

Theorem 3.3.18 (Compatibility of $\succ^*$ with β-Substitutions) Let a and a' as well as b and b' be terms, then the following holds.

$$a \succ^* a' \implies b \succ^* b' \implies a[b \cdot \mathsf{id}] \succ^* a'[b' \cdot \mathsf{id}]$$

Proof Given Theorem 3.3.17, it is sufficient to establish

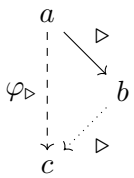
$$a \rhd^* a' \implies b \rhd^* b' \implies a[b \cdot \mathsf{id}] \rhd^* a'[b' \cdot \mathsf{id}].$$

Now thanks to Lemma 2.4.29 this can be further reduced to

$$a \rhd a' \implies a[b \cdot \mathsf{id}] \rhd a'[b \cdot \mathsf{id}] \qquad \text{and} \qquad d \rhd d' \implies c[d \cdot \mathsf{id}] \rhd c[d' \cdot \mathsf{id}],$$

which are both trivial instances of Theorem 3.3.14. ∎

The last missing ingredient in our confluence proof is now a suitable triangle function for $\rhd$. That is, we need a function $\varphi_\rhd$ that satisfies the following reduction diagram.



Recall that $\rhd$ reduces up to all immediately visible β-redices in one step, but it may also reduce fewer. We now define $\varphi_\rhd$ to *exactly* reduce all immediately visible β-redices, such that $\rhd$ is always capable to reduce those redices it missed in a first step with a single second step to close the triangle.

Definition 3.3.19 The triangle function $\varphi_\rhd$ is defined recursively as follows.

$$\begin{aligned}
\varphi_\rhd(n) &:= n \\
\varphi_\rhd(\mathsf{s}) &:= \mathsf{s} \\
\varphi_\rhd(\Pi a.\, b) &:= \Pi \varphi_\rhd(a).\, \varphi_\rhd(b) \\
\varphi_\rhd((\lambda a.\, b)\, c) &:= \varphi_\rhd(b)[\varphi_\rhd(c) \cdot \mathsf{id}] \\
\varphi_\rhd(a\, b) &:= \varphi_\rhd(a)\, \varphi_\rhd(b) \\
\varphi_\rhd(\lambda a.\, b) &:= \lambda \varphi_\rhd(a).\, \varphi_\rhd(b)
\end{aligned}$$

Lemma 3.3.20 The function $\varphi_\rhd$ satisfies the triangle property for $\rhd$.

$$a \rhd b \implies b \rhd \varphi_\rhd(a)$$

Proof Basic induction on $a \rhd b$, using Theorem 3.3.14. The case where $b = c\, d$ requires an additional case analysis of the term c, which again needs the compatibility with β-substitutions to handle the potentially occurring β-redex. ∎

Theorem 3.3.21 (Confluence of $\succ$) The PTS one-step β-reduction $\succ$ is confluent, that is, diverging reduction sequences can always be joined using $\succ^*$. Equivalently, $\succ$ is CR, that is, convertible terms can be joined using $\succ^*$.

Proof We rely on the generic Takahashi/Tait/Martin-Löf proof of confluence (Theorem 2.4.26), instantiated for $\succ$, $\triangleright$ and $\varphi_{\triangleright}$. Theorem 3.3.15 and Lemma 3.3.20 yield the required conditions. The fact that $\succ$ is CR follows from confluence and Lemma 2.4.21. ∎

Now that we know that $\succ$ is CR, we can establish further useful facts.

Fact 3.3.22 $\lfloor b \rfloor \implies a \equiv b \implies a \succ^* b$

Fact 3.3.23 $\lfloor a \rfloor \implies \lfloor b \rfloor \implies a \equiv b \implies a = b$

Fact 3.3.24 $a \equiv \mathsf{s} \implies a \succ^* \mathsf{s}$

Fact 3.3.25 $\mathsf{s} \equiv a \implies a \succ^* \mathsf{s}$

Fact 3.3.26 $\lfloor a \rfloor \implies a \equiv \mathsf{s} \implies a = \mathsf{s}$

Fact 3.3.27 $\lfloor a \rfloor \implies \mathsf{s} \equiv a \implies a = \mathsf{s}$

Fact 3.3.28 $\mathsf{s}_1 \equiv \mathsf{s}_2 \implies \mathsf{s}_1 = \mathsf{s}_2$

The last properties of interest in this section pertain to the interaction of conversion with dependent function types and β-substitutions.

Lemma 3.3.29 Dependent function types and sorts are not convertible.

$$\Pi a.\, b \not\equiv \mathsf{s}$$

Proof Assume $\Pi a.\, b \equiv \mathsf{s}$, then necessarily $\Pi a.\, b \succ^* \mathsf{s}$. But from Lemma 3.3.7 it then follows that there are some a', b', such that $\Pi a'.\, b' = \mathsf{s}$. Contradiction. ∎

Lemma 3.3.30 Conversion is a congruence with respect to dependent function types.

$$\Pi a.\, b \equiv \Pi c.\, d \implies a \equiv c \wedge b \equiv d$$

Proof From Lemma 3.3.7 and CR we know that there are terms a' and b', such that both $\Pi a.\, b \succ^* \Pi a'.\, b'$ and $\Pi c.\, d \succ^* \Pi a'.\, b'$. We further have that $a \succ^* a'$ and $c \succ^* a'$ as well as $b \succ^* b'$ and $d \succ^* b'$. Hence clearly $a \equiv c$ and $b \equiv d$. ∎

We also observe that conversion is compatible with β-subtitutions. We have postponed this result all the way to the end, since a priori, conversion may also contain β-expansions for which we do not obtain compatibility. As $\succ$ is CR, however, we can reduce convertibility to joinability and do not run into this complication.

Theorem 3.3.31 (Compatibility of $\equiv$ with β-Substitutions) Let a and a' as well as b and b' be terms, then the following holds.

$$a \equiv a' \implies b \equiv b' \implies a[b \cdot \mathsf{id}] \equiv a'[b' \cdot \mathsf{id}]$$

Proof Clearly we have an a'' joining a and a', as well as a b'' joining b and b'. Hence we can construct $a''[b'' \cdot \mathsf{id}]$ as a common reduct of $a[b \cdot \mathsf{id}]$ and $a'[b' \cdot \mathsf{id}]$ according to Theorem 3.3.18. The two expressions are therefore convertible. ∎

3.4 Free Variables

When we work with de Bruijn syntax it is rarely necessary to explicitly talk about the free variables of an expression, sometimes referred to as **dangling** de Bruijn indices. In particular, when the natural notion of parallel substitutions is used. There are, however, certain cases where the need to talk about free variables does arise and there are a number of approaches which appear useful. The two obvious choices are either a recursive function which computes the exact set of free variables or a predicate which fixes an approximation in terms of an upper bound on the variables which may occur freely. As it turns out, both approaches are problematic. The approximation of the latter is always a continuous range of variables starting at 0, which is too imprecise for our purposes, while the former leads to technically heavy and inelegant formalisations.

The approach which we found to work best, since it nicely interacts with our definition of instantiation, is an intermediate solution. We still work with an over-approximation of the set of free variables but we do not require it to be a continuous range. We achieve this by phrasing the set as a predicate $P : \mathbb{N} \to \mathbb{P}$ which should be satisfied by all free variables. This universal satisfaction is then captured in a recursively defined predicate $\mathsf{all} : (\mathbb{N} \to \mathbb{P}) \to \mathsf{Tm}_\lambda \to \mathbb{P}$ over general PTS terms.[4] The key insight here is that a predicate on de Bruijn indices is essentially a parallel substitution that maps variables to truth values. The all predicate is then structurally similar to instantiation in that it combines the obtained truth values at each variable into a single truth value for a given PTS term. When we adopt the view that our predicate on variables is a substitution, or equivalently a sequence of truth values, it also makes sense to adopt the primitives and operations of the σ-calculus, which we introduced in Section 2.3.1. That is, we write $\xi \circ P$ for pre-composing a renaming ξ to a predicate P, and $\top \cdot P$ for the predicate that evaluates to $\top$ at index 0 and to $P\,n$ at index $n+1$. The consing is, as usual, useful for the binder cases, where the constant $\top$ for index 0 indicates that we do not care about the value of the predicate on bound variables (also recall that $\top$ is the neutral element of conjunction, which we will use to combine truth values evaluated for subexpressions).

Definition 3.4.1 Let $P : \mathbb{N} \to \mathbb{P}$ be a predicate on variables. We can recursively lift it to a predicate $\mathsf{all}\,P : \mathsf{Tm}_\lambda \to \mathbb{P}$ on PTS terms.

$$
\begin{aligned}
\mathsf{all}\,P\,n &\coloneqq P\,n \\
\mathsf{all}\,P\,\mathsf{s} &\coloneqq \top \\
\mathsf{all}\,P\,(\Pi a.\,b) &\coloneqq \mathsf{all}\,P\,a \wedge \mathsf{all}\,(\top \cdot P)\,b \\
\mathsf{all}\,P\,(a\,b) &\coloneqq \mathsf{all}\,P\,a \wedge \mathsf{all}\,P\,b \\
\mathsf{all}\,P\,(\lambda a.\,b) &\coloneqq \mathsf{all}\,P\,a \wedge \mathsf{all}\,(\top \cdot P)\,b
\end{aligned}
$$

[4] The original idea of the all predicate, its properties and its use in a simply typed de Bruijn context is due to Steven Schäfer and was communicated privately. We scale the approach in this section to the dependently typed setting.

One of the key properties of this setup, and one of the main reasons, why this approach plays so well with our parallel substitution setup, is the fact that instantiations (with renamings as well as full substitutions) can be folded into the argument predicate of an all statement. This is witnessed by the following two lemmas.

Lemma 3.4.2 The $\mathsf{all}\,P$ predicate and renaming interact nicely.

$$\mathsf{all}\,P\,a[\xi] = \mathsf{all}\,(\xi \circ P)\,a$$

Proof By induction on a using the Autosubst decision procedure. ∎

Lemma 3.4.3 The $\mathsf{all}\,P$ predicate and substitution also interact nicely.

$$\mathsf{all}\,P\,a[\sigma] = \mathsf{all}\,(\sigma \circ \mathsf{all}\,P)\,a$$

Note that $\sigma \circ \mathsf{all}\,P$ is plain forward function composition.

Proof By induction on a. In the binder cases we need to solve the following equality.

$$\top \cdot \sigma \circ \mathsf{all}\,P = \Uparrow\sigma \circ \mathsf{all}\,(\top \cdot P)$$

Using functional extensionality and the Autosubst decision procedure we can reduce this in the $n = 0$ case to $\top = \top$. In the non-zero case we have to show

$$\mathsf{all}\,P\,(\sigma\,n) = \mathsf{all}\,(\top \cdot P)\,(\sigma\,n)[\uparrow],$$

which Lemma 3.4.2 reduces to

$$\mathsf{all}\,P\,(\sigma\,n) = \mathsf{all}\,(\uparrow \circ\,(\top \cdot P))\,(\sigma\,n)$$

and $\uparrow \circ\,(\top \cdot P) = P$ clearly holds. ∎

We also observe the following monotonicity property, which allows us to establish the interaction of all and β-substitutions. Predicate inclusion $P \subseteq Q$ is defined as usual as $\forall x.P\,x \implies Q\,x$.

Fact 3.4.4 (Monotonicity of all) $P \subseteq Q \implies \mathsf{all}\,P \subseteq \mathsf{all}\,Q$

Lemma 3.4.5 $\mathsf{all}\,(\top \cdot P)\,a \implies \mathsf{all}\,P\,c \implies \mathsf{all}\,P\,a[c \cdot \mathsf{id}]$

Proof We can use Lemma 3.4.3, Autosubst and the fact that $\mathsf{id} \circ \mathsf{all}\,P = P$ to rewrite our goal to $\mathsf{all}\,(\mathsf{all}\,P\,c \cdot P)\,a$. From $\mathsf{all}\,P\,c$ it clearly follows that $\top \cdot P \subseteq \mathsf{all}\,P\,c \cdot P$ and hence monotonicity closes the proof. ∎

Another interesting property of the $\mathsf{all}\,P$ predicate is that it is preserved under β-reduction. This is due to the fact that reduction will never introduce new variables. While it may remove free variables, this will not affect the truth value of the $\mathsf{all}\,P$ predicate. In particular, it is preserved under a single β-reduction step, so we do not need to delegate to $\triangleright$ to complete the following proof.

Theorem 3.4.6 The set of free variables of a term a does not increase under β-reduction, hence the following hold.

$$a \succ a' \implies \mathsf{all}\, P\, a \implies \mathsf{all}\, P\, a' \quad \text{(i)}$$
$$a \succ^* a' \implies \mathsf{all}\, P\, a \implies \mathsf{all}\, P\, a' \quad \text{(ii)}$$

Proof We proof (i) by induction on $a \succ a'$. Most cases are trivial. For the β-case we use Lemma 3.4.5. We lift (i) to (ii) with the reduction-generic Lemma 2.4.27. ∎

One scenario where a notion of free variables is essential is the concept of a vacuous binder, that is, a binder, whose bound variable does not occur in its body. In our de Bruijn setting, this means that the index 0 does not occur. The easiest way to deal with this situation is to define a suitable predicate NZ as

$$\mathsf{NZ} \coloneqq \lambda n \Rightarrow n \neq 0$$

and then express the non-occurrence of 0 in a term a as

$$\mathsf{all}\, \mathsf{NZ}\, a.$$

We will, in particular, need the following result.

Lemma 3.4.7 A term a with at least one applied shift does not contain 0 freely.

$$\mathsf{all}\, \mathsf{NZ}\, a[\uparrow]$$

Proof With Lemma 3.4.2 we reduce this to $\mathsf{all}\,(\uparrow \circ \mathsf{NZ})\, a$ and basic arithmetic establishes that $\uparrow \circ \mathsf{NZ}$ is equal to $(\lambda n \Rightarrow \top)$. Clearly $\mathsf{all}\,(\lambda n \Rightarrow \top)\, a$ holds by an inductive argument on a. ∎

3.5 Typing

So far we have only looked at the syntactic aspects of a PTS. The name however, already suggests that the focus of this formalism is its static semantics, that is, its type system. We are, in particular, interested in the formation and population of dependent function types, or, more precisely, exactly what forms of dependencies are expressible.

The named PTS typing judgement $\Psi \vdash^{\mathsf{nmd}}_{\lambda} a : b$ is inductively defined according to the rules of Figure 3.3. Note that the named rules are mostly given for reference, as the remainder of this chapter focuses on the de Bruijn version.

We use terms both in the subject and the predicate (or type) positions, here a and b, and we say that a inhabits, lives in, or is contained in, b. We also have a typing context Ψ[5] which provides the (semantic) types of the free variables in a and b.

[5] We will use Ψ (and later also Ξ) to denote PTS contexts. This allows us to reserve the more common Γ and Δ for stratified STLC and PLC. The goal here is to avoid confusion when judgements from different systems and their respective contexts are placed in close proximity.

$$\text{PTS-N-AX}\ \frac{(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{A}}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} \mathsf{s}_1 : \mathsf{s}_2} \qquad \text{PTS-N-VAR}\ \frac{x{:}a \in \Psi \qquad \Psi \vdash^{\mathsf{nmd}}_{\lambda} a : \mathsf{s}}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} x : a}$$

$$\text{PTS-N-PI}\ \frac{(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R} \qquad \Psi \vdash^{\mathsf{nmd}}_{\lambda} a : \mathsf{s}_1 \qquad \Psi, x{:}a \vdash^{\mathsf{nmd}}_{\lambda} b : \mathsf{s}_2}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} \Pi x{:}a.\, b : \mathsf{s}_3}\ x \notin \mathsf{dom}\ \Psi$$

$$\text{PTS-N-LAM}\ \frac{(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R} \qquad \Psi \vdash^{\mathsf{nmd}}_{\lambda} a : \mathsf{s}_1 \qquad \Psi, x{:}a \vdash^{\mathsf{nmd}}_{\lambda} c : \mathsf{s}_2 \qquad \Psi, x{:}a \vdash^{\mathsf{nmd}}_{\lambda} b : c}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} \lambda x{:}a.\, b : \Pi x{:}a.\, c}\ x \notin \mathsf{dom}\ \Psi$$

$$\text{PTS-N-APP}\ \frac{\Psi \vdash^{\mathsf{nmd}}_{\lambda} a : \Pi x{:}c.\, d \qquad \Psi \vdash^{\mathsf{nmd}}_{\lambda} b : c}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} a\, b : d[b/x]}$$

$$\text{PTS-N-CONV}\ \frac{\Psi \vdash^{\mathsf{nmd}}_{\lambda} a : b \qquad \Psi \vdash^{\mathsf{nmd}}_{\lambda} c : \mathsf{s} \qquad b \equiv c}{\Psi \vdash^{\mathsf{nmd}}_{\lambda} a : c}$$

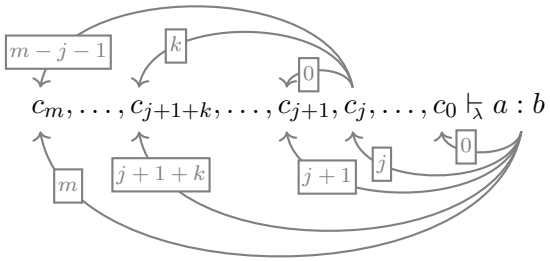
Figure 3.3: The named PTS typing rules for $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

In the named setting, such a typing context is usually represented as a mapping from variable names $x \in \mathcal{V}$ to their associated types $c \in \mathsf{Tm}^{\mathsf{nmd}}_{\lambda}$, and care has to be taken that no name aliasing is introduced upon context extension. In the de Bruijn setting on the other hand, typing contexts are simply lists of types $c \in \mathsf{Tm}_{\lambda}$ and the type of a free variable n is found at position n in this list. Note that to conform to the usual presentation of typing contexts we write these context lists with their head to the right. Thus named and de Bruijn typing contexts are formed according to the following grammars.

$$\Psi ::= \bullet \mid \Psi, x{:}c \qquad \text{(named)}$$
$$\Psi ::= \bullet \mid \Psi, c \qquad \text{(de Bruijn)}$$

We use $\mathsf{dom}\ \Psi$ to denote the domain of a context. For the named setting this is the set of declared variable names, while for a de Bruijn context we simply get $\mathsf{dom}\ \Psi := |\Psi| - 1$.

Note that the types in our typing contexts can very well themselves contain free variables, which are taken to refer back into the same context. In other words, we are dealing with potentially self-referential **dependent typing contexts**, which immediately poses the danger of circular dependencies. In the named setting separate conditions are needed to prevent this from happening. In the de Bruijn setting we prevent it by adapting the interpretation of positional indexing as follows. Let k be a free variable in a type c_j at position j in Ψ, then its type can be found at position $j+1+k$. This referencing policy can be visualised with the following diagram.



A consequence of this policy is that whenever we consider a type from a typing context in some different position we have to adjust the free variables correctly for this new position. The prime example here is context extraction where we need to assign a type to a free variable. This leads to the following inductive definition of **dependent context lookup** which performs the necessary adjustment with incremental shifts.

$$\text{L-ZERO}\ \frac{}{\Psi, a \vdash_\forall 0 : a[\uparrow]} \qquad \text{L-SUCC}\ \frac{\Psi \vdash_\forall n : a}{\Psi, b \vdash_\forall n + 1 : a[\uparrow]}$$

Fact 3.5.1 Dependent context lookup $(\Psi \vdash_\forall \_ : \_) \subseteq \mathbb{N} \times \mathsf{Tm}_\lambda$ is a **functional** relation.

Now that we have a notion of typing contexts, it appears prudent to also extend our treatment of free variables to such contexts. We construct a **context closure** by lifting our predicate $\mathsf{all}\, P$ partially pointwise, taking care to only consider entries which satisfy our predicate P in the first place.

Definition 3.5.2 (Context Closure of all P) The notion of all relevant, freely occurring variables in a typing context Ψ is defined as follows.

$$\mathsf{all}\, P\, \Psi := \forall na.\ P\, n \implies \Psi \vdash_\forall n : a \implies \mathsf{all}\, P\, a$$

Lemma 3.5.3 The $\mathsf{all}\, P$ predicate for contexts is preserved under context extension by suitably adjusting P.

$$\mathsf{all}\, P\, \Psi \implies \mathsf{all}\, P\, a \implies \mathsf{all}\, (\top \cdot P)\, (\Psi, a)$$

Proof Trivial using Lemma 3.4.2. ∎

We can now express the following interesting fact: no type a in a given context Ψ, even the rightmost type, can reference the head position of the same context. This is an immediate consequence of the definition of our dependent context lookup and the non-occurrence of 0 after shifting. In a way this expresses the non-circularity of dependent de Bruijn contexts.

Fact 3.5.4 (Non-Circularity) $\mathsf{all}\, \mathsf{NZ}\, \Psi$

$$\text{PTS-AX}\ \frac{(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{A}}{\Psi \vdash_\lambda \mathsf{s}_1 : \mathsf{s}_2} \qquad \text{PTS-VAR}\ \frac{\Psi \vdash_\mathsf{v} n : a \qquad \Psi \vdash_\lambda a : \mathsf{s}}{\Psi \vdash_\lambda n : a}$$

$$\text{PTS-PI}\ \frac{(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R} \qquad \Psi \vdash_\lambda a : \mathsf{s}_1 \qquad \Psi, a \vdash_\lambda b : \mathsf{s}_2}{\Psi \vdash_\lambda \Pi a.\, b : \mathsf{s}_3} \qquad \text{PTS-LAM}\ \frac{(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R} \qquad \Psi \vdash_\lambda a : \mathsf{s}_1 \qquad \Psi, a \vdash_\lambda b : c \qquad \Psi, a \vdash_\lambda c : \mathsf{s}_2}{\Psi \vdash_\lambda \lambda a.\, b : \Pi a.\, c}$$

$$\text{PTS-APP}\ \frac{\Psi \vdash_\lambda a : \Pi c.\, d \qquad \Psi \vdash_\lambda b : c}{\Psi \vdash_\lambda a\, b : d[b \cdot \mathsf{id}]} \qquad \text{PTS-CONV}\ \frac{\Psi \vdash_\lambda a : b \qquad \Psi \vdash_\lambda c : \mathsf{s} \qquad b \equiv c}{\Psi \vdash_\lambda a : c}$$

Figure 3.4: The de Bruijn PTS typing rules for $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

With the proper notions of dependent typing contexts settled we can finally introduce and discuss the actual de Bruijn PTS typing rules.

Definition 3.5.5 (PTS Typing Judgement.) The PTS typing judgement $\Psi \vdash_\lambda a : b$ for terms a and b and de Bruijn typing context Ψ is inductively defined according to the rules of Figure 3.4.

The axiom rule PTS-AX seeds the set of derivable typing judgements by injecting the assumptions about universe containment recorded in $\mathcal{A}$ into the type system. A trivial inductive argument ascertains that for $\mathcal{A} = \emptyset$, no judgement of the form $\Psi \vdash_\lambda a : b$ is derivable.

The rule PTS-VAR takes care of variables, which at the point of the rule invocation are free. The type of the respective variable is determined via the dependent context lookup $\Psi \vdash_\mathsf{v} n : a$ introduced above. We would like to point out that the n in the premise of the rule is an index, while the n in the conclusion is a term. If we were to make the variable constructor explicit, the rule would read as follows.

$$\text{PTS-VAR}\ \frac{\Psi \vdash_\mathsf{v} n : a \qquad \Psi \vdash_\lambda a : \mathsf{s}}{\Psi \vdash_\lambda \mathsf{v}_n : a}$$

Note that we also require that the extracted type a inhabits some universe s. This is in fact the reason why we refer to the constants $\mathsf{s} \in \mathcal{S}$ as universes. They are the **type universes** inhabited by (almost) exactly those terms which may be used as a type in the predicate position of the typing judgement. The only exceptions to this rule are universes that appear as the top element in a given universe stack, e.g. s_3 in $\mathsf{s}_1 : \mathsf{s}_2 : \mathsf{s}_3$, if such finite stacks exist in the given PTS.

The next two rules, PTS-PI and PTS-LAM, are best viewed together, as they share a lot of common structure. Both are parametric in $\mathcal{R}$, the third component of our PTS specification. The formation of a dependent function type $\Pi a.\, b$ involves three types. The first two are the domain a and dependent codomain b. The third is the resulting function type itself. As types, all three have to live in some universe and

a each triple $(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}$ yields the admissibility of forming a particular class of function types. Meanwhile, the abstraction rule ensures that the abstraction $\lambda a.\, b$ inhabits $\Pi a.\, c$, whenever two conditions are met: first, the function type should be formable so we duplicate the premises from the function type rule PTS-PI, and second, we require that the body of the abstraction b does in fact inhabit the codomain c. Note also, how the dependencies in the codomain and the abstraction body are elegantly handled by simply pushing the type of the bound variable onto the context. This maintains the de Bruijn binding discipline without requiring further adjustments. We can visualise this as

$$\text{PTS-PI}\ \frac{\ldots \qquad \Psi, a \overset{\curvearrowleft}{\vdash_\lambda} b : \mathsf{s}_2}{\Psi \overset{\curvearrowleft}{\vdash_\lambda} \Pi a.\, b : \mathsf{s}_3} \qquad \text{PTS-LAM}\ \frac{\ldots \qquad \Psi, a \overset{\curvearrowleft}{\vdash_\lambda} b : c \qquad \Psi, a \overset{\curvearrowleft}{\vdash_\lambda} c : \mathsf{s}_2}{\Psi \overset{\curvearrowleft}{\vdash_\lambda} \lambda a.\, b : \Pi a.\, c}$$

where the arrows above and below the rules all illustrate the correct dereferencing of the index 0.

The application rule PTS-APP is relatively straightforward. The obvious parts are that the function subterm a of the application has to have a function type and that the argument subterm supplied to this function has to be typeable to the correct domain. Slightly more interesting is the effective type of the application, which is the codomain d of the function with the dependency instantiated with the actual argument b. We implement this using the β-substitution $b \cdot \mathsf{id}$ which also takes care of suitably lowering the non-zero free variables of d. The latter is necessary, since we remove a binder without extending the context.

Lastly we have the conversion rule PTS-CONV, which allows us to change the type b of a derivable judgement to another type c as long as b and c are convertible and the new type c is demonstrably an admissible type (i.e. it lives in some sort s). While this rule may appear straightforward at first it carries some subtleties. First, we have made the decision to work with untyped, external conversion, rather than a so-called judgemental equality, to be closer to practically implemented type systems and to decouple the properties we are going to prove in the following. This however entails that a conversion chain between b and c may a priori contain β-expansion steps to untypeable terms. For a detailed discussion on the trade-offs between the two approaches to treating conversion and their correspondence see [SH12]. Second, since the types in the premise and the conclusion are only connected via this untyped conversion chain, several of the key lemmas need to be generalised to work at least modulo reduction in type position (which includes, among others, the notion of predicate reduction).

The conversion rule also complicates inversion on derivable judgements as it is not structural in the subject of the judgement. This means that whenever we have a judgement where the subject has a particular head constructor then we can exactly pinpoint the rule that introduced this construction. There may, however, be an arbitrary number of instances of the conversion rule between the construction of the

subject and its appearance in the analysed judgement. Since conversion is transitive, we can compress adjacent conversion steps into a single step. This leads to a collection of inversion principles, also known as **stripping laws**, that we briefly introduce here and later often use without explicitly referencing them.

Lemma 3.5.6 (PTS Stripping Laws) The following inversion principles hold.

$$\Psi \vdash_\lambda \mathsf{s} : a \implies \exists \mathsf{s}'.\ (\mathsf{s}, \mathsf{s}') \in \mathcal{A} \wedge a \equiv \mathsf{s}' \tag{i}$$

$$\Psi \vdash_\lambda n : a \implies \exists a'.\ \Psi \vdash_\mathsf{v} n : a' \wedge a \equiv a' \tag{ii}$$

$$\Psi \vdash_\lambda \Pi a.\, b : c \implies \exists (\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}.\ \Psi \vdash_\lambda a : \mathsf{s}_1 \wedge \Psi, a \vdash_\lambda b : \mathsf{s}_2 \wedge c \equiv \mathsf{s}_3 \tag{iii}$$

$$\Psi \vdash_\lambda \lambda a.\, b : c \implies \exists d, (\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}.\ \Psi \vdash_\lambda a : \mathsf{s}_1 \wedge \Psi, a \vdash_\lambda d : \mathsf{s}_2 \tag{iv}$$
$$\wedge\ \Psi, a \vdash_\lambda b : d \wedge \Psi \vdash_\lambda \Pi a.\, d : \mathsf{s}_3 \wedge c \equiv \Pi a.\, d$$

$$\Psi \vdash_\lambda a\, b : c \implies \exists d e.\ \Psi \vdash_\lambda a : \Pi d.\, e \wedge \Psi \vdash_\lambda b : d \wedge c \equiv e[b \cdot \mathsf{id}] \tag{v}$$

Proof All by induction on the given derivation. In each induction we have to deal either with the correct structural rule, where we exploit that conversion is reflexive, or an instance of the conversion rule, where we can exploit the inductive hypothesis and the transitivity of conversion to obtain the desired result. ∎

In the context of formalised developments, our presentation of PTS typing is relatively standard, mostly due to the idiosyncrasies of working with a de Bruijn encoding. Note, however, that there are two notable differences with respect to the original, named formulations of the system [Bar91, GN91]. These relate to the notions of context validity and the process of weakening a given judgement by adding extra contextual assumptions without affecting derivability. Validity is a context property which states that valid contexts only contain well-formed types. The traditional presentations usually add a restricted weakening rule that adds a single element to the head of a given context and adjust the axiom and variable rules as follows (variation on the rules of Figure 3.3).

$$\textsc{pts-nt-ax}\ \frac{(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{A}}{\bullet \vdash^{\mathsf{nmd}}_\lambda \mathsf{s}_1 : \mathsf{s}_2} \qquad \textsc{pts-nt-var}\ \frac{\Psi \vdash^{\mathsf{nmd}}_\lambda a : \mathsf{s} \qquad x \notin \mathsf{dom}\ \Psi}{\Psi, x{:}a \vdash^{\mathsf{nmd}}_\lambda x : a}$$

$$\textsc{pts-nt-weak}\ \frac{\Psi \vdash^{\mathsf{nmd}}_\lambda a : b \qquad \Psi \vdash^{\mathsf{nmd}}_\lambda c : \mathsf{s} \qquad x \notin \mathsf{dom}\ \Psi}{\Psi, x{:}c \vdash^{\mathsf{nmd}}_\lambda a : b}$$

...

Further adjustments are the removal of the freshness conditions ($x \notin \mathsf{dom}\ \Psi$) in the rules PTS-N-PI and PTS-N-LAM. The freshness conditions of these rules can be omitted as the traditional system enforces the validity of contexts in every derivable judgement, while our variant admits non-valid contexts. This simplifies the proofs of metatheoretic properties of the typing judgement that do not really rely on validity at the cost of

ensuring that we only ever access valid portions of the context. However, since in the de Bruijn setting such freshness assumptions are dealt with automatically, we actually do not have to pay this cost. A further downside of the traditional formulation is the fact that the validity is verified at every axiomatic leaf of a derivation leading to massive derivation trees.

In our setting, we can define context validity as a separate inductive property for those cases where we do require it.

Definition 3.5.7 (PTS Context Validity) A de Bruijn PTS context Ψ is **valid** whenever $\mathsf{val}\ \Psi$ is derivable from the following rules.

$$\textsc{V-empty}\ \frac{}{\mathsf{val}\ \bullet} \qquad \textsc{V-ext}\ \frac{\mathsf{val}\ \Psi \qquad \Psi \vdash_\lambda a : \mathsf{s}}{\mathsf{val}\ \Psi, a}$$

In the following, whenever validity is relevant for a given result we make this dependency explicit. For our first main result, however, namely the compatibility of the typing judgement with instantiation, it does not come into play until the very end. We will also see that a noticeably generalised form of the weakening rule introduced above appears as an admissible rule on our way towards the compatibility result.

3.6 Context Morphism Lemmas

Our goal now is to establish the admissibility of the following substitution lemma. Since we deal with various forms of instantiation compatibility throughout this work, we will occasionally refer to the present result more precisely as the **compatibility of the PTS typing judgement with β-substitutions**.

$$\frac{\Psi, d \vdash_\lambda a : b \qquad \Psi \vdash_\lambda c : d \qquad \mathsf{val}\ \Psi}{\Psi \vdash_\lambda a[c \cdot \mathsf{id}] : b[c \cdot \mathsf{id}]}$$

A proof of admissibility will clearly involve an inductive argument on the derivation of the first premise. This proof will not succeed in the presented form due to two reasons. First, various quantities, like the context Ψ, d or the β-substitution $c \cdot \mathsf{id}$ are too specific to deliver a suitable inductive hypothesis, in particular with respect to the rules involving variable binding where these quantities change. The second concern relates to the fact that the termination order of the typing judgement follows the recursive structure of terms while the recursive definition of instantiation goes through a special variant of instantiation that is only defined for renamings as outlined in Section 2.3.1.

The first problem calls for a suitable generalisation of our statement and we have two options. We could either try to modify the statement as little as possible or we could be more radical. As little as possible in this context would require us to consider instantiating k rather than 0 with the term c. This would then entail that we have to split the context Ψ into two halves around position k. It would also massively

complicate the substitutions in the conclusion, as indices below k have to be preserved, while those above k need to be lowered. The real problem here is that this approach is strongly geared towards the notion of single-point substitutions, which is simply not native to the de Bruijn world. The alternative then is to adopt a **big-step approach**, a phrase coined by Adams in [Ada04], and allow the rule to completely change the **initial** typing context Ψ into another **target** context Ξ, rather than just remove a single entry. We then formulate a property on a substitution σ which ensures that the mapping from one context into the other is type-preserving. This property, denoted by $\sigma \Vdash_\lambda \Psi \to \Xi$, captures the inductive invariant which has to be maintained throughout the admissibility proof. Substitutions which satisfy such an invariant are referred to as **context morphisms** and the corresponding generalised big-step lemma, called a **context morphism lemma** (CML), takes the following form.

$$\frac{\Psi \vdash_\lambda a : b \qquad \sigma \Vdash_\lambda \Psi \to \Xi}{\Xi \vdash_\lambda a[\sigma] : b[\sigma]}$$

Note that at this point we neither demand the validity of Ψ nor of Ξ. Lemmas of this form have successfully been employed in similar scenarios in [GM97, Ada04] and we will encounter them repeatedly throughout this work. Let us therefore briefly summarise the resulting two-stage proof structure of the admissibility proof before we proceed with the concrete result:

- The renaming stage:
  1. Define the notion of a context renaming.
  2. Show that context renamings are stable under context extension. This ensures that the defined property really is an invariant.
  3. Prove the CML with instantiation restricted to renaming.
- The substitution stage:
  1. Define the notion of a context morphism.
  2. Show that context morphisms are stable under context extension, using the renaming variant of the CML from step 3 above.
  3. Prove the full CML.

Note that for the particular CML we consider here, weakening will occur as an instance of the renaming CML using $\uparrow$ as a context renaming. Similarly, the compatibility with β-substitutions will appear as an instance of the full CML.

Let us briefly consider where the present terminology originates from. The reason, why certain substitutions are referred to as *morphisms* relates to Lambek's observation that any **cartesian closed category** (CCC) can be used as a model for STLC [Lam80]. Such categorical semantics interpret types as objects of the category and well-typed terms as arrows, or morphisms, into their types. A CCC has binary products, which admits the interpretation of typing contexts Γ also as objects via recursion on the

context (which as we recall is a list of types). The empty context is interpreted as the terminal object 1. The interpretation of well-typed terms is then by recursion on the typing judgement which amounts to the following categorical situation.

$$[\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash t : A]\!]} [\![A]\!]$$

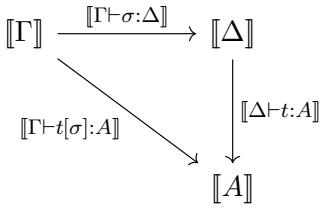
Since contexts map to categorical objects, it makes sense to consider arrows between two contexts. This motivates a pointwise extension of types to contexts and terms to substitutions with the following judgement.

$$\Gamma \vdash \sigma : \Delta$$

When this judgement holds, σ is referred to as a context morphism, due to the following categorical interpretation.

$$[\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash \sigma : \Delta]\!]} [\![\Delta]\!]$$

When we now pre-compose the interpretation of a substitution to the interpretation of a term, we end up with the following diagram.



The commuting property of this diagram is then the semantic counterpart to our notion of a context morphism lemma. Note that what we introduced here as a pointwise extension of the typing judgement to substitutions and contexts, $\Gamma \vdash \sigma : \Delta$, is exactly the morphism condition which we formulated above as the required inductive invariant $\sigma \Vdash_\lambda \Delta \to \Gamma$. There exist quit a number of different notations for this property throughout the literature and one aspect that often tends to be confusing is the direction of the arrow. When the focus of the discussion lies on the categorical semantics it is common practise to have the arrow point from the target to the initial context. This is for example done in [GM97] where the morphism condition is expressed as $\sigma : \Gamma \to \Delta$, or at least hinted at in the notation $\Gamma \vdash \sigma : \Delta$ used above and also in Pitts' excellent lecture notes on the categorical semantics of STLC [Pit17]. If on the other hand we are more interested in the operational nature of the context morphism lemma as an admissible inference rule of the type system it appears more sensible to have the arrow point from the initial context to the target context. This corresponds to our present focus. The operational view is further justified by the fact that we adapt the technique to systems and situations where it is not clear if the categorical interpretation is still applicable.

This concludes our brief excursion into the land of category theory and we return to the challenge at hand, that is the proof of the CML for the PTS typing judgement.

The Renaming Stage

Since renamings ξ map variables to variables, we are effectively dealing with a reorganisation of a given de Bruijn context Ψ. Recall that de Bruijn judgements are not invariant under context modifications (including reorderings). The proof of the renaming variant of the CML will make precise in what sense judgements vary under such changes. The basic idea is that whenever Ψ assigns type a to a variable k, then Ξ should assign the same type to variable $\xi\, k$. Clearly here *same*, means *same up to a suitable adjustment of free variables*. Also note that we only want to restrict the behaviour of the renaming ξ on those variables that actually occur in a typing under Ψ. The domain of Ψ is a safe approximation. In total, we obtain the following definition.

Definition 3.6.1 (PTS Context Renaming) A renaming ξ is a context renaming from Ψ to Ξ, whenever it satisfies the following condition.

$$\xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi \quad :\Leftrightarrow \quad \forall n a.\ \Psi \vdash_\forall n : a \implies \Xi \vdash_\forall \xi\, n : a[\xi]$$

Lemma 3.6.2 Lifting yields context renamings that satisfy the simultaneous extension of source and target context.

$$\xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi \implies \Uparrow\xi \Vdash_\lambda \Psi, a \xrightarrow{r} \Xi, a[\xi]$$

Proof Assume $\xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi$ and $\Psi, a \vdash_\forall n : b$ for some n and b. We have to show

$$\Xi, a[\xi] \vdash_\forall (\Uparrow\xi)\, n : b[\Uparrow\xi].$$

For $n = 0$ we obtain $b = a[\uparrow]$ and our goal reduces to

$$\Xi, a[\xi] \vdash_\forall 0 : a[\uparrow][\Uparrow\xi] \quad \iff \quad \Xi, a[\xi] \vdash_\forall 0 : a[\xi][\uparrow].$$

The RHS trivially holds and the equivalence is justified since $a[\uparrow][\Uparrow\xi] = a[\xi][\uparrow]$ is a theorem of the σ-calculus.

Now let $n = m+1$, then $b = b'[\uparrow]$ for some b' with $\Psi \vdash_\forall m : b'$. From our assumption we obtain $\Xi \vdash_\forall \xi\, m : b'[\xi]$ and from the definition of dependent context lookup we have

$$\Xi, a[\xi] \vdash_\forall (\xi\, m) + 1 : b'[\xi][\uparrow]$$

which, as before thanks to the σ-calculus, is equivalent to our goal

$$\Xi, a[\xi] \vdash_\forall \Uparrow\xi\, (m+1) : b'[\uparrow][\Uparrow\xi].$$ ∎

In the following we will encounter further such extension lemmas that all follow the same proof structure. We will therefore subsequently only point out that we proceed by case analysis on the quantified variable and focus on where additional assumptions are needed.

Lemma 3.6.3 (PTS Renaming CML) PTS typing is preserved under instantiation with context renamings.

$$\textsc{pts-rcml}\ \frac{\Psi \vdash_\lambda a : b \qquad \xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi}{\Xi \vdash_\lambda a[\xi] : b[\xi]}$$

Proof By induction on the derivation of $\Psi \vdash_\lambda a : b$. The renaming assumption closes the variable case and Lemma 3.6.2 is needed to instantiate the inductive hypotheses when descending under binders. We also need Lemma 3.3.10 for the conversion case. ■

Before we now commence with stage two of our proof we briefly remark upon the following fact about $\uparrow$, which is easy to verify.

Fact 3.6.4 $\uparrow \Vdash_\lambda \Psi \xrightarrow{r} \Psi, a$

We thus obtain weakening for our PTS as a derived concept, rather than as part of the initial definition.

Corollary 3.6.5 (PTS Weakening) The weakening rule PTS-WEAK is admissible.

$$\textsc{pts-weak}\ \frac{\Psi \vdash_\lambda a : b}{\Psi, c \vdash_\lambda a[\uparrow] : b[\uparrow]}$$

Proof

$$\textsc{pts-rcml}\ \frac{\Psi \vdash_\lambda a : b \qquad \overline{\uparrow \Vdash_\lambda \Psi \xrightarrow{r} \Psi, c}}{\Psi, c \vdash_\lambda a[\uparrow] : b[\uparrow]}$$

The second premise is an instance of Fact 3.6.4. ■

Now that we have weakening we can also establish that a successfull lookup from a valid context is sufficient to construct an admissible variable typing, that is, context validity yields the required well-formedness of the type.

Lemma 3.6.6 A successfull lookup in a valid context Ψ admits a variable typing.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_\forall n : a \implies \exists \mathsf{s}.\ \Psi \vdash_\lambda a : \mathsf{s} \tag{i}$$

$$\mathsf{val}\ \Psi \implies \Psi \vdash_\forall n : a \implies \Psi \vdash_\lambda n : a \tag{ii}$$

Proof Claim (i) is by induction on the derivation of $\Psi \vdash_\forall n : a$ and case analysis on the validity assumption. Both cases rely on weakening. Claim (ii) is a trivial corollary of (i) using the rule PTS-VAR. ■

The Substitution Stage

Our next order of business is to define what it means for a substitution σ to be considered as a context morphism. Recall that for renamings we simply mapped context lookups under the initial context to context lookups under the target context. Since substitutions map into the codomain Tm_λ we instead have to require that context morphisms yield typings, rather than lookups. We therefore use the following definition and then derive the full CML.

Definition 3.6.7 (PTS Context Morphism) A substitution σ is a context morphism from Ψ to Ξ, whenever it satisfies the following condition.

$$\sigma \Vdash_\lambda \Psi \to \Xi \quad := \quad \forall na.\ \Psi \vdash_\forall n : a \implies \Xi \vdash_\lambda \sigma\, n : a[\sigma]$$

Lemma 3.6.8 Lifting yields context morphisms that satisfy the simultaneous extension of source and target context.

$$\sigma \Vdash_\lambda \Psi \to \Xi \implies \Uparrow\sigma \Vdash_\lambda \Psi, a \to \Xi, a[\sigma]$$

Proof By case analysis on the quantified variable. In both cases we require Corollary 3.6.5 to weaken a given typing under Ξ to a typing under $\Xi, a[\sigma]$ in order to satisfy the conclusion. ∎

Lemma 3.6.9 (PTS CML) PTS typing is preserved under instantiation with context morphisms.

$$\text{PTS-CML}\ \frac{\Psi \vdash_\lambda a : b \qquad \sigma \Vdash_\lambda \Psi \to \Xi}{\Xi \vdash_\lambda a[\sigma] : b[\sigma]}$$

Proof By induction on the derivation of $\Psi \vdash_\lambda a : b$. The context morphism assumption closes the variable case. As before we use Lemma 3.6.8 to handle the two binder cases and Lemma 3.3.10 for the conversion case. ∎

We can finally tackle our desired substitution lemma as a special instance of the CML. We therefore have to show that β-substitutions constitute context morphisms of the correct form.

Lemma 3.6.10 β-substitutions are context morphisms.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : b \implies a \cdot \mathsf{id} \Vdash_\lambda \Psi, b \to \Psi$$

Proof By case analysis on the quantified variable. The $n = 0$ case is direct from the typing assumption. For the remaining case we obtain a lookup under Ψ and have to provide a variable typing under Ψ. For this we need the validity assumption and Lemma 3.6.6. ∎

Observe how the asymmetry in our context morphism definition, that is the mapping from lookups to variable typings, forces us here to require context validity. One might think that this could be alleviated by simply redefining context morphisms in a more symmetrical way, mapping variable typings to variable typings. This however would prevent us from proving the lifting lemma (Lemma 3.6.8) unless we have subject reduction. And as it will turn our shortly, subject reduction relies on the compatibility with β-substitutions. Hence the asymmetry in the definition was carfully chosen to prevent this circularity and we have to live with the imposed validity assumption.[6]

The actual substitution lemma is now a simple corollary.

Corollary 3.6.11 (PTS Typing Compatibility with β-Substitutions) The following rule is admissible.

$$\text{PTS-BETA}\ \frac{\Psi, d \vdash_\lambda a : b \qquad \mathsf{val}\ \Psi \qquad \Psi \vdash_\lambda c : d}{\Psi \vdash_\lambda a[c \cdot \mathsf{id}] : b[c \cdot \mathsf{id}]}$$

Proof

$$\text{PTS-CML}\ \frac{\Psi, d \vdash_\lambda a : b \qquad \dfrac{\mathsf{val}\ \Psi \qquad \Psi \vdash_\lambda c : d}{c \cdot \mathsf{id} \Vdash_\lambda \Psi, d \to \Psi}}{\Psi \vdash_\lambda a[c \cdot \mathsf{id}] : b[c \cdot \mathsf{id}]}$$

The unnamed rule is an instance of Lemma 3.6.10. ∎

3.6.1 Correctness of Typing

With the compatibility with β-substitutions, we are finally in a position to prove a property of our type system that was implicitly suggested from the outset but has so far not been made precise. When we look at a derivable typing judgement $\Psi \vdash_\lambda a : b$, then we refer to the term b as a type. We have suggested that types live in type universes, that is the constants $\mathsf{s} \in \mathcal{S}$. The universes themselves can also act as types. We now formally prove this claim, which is variously known as **propagation**, **correctness of types** or grouped in with the stripping laws.

Lemma 3.6.12 (PTS Propagation) Derivable judgements have well-formed types.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : b \implies \exists \mathsf{s}.\ b = \mathsf{s} \vee \Psi \vdash_\lambda b : \mathsf{s}$$

Proof By induction on the derivation of $\Psi \vdash_\lambda a : b$. The application case involves a β-substitution which can be handled with Corollary 3.6.11. ∎

Propagation allows us to lift the stripping laws to the predicate/type position. For dependent function types in particular this yields a rather useful inversion principle.

[6] See also Section 3.10 for further comments.

Lemma 3.6.13 (Inversion Principle for Function Types in Type Position)

$$\mathsf{val}\ \Psi \implies \Psi \mathrel{\underset{\lambda}{\vdash}} a : \Pi c.\, d \implies \\ \exists (\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}.\ \Psi \mathrel{\underset{\lambda}{\vdash}} c : \mathsf{s}_1 \wedge \Psi, c \mathrel{\underset{\lambda}{\vdash}} d : \mathsf{s}_2 \wedge \Psi \mathrel{\underset{\lambda}{\vdash}} \Pi c.\, d : \mathsf{s}_3$$

Proof From Lemma 3.6.12 it clearly follows that we have $\Psi \mathrel{\underset{\lambda}{\vdash}} \Pi c.\, d : \mathsf{s}$ for some s. Thus stripping allows us to obtain all desired properties. ∎

An immediate use of this result deals with the instantiation of bodies of dependent function types. In particular, we can infer that if a function type could be used as a type, then its instantiated codomain is well-sorted. This will turn out to be useful for handling certain application cases later on.

Lemma 3.6.14 (Well-Sortedness of Instantiated Codomains)

$$\mathsf{val}\ \Psi \implies \Psi \mathrel{\underset{\lambda}{\vdash}} a : \Pi c.\, d \implies \Psi \mathrel{\underset{\lambda}{\vdash}} b : c \implies \exists (\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}.\ \Psi \mathrel{\underset{\lambda}{\vdash}} d[b \cdot \mathsf{id}] : \mathsf{s}_2$$

Proof Straightforward combination of Lemma 3.6.13 and compatibility with β-substitutions (Corollary 3.6.11), using the assumed typing for b. ∎

3.7 Subject Reduction

Let us now consider the interaction of typing and reduction. More precisely, the type b of a derivable judgement $\Psi \mathrel{\underset{\lambda}{\vdash}} a : b$ is preserved when the subject a reduces according to $\succ^*$. To prove this result, commonly known as **subject reduction** or **preservation**, we again have to fall back to the parallel reduction relation $\triangleright$. The argument will again be inductive and the binder cases extend the context of the two judgements with a pair of terms, where one is the parallel reduct of the other. Hence for a suitable generalisation we have to lift the notion of parallel stepping to contexts.

Note that the following definition is closely related to the more familiar notion of context conversion, which states that all matching components of two contexts are pairwise β-convertible. Our notion of parallel context stepping plays approximately the same role in the de Bruijn subject reduction proof, that context conversion plays in the named setting. For the proof in a named PTS setting, see for example [SH12]. The reason we use the parallel stepping variant follows from the decision to first establish subject reduction for $\triangleright$ and then lift it to $\triangleright^*$ and finally $\succ^*$ with the generic constructions from Section 2.4. In addition, we found that working with asymmetric reduction relations rather than symmetric conversion simplifies matters somewhat, since it prevents unwanted β-expansions that we otherwise would need to rule out.

Definition 3.7.1 (Parallel One-Step Context Reduction) A context Ψ can pointwise and in parallel step to another context Ξ, whenever the two contexts satisfy the following condition.

$$\Psi \triangleright \Xi \quad := \quad \forall n a.\ \Psi \mathrel{\underset{\forall}{\vdash}} n : a \implies \exists a'.\ \Xi \mathrel{\underset{\forall}{\vdash}} n : a' \wedge a \triangleright a'$$

Lemma 3.7.2 Parallel stepping is preserved under context extension in the obvious way.

$$\Psi \rhd \Xi \implies a \rhd a' \implies \Psi, a \rhd \Xi, a'$$

Proof Straightforward using Lemma 3.3.10 to deal with the renamings that are caused by context lookups. ∎

The key lemma in our proof of subject reduction can now be formulated. Its proof is somewhat technical due to various moving parts.

Lemma 3.7.3 (PTS Subject Reduction under $\rhd$)

$$\Psi \vdash_\lambda a : b \implies \mathsf{val}\,\Xi \implies \Psi \rhd \Xi \implies a \rhd a' \implies \Xi \vdash_\lambda a' : b$$

Proof By induction on $\Psi \vdash_\lambda a : b$. The axiom case is trivial.

Let $a = n$ with $b = c$ for some c and s satisfying $\Psi \vdash_\forall n : c$ and $\Psi \vdash_\lambda c : \mathsf{s}$. From $\Psi \rhd \Xi$ it follows that $\Xi \vdash_\forall n : c'$ for some c' satisfying $c \rhd c'$. We clearly also have $c \rhd c$. Using the inductive hypothesis twice we obtain $\Xi \vdash_\lambda c : \mathsf{s}$ and $\Xi \vdash_\lambda c' : \mathsf{s}$. We have to show $\Xi \vdash_\lambda n : c$. By conversion ($c \equiv c'$) it is sufficient to show $\Xi \vdash_\lambda n : c'$, which can be derived from our assumptions using the rule PTS-VAR.

Let $a = \Pi c.\, d$. This case is a direct consequence of the inductive hypotheses and rule PTS-PROD. Lemma 3.7.2 is needed to accommodate the potential stepping of the domain c, when dealing with the codomain d.

Let $a = \lambda a'.\, b'$ with $b = \Pi a'.\, c$. We also have a'', b'' satisfying $a' \rhd a''$ and $b' \rhd b''$. Thus clearly $\Pi a'.\, c \rhd \Pi a''.\, c$ and hence $\Pi a'.\, c \equiv \Pi a''.\, c$. Our goal is $\Xi \vdash_\lambda \lambda a''.\, b'' : \Pi a'.\, c$. With conversion, it is sufficient to show $\Xi \vdash_\lambda \lambda a''.\, b'' : \Pi a''.\, c$. This, as well as the well-formedness of $\Pi a'.\, c$, are both derivable from the inductive hypotheses and Lemma 3.7.2.

Let $a = a$ and $b = c$ and we have some a' such that $a \rhd a'$. The inductive hypotheses are given for $\Psi \vdash_\lambda a : b'$ with $b' \equiv c$ and some s satisfying $\Psi \vdash_\lambda c : \mathsf{s}$. We can thus easily obtain $\Xi \vdash_\lambda a' : b'$ and with $c \rhd c$ also $\Xi \vdash_\lambda c : \mathsf{s}$. The goal $\Xi \vdash_\lambda a' : c$ is then a simple consequence of conversion.

Now let $a = a'\,b'$ and $b = d[b' \cdot \mathsf{id}]$. We know that $\Psi \vdash_\lambda a' : \Pi c.\, d$ and $\Psi \vdash_\lambda b' : c$. Again using reflexive instances of $\rhd$ we can already assert that $\Xi \vdash_\lambda a' : \Pi c.\, d$ and $\Xi \vdash_\lambda b' : c$. Using Lemma 3.6.14 we therefore also know that $\Xi \vdash_\lambda d[b' \cdot \mathsf{id}] : \mathsf{s}$ for some s. We now have to consider two cases, depending on how the application steps. We consider the congruence case first and consider terms a'', b'' such that $a' \rhd a''$ and $b' \rhd b''$. Here we have to show that $\Xi \vdash_\lambda a''\,b'' : d[b' \cdot \mathsf{id}]$. With Theorem 3.3.14 we can derive that $d[b' \cdot \mathsf{id}] \rhd d[b'' \cdot \mathsf{id}]$ and hence $d[b' \cdot \mathsf{id}] \equiv d[b'' \cdot \mathsf{id}]$, which we use to convert our goal to $\Xi \vdash_\lambda a''\,b'' : d[b'' \cdot \mathsf{id}]$. This, then, follows from our assumptions and inductive hypotheses. If on the other hand we have $(\lambda a''.\, e)\,b' \rhd e'[b'' \cdot \mathsf{id}]$ for some a'', e, e' and b'' with $e \rhd e'$ and $b' \rhd b''$, then we have to work a bit harder to obtain our goal $\Xi \vdash_\lambda e'[b'' \cdot \mathsf{id}] : d[b' \cdot \mathsf{id}]$. First of all we clearly have $\lambda a''.\, e \rhd \lambda a''.\, e'$ and therefore from our inductive hypothesis it follows that $\Xi \vdash_\lambda \lambda a''.\, e' : \Pi c.\, d$ and

we hence obtain $\Xi \vdash_\lambda a'' : \mathsf{s}'$ for some universe s', $\Xi, a'' \vdash_\lambda e' : d'$ for some d' as well as $c \equiv a''$ and $d \equiv d'$. We can thus construct $d'[b'' \cdot \mathsf{id}] \equiv d[b' \cdot \mathsf{id}]$ and convert our goal to $\Xi \vdash_\lambda e'[b'' \cdot \mathsf{id}] : d'[b'' \cdot \mathsf{id}]$. Also by induction we have $\Xi \vdash_\lambda b'' : c$, which we can convert to $\Xi \vdash_\lambda b'' : a''$. Corollary 3.6.11 now closes this case. ∎

Lemma 3.7.3 encapsulates the inductive nature of the subject reduction proof, while the lifting to $\triangleright^*$ and $\succ^*$ will be direct. The inductive argument introduced the need to deal with contexts which are separated by reduction steps and hence required the lifting of $\triangleright$ to contexts. Now, since $\triangleright$ is reflexive on terms, it is also reflexive on contexts, as witnessed by the following fact.

Fact 3.7.4 $\Psi \triangleright \Psi$

These considerations allow us to work, from now on, with the same context before and after the reduction of the subject term in question.

Lemma 3.7.5 Subject reduction under $\triangleright$ can be lifted to $\triangleright^*$, that is we obtain the following.

$$\mathsf{val}\ \Psi \implies a \triangleright^* a' \implies \Psi \vdash_\lambda a : b \implies \Psi \vdash_\lambda a' : b$$

Proof We use Lemma 2.4.27 with P instantiated to $\lambda c \Rightarrow \Psi \vdash_\lambda c : b$ and use Lemma 3.7.3 to ensure that P is preserved under a single $\triangleright$ step. ∎

Theorem 3.7.6 (PTS Subject Reduction) Typing is preserved under β-reduction of the subject of the judgement.

$$\mathsf{val}\ \Psi \implies a \succ^* a' \implies \Psi \vdash_\lambda a : b \implies \Psi \vdash_\lambda a' : b$$

Proof Trivial using Lemma 3.7.5 and Theorem 3.3.17. ∎

We can now also obtain a variant of the conversion rule which is sometimes referred to as **predicate reduction**. Note that we have replaced the existence of a type universe for the new type with a validity assumption on the context.

Corollary 3.7.7 (PTS Predicate Reduction) Typing is preserved under β-reduction of the predicate/type of the judgement.

$$\mathsf{val}\ \Psi \implies b \succ^* b' \implies \Psi \vdash_\lambda a : b \implies \Psi \vdash_\lambda a : b'$$

Before we close this section it is worth to also consider the following property which will help us in the next two chapters.

Lemma 3.7.8 When the codomains of dependent function types are known to never be universes themselves, then all universes are only inhabited by normal terms.

$$(\forall(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}, \mathsf{s}_4.\ (\mathsf{s}_4, \mathsf{s}_2) \notin \mathcal{A}) \implies \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : \mathsf{s} \implies \lfloor a \rfloor$$

Proof By induction on $\Psi \vdash_\lambda a : \mathsf{s}$. The case where a is an application is refuted by using Lemma 3.6.14 and subject reduction to construct a counterexample to the universe assumption. ∎

3.8 Functional Pure Type Systems and Strengthening

Recall that the admissibility of weakening appeared as a by-product of our proof of the main substitution lemma, and as such was relatively easy to obtain. What about the inverse direction, that is the following **strengthening** principle?

$$\frac{\Psi, c \vdash_\lambda a[\uparrow] : b[\uparrow]}{\Psi \vdash_\lambda a : b}$$

While it is intuitively clear that this property should hold, it turns out to be surprisingly tricky to prove. A naive approach is to recycle our CML for renamings, since $\downarrow := 0 \cdot \mathsf{id}$ is a right inverse of $\uparrow$, that is we have $\uparrow \circ \downarrow = \mathsf{id}$. Unfortunately though, $\downarrow \Vdash_\lambda \Psi, c \xrightarrow{r} \Psi$ fails to hold, since the invariant demands that the final context Ψ carries a typing for every entry in the initial context Ψ, c, including the vacuous entry at position 0. And there is no reason why $\Psi \vdash_\forall 0 : c[\uparrow][\downarrow]$ should be given.

We can recover from this problem by weakening the definition of the morphism invariant $\xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi$ in a certain way. This may at first look like a simple generalisation of the renaming CML we proved in Section 3.6, but things are unfortunately not quite as straightforward as that. The key problem is that in order to establish the (supposedly) generalised statement as a functioning renaming CML we indirectly require compatibility of typing with full substitutions (Lemma 3.6.9). The latter however, in turn, relies on the plain renaming CML introduced above (Lemma 3.6.3). Due to this circularity it appears unlikely that we could replace the plain renaming CML and start directly with the more general variant that is introduced here. We will expand on this issue in Section 3.10.

Before we make the modified construction precise, however, we would like to consider two strengthening proofs for the named PTS presentation which are present in the literature. The first is due to Geuvers and Nederhof [GN91] and appears in the context of a strong normalisation proof for the Calculus of Constructions. The Geuvers and Nederhof proof (GN) contains one crucial step that relies on uniqueness of type assignment, which however does not hold for a general PTS. They recover uniqueness of typing by restricting themselves to the subclass of functional pure type systems (FPTSs). The restriction is relatively weak since most systems of interest turn out to be functional.

A more general proof was later presented in [vBJ93], which does not rely on type uniqueness. It instead builds on the weaker notion of domain uniqueness, which is shown to hold for an arbitrary PTS. A crucial component of this proof is a precise notion of n-ary dependent function types as n-fold iteration of the unary function type construction.

We are going to combine our initial idea of a generalised renaming CML and the GN proof to construct a solution for our de Bruijn setting. Since we will in this thesis only look at concrete PTS instances that are functional, this design decision will not negatively impact our development.

The first thing we have to clarify is the notion of a functional PTS.

Definition 3.8.1 (Functional Pure Type System) A PTS $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is functional iff its components satisfy the following two properties.

$$(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{A} \implies (\mathsf{s}_1, \mathsf{s}_2') \in \mathcal{A} \implies \mathsf{s}_2 = \mathsf{s}_2'$$
$$(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R} \implies (\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3') \in \mathcal{R} \implies \mathsf{s}_3 = \mathsf{s}_3'$$

Formally, we give a signature extension FPTS, which includes the PTS signature and in addition requires proofs of the two functionality principles. We thus automatically obtain all the results of the previous sections for any given FPTS $\mathcal{P}$ and use them here to derive strengthening.

It is a relatively straightforward inductive argument to establish that under these conditions we obtain uniqueness of type assignment, that is, if a term a has both b and c as types under a given context Ψ, then $b \equiv c$. Unfortunately though, it turns out that this result is insufficient in the de Bruijn setting and we will need something more general. We thus defer the proof of type uniqueness as a corollary of the more general result towards the end of this section.

When we go back to our failed proof attempt based on the standard renaming CML we notice that the employed morphism invariant is too strong to be suitably instantiated. In particular, it demands information about variables in the initial context Ψ, which may not even be accessed in typings under Ψ. And it was one of those non-occurring variables that was causing problems, namely the type c at index 0 of Ψ, c, when both $a[\uparrow]$ and $b[\uparrow]$ clearly cannot have index 0 occurring freely. At the heart of it, we are faced with the problem that our present morphism condition is essentially a property of the context Ψ *on its own*, while the set of **relevant** context entries crucially depends on *concrete typings under* Ψ. The solution to this is an invariant that is instrumented in such a way that the missing information can be provided to close the mismatch.

The next question we have to answer is: given a typing $\Psi \vdash_\lambda a : b$, which part of $\mathsf{dom}\ \Psi$ matters? The obvious part of the answer is "the free variables of a and b". Recall however, that Ψ is a dependent context with potential self-references. Thus we also have to include the free variables of every type c, which we do extract from Ψ, recursively.

At this point we recall our treatment of free variables in terms of the all predicate. Let $P : \mathbb{N} \to \mathbb{P}$ be a predicate where we interpret $P\,n$ as "variable n is relevant". When we have $\mathsf{all}\,P\,a$, $\mathsf{all}\,P\,b$ and $\mathsf{all}\,P\,\Psi$, then P holds at least on all relevant free variables of the judgement $\Psi \vdash_\lambda a : b$. We recall in particular the definition of $\mathsf{all}\,P\,\Psi$, namely

$$\mathsf{all}\,P\,\Psi := \forall n a.\ P\,n \implies \Psi \vdash_\forall n : a \implies \mathsf{all}\,P\,a,$$

and observe that the guard $P\,n$ ensures that only the P-relevant fragment of Ψ is checked. That is, we only *partially* check Ψ as mediated by P.

The solution to our problem now is to use a similar idea for our morphism invariant. That is, we want it to be a property of only a certain subset of the context Ψ, not of the whole context. The concrete subset itself is factored out as a separate parameter

in the form of the relevance predicate P on variables. Since P will be chosen to contain at least all relevant variables, this will turn out to be a safe weakening of the definition. In this sense we are now dealing with **partial** context renaming, where we should point out that the partiality is meant with respect to the initial context, not the renaming itself. The latter is still a total function that acts on all variables, but we do no longer care what it does to those variables that are not considered relevant.

Definition 3.8.2 (Partial Context Renaming) We say that ξ is a partial context renaming from initial context Ψ to final context Ξ mediated by a predicate P, whenever the following property holds.

$$\xi \Vdash_\lambda [P]\,\Psi \xrightarrow{r} \Xi \quad :=\quad \forall n a.\ P\,n \implies \Psi \vdash_{\mathsf{V}} n : a \implies \Xi \vdash_{\mathsf{V}} \xi\,n : a[\xi]$$

Observe the guard $P\,n$ that was added relative to the definition of $\xi \Vdash_\lambda \Psi \xrightarrow{r} \Xi$.

Lemma 3.8.3 Lifting yields partial context renamings that satisfy the simultaneous extension of source and target context, if the predicate P is adjusted accordingly.

$$\xi \Vdash_\lambda [P]\,\Psi \xrightarrow{r} \Xi \implies \Uparrow\xi \Vdash_\lambda [\top \cdot P]\,\Psi, a \xrightarrow{r} \Xi, a[\xi]$$

Proof Similar to Lemma 3.6.2 by discriminating on the quantified variable. ∎

We would now like to obtain a partial renaming CML, analogue to Lemma 3.6.3, which will subsequently yield strengthening as a special case. In doing so, we have to overcome a number of challenges, some of which are already present and handled in the on-paper GN proof, while others are consequences of our de Bruijn setting.

In the GN proof, which works in the named setting, there occur two typings $\Psi, x{:}b \vdash^{\mathsf{nmd}}_\lambda a : \mathsf{s}_1$ and $\Psi \vdash^{\mathsf{nmd}}_\lambda a : \mathsf{s}_2$. In order to proceed, it is necessary to infer that $\mathsf{s}_1 \equiv \mathsf{s}_2$ and thus $\mathsf{s}_1 = \mathsf{s}_2$ (convertible universes are equal). The proof achieves this by weakening the second typing such that the contexts match and then employs uniqueness of typing.

The problem with this is that the first typing under context $\Psi, x{:}b$ is obtained by discriminating on the premise in the inductive proof, while the second typing under Ψ is an instance of the inductive hypothesis. The CML that we are planning to prove will have to generalise this to potentially completely different contexts Ψ and Ξ, whose only connection is a partial context renaming. And to align these for the ordinary uniqueness result to be applicable, we would need the partial renaming CML itself, which we are in the process of proving and do not yet have available.

The solution to this dilemma is to simply accept that we cannot yet align the contexts and instead establish uniqueness of type assignment *modulo partial context renaming*.

Lemma 3.8.4 (PTS Type Uniqueness up to Partial Context Renaming) Let ξ be a partial context renaming from Ψ to Ξ mediated by a predicate P, which covers all free variables of a term a. Then whenever a has type b under Ψ and $a[\xi]$ has type c under Ξ, we have $b[\xi] \equiv c$.

$$\mathsf{all}\,P\,a \implies \xi \Vdash_\lambda [P]\,\Psi \xrightarrow{r} \Xi \implies \Psi \vdash_\lambda a : b \implies \Xi \vdash_\lambda a[\xi] : c \implies b[\xi] \equiv c$$

Proof By induction on the derivation of $\Psi \vdash_\lambda a : b$, and in all but the conversion case, discriminating on $\Xi \vdash_\lambda a[\xi] : c$.

In the axiom case we use the functionality of $\mathcal{A}$.

For the variable case we know $P\, n$ and hence obtain two variable lookups from Ξ at $\xi\, n$, one by inversion and one through our renaming invariant. As lookup is functional the types are clearly convertible.

For the dependent function types we use the inductive hypotheses, the functionality of $\mathcal{R}$, the fact that convertible universes are equal, and Lemma 3.8.3 to deal with the context extension. The abstraction case is similar with the added complexity, that we need to establish the convertibility of two dependent function types. We do however know that their domains coincide, while their codomains are convertible. We use Lemma 2.4.31 to lift the componentwise congruence of $\succ$ to full congruence of $\equiv$. This allows us to obtain the convertibility of the two function types.

The application case also mostly follows from the inductive assumptions. In addition we have to perform the inverse operation from what we did in the abstraction case and decompose the convertibility of two function types into the convertibility of their domains and codomains, respectively, using Lemma 3.3.30.

The conversion case reduces to the transitivity of $\equiv$ as well as its compatibility with uniformly adding renamings (Lemma 3.3.10). ∎

While not technically relevant for our present proof of strengthening it is helpful to see how the standard notion of type uniqueness is simply a special case of the preceding result.

Theorem 3.8.5 (PTS Type Uniqueness) Under a given context Ψ, a term a has at most one type, up to conversion.

$$\Psi \vdash_\lambda a : b \implies \Psi \vdash_\lambda a : c \implies b \equiv c$$

Proof We start by observing that the identity renaming id is a (partial) context renaming from Ψ to Ψ for an arbitrary predicate P:

$$\mathsf{id} \Vdash_\lambda [P]\, \Psi \xrightarrow{r} \Psi.$$

We further recall that

$$\mathsf{all}\, (\lambda n \Rightarrow \top)\, a$$

holds for all a. Instantiating Lemma 3.8.4 with these yields the desired result. ∎

We now return to our proof of the partial renaming CML and consider a second major obstacle concerning the collection of *relevant* free variables. Above we have claimed that we care about the free variables of the term, the type and those in the reachable portion of the context. While technically correct for our final theorem, this leads to an inductive invariant that is too strong. For some of the inductive cases we do not have enough information to a priori know that P covers all free variables

of the *type*. Note however, that all types in derivable judgements are build up from universes or types from the context with the help of coverage-preserving constructors. Universes do not have free variables and those types extracted from the context are covered by P thanks to our assumption of context closure. We can therefore shift the coverage of the type from the assumptions to the conclusion. Note that we cannot fully drop it, as we need it to instantiate our uniqueness result.

In doing so, we have introduced our final complication, because we cannot satisfy this additional conclusion in the conversion case, as $\mathsf{all}\, P$ is not preserved under conversion. We do however know that $\mathsf{all}\, P$ is preserved under reduction (Theorem 3.4.6). We can thus further adjust our lemma statement to only obtain the final typing with a reduct of the type of the original typing.

Both modifications are present in the GN proof, where they are attributed to Luo and his work on the strong normalisation of an extend calculus of constructions (see [Luo94, p. 60, Lemma 3.17]). We adapt those ideas here to the de Bruijn setting, in particular with respect to the tracking of free variables, and arrive at the following carefully engineered statement.

Lemma 3.8.6 Let ξ be a partial context renaming from Ψ to Ξ mediated by P and let $\Psi \vdash_\lambda a : b$ be given. Let P cover the free variables of the term a, and let P further be closed under context extraction. Then ξ constructs a typing under Ξ, up to reduction in the type. That is we have the following.

$$\mathsf{val}\ \Psi \implies \mathsf{val}\ \Xi \implies \mathsf{all}\, P\, a \implies \mathsf{all}\, P\, \Psi \implies$$
$$\Psi \vdash_\lambda a : b \implies \xi \Vdash_\lambda [P]\, \Psi \xrightarrow{r} \Xi \implies \exists b'.\ b \succ^* b' \wedge \mathsf{all}\, P\, b \wedge \Xi \vdash_\lambda a[\xi] : b'[\xi]$$

Proof By induction on the derivation of $\Psi \vdash_\lambda a : b$. We consider each case carefully.

In the axiom case we have $b = \mathsf{s}$. We use $b' := \mathsf{s}$ as witness and easily obtain the desired properties.

In the variable case we have $b = c$ for some type c at position n in the context Ψ. We further know $P\, n$ and thus from our renaming invariant $\Xi \vdash_\mathsf{v} \xi\, n : c[\xi]$. With context closure we also know $\mathsf{all}\, P\, c$ and we have some sort s, such that $\Psi \vdash_\lambda c : \mathsf{s}$. By induction it therefore follows that $\Xi \vdash_\lambda c[\xi] : \mathsf{s}$ holds and hence $\Xi \vdash_\lambda n[\xi] : c[\xi]$ from the variable rule. Let $b' := c$ to close the case.

The case for dependent function types is straightforward since all occurring types are sorts and therefore coincide with their respective reducts. We have $b = \mathsf{s}$ for some sort s which we can use directly as a witness. The desired properties then follow directly from the inductive hypotheses and suitable context extension lemmas for the various invariants.

The application case is slightly more involved. We have $a = a_1\, a_2$ and $b = d[a_2 \cdot \mathsf{id}]$, with $\mathsf{all}\, P\, a_1$, $\mathsf{all}\, P\, a_2$, $\Psi \vdash_\lambda a_1 : \Pi c.\, d$ and $\Psi \vdash_\lambda a_2 : c$. By induction and Lemma 3.3.7 we have c', d' and c'' such that $c \succ^* c'$, $d \succ^* d'$ and $c \succ^* c''$, as well as $\mathsf{all}\, P\, c'$, $\mathsf{all}\, (\top \cdot P)\, d'$ and $\mathsf{all}\, P\, c''$, together with the typings $\Xi \vdash_\lambda a_1[\xi] : \Pi c'[\xi].\, d'[\Uparrow \xi]$ and $\Xi \vdash_\lambda a_2 : c''[\xi]$. Note that the domain types have diverged. Using confluence we join them at some $e[\xi]$ and with Lemma 2.4.29 and Theorem 3.7.7 we obtain $\Xi \vdash_\lambda a_1[\xi] : \Pi e[\xi].\, d'[\Uparrow \xi]$ and

$\Xi \vdash_\lambda a_2[\xi] : e[\xi]$. With the application rule we thus have $\Xi \vdash_\lambda a_1[\xi]\, a_2[\xi] : d'[\Uparrow\xi][a_2[\xi] \cdot \mathsf{id}]$, or equivalently $\Xi \vdash_\lambda (a_1\, a_2)[\xi] : d'[a_2 \cdot \mathsf{id}][\xi]$. Clearly $d'[a_2 \cdot \mathsf{id}]$ is a reduct of $d[a_2 \cdot \mathsf{id}]$ and therefore our witness. Lemma 3.4.5 yields $\mathsf{all}\, P\, d'[a_2 \cdot \mathsf{id}]$ and closes the case.

The abstraction case is the tricky one. We have $a = \lambda c.\, a'$ and $b = \Pi c.\, d$ with $\mathsf{all}\, P\, c$ and $\mathsf{all}\, (\top \cdot P)\, a'$, as well as $(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}$ and typings $\Psi \vdash_\lambda c : \mathsf{s}_1$, $\Psi, c \vdash_\lambda d : \mathsf{s}_2$ and $\Psi, c \vdash_\lambda a' : d$. For the first and third typing we obtain our regular inductive assumptions, using extension lemmas where needed, namely $\Xi \vdash_\lambda c[\xi] : \mathsf{s}_1$ and

$$\Xi, c[\xi] \vdash_\lambda a'[\Uparrow\xi] : d'[\Uparrow\xi] \tag{$\dagger$}$$

for some d' with $d \succ^* d'$ and $\mathsf{all}\, (\top \cdot P)\, d'$. Note that we do not know whether P covers all free variables of d, only that it covers its reduct d', hence we do not obtain any inductive assumptions for the second typing under Ψ. For now, we clearly get $\Pi c.\, d \succ^* \Pi c.\, d'$ and $\mathsf{all}\, P\, \Pi c.\, d'$ and hence use $b' \coloneqq \Pi c.\, d'$ as the witness. In order to close the case with an application of the abstraction rule we do, however, need to know that

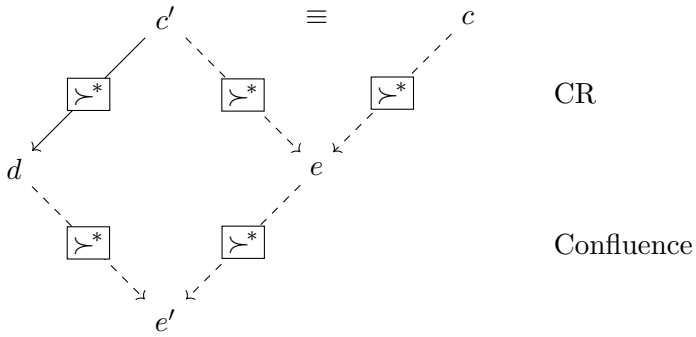
$$\Xi, c[\xi] \vdash_\lambda d'[\Uparrow\xi] : \mathsf{s}_2,$$

which we precisely did not obtain by induction. By subject reduction (Theorem 3.7.6) we do, however, know that $\Psi, c \vdash_\lambda d' : \mathsf{s}_2$. Next we consider propagation (Lemma 3.6.12) on ($\dagger$) and obtain two subcases:

- We either have $d'[\Uparrow\xi] = \mathsf{s}$ for some sort s, and hence $d' = \mathsf{s}$. We thus have $(\mathsf{s}, \mathsf{s}_2) \in \mathcal{A}$ and $\Xi, c[\xi] \vdash_\lambda \mathsf{s} : \mathsf{s}_2$ holds by the axiom rule, closing the case.
- Otherwise we know that $\Xi, c[\xi] \vdash_\lambda d'[\Uparrow\xi] : \mathsf{s}$ for some sort s. Now by uniqueness modulo partial renaming (Lemma 3.8.4) we obtain $\mathsf{s}_2[\Uparrow\xi] \equiv \mathsf{s}$, that is $\mathsf{s}_2 = \mathsf{s}$, and hence again $\Xi, c[\xi] \vdash_\lambda d'[\Uparrow\xi] : \mathsf{s}_2$, as required.

This closes the abstraction case.

Finally, in the conversion case we have $b = c$ and some c' such that $c' \equiv c$. By induction we have some d satisfying $c' \succ^* d$, $\mathsf{all}\, P\, d$ and $\Xi \vdash_\lambda a[\xi] : d[\xi]$, but we need these properties instead for a reduct of c. We cannot transport the variable coverage across the conversion, which is why we reason modulo reduction in the first place. We construct the following reduction diagram using the fact that $\succ$ is both confluent and CR (Theorem 3.3.21).



By transitivity we have $c \succ^* e'$ and by preservation under reduction we also get $\mathsf{all}\, P\, e'$. In addition, we further obtain $d[\xi] \succ^* e'[\xi]$ and therefore $\Xi \vdash_\lambda a[\xi] : e'[\xi]$ with Theorem 3.7.7. Thus $b' \coloneqq e'$ is the required witness. ∎

We are now in a position to clean up some of the structural complications of the preceding development and formulate the partial renaming CML that we are really after. And with that we can finally obtain strengthening, the main goal of this section.

Lemma 3.8.7 (PTS Partial Renaming CML) PTS typing is preserved under instantiation with partial context renamings.

$$\text{PTS-PRCML}\ \frac{\mathsf{val}\ \Psi \qquad \mathsf{val}\ \Xi \qquad \mathsf{all}\, P\, \Psi \qquad \mathsf{all}\, P\, a \qquad \mathsf{all}\, P\, b \qquad \Psi \vdash_\lambda a : b \qquad \xi \Vdash_\lambda [P]\, \Psi \xrightarrow{r} \Xi}{\Xi \vdash_\lambda a[\xi] : b[\xi]}$$

Proof From Lemma 3.8.6 and the given assumptions it follows, that there is some b' with $b \succ^* b'$ and $\Xi \vdash_\lambda a[\xi] : b'[\xi]$. With propagation (Lemma 3.6.12) it follows that either $b = \mathsf{s}$ or $\Psi \vdash_\lambda b : \mathsf{s}$ for some universe s. In the first case, we clearly also have $b' = \mathsf{s}$, since universes are normal, closing the proof.

Otherwise, we can use Lemma 3.8.6 again with the extra assumption that P also covers the type b and obtain some c satisfying $\mathsf{s} \succ^* c$ and $\Xi \vdash_\lambda b[\xi] : c[\xi]$. As before we infer that $c = \mathsf{s}$ since universes do not reduce, and hence have $\Xi \vdash_\lambda b[\xi] : \mathsf{s}$. We further note that reduction is contained in conversion, that is we have $b \equiv b'$, and thus, with Lemma 3.3.10, $b[\xi] \equiv b'[\xi]$. One application of the conversion rule now closes the proof. ∎

Note that if we instantiate P in Lemma 3.8.7 to the constant true predicate $(\lambda n \Rightarrow \top)$, then we almost recover our original renaming CML (Lemma 3.6.3), since in this case the two respective morphism conditions coincide, and the three instances of the all predicate all hold vacuously. The only, but crucial, differences that remain are the two validity assumptions on the initial and the final context. We recall that the presence of these assumptions arose from various invocations of propagation, as well as subject- and predicate-reduction, which each trace the requirement of validity back to the compatibility of typing with β-substitutions. All in all, the present construction seems to be yet another variation of the prevalent theme that a special case of a general theorem is needed in the proof of the general result.

When we recall that the renaming CML was a generalisation of weakening, we can now show that the *partial* renaming CML is additionally a generalisation of strengthening, since it is sufficiently instrumented to feed in the additionally required information about non-occurrence.

Theorem 3.8.8 (PTS Strengthening) Strengthening is admissible for every functional PTS.

$$\text{PTS-STR}\ \frac{\mathsf{val}\ \Psi, c \qquad \Psi, c \vdash_\lambda a[\uparrow] : b[\uparrow]}{\Psi \vdash_\lambda a : b}$$

Proof As a first step it is easy to verify that $\downarrow = 0 \cdot \mathsf{id}$ is a partial context renaming as long as we ignore the index 0 via the non-zero predicate NZ, that is we have

$$\downarrow \Vdash_\lambda [\mathsf{NZ}]\, \Psi, c \xrightarrow{r} \Psi.$$

Next, we recall Lemma 3.4.7 as well as Fact 3.5.4 and hence also have the following.

$$\mathsf{all}\,\mathsf{NZ}\, a[\uparrow] \qquad \mathsf{all}\,\mathsf{NZ}\, b[\uparrow] \qquad \mathsf{all}\,\mathsf{NZ}\, \Psi$$

Since we also clearly have $\mathsf{val}\ \Psi$ we can instantiate Lemma 3.8.7 and obtain

$$\Psi \vdash_\lambda a[\uparrow][\downarrow] : b[\uparrow][\downarrow] = \Psi \vdash_\lambda a[\uparrow \circ \downarrow] : b[\uparrow \circ \downarrow] = \Psi \vdash_\lambda a : b,$$

as required. ∎

3.9 The λ-Cube

One of the most prominent uses of the PTS framework is Barendregt's famous λ-cube introduced in [Bar91]. We want to briefly recap the basic notions here.

The purpose of the cube is the analysis of the fine-structure of the Calculus of Constructions (CC) introduced in [CH88], that is a careful study of how terms and types of CC are constructed, with a particular focus on the formation and population of function types. One key notion is that of dependencies among terms. The simplest form are terms depending on terms as represented by basic abstraction. In addition, one may add any number of the following:

- terms depending on types, that is **polymorphism**
- types depending on types, that is **type constructors**
- types depending on terms, that is **dependent types**

The cube identifies eight subsystems of CC, ordered by inclusion and based on which of these three additional dependencies are admitted. CC admits all three forms of dependencies.

With the PTS framework these ideas are captured by considering all $\mathcal{P} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ which satisfy the following.

- A fixed set of sorts $\mathcal{S} = \{*, \Box\}$, where $*$ is the designated universe of **types** and $\Box$ is the universe of **kinds**.
- A single axiom $*{:}\Box$, that is $\mathcal{A} = \{(*, \Box)\}$.
- For all rules $(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}$ we have $\mathsf{s}_2 = \mathsf{s}_3$ and hence write them as tuples $(\mathsf{s}_1, \mathsf{s}_2) \in \mathcal{R}$. We also require at least $(*, *) \in \mathcal{R}$.

Note that we do not explicitly express this particular PTS subclass in our formalisation, and therefore also do not formally prove the following claim. We do instead formally proof functionality separately for each concrete PTS instance that we consider later-on.

Fact 3.9.1 (Functionality of the λ-cube) Every PTS that satisfies the above constraints is in fact an FPTS, since there is only one axiom and the third component of every rule is determined by its second component.

The collection of all eight possible systems admissible under the constraints is usually tabulated as follows and arranged in a cube as depicted in Figure 3.5, where the arrow direction indicates system inclusion.

System	$\mathcal{R}$			
$\lambda_{\to}$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
$\lambda\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	
$\lambda\omega$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	
λP	$(*,*)$			$(*,\Box)$
$\lambda P2$	$(*,*)$	$(\Box,*)$		$(*,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(\Box,\Box)$	$(*,\Box)$
$\lambda P\omega = \lambda C$	$(*,*)$	$(\Box,*)$	$(\Box,\Box)$	$(*,\Box)$

Various corners of the cube correspond to well-known systems in the literature. The system $\lambda_{\to}$ is essentially the simply typed λ-calculus, while $\lambda 2$ corresponds to the polymorphic or second-order typed λ-calculus, also known as System F, of Girard [Gir72] and Reynolds [Rey74]. In a similar fashion, $\lambda\omega$ appears as System F_ω in [Gir72] and λP appears under the name LF in [HHP87]. And $\lambda P\omega = \lambda C$ is of course the Calculus of Constructions.

In [Bar91], Barendregt goes into a lot of detail and considers several further PTSs. His study includes, among other examples, another cube of PTS-encoded logics that mirrors the λ-cube along a propositions-as-types interpretation, which is cornerwise sound and at most corners also complete.

3.10 Discussion

The formalisation we have presented in this chapter is quite close to another Coq formalisation of the PTS metatheory by Adams [Ada04]. There are, however, a number of differences which we want to briefly discuss.

We have used plain inductive types to encode our de Bruijn syntax. Adams, on the other hand, works with so-called **well-scoped** de Bruijn syntax, which means that his expressions are taken from a $\mathbb{N}$-indexed inductive family of types. The index is used to build an upper bound on the free variables directly into the expression type. The trade-off among the two approaches amounts to the choice between flexibility on

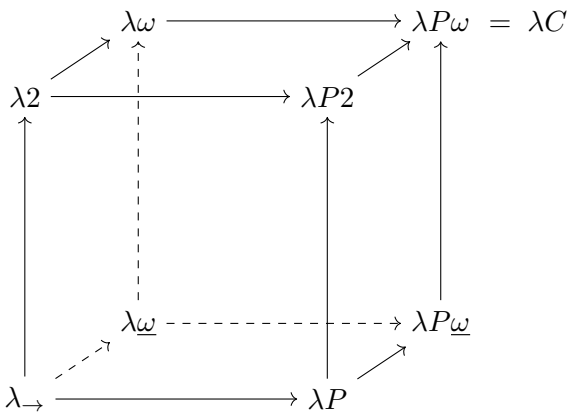


Figure 3.5: Barendregt's λ-cube.

the one hand and stronger support from the type system on the other. Note though that the well-scoped approach does not yield the flexibility of the all predicate, which we used in the strengthening result.

With respect to the treatment of strengthening there are also differences. Adams establishes the fully generic strengthening result of [vBJ93], and a significant portion of his development is dedicated solely to this proof, with the n-ary dependent function types causing most of the trouble. He additionally mentions in passing that his development also includes the simpler strengthening proof for FPTSs, but gives no further explanation. From the accompanying development it is difficult to extract his exact proof structure but he certainly does not use the notion of partial context renamings which we have introduced here and believe to be novel. Since the PTS metatheory plays a main, but not primary role, and since all the PTS instances we consider later on are functional, we have restricted ourselves to the simpler proof of strengthening from [GN91].

Finally, Adams stays close to the original literature and formulates the PTS type system with built-in weakening and validity, while we have decoupled these concerns. This allows us to isolate the situations where validity is needed from those, where it only causes unnecessary overhead. We believe that this helps in clarifying dependencies in the proof structure. The different approaches are further discussed in [Luo90]. In hindsight it might be possible to decouple the notions even further by not requiring well-sortedness in the variable rule PTS-VAR. This would lead to a more symmetric context morphism definition in Section 3.6, and thus drop the validity requirement from a number of lemmas. It is possible that this might even lead to a situation where we could prove the partial renaming CML for the FPTS case without resorting to the standard renaming CML first. We have, however, not investigated this avenue further.

The main contributions of this chapter are probably a clear exposition of the CML-based proof technique, the extension to partial context renamings and the

identification of the all predicate as the most natural way to handle free variables of de Bruijn terms also in the dependently typed setting. Also novel in this context is the guarded lifting of the all predicate to contexts as a closure operation that captures the reachable fragment of a given context in terms of used variables. The interplay of the two all predicates and partial context renaming was crucial for the adaptation of Luo's key lemma in the strengthening proof of [GN91] to the de Bruijn setting.

Finally, we would like to point out that the inclusion of the conversion rule into our PTS development was a significant extension, since it not only required the addition of the abstract reduction systems and several results particular to reduction itself, but also caused changes in several lemmas, where equality had to be changed to convertibility. To get an idea of the magnitude of differences it is interesting to compare the present work to the considerably shorter PTS metatheory used in [KTS17], where we omitted conversion and only looked at the particular PTS instance $\lambda 2$.

4 Simply Typed Lambda-Calculus

We now turn to our first correspondence proof, which is captured by the following claim: "The system $\lambda_{\to}$ is essentially the simply typed λ-calculus." How do we translate this into a formal statement? And how do we then prove it?

Before we can tackle these questions properly we should briefly recap the simply typed λ-calculus (STLC). Its term syntax is mostly that of the untyped λ-calculus (ULC), which we have seen in Chapter 2. On top of that we add a very minimal typing discipline, where types are either abstract base types $\mathsf{b} \in \mathcal{B}$ or are constructed from the single type constructor for function types, $A \to B$. That is, we have a 2-sorted system with separate syntactic sorts for terms and types. We also equip abstractions with a type annotation for their argument. The full, named system, including typing contexts and the typing judgement, is summarised in Figure 4.1. Note that the system also carries some logical content, since the Curry-Howard correspondence [How80] connects STLC to intuitionistic propositional logic. To visualise the connection, we can take the typing rules and only focus on the type level. This turns the abstraction rule into *implication-introduction* and the application rule into *implication-elimination*, better known as *modus ponens*.

Now, as stated in the introduction, our primary notion of system-correspondence is co-typeability. That is, a given typing judgement in one system is derivable if and only if there exists a corresponding, derivable judgement in the second system. This, of course, immediately raises the question of what exactly is meant by *corresponding judgement*? The approach that in our opinion works best is the inductive construction of syntactic correspondence relations for each semantic class of expressions, here terms and types. We say **semantic** here to emphasise that auxiliary information like typing derivations may be necessary to determine if a given **syntactic** expression belongs to one class or another. With respect to our correspondence relations, this means that they will relate syntactic expressions, but only those that are semantically meaningful. In the following chapters we will repeatedly come back to this distinction between the syntactic and the semantic level. It is of particular relevance for the PTS side of our correspondences where the semantic notions of types and terms are not syntactically separated (recall also Section 3.2).

Given such relations, say $\models_{\mathsf{S}} A \sim b$ and $\models_{\mathsf{S}} s \approx a$ for *semantic* types and terms respectively, we can formulate our correspondence result as follows.

$$\vdash_{\mathsf{S}}^{\mathsf{nmd}} s : A \iff \exists ab.\ \models_{\mathsf{S}} A \sim b \wedge \models_{\mathsf{S}} s \approx a \wedge \vdash_{\lambda}^{\mathsf{nmd}} a : b$$
$$\vdash_{\lambda}^{\mathsf{nmd}} a : b \iff \exists sA.\ \models_{\mathsf{S}} A \sim b \wedge \models_{\mathsf{S}} s \approx a \wedge \vdash_{\mathsf{S}}^{\mathsf{nmd}} s : A$$

$$\boxed{\mathsf{Ty}_{\mathsf{S}}^{\mathsf{nmd}}} \quad A, B ::= \mathsf{b} \mid A \to B \qquad \mathsf{b} \in \mathcal{B} \qquad \boxed{\mathsf{C}_{\mathsf{S}}^{\mathsf{nmd}}} \quad \Gamma ::= \bullet \mid \Gamma, x{:}A$$

$$\boxed{\mathsf{Tm}_{\mathsf{S}}^{\mathsf{nmd}}} \quad s, t ::= x \mid s\ t \mid \lambda x{:}A.\ s \qquad x \in \mathcal{V}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} x : A} \qquad \frac{\Gamma \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} s : A \to B \qquad \Gamma \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} t : A}{\Gamma \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} s\ t : B} \qquad \frac{\Gamma, x{:}A \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} s : B}{\Gamma \mathrel{|\!\!\sim^{\mathsf{nmd}}_{\mathsf{S}}} \lambda x{:}A.\ s : A \to B}\ x \notin \mathsf{dom}\ \Gamma$$

Figure 4.1: The simply typed λ-calculus.

We are at this point, of course, glossing over a few aspects that will be made precise over the course of this chapter, but the given statement should yield a rough idea of what we are after. It also allows us to give a high-level sketch of our proof strategy.

First of all we note that the result decomposes into four implications. To prove all of them, we have to ensure that our correspondence relations exhibit certain properties. The two implications from left to right claim the existence of related expressions, so we require $\sim$ and $\approx$ to be left-total and respectively right-total, as well as in each case judgement-preserving on a suitable subset of the involved syntactic sorts. Once we have these forward implications we can also use them to obtain the inverse implications. Closing the proofs then, however, relies on the uniqueness of the existentials, which translates into $\sim$ and $\approx$ being injective and, respectively, functional.

So in total we are interested in **four key properties** for each of the two correspondence relations.

- The relation is functional.
- The relation is injective.
- The relation is left-total and judgement-preserving.
- The relation is right-total and judgement-preserving.

Throughout this chapter we will carefully develop a correspondence result of this form. In order to do so, we will first present our de Bruijn encoding of STLC, with a focus on the treatment of base types, and then introduce some results that pertain to the particular PTS $\lambda_{\to}$, where the semantic classification of certain PTS terms as types will play a crucial role. For both systems we also have to discuss the notion of context internalisation. Next we precisely introduce our notion of syntactic correspondence, first on de Bruijn indices, then on types and finally on terms. Once all of these notions are defined, we proceed to formulate and establish the four key properties mentioned above for each of the relations. A major component here is the construction of invariants for the inductive preservation proofs. These take the form of context correspondence conditions, which we identify as *relational and inter-system generalisations of context morphisms*. The preservation proofs can accordingly be seen

as a generalisation of the CML proof technique. We close the chapter by putting all pieces together along the aforementioned top-level proof strategy and then demonstrate the usability of the result with a transfer of two metatheoretic properties from $\lambda_{\to}$ to STLC.

4.1 STLC

The defining aspect of STLC is its typing discipline, where the function types ascribed to abstractions are the main feature. Without base types, however, the language of types is empty, so we pose a finite set $\mathcal{B}$ of such types. In our de Bruijn encoding of STLC we are going to treat these base types as free type variables, represented again using numerical indices, similar to our implementation of the set of free variables $\mathcal{V}$.

The presence of type variables may feel arcane, since STLC can neither abstract over them nor sensibly instantiate them. The main advantages of this design are the ability to fold the set $\mathcal{B}$ into the typing judgement, in the form of a separate type variable context. This, in particular, allows us to elegantly scale the results for STLC to System F where type variables are playing a much more active role. The design also allows for a closer alignment with $\lambda_{\to}$ where base types necessarily appear as variables. Lastly, having the instantiation machinery, and in particular renaming, available for types will help us to get rid of certain spurious identity renamings.

Definition 4.1.1 (STLC Syntax) The de Bruijn syntax of types and terms of the simply typed λ-calculus is given by the following grammar.

$$\boxed{\mathsf{Ty_S}} \qquad A, B ::= n \mid A \to B \qquad n : \mathbb{N}$$

$$\boxed{\mathsf{Tm_S}} \qquad s, t ::= n \mid s\, t \mid \lambda A.\, s \qquad n : \mathbb{N}$$

The first thing to note about this syntax definition is that it is **two-sorted**, or stratified. That is, we have separate syntactic classes for terms and types, in contrast to the unified syntactic language of the PTS formalism (see Section 3.2). Due to this we also consider **two separate scopes of variables**, both represented by de Bruijn indices. In particular, when we make the variable constructors explicit, we observe that $\mathsf{V}_n : \mathsf{Ty_S}$ and $\mathsf{v}_n : \mathsf{Tm_S}$ are distinct syntactic entities, despite the fact that both utilise the index n. Since the kind of a given variable is usually clear from the context, we will, as before, mostly omit the variable constructor. In accordance with the fact that STLC cannot internally deal with type variables, it should be clear that abstractions $\lambda A.\, s$ only affect the term variable scope (variables of the form v_n), while variables of the form V_n are unaffected and denote the same type variable, below and above a given binder. As such, from an internal perspective, the V_n can be seen as a countably infinite set of constants, which capture our aforementioned treatment of base types.

Let us next consider instantiation for our two syntactic classes, which sheds further light on the roles of the two kinds of variables. As before we are going to work with

parallel substitutions and first consider the trivial case for type instantiation, which is defined as follows, for $\tau : \mathbb{N} \to \mathsf{Ty_S}$.

$$\begin{aligned} \mathsf{V}_n[\tau] &\coloneqq \tau\, n \\ (A \to B)[\tau] &\coloneqq A[\tau] \to B[\tau] \end{aligned}$$

Note that due to the absence of binders for type variables, we do not even have to employ the trick of first defining the machinery for renaming in order to break the mutual recursion with forward composition. The latter can easily and directly be defined as

$$(\tau \circ \tau')\, n \coloneqq (\tau\, n)[\tau'].$$

Instantiation for the term level with $\sigma : \mathbb{N} \to \mathsf{Tm_S}$ is a direct adaptation of the definition we gave for ULC in Chapter 2, which again exhibits the usual mutual recursion with forward composition:

$$\begin{aligned} \mathsf{v}_n[\sigma] &\coloneqq \sigma\, n & (\sigma \circ \sigma')\, n &\coloneqq (\sigma\, n)[\sigma'] \\ (s\, t)[\sigma] &\coloneqq s[\sigma]\, t[\sigma] & & \\ (\lambda A.\, s)[\sigma] &\coloneqq \lambda A.\, s[\Uparrow \sigma] & \Uparrow \sigma &\coloneqq \mathsf{v}_0 \cdot \sigma \circ \uparrow \end{aligned}$$

The abstraction case of this definition is interesting: we do not push the term substitution σ into the type annotation A, since the latter can only contain type variables. As this point, the astute reader might wonder about a third form of instantiation which could be of relevance, namely that of instantiating type variables in terms (which potentially exist due to the type annotations on abstractions). As it turns out, we never have to perform such an instantiation, at least for the STLC correspondence proof. We therefore postpone the treatment of such heterogeneous instantiations to our discussion of System F in Chapter 5.

The two forms of substitution that we have just laid out are not manually defined in the underlying formalisation but instead obtained from the Autosubst library, which additionally generates the corresponding equational theory.

Before we can turn our attention to the definition of typing, we need to briefly consider typing contexts. Since we have two variable scopes, it should come as no surprise that we will also consider two forms of contexts.

For term variables we adopt the usual de Bruijn view that typing contexts are simply lists of types, which are referenced by position. The fact that we are currently dealing with a stratified setting makes this rather simple. Term variable contexts Γ only contain types, and therefore only type variables, and hence do not reference back into themselves. We thus do not need the inductively defined notion of dependent context lookup that was introduced for the PTS setting ($\Psi \vdash_{\mathsf{V}} n : a$) and instead rely on plain positional indexing. The occurrence of type $A : \mathsf{Ty_S}$ at position n in term variable context Γ is expressed as

$$\Gamma_n = A.$$

In line with the usual presentation of typing contexts, we write extension to the right. That is, we have Γ, A for the context extension of Γ with a new type A at index 0.

In contrast to term variables, type variables do not come with any form of ascription. Instead, for the purpose of typing, all we care about is whether a particular type variable, that is base type, exists. In the named setting, the collection of base types, which are assumed to exists, was captured by the finite set $\mathcal{B}$. To reflect this in the de Bruijn setting, we simply number the base types sequentially, starting from 0. Due to this, a single natural number N, interpreted as an exclusive upper bound to the range of admissible type variables, can play the role of the set $\mathcal{B}$. Based on this, context lookup for V_k, reduces to the simple arithmetic check $k < N$. The case $N = 0$ encodes the empty type variable context.

The fact that a type could potentially contain non-existing type variables leads to the notion of a dedicated **type formation** judgement, which lifts the existence check for type variables to general types in the obvious way. Note that, more generally, type formation constitutes an integral part of type systems for stratified languages when the correctness of an employed type is not syntactically enforced.

With these preliminaries it should now be straightforward to grasp the following definition of STLC typing.

Definition 4.1.2 (STLC Typing) Let $N : \mathbb{N}$ be a de Bruijn type variable context, that is an exclusive upper bound on the admissible type variable indices, and let Γ be a de Bruijn term variable context, that is a list of types supporting positional lookup, then type formation, $N \vDash_{\mathsf{S}} A$, and typing, $N; \Gamma \vDash_{\mathsf{S}} s : A$, are inductively characterised by the rules of Figure 4.2.

While most rules are straightforward, it is worth to take a closer look at the abstraction rule:

$$\frac{N; \Gamma, A \vDash_{\mathsf{S}} s : B \qquad N \vDash_{\mathsf{S}} A}{N; \Gamma \vDash_{\mathsf{S}} \lambda A.\, s : A \to B}$$

Here we note that descending into the body of the abstraction extends the term variable context Γ with the type annotation A, while the type variable context N remains unchanged. This reflects the fact that the interpretation of de Bruijn encoded type variables is unaffected by term-level binding.

Since types now have the potential to be ill-formed, we again have to consider the notion of context validity. Here it simply states that all types $A \in \Gamma$ are well-formed under a given type variable context N, which we can inductively define as follows.

$$\frac{}{\mathsf{val}\ N; \bullet} \qquad \frac{\mathsf{val}\ N; \Gamma \qquad N \vDash_{\mathsf{S}} A}{\mathsf{val}\ N; \Gamma, A}$$

Interestingly, we need to know surprisingly little about the system we have just defined in order to establish its correspondence with $\lambda_{\to}$. We do not, in particular,

$$\frac{n < N}{N \models_{\mathsf{S}} n} \qquad \frac{N \models_{\mathsf{S}} A \qquad N \models_{\mathsf{S}} B}{N \models_{\mathsf{S}} A \to B} \qquad \frac{\Gamma_n = A \qquad N \models_{\mathsf{S}} A}{N; \Gamma \models_{\mathsf{S}} n : A}$$

$$\frac{N; \Gamma \models_{\mathsf{S}} s : A \to B \qquad N; \Gamma \models_{\mathsf{S}} t : A}{N; \Gamma \models_{\mathsf{S}} s\, t : B} \qquad \frac{N; \Gamma, A \models_{\mathsf{S}} s : B \qquad N \models_{\mathsf{S}} A}{N; \Gamma \models_{\mathsf{S}} \lambda A.\, s : A \to B}$$

Figure 4.2: The STLC de Bruijn type system.

need to give direct proofs of propagation or β-substitutivity, even though these would not be too involved. As indicated before, we are going to inherit these results from $\lambda_{\to}$ via the correspondence.[1]

The only point where some infrastructure is needed is the construction of a corresponding valid STLC context from a given valid $\lambda_{\to}$ context. The construction inductively traces the order of the PTS context, where (semantic) type and term variables are mixed. Thus it will be necessary to add a type variable after some term variables have already been introduced. Since the inductive definition of STLC validity assumes that all type variables are fixed a priori, a separate extension principle is in order. The required shifting result in turn follows from a renaming CML for STLC type formation. We briefly illustrate the corresponding proofs, but the employed proof structures are similar to those used for generic PTSs in Chapter 3. The proofs are notably simpler though, due to fewer cases in the inductive definitions, so we will not go into too much detail.

Context lookup for type variables is a simple arithmetic check, which is why the correct notion of a context renaming for type formation and the corresponding renaming CML take the following form. Note also that we do not require any form of lifting/extension lemmas, since there are no binders for type variables.

Definition 4.1.3 (STLC Context Renaming for Type Formation) A given renaming ξ is a context renaming from type variable context N to M, whenever it satisfies the following condition.

$$\xi \Vdash_{\mathsf{S}} N \xrightarrow{r} M \quad := \quad \forall n.\ n < N \implies \xi\, n < M$$

Lemma 4.1.4 (STLC Renaming CML for Type Formation) STLC type formation is preserved under instantiation with context renamings.

$$\text{STLC-RCML}\ \frac{N \models_{\mathsf{S}} A \qquad \xi \Vdash_{\mathsf{S}} N \xrightarrow{r} M}{M \models_{\mathsf{S}} A[\xi]}$$

Proof By induction on the derivation of $N \models_{\mathsf{S}} A$. ∎

[1] The accompanying development includes the direct proofs as well, since they illustrate the CML proof pattern in a setting without too many moving parts, but the subsequent proofs do not utilise them.

Lifting the result to context validity is now routine. Instantiation on term variable contexts, $\Gamma[\xi]$, is the pointwise lifting of type instantiation.

Lemma 4.1.5 (STLC Renaming CML for Context Validity) STLC context validity is preserved under instantiation with context renamings.

$$\frac{\mathsf{val}\ N; \Gamma \qquad \xi \Vdash_{\mathsf{S}} N \xrightarrow{r} M}{\mathsf{val}\ M; \Gamma[\xi]}$$

Proof By induction on the derivation of $\mathsf{val}\ N; \Gamma$, using Lemma 4.1.4 for context extension with a new term variable. ∎

Lemma 4.1.6 Validity of STLC contexts is preserved under the addition of a new type variable. That is the following rule is admissible.

$$\frac{\mathsf{val}\ N; \Gamma}{\mathsf{val}\ N+1; \Gamma[\uparrow]}$$

Proof Basic arithmetic yields $\uparrow \Vdash_{\mathsf{S}} N \xrightarrow{r} N+1$. Then instantiate Lemma 4.1.5. ∎

Lastly we observe that STLC can fully internalise its term variable context Γ, using abstractions and implications. This means that for every judgement $N; \Gamma \vdash_{\mathsf{S}} s : A$ there exists a judgement $N; \bullet \vdash_{\mathsf{S}} t : B$, such that they are equi-derivable. That is, either both are derivable or both are not derivable. The terms t and B are easily computable by recursion on Γ with

$$\begin{aligned} \mathsf{intern}\ \bullet\ s\ A &:= (s, A) \\ \mathsf{intern}\ (\Gamma, C)\ s\ A &:= \mathsf{intern}\ \Gamma\ (\lambda C.\ s)\ (C \to A) \end{aligned}$$

Lemma 4.1.7 (STLC Context Internalisation) The internalisation function intern yields equi-derivable judgements.

$$\begin{aligned} &\mathsf{val}\ N; \Gamma \implies \mathsf{intern}\ \Gamma\ s\ A = (t, B) \implies \\ &\qquad (N \vdash_{\mathsf{S}} A \iff N \vdash_{\mathsf{S}} B) \wedge (N; \Gamma \vdash_{\mathsf{S}} s : A \iff N; \bullet \vdash_{\mathsf{S}} t : B) \end{aligned}$$

Proof By induction on the derivation of $\mathsf{val}\ N; \Gamma$ and using the transitivity of $\iff$. The interesting case is

$$N; \Gamma, C \vdash_{\mathsf{S}} s : A \iff N; \Gamma \vdash_{\mathsf{S}} \lambda C.\ s : C \to A.$$

The forward direction is simply the abstraction rule, where $N \vdash_{\mathsf{S}} C$ is one of our inductive assumptions. For the inverse direction we simply discriminate on the derivation of $N; \Gamma \vdash_{\mathsf{S}} \lambda C.\ s : C \to A$. ∎

As we have seen, context internalisation depends on two ingredients, the internalisation function and a correctness proof which establishes equi-derivability. The function itself only relies on the presence of suitable syntactic abstraction mechanisms for the variables in the context and is usually relatively easy to define. For the correctness proof, two further aspects come into play, both of which are related to the equivalence we highlighted in the previous proof. The first concerns the forward use of the respective abstraction rule, which carries a side-condition on the well-formedness of the domain. While the side condition in the present scenario was trivial to satisfy, this is not always the case. The second concerns the invertibility of the abstraction rule, which again holds in the present scenario but is not always a given.

Consider for example the generic PTS development in Chapter 3, where both issues are problematic. For the forward direction, the formability of the respective dependent function type depends on the concrete PTS specification, while the invertibility is hindered by the presence of the conversion rule.

4.2 The PTS $\lambda_{\to}$

The system $\lambda_{\to}$ is the PTS that occupies the lowest corner of the λ-cube. It is obtained with the PTS specification $\mathcal{P}_{\to} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where the components are defined as follows.

$$\begin{array}{lcl} \mathcal{S} & \coloneqq & \{*, \Box\} \\ \mathcal{A} & \coloneqq & \{(*, \Box)\} \\ \mathcal{R} & \coloneqq & \{(*, *, *)\} \end{array}$$

It should be clear that the universe of kinds $\Box$ is devoid of any function types, since there are no rules in $\mathcal{R}$ admitting their formation. We further note that since there is no rule in $\mathcal{R}$ with $\Box$ as its first component, we will not be able to abstract over type variables, that is construct abstractions with $*$ as their domain. Somewhat more subtle is the fact that all dependent function types of $\lambda_{\to}$ are in fact vacuous, or non-dependent. The specification $\mathcal{P}_{\to}$ is such that a given dependent function type is only typeable in $\lambda_{\to}$, when its body does *not* refer to the bound variable, which hence renders the binding vacuous. We are not going to establish this explicitly, since our proof structures will implicitly take care of the aspect. We do give the following informal justification.

Assume we have a well-formed dependent function type $\Psi \vdash_{\lambda} \Pi a.\, b : \mathsf{s}$. Then by stripping we can infer that $\Psi \vdash_{\lambda} a : *$ and $\Psi, a \vdash_{\lambda} b : *$. We can now inductively argue that if 0 would occur in b, then necessarily $\Psi, a \vdash_{\lambda} 0 : *$ and therefore $a = *$. This however contradicts $\Psi \vdash_{\lambda} a : *$. The inductive argument that type variables do not occur in types is rather involved, since we have to consider the non-occurrence of type variables at arbitrary indices.

Let us now turn to those properties that we do establish formally. First of all, $\Box$ is the top universe and hence itself not typeable.

Fact 4.2.1 $\Psi \vdash_{\lambda} \Box : a \implies \bot$

Moreover, the universe $\Box$ is degenerate, since the universe $*$ is its only inhabitant.

Lemma 4.2.2 (Degeneracy of the Universe of Kinds $\Box$) The universe $*$ is the only kind under a valid context Ψ.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : \Box \implies a = *$$

Proof By discriminating on the derivation of $\Psi \vdash_{\lambda} a : \Box$. Most spurious cases are discharged with Fact 4.2.1. For the case where a is an application we use Lemma 3.6.14 to infer an impossible typing for $\Box$. ∎

Another frequently used fact ensures that the universe of types $*$ does not inhabit itself, which prevents certain logical inconsistencies.

Fact 4.2.3 $\Psi \vdash_{\lambda} * : * \implies \bot$

When we take another close look at our universes, we see that $\Box$ is only inhabited by $*$, while $*$ is only inhabited by variables and function types, where the domain and codomain are again taken from the universe $*$. All these inhabitants turn out to be normal.

Lemma 4.2.4 (Normality of $\lambda_{\to}$ Universes)

$$\mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : \mathsf{s} \implies \lfloor a \rfloor$$

Proof Special instance of Lemma 3.7.8. ∎

Lemma 4.2.5 (Normality of $\lambda_{\to}$ Types)

$$\mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : b \implies \lfloor b \rfloor$$

Proof Consequence of propagation and Lemma 4.2.4. ∎

The significance of this observation is that there are no non-trivial uses of the conversion rule in any given derivation of $\lambda_{\to}$. This suggests the usefulness of an auxiliary induction principle for typing of the PTS $\lambda_{\to}$, where the conversion rule is removed. We thus move one step towards the stratified presentation of STLC, where no concept of conversion was introduced in the first place. And since we are already considering the introduction of an alternate induction principle, we may as well also properly separate the notions of terms and types. Recall that these are semantic notions for a PTS. In particular, when Ψ is a valid context, then we say that a term a is a Ψ-type, whenever $\Psi \vdash_{\lambda} a : *$. Similarly, a term b is a Ψ-term, whenever there is a Ψ-type c such that $\Psi \vdash_{\lambda} b : c$ holds. We will later omit the explicit mention of the context when it is clear that we talk about the semantic, rather than the syntactic view on expressions.

We recall that STLC has two judgements, one for type formation and one for typing. To mirror this structure we introduce two auxiliary PTS induction principles, one for Ψ-types and another one for Ψ-terms. Note that we incorporate some inversion results already into these principles. This allows us to later avoid redoing the inversion steps whenever we use the custom induction principles.

Lemma 4.2.6 ($\lambda_\to$ Induction Principle for Type Formation.) Let Q be a property of PTS contexts and PTS terms. Then in order to prove that Q holds for all Ψ-types, it is sufficient to only consider type variables and function types, as these are the only possible constructions of $\lambda_\to$ types. This leads to the following admissible induction principle for $\lambda_\to$ types.

$$\begin{aligned}
&(\forall \Psi n.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\mathsf{V}} n : * \implies Q\,\Psi\,n) \implies && (\mathrm{H}_V)\\
&(\forall \Psi a b.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : * \implies \Psi, a \vdash_{\lambda} b : * \implies \\
&\qquad Q\,\Psi\,a \implies Q\,(\Psi, a)\,b \implies Q\,\Psi\,(\Pi a.\, b)) \implies && (\mathrm{H}_\to)\\
&(\forall \Psi a.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : * \implies Q\,\Psi\,a)
\end{aligned}$$

Proof Assume H_V and $\mathrm{H}_\to$. Then by induction on the derivation of $\Psi \vdash_\lambda a : *$. The axiom and the abstraction case are easy to discharge. For the application case we can use Lemma 3.6.14 to construct the impossible typing $\Psi \vdash_\lambda * : *$ and are again done. In the conversion case, we obtain a type $b \equiv *$, but from Lemma 4.2.5 it follows that b is normal and hence $b = *$. This allows us to instantiate the inductive hypothesis and close this case as well.

We are left with the two actual cases. The variable case is immediate from H_V. For the function type case we observe that $(*, *, *)$ is the only rule in $\mathcal{R}$. This allows us to easily instantiate the inductive hypotheses and then close the case with $\mathrm{H}_\to$. ■

Lemma 4.2.7 ($\lambda_\to$ Induction Principle for Typing.) Let Q be a property of PTS contexts and two PTS terms. Then in order to prove that Q holds for all Ψ-terms and their corresponding Ψ-types, it is sufficient to only consider term variables, abstractions and applications, as these are the only possible constructions of $\lambda_\to$ terms. This leads to the following admissible induction principle for $\lambda_\to$ terms.

$$\begin{aligned}
&(\forall \Psi n a.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\mathsf{V}} n : a \implies \Psi \vdash_{\lambda} a : * \implies Q\,\Psi\,n\,a) \implies && (\mathrm{H}_V)\\
&(\forall \Psi a b c d d'.\ \mathsf{val}\ \Psi \implies \\
&\qquad \Psi \vdash_{\lambda} a : \Pi c.\, d \implies \Psi \vdash_{\lambda} c : * \implies \Psi, c \vdash_{\lambda} d : * \implies \Psi \vdash_{\lambda} b : c \implies \\
&\qquad Q\,\Psi\,a\,(\Pi c.\, d) \implies Q\,\Psi\,b\,c \implies d' = d[b \cdot \mathsf{id}] \implies Q\,\Psi\,(a\,b)\,d') \implies && (\mathrm{H}_A)\\
&(\forall \Psi a b c.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : * \implies \Psi, a \vdash_{\lambda} c : * \implies \Psi, a \vdash_{\lambda} b : c \implies \\
&\qquad Q\,(\Psi, a)\,b\,c \implies Q\,\Psi\,(\lambda a.\, b)\,(\Pi a.\, c)) \implies && (\mathrm{H}_\lambda)\\
&(\forall \Psi a b.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} b : * \implies \Psi \vdash_{\lambda} a : b \implies Q\,\Psi\,a\,b)
\end{aligned}$$

Proof Assume H_V, H_A and H_λ. Then by induction on the derivation of $\Psi \vdash_\lambda a : b$.

The axiom case is easy to discharge, since there is no axiom $(\mathsf{s}_1, \mathsf{s}_2)$ with $* = \mathsf{s}_2$. For the same reason, we can easily discard the case for function type formation. The conversion case is handled as before, but we have to force a conversion $b \equiv c$ down to an equality, which requires both Lemma 4.2.4 and Lemma 4.2.5.

We are left with the three actual cases. The variable case is again immediate from H_V. The application case is of course handled with H_A, where all premises follow from one use of Lemma 3.6.13 and the inductive hypotheses. Lastly for the abstraction case we again observe that $(*, *, *)$ is the only rule in $\mathcal{R}$. This allows us to instantiate all inductive hypotheses and then close the case with H_λ. ∎

We recall that $\lambda_{\to}$ is a functional PTS and therefore also obtain the respective strengthening result (Theorem 3.8.8). We further observe that universes are invariant under renaming, so we can easily replace an occurrence of s with $\mathsf{s}[\uparrow]$. This allows us to establish the following fact as a slight variation of the generic strengthening result.

Fact 4.2.8 (Strengthening for $\lambda_{\to}$ Type Formation)

$$\mathsf{val}\ \Psi, a \implies \Psi, a \vdash_\lambda b[\uparrow] : * \implies \Psi \vdash_\lambda b : *$$

In Section 4.1 we discussed to what extent STLC is capable of internalising its context. As it turns out, a similar result is surprisingly tricky to obtain for $\lambda_{\to}$. The main problem here is the fact that type and term variables are mixed arbitrarily in a given PTS context Ψ. And while all PTSs are syntactically equipped to fully internalise their contexts this does not ensure co-derivability of the judgements before and after internalisation. In particular, the process constructs abstractions and corresponding function types that may not be admitted by the available rules in $\mathcal{R}$. Our PTS $\lambda_{\to}$, for example, does not admit the abstraction over a type variable.

Thus any attempt to internalise an arbitrary context Ψ would first involve reordering it into the following shape where all a_i are types (i.e. $a_i \neq *$).

$$\Psi' = \underbrace{*, \ldots, *}_{N}, a_{k-1}, \ldots, a_0$$

Now in theory, this should always be possible, but recall that de Bruijn judgements are not stable under context reorderings. Hence we would need to find a context renaming $\xi \Vdash_\lambda \Psi \xrightarrow{r} \Psi'$ which, moreover, is a permutation and therefore invertible.

We could then construct an internalisation function that only consumes the right half of such a structured context and leaves behind what could feasibly be considered as an "empty" context for $\lambda_{\to}$:

$$\Psi'' = \underbrace{*, \ldots, *}_{N}$$

This notion of emptiness is in fact useful and we adopt it below when we state our correspondence results for the special case where the contexts are empty, but we refrain from actually constructing the internalisation, and particularly the initial reordering, for one simple reason. The construction and the associated correctness

proof are rather difficult, while the payoff is surprisingly small. This is due to the fact that (a) the correspondence statements for closed contexts are a lot less useful than their generalisations to open judgements (which we have to prove anyway) and (b) we can show that valid contexts of one system always have a suitable counterpart in the respective other system, where suitable means that all required proof invariants hold (we will make this precise below). In combination we conclude that all we really gain from closing the judgements are theorem statements that are slightly less cluttered.

We thus conclude our exposition of $\lambda_{\to}$ by observing a few properties of "empty" $\lambda_{\to}$ contexts.

Definition 4.2.9 PTS contexts which exclusively consist of N type variables, written T_N, are recursively defined as follows.

$$\begin{aligned} \mathsf{T}_0 &:= \bullet \\ \mathsf{T}_{(1+N)} &:= \mathsf{T}_N, * \end{aligned}$$

Fact 4.2.10 $\mathsf{val}\ \mathsf{T}_N$

Fact 4.2.11 $\mathsf{T}_N \vdash_{\forall} n : a \implies a = *$

The context T_N is of course designed to mirror the de Bruijn type variable context N of our STLC judgements.

4.3 Relating de Bruijn Indices

Our inductive correspondence relations necessarily involve open terms and types. And similar to the need for typing contexts in the definition of the typing judgements we now also have to track some contextual information for our correspondence result. This time, however, we are not primarily interested in the types of the free variables. We instead would like to know which free variables in one system correspond to which free variables in another. For the remainder of this section, let us refer to the two systems abstractly as $\mathsf{S_L}$ and $\mathsf{S_R}$ for the systems that are placed on the left and the right side of the correspondence relations respectively. Since both systems are considered in their de Bruijn presentation, this amounts to the tracking of pairs of related indices.

Definition 4.3.1 (Relational Context) A relational context Θ is a sequence of pairs of de Bruijn indices, which can be visually represented as follows.

$$\Theta\colon \quad \begin{array}{ccccccc} n_1 & n_2 & n_3 & \cdots & n_k \\ | & | & | & & | \\ m_1 & m_2 & m_3 & \cdots & m_k \end{array}$$

The first components, n_i, of each pair $p \in \Theta$ are taken from $\mathsf{S_L}$, the system to the left of the correspondence relations, while the second components, m_i, are taken from $\mathsf{S_R}$ on the right of the correspondences. Paired indices encode related free variables of the respective systems. We write

$$\Theta \vdash n \simeq m$$

for the lookup operation in relational contexts, that is to express that, according to Θ, index n in $\mathsf{S_L}$ and index m in $\mathsf{S_R}$ encode related variables. We again denote empty contexts by $\bullet$.

In the formalisation, we represent relational contexts as values of type $\mathsf{list}(\mathsf{var} \times \mathsf{var})$ and implement the lookup predicate simply as list membership.

Our first challenge now is the question of variable scopes. Whenever we look at two systems with variable binding, we certainly may have multiple variable scopes, at least semantically. At the semantic level, we also expect the same set of scopes on both sides of our correspondence relation. At the syntactic level, however, the collection of variable scopes may not match up. We are here for example faced with the difficulty that a PTS only knows a single syntactic scope of variables, while $\lambda_{\to}$ clearly supports a distinction of term and type variables at the semantic level. We now have two options to handle this.

We could track indices at the syntactic level. We are then however faced with the problem that index n is available in each scope. It is thus possible that index n in the system with multiple scopes may be related to multiple indices m_i in the system with a single scope. The choice of the correct mapping, when it comes to relating variables, then crucially depends on auxiliary information to disambiguate.

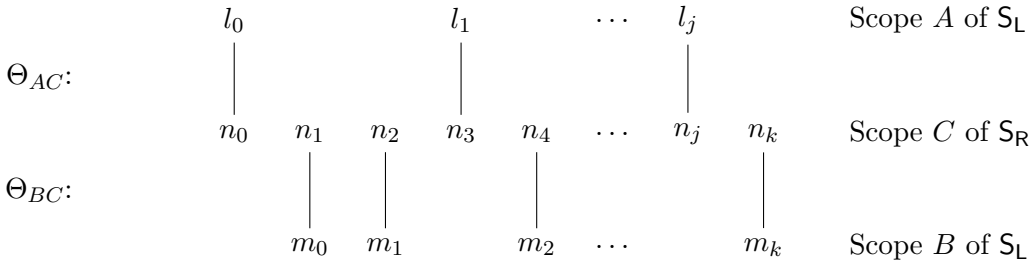
Alternatively, we could track indices at the semantic level, meaning that we have one context of related indices *per semantic variable scope*, e.g. one relational context for type variables and a second relational context for term variables. The cost of maintaining multiple contexts is easily offset by two factors. First, lookup is easy since all mappings in a given context are of the same kind and no auxiliary information for disambiguation is required. Second, when it comes to extending these contexts it is often easy to determine which context obtains a new mapping since binders allow for easy classification into one semantic variable scope or another, either directly by nature of their syntax or through attached typing information.

Here we adopt the second, semantic approach and now turn to the issue of context extension. We are, in particular, concerned with the traversal of binders, since it adds new variables to the scope and affects the interpretation of indices already in scope. To grasp the following, it is important to keep in mind that a given pair $p \in \Theta$ should always represent a fixed free variable, while the components of p yield the current de Bruijn encodings of this variable in $\mathsf{S_L}$ and $\mathsf{S_R}$, respectively.

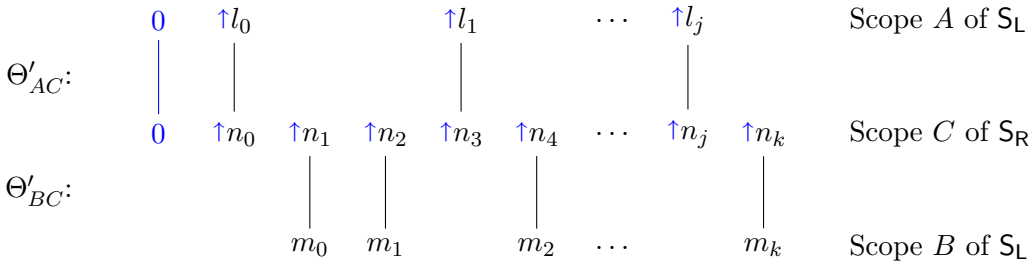
Let us consider the scenario where $\mathsf{S_L}$ is equipped with two syntactic variable scopes A and B and two corresponding de Bruijn binders[2] $\lambda_A.\, s$ and $\lambda_B.\, s$. $\mathsf{S_R}$, on

[2] Note that we presently do not care about any ascriptions on the bound variables and have hence elided them from the chosen notation.

the other hand, only comes with a single syntactic scope C and corresponding binder $\lambda_C.\, s$. We are thus tracking two semantic variable scopes with two relational contexts Θ_{AC} and Θ_{BC}. At any given point in our derivation of correspondence, our contextual information may look as follows, where we have vertically flipped the depiction of Θ_{BC} to keep the single scope C of $\mathsf{S_R}$ in the centre.



Now what happens when we descend underneath binders in both systems, say $\lambda_A.\, s$ in $\mathsf{S_L}$ and $\lambda_C.\, s'$ in $\mathsf{S_R}$? With respect to Θ_{AC} the situation is very similar to what happens when a parallel substitution is pushed underneath a binder. That is, we have a new free variable represented by index 0, both in scope A and scope C, and all other encodings in these scopes have to be shifted to acknowledge the additional binder. Let us add these modifications to our diagram to see how Θ_{BC} will be affected.



Note that Θ_{BC} does not obtain a new pair of indices since there is no new free variable of the corresponding semantic scope. The interpretation of scope C has changed, however, so in order to maintain the invariant that each pair $p \in \Theta_{BC}$ represents a fixed variable we have to incorporate this change via a suitable adjustment to Θ_{BC}. In particular, we need a form of one-sided shifting of pair components.

We further observe that the situation would have been symmetric if we were to instead traverse binder $\lambda_B.\, s$ in $\mathsf{S_L}$ and again $\lambda_C.\, s'$ in $\mathsf{S_R}$.

Based on these considerations we now have to define two operations, the **extension** of the context with a new variable, written $\Theta^{\Uparrow}$, and the one-sided shifting, or **skewing**, of all other contexts that have a syntactic scope overlap with the main affected context, expressed as $\Theta^{\uparrow}$. We use list operations like map and cons to define our two operations as follows.

$$\Theta^{\Uparrow} \coloneqq (0,0) :: \mathsf{map}\,(\uparrow \times \uparrow)\,\Theta \qquad \text{(extension)}$$

$$\Theta^{\uparrow} \coloneqq \mathsf{map}\,(\mathsf{id} \times \uparrow)\,\Theta \qquad \text{(right skewing)}$$

Here $\mathsf{map}\,(\xi \times \zeta)\,\Theta$ denotes a componentwise map, that is the operation that applies ξ to all first projections and ζ to all second projections of pairs $p \in \Theta$. Note that $\Theta^{\uparrow}$ is an asymmetric skewing of the scope of $\mathsf{S_R}$. Throughout this work we will always place the PTS variant of a system in question on the right, and therefore also the unified variable scope. Hence we do not need a symmetric skewing operation on the left, where the syntactic scopes, in our case, will always match the semantic scopes.

We lift a number of facts about lists, map and lookup to our componentwise map. All results are straightforward. Note that in Fact 4.3.4 we again use $\circ$ to denote forward composition.

Fact 4.3.2 $\Theta \vdash n \simeq m \implies \mathsf{map}\,(\xi \times \zeta)\,\Theta \vdash \xi\, n \simeq \zeta\, m$

Fact 4.3.3 $\mathsf{map}\,(\xi \times \zeta)\,\Theta \vdash n \simeq m \implies \exists n' m'.n = \xi\, n' \wedge m = \zeta\, m' \wedge \Theta \vdash n' \simeq m'$

Fact 4.3.4 $\mathsf{map}\,(\xi_2 \times \zeta_2)\,(\mathsf{map}\,(\xi_1 \times \zeta_1)\,\Theta) = \mathsf{map}\,(\xi_1 \circ \xi_2 \times \zeta_1 \circ \zeta_2)\,\Theta$

Lists support a natural notion of containment, $\Theta_1 \sqsubseteq \Theta_2$, which for our definitions is captured by the following.

Fact 4.3.5 $\Theta_1 \sqsubseteq \Theta_2 \implies \Theta_1 \vdash n \simeq m \implies \Theta_2 \vdash n \simeq m$

Clearly, mapping over lists preserves containment. Thus our componentwise map and subsequently the extension operation $\Theta^{\Uparrow}$ are monotone with respect to list containment.

Lemma 4.3.6 $\Theta_1 \sqsubseteq \Theta_2 \implies \mathsf{map}\,(\xi \times \zeta)\,\Theta_1 \sqsubseteq \mathsf{map}\,(\xi \times \zeta)\,\Theta_2$

Proof Consequence of Facts 4.3.2 and 4.3.3. ∎

Lemma 4.3.7 $\Theta_1 \sqsubseteq \Theta_2 \implies \Theta_1^{\Uparrow} \sqsubseteq \Theta_2^{\Uparrow}$

Proof The contexts $\Theta_1^{\Uparrow}$ and $\Theta_2^{\Uparrow}$ both contain the pair $(0, 0)$, while the remainders have the form $\mathsf{map}\,(\uparrow \times \uparrow)\,\Theta_1$ and respectively $\mathsf{map}\,(\uparrow \times \uparrow)\,\Theta_2$. The former is included in the latter due to Lemma 4.3.6. ∎

We can combine the preceding results to obtain a number of useful facts about our skewing and extension operations. These include simple forward facts, inversion principles and also the preservation of functionality and injectivity. All of these will be relevant in the later development.

Lemma 4.3.8 Inversion principles for context extension operations.

$$\Theta^{\uparrow} \vdash n \simeq m \implies \exists m'.\ m = \uparrow m' \wedge \Theta \vdash n \simeq m' \tag{i}$$

$$\Theta^{\Uparrow} \vdash n \simeq m \implies (n = 0 \wedge m = 0) \vee \exists n' m'.\ n = \uparrow n' \wedge m = \uparrow m' \wedge \Theta \vdash n' \simeq m' \tag{ii}$$

Proof Trivial. ∎

Lemma 4.3.9 The following extension principles hold.

$$\Theta \vdash n \simeq m \implies \Theta^{\uparrow} \vdash n \simeq \uparrow m \tag{i}$$

$$\Theta^{\Uparrow} \vdash 0 \simeq 0 \tag{ii}$$

$$\Theta \vdash n \simeq m \implies \Theta^{\Uparrow} \vdash \uparrow n \simeq \uparrow m \tag{iii}$$

Proof Trivial. ∎

Lemma 4.3.10 Extension and mapping commute according to the following laws.

$$(\mathsf{map}\,(\xi \times \zeta)\,\Theta)^{\uparrow} = \mathsf{map}\,(\xi \times \Uparrow\zeta)\,\Theta^{\uparrow} \tag{i}$$

$$(\mathsf{map}\,(\xi \times \zeta)\,\Theta)^{\Uparrow} = \mathsf{map}\,(\Uparrow\xi \times \Uparrow\zeta)\,\Theta^{\Uparrow} \tag{ii}$$

Proof Trivial using composition of mappings. ∎

Fact 4.3.11 Let Θ be a functional (respectively injective) relation on de Bruijn indices. Then both $\Theta^{\uparrow}$ and $\Theta^{\Uparrow}$ are functional (respectively injective).

Recall that our relational contexts capture semantically disjoint scopes of variables. When $\mathsf{S_L}$ is equipped with two syntactic scopes, while $\mathsf{S_R}$ has only one, then we do of course consider two relational contexts. For this setup to make sense, all variables that are in scope in $\mathsf{S_R}$ should correspond to exactly one variable in one of the two scopes of $\mathsf{S_L}$, and particularly not to something in both. To capture this idea formally we introduce the notion of **range-disjoint** relational contexts.

Definition 4.3.12 Two contexts Θ_1 and Θ_2 are range-disjoint, written $\Theta_1 \parallel \Theta_2$, whenever they satisfy the following.

$$\Theta_1 \parallel \Theta_2 := \forall nn'm.\ \Theta_1 \vdash n \simeq m \implies \Theta_2 \vdash n' \simeq m \implies \bot$$

This property is symmetric and we observe that the empty context is always range-disjoint with an arbitrary context. One crucial result for our subsequent development is the way, range-disjointness is preserved under context modifications.

Lemma 4.3.13 Let Θ_1 and Θ_2 be two relational contexts which are range-disjoint. When we extend one of them and skew the other, the two resulting contexts are still range-disjoint. Formally we have

$$\Theta_1 \parallel \Theta_2 \implies \Theta_1^{\Uparrow} \parallel \Theta_2^{\uparrow}$$

and by symmetry also $\Theta_1^{\uparrow} \parallel \Theta_2^{\Uparrow}$.

Proof By case analysis on $\Theta_1^{\Uparrow} \vdash n \simeq m$ and $\Theta_2^{\uparrow} \vdash n' \simeq m$. Clearly $m \neq 0$ due to skewing. But then there are n'',m' such that $\Theta_1 \vdash n'' \simeq m'$ and $\Theta_2 \vdash n' \simeq m'$, which contradicts our assumption. ∎

Note that this result will only be relevant for the System F development in Chapter 5 where we have to lift the range-disjointness from relational contexts to the relations that place the terms, and respectively types, in correspondence. There we establish that a PTS term may either be related to a PLC type or a PLC term, but certainly not both. That is, the term and type relations are going to be range-disjoint, provided their contextual assumptions are. For the present development for STLC this issue does not come into play.

As mentioned above, we use free type variables to abstractly represent STLC base types. Also recall that STLC is not capable of internalising these type variables. Our final result of this chapter will therefore assume the correspondence of typings under n a priori fixed abstract base types. Here we adopt the initial setup that any given base type is encoded by the respective *same* de Bruijn index in both systems. We express this with an identity relation of size n as the initial relational type variable context. This special identity context is constructed as the n-fold iteration of context extension.

$$\mathsf{id}_{\mathsf{R}}\, 0 \coloneqq \bullet$$
$$\mathsf{id}_{\mathsf{R}}\, 1 + n \coloneqq (\mathsf{id}_{\mathsf{R}}\, n)^{\Uparrow}$$

A straightforward induction on n yields the fact that $\mathsf{id}_{\mathsf{R}}\, n$ is both functional and injective.

4.4 Proof of Correspondence

We now have all the ingredients to tackle our first relational correspondence proof. The main idea is to place each well-behaved fragment of the expressions of one system in bijective correspondence with a well-behaved fragment of the expressions of the other system. Concretely, this means that we want to relate the well-formed STLC types with the Ψ-types of $\lambda_{\to}$, and the well-typed STLC terms with the Ψ-terms, assuming the respective contexts are related. The relations are build up inductively, and follow the structure of the involved type systems.

Definition 4.4.1 (STLC Correspondence Relation for Types) Let Θ be a relational type variable context. Then an STLC type A and a (semantic) $\lambda_{\to}$ type a are related whenever $\Theta \models_{\mathsf{S}} A \sim a$ is derivable from the rules in Figure 4.3.

Definition 4.4.2 (STLC Correspondence Relation for Terms) Let Θ and Σ be relational type and term variable contexts respectively. Then an STLC term s and a (semantic) $\lambda_{\to}$ term b are related whenever $\Theta; \Sigma \models_{\mathsf{S}} s \approx b$ is derivable from the rules in Figure 4.3.

There are a few things that are worth noting about these definitions. First of all, the relational type variable context Θ, that is the contextual information about type variables is only skewed, while Σ is only extended. This reflects the situation that there are two scopes of variables, but terms can only abstract over one of them,

$$\frac{\Theta \vdash n \simeq m}{\Theta \models_{\mathsf{S}} n \sim m} \qquad \frac{\Theta \models_{\mathsf{S}} A \sim a \qquad \Theta^{\uparrow} \models_{\mathsf{S}} B \sim b}{\Theta \models_{\mathsf{S}} A \to B \sim \Pi a.\, b}$$

$$\frac{\Sigma \vdash n \simeq m}{\Theta; \Sigma \models_{\mathsf{S}} n \approx m} \qquad \frac{\Theta; \Sigma \models_{\mathsf{S}} s \approx a \qquad \Theta; \Sigma \models_{\mathsf{S}} t \approx b}{\Theta; \Sigma \models_{\mathsf{S}} s\, t \approx a\, b} \qquad \frac{\Theta \models_{\mathsf{S}} A \sim a \qquad \Theta^{\uparrow}; \Sigma^{\Uparrow} \models_{\mathsf{S}} s \approx b}{\Theta; \Sigma \models_{\mathsf{S}} \lambda A.\, s \approx \lambda a.\, b}$$

Figure 4.3: Inductive correspondence relation for STLC.

namely the term variables. Secondly, the two relations separate the semantic notions of types and terms of $\lambda_{\to}$. Note here that for any derivable judgement $\Theta \models_{\mathsf{S}} A \sim a$, the term a can only take the form of a variable or a dependent function type. The term relation is similarly restricted to semantic $\lambda_{\to}$ terms on the PTS side.

It is also worth pointing out that terms (here in particular, abstractions) carry type annotations, and therefore the definition of the term relation internally depends on the type relation. This dependency extends to the subsequent developments where we often find that certain properties of the term relation depend on the fact that similar results have already been established for the type level.

Based on these considerations we now first handle the type relation $\sim$ and establish that it satisfies all required properties: injectivity, functionality, left-totality and judgement-preservation, as well as right-totality and judgement-preservation. We will then use these results to obtain the same set of properties for the term relation $\approx$.

4.4.1 Four Properties for $\sim$

Of the four properties, functionality and injectivity are the easier ones and hence a good starting point to gain some intuition with respect to our relations.

Lemma 4.4.3 (Functionality of $\Theta \models_{\mathsf{S}} A \sim a$) The type relation is functional, provided that Θ is a functional relation.

$$\Theta\ \mathsf{func} \implies \Theta \models_{\mathsf{S}} A \sim a_1 \implies \Theta \models_{\mathsf{S}} A \sim a_2 \implies a_1 = a_2$$

Proof By induction on the derivation of $\Theta \models_{\mathsf{S}} A \sim a_1$ and discriminating on the derivation of $\Theta \models_{\mathsf{S}} A \sim a_2$. In the variable case we use the functionality of Θ. The case for function types follows from the inductive hypotheses and the preservation of functionality under skewing (Fact 4.3.11). ∎

Lemma 4.4.4 (Injectivity of $\Theta \models_{\mathsf{S}} A \sim a$) The type relation is injective, provided that Θ is an injective relation.

$$\Theta\ \mathsf{inj} \implies \Theta \models_{\mathsf{S}} A_1 \sim a \implies \Theta \models_{\mathsf{S}} A_2 \sim a \implies A_1 = A_2$$

Proof By induction on the derivation of $\Theta \models_{\mathsf{S}} A_1 \sim a$ and discriminating on the derivation of $\Theta \models_{\mathsf{S}} A_2 \sim a$. In the variable case we use the injectivity of Θ. The case for

function types follows from the inductive hypotheses and the preservation of injectivity under skewing (Fact 4.3.11). ∎

Our next goal is to prove the two totality and preservation results. Loosely speaking they state that for every well-formed type of one system there exists a well-formed type in the other, given suitably related contexts.

There are two crucial aspects here. The first is the restriction of totality to well-formed types, as these in a sense are the only *real* types we care about. In particular for the direction from $\lambda_{\to}$ to STLC, this restriction is essential. The second is the requirement of "suitably related contexts". As it turns out this phrase encompasses a proof invariant that is reminiscent of the context morphism conditions we have encountered in Section 3.6. This is not very surprising as our theorem states that one judgement will entail another. In contrast to the CMLs we have seen so far, however, the two judgements now belong to two separate systems with distinct syntactic languages. Interestingly though, the same proof techniques that worked for CMLs still apply when suitably generalised. That is, we first define our invariant, then show that it is preserved under certain context extensions and finally state and prove the associated CML with a basic induction that uses the invariant extension principles for the binder cases. The four invariants which we will introduce throughout this section are summarised in Figure 4.4 for reference and comparison.

We start with the left-totality and preservation of type formation, that is the direction from STLC to $\lambda_{\to}$. Observe how the type variables of $\lambda_{\to}$ are captured with the condition $\Psi \vdash_{\mathsf{V}} m : *$.

Definition 4.4.5 (Type Formation Invariant: STLC to $\lambda_{\to}$) A $\lambda_{\to}$ context Ψ is suitably related to a given STLC type variable context N as mediated by a relational type variable context Θ, whenever the following property is satisfied.

$$\Theta \Vdash_{\mathsf{S}} N \to \Psi \quad :\Longleftrightarrow \quad \forall n < N.\ \exists m.\ \Theta \vdash n \simeq m \wedge \Psi \vdash_{\mathsf{V}} m : *$$

Lemma 4.4.6 (Skewing law) Skewing of the relational context Θ preserves the invariant under context extension with a new term variable.

$$\Theta \Vdash_{\mathsf{S}} N \to \Psi \implies \Theta^{\uparrow} \Vdash_{\mathsf{S}} N \to \Psi, a$$

Proof Let $n < N$ be an STLC type variable index and m the corresponding index in Ψ, then $\uparrow m$ is the corresponding index in Ψ, a. Clearly $\Psi, a \vdash_{\mathsf{V}} \uparrow m : *$ follows from $\Psi \vdash_{\mathsf{V}} m : *$ and $\Theta^{\uparrow} \vdash n \simeq \uparrow m$ follows with Lemma 4.3.9. ∎

Lemma 4.4.7 (Extension Law) Extension of the relational context Θ preserves the invariant under context extension with a new type variable.

$$\Theta \Vdash_{\mathsf{S}} N \to \Psi \implies \Theta^{\Uparrow} \Vdash_{\mathsf{S}} N + 1 \to \Psi, *$$

$$\Theta \Vdash_{\mathsf{S}} N \to \Psi := \forall n < N.\ \exists m.\ \Theta \vdash n \simeq m \wedge \Psi \vdash_{\mathsf{V}} m : * \qquad \text{(Def. 4.4.5)}$$

$$\Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi := \forall m.\ \Psi \vdash_{\mathsf{V}} m : * \implies \exists n.\ \Theta \vdash n \simeq m \wedge n < N \qquad \text{(Def. 4.4.9)}$$

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi := \forall n A.\ \Gamma_n = A \implies \exists m a.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Psi \vdash_{\mathsf{V}} m : a \qquad \text{(Def. 4.4.25)}$$

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi := \forall m a.\ \Psi \vdash_{\mathsf{V}} m : a \implies \Gamma \vdash_{\lambda} a : * \implies \exists n A.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Gamma_n = A \qquad \text{(Def. 4.4.29)}$$

Figure 4.4: STLC preservation invariants.

Proof Let $n < N + 1$ be an STLC type variable index. For $n = 0$, we can take 0 as the corresponding index in $\Psi, *$, since $\Theta^{\Uparrow} \vdash 0 \simeq 0$ and $\Psi, * \vdash_{\mathsf{V}} 0 : *$ clearly hold. Otherwise we have $n = \uparrow n'$ and thus $n' < N$ with m as the corresponding index in Ψ. Then $\uparrow m$ is the corresponding index in $\Psi, *$ and we again obtain $\Psi, * \vdash_{\mathsf{V}} \uparrow m : *$ from $\Psi \vdash_{\mathsf{V}} m : *$ and $\Theta^{\Uparrow} \vdash \uparrow n' \simeq \uparrow m$ with Lemma 4.3.9. ∎

Note that we do not yet need Lemma 4.4.7, due to the fact that the induction of the present CML proof never extends contexts with type variables. Later though, we will establish the existence of suitably related contexts and there the result is required for the construction of the initial type variable contexts. We give it here to keep the two extension principles together. Moreover when we later scale this development to System F, both skewing and extension will be required directly for the corresponding CML.

For now, though, the associated CML is the following.

Lemma 4.4.8 (Left-Totality and Preservation of $\Theta \vdash_{\mathsf{S}} A \sim a$) For every well-formed STLC type A there exists a corresponding Ψ-type a in $\lambda_{\to}$.

$$N \vdash_{\mathsf{S}} A \implies \forall \Theta \Psi.\ \Theta \Vdash_{\mathsf{S}} N \to \Psi \implies \exists a.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Psi \vdash_{\lambda} a : *$$

Proof By induction on the derivation of $N \vdash_{\mathsf{S}} A$. The invariant handles the variable case while Lemma 4.4.6 is used to incorporate the context extension of Ψ when descending into the codomain of the function type case, while N remains unchanged for the non-dependent STLC function type. ∎

The last of our four key properties is now the right-totality and preservation result. The construction is very similar to what we have seen above. The two differences that are most interesting are a side condition on one of the extension principles and the use of the custom PTS type induction principle (Lemma 4.2.6) to prove the CML.

Definition 4.4.9 (Type Formation Invariant: $\lambda_{\to}$ to STLC) An STLC type variable context N is suitably related to a given $\lambda_{\to}$ context Ψ as mediated by a relational type variable context Θ, whenever the following property is satisfied.

$$\Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi \quad := \quad \forall m.\ \Psi \vdash_{\mathsf{V}} m : * \implies \exists n.\ \Theta \vdash n \simeq m \wedge n < N$$

Lemma 4.4.10 (Skewing law) Skewing of the relational context Θ preserves the invariant under context extension with a new term variable.

$$\Psi \vdash_\lambda a : * \implies \Theta \Vdash_\mathsf{S} N \leftarrow \Psi \implies \Theta^\uparrow \Vdash_\mathsf{S} N \leftarrow \Psi, a$$

Proof Let m be such that $\Psi, a \vdash_\mathsf{V} m : *$. Then clearly $m \neq 0$ for otherwise we would obtain $a = *$ and hence $\Psi \vdash_\lambda * : *$ which is impossible. So $m = \uparrow m'$ for some m' with $\Psi \vdash_\mathsf{V} m' : *$. But then there is an n such that $\Theta \vdash n \simeq m'$ and $n < N$. We clearly also have $\Theta^\uparrow \vdash n \simeq \uparrow m'$, as required. ∎

Lemma 4.4.11 (Extension Law) Extension of the relational context Θ preserves the invariant under context extension with a new type variable.

$$\Theta \Vdash_\mathsf{S} N \leftarrow \Psi \implies \Theta^{\Uparrow} \Vdash_\mathsf{S} N + 1 \leftarrow \Psi, *$$

Proof Let m be such that $\Psi, * \vdash_\mathsf{V} m : *$. When $m = 0$, then the corresponding index is clearly 0 since $0 < N + 1$ trivially holds and $\Theta^{\Uparrow} \vdash 0 \simeq 0$ is also satisfied. Otherwise $m = \uparrow m'$ where $\Psi \vdash_\mathsf{V} m' : *$ holds. Now let $n < N$ be the index related to m' according to Θ, then clearly $\Theta^{\Uparrow} \vdash \uparrow n \simeq \uparrow m'$ and we also have $\uparrow n < N + 1$, as required. ∎

Lemma 4.4.12 (Right-Totality and Preservation of $\Theta \vdash_\mathsf{S} A \sim a$) For every Ψ-type a in $\lambda_\to$ there exists a corresponding well-formed STLC type A.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : * \implies \forall \Theta N.\ \Theta \Vdash_\mathsf{S} N \leftarrow \Psi \implies \exists A.\ \Theta \vdash_\mathsf{S} A \sim a \wedge N \vdash_\mathsf{S} A$$

Proof We use Lemma 4.2.6 to perform an induction on the fact that a is a Ψ-type. In the variable case we use the invariant to obtain an index n, corresponding to the given PTS variable m, and thus construct the related STLC type variable n, which is clearly well-formed under N. Otherwise a is of the form $\Pi c.\, d$ and we obtain by induction some C such that $\Theta \vdash_\mathsf{S} C \sim c$ and with Lemma 4.4.10 also some D such that $\Theta^\uparrow \vdash_\mathsf{S} D \sim d$, so we clearly have $\Theta \vdash_\mathsf{S} C \to D \sim \Pi c.\, d$. In addition, we also know $N \vdash_\mathsf{S} C$ and $N \vdash_\mathsf{S} D$, hence $C \to D$ is the required, well-formed STLC function type. ∎

For the invariants we consider certain ground instances, in particular when the initial context is empty. Here the invariants hold vacuously since they only quantify over elements of the initial contexts. This is also reminiscent of the standard CML proofs.

Fact 4.4.13 (Ground Invariant Instances for Type Formation) For empty initial contexts, the following hold vacuously.

$$\Theta \Vdash_\mathsf{S} 0 \to \Psi \tag{i}$$

$$\Theta \Vdash_\mathsf{S} N \leftarrow \bullet \tag{ii}$$

Recall that STLC is unable to internalise its type variables, so for a closed STLC judgement, N will not necessarily be zero. We have introduced contexts T_N as the PTS counterpart and the relation $\mathsf{id}_\mathsf{R}\, N$ to match them. We can now make this connection precise.

Lemma 4.4.14 (Identity Invariant Instances for Type Formation) For non-empty initial contexts, the following instances hold.

$$\mathsf{id}_{\mathsf{R}}\, N \Vdash_{\mathsf{S}} N \to \mathsf{T}_N \tag{i}$$
$$\mathsf{id}_{\mathsf{R}}\, N \Vdash_{\mathsf{S}} N \leftarrow \mathsf{T}_N \tag{ii}$$

Proof Both are by induction on N. For $N = 0$ we use Fact 4.4.13, while for $N = N'+1$ we use Lemmas 4.4.7 and 4.4.11, respectively. ∎

Note that this proof utilises the extension principles for new type variables, that is exactly those that were not needed for the CML proofs above. This clearly illustrates that the type variables are treated a priori, while the set of free term variables is modified over the course of a given derivation.

4.4.2 Instantiation Compatibility for $\sim$

Recall that one of our prime examples will be the transfer of compatibility with β-substitutions from the PTS to the stratified version. To facilitate this we have to ensure that our relation is strong enough to support it. To be precise, we need a notion of substitution compatibility for our correspondence relation, and as usual, we first need it for the type level before we can consider the term level. In addition, some of the inversion principles that fall out of the treatment of the interaction of the type relation and renaming will be quite useful when we tackle the four main properties for the term relation.

We therefore now take a short detour, prior to the discussion of $\Theta; \Sigma \Vdash_{\mathsf{S}} t \approx b$ and start by considering Fact 4.3.2, which states that related variables are preserved under mapping renamings over the relation. This can be lifted to the type relation.

Lemma 4.4.15 Related types are preserved under renaming, when the contextual information about related free variables is adjusted accordingly.

$$\Theta \Vdash_{\mathsf{S}} A \sim a \implies \mathsf{map}\,(\xi \times \zeta)\,\Theta \Vdash_{\mathsf{S}} A[\xi] \sim a[\zeta]$$

Proof By induction on the derivation of $\Theta \Vdash_{\mathsf{S}} A \sim a$, using Fact 4.3.2 for the variable case and Lemma 4.3.10 for function types to move the skewing operation on Θ through the map. ∎

In a similar fashion we can also lift our monotonicity result for relational contexts to the type relation. This can be seen as a form of weakening for $\sim$.

Lemma 4.4.16 Related types are preserved under monotone extension of the relational context.

$$\Theta_1 \sqsubseteq \Theta_2 \implies \Theta_1 \Vdash_{\mathsf{S}} A \sim a \implies \Theta_2 \Vdash_{\mathsf{S}} A \sim a$$

Proof By induction on the derivation of $\Theta_1 \models_{\mathsf{S}} A \sim a$ using Fact 4.3.5 for the variable case. For the binder case we need Lemma 4.3.6 to ensure that skewing preserves context inclusion. ∎

We can combine the two preceding results to obtain two useful principles for the interaction of the type correspondence relation with context skewing and extension.

Lemma 4.4.17 (Preservation of $\Theta \models_{\mathsf{S}} A \sim a$ under Skewing and Extension) The following principles hold.

$$\Theta \models_{\mathsf{S}} A \sim a \implies \Theta^{\uparrow} \models_{\mathsf{S}} A \sim a[\uparrow] \tag{i}$$

$$\Theta \models_{\mathsf{S}} A \sim a \implies \Theta^{\Uparrow} \models_{\mathsf{S}} A[\uparrow] \sim a[\uparrow] \tag{ii}$$

Proof Property (i) is a direct instance of Lemma 4.4.15. For (ii), we first observe that $\mathsf{map}\,(\uparrow \times \uparrow)\,\Theta \sqsubseteq \Theta^{\Uparrow}$, and then use both Lemma 4.4.15 and Lemma 4.4.16, to obtain the desired result. ∎

We are now in a position to define what it means for the type relation to be compatible with instantiation, which is yet another CML-style result. Clearly, instantiating all free variables of two related types with parallel substitutions has the potential to introduce new sets of free variables and we should better know how these are related if we want to say anything about relatedness after the instantiation. This motivates the following invariant, which is preserved under skewing.

Definition 4.4.18 (Type Relation Invariant)

$$\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{S}} \Theta_1 \to \Theta_2 \quad := \quad \forall nm.\ \Theta_1 \vdash n \simeq m \implies \Theta_2 \models_{\mathsf{S}} \sigma\, n \sim \tau\, m$$

Lemma 4.4.19 (Skewing Law) One-sided lifting of two related type substitutions preserves the invariant under context skewing.

$$\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{S}} \Theta_1 \to \Theta_2 \implies \langle \sigma \sim \Uparrow \tau \rangle \Vdash_{\mathsf{S}} \Theta_1^{\uparrow} \to \Theta_2^{\uparrow}$$

Proof Straightforward using Lemma 4.3.8 and Lemma 4.4.17. ∎

Lemma 4.4.20 (CML for $\Theta \models_{\mathsf{S}} A \sim a$) Related types remain related under instantiation with related type substitutions.

$$\Theta_1 \models_{\mathsf{S}} A \sim a \implies \langle \sigma \sim \tau \rangle \Vdash_{\mathsf{S}} \Theta_1 \to \Theta_2 \implies \Theta_2 \models_{\mathsf{S}} A[\sigma] \sim a[\tau]$$

Proof By induction on the derivation of $\Theta_1 \models_{\mathsf{S}} A \sim a$. As usual, the invariant handles the variable case, while preservation of the invariant under skewing is needed for the binder case. ∎

In the current setting, that is the discussion of STLC, this CML is only needed to obtain the corresponding result for the term relation below. When we consider System F instead, it will have more immediate applications.

We can also obtain an analogue to Fact 4.3.3, that is, we can strip renamings from related types. This is the inverse to our renaming result (Lemma 4.4.15) and provides a form of strengthening, which will be crucial later.

Lemma 4.4.21 Whenever types are related under a mapped variable context, it is possible to strip the mappings.

$$\mathsf{map}\,(\xi \times \zeta)\,\Theta \vdash_{\mathsf{S}} A \sim a \implies \exists A' a'.\ A = A'[\xi] \wedge a = a'[\zeta] \wedge \Theta \vdash_{\mathsf{S}} A' \sim a'$$

Proof By induction on the derivation of $\mathsf{map}\,(\xi \times \zeta)\,\Theta \vdash_{\mathsf{S}} A \sim a$ using Fact 4.3.3 for the variable case and Lemma 4.3.10 to commute mapping and skewing in the binder case. ∎

Corollary 4.4.22 We can strip skewing from related types.

$$\Theta^{\uparrow} \vdash_{\mathsf{S}} A \sim a \implies \exists a'.\ a = a'[\uparrow] \wedge \Theta \vdash_{\mathsf{S}} A \sim a'$$

Proof Direct instance of Lemma 4.4.21. ∎

This concludes our small detour with respect to substitutivity properties of the type relation.

4.4.3 Four Properties for $\approx$

We next turn our attention to the relation on terms where we need to handle embedded type-level derivations. The constructions are, however, mostly similar to what we have seen before.

As above, the functionality and injectivity are the easier of the four required properties, so we again deal with them first.

Lemma 4.4.23 (Functionality of $\Theta; \Sigma \vdash_{\mathsf{S}} t \approx b$) The term relation is functional, provided that both Θ and Σ are functional relations.

$$\Theta\ \mathsf{func} \implies \Sigma\ \mathsf{func} \implies \Theta; \Sigma \vdash_{\mathsf{S}} t \approx b_1 \implies \Theta; \Sigma \vdash_{\mathsf{S}} t \approx b_2 \implies b_1 = b_2$$

Proof By induction on the derivation of $\Theta; \Sigma \vdash_{\mathsf{S}} t \approx b_1$ and discriminating on the derivation of $\Theta; \Sigma \vdash_{\mathsf{S}} t \approx b_2$. In the variable case we use the functionality of Σ and the application case is a trivial consequence of the inductive hypotheses. The case for abstractions follows from the inductive hypotheses and the preservation of functionality under skewing of Θ and extension of Σ (Fact 4.3.11). The type annotations on the abstractions coincide due to functionality of the type relation (Lemma 4.4.3). ∎

Lemma 4.4.24 (Injectivity of $\Theta; \Sigma \models_{\mathsf{S}} t \approx b$) The term relation is injective, provided that both Θ and Σ are injective relations.

$$\Theta \text{ inj} \implies \Sigma \text{ inj} \implies \Theta; \Sigma \models_{\mathsf{S}} t_1 \approx b \implies \Theta; \Sigma \models_{\mathsf{S}} t_2 \approx b \implies t_1 = t_2$$

Proof Analogue to the functionality proof above, with injectivity used in place of functionality, e.g. Lemma 4.4.4 in place of Lemma 4.4.3. ∎

For the totality and preservation results we also proceed similarly to the type level, but due to the embedded type-level derivations we obtain invariants that are slightly more involved. In particular, for the type level we simply expected related type variables while here we require related term variables *with related types*.

Definition 4.4.25 (Typing Invariant: STLC to $\lambda_{\to}$) A $\lambda_{\to}$ context Ψ is suitably related to a given STLC term variable context Γ as mediated by relational contexts Θ and Σ, whenever the following property is satisfied.

$$\begin{aligned}\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi \quad :=\quad & \forall n A.\ \Gamma_n = A \implies \\ & \exists m a.\ \Theta \models_{\mathsf{S}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Psi \models_{\mathsf{V}} m : a\end{aligned}$$

Lemma 4.4.26 (Skewing/Extension Law) Skewing of the relational type variable context Θ and extension of the relational term variable context Σ preserves the invariant under context extension with a new term variable.

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi \implies \Theta \models_{\mathsf{S}} A \sim a \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \Vdash_{\mathsf{S}} \Gamma, A \to \Psi, a$$

Proof Let $(\Gamma, A)_n = B$.

We either have $n = 0$ and $A = B$, in which case 0 is the corresponding related PTS de Bruijn index with type $a[\uparrow]$ since $\Theta^{\uparrow} \models_{\mathsf{S}} B \sim a[\uparrow]$ and $\Psi, a \models_{\mathsf{V}} 0 : a[\uparrow]$ clearly hold.

Otherwise we have $n = \uparrow n'$ for some n' satisfying $\Gamma_{n'} = B$. Then from our assumption we have some m and b satisfying $\Theta \models_{\mathsf{S}} B \sim b$, $\Sigma \vdash n' \simeq m$, and $\Psi \models_{\mathsf{V}} m : b$. The corresponding PTS index is thus $\uparrow m$ with type $b[\uparrow]$ since $\Theta^{\uparrow} \models_{\mathsf{S}} B \sim b[\uparrow]$, $\Sigma^{\Uparrow} \vdash \uparrow n' \simeq \uparrow m$, and $\Psi, a \models_{\mathsf{V}} \uparrow m : b[\uparrow]$ all clearly hold. ∎

Lemma 4.4.27 (Extension/Skewing Law) Extension of the relational type variable context Θ and skewing of the relational term variable context Σ preserves the invariant under context extension with a new type variable.

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi \implies \Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_{\mathsf{S}} \Gamma[\uparrow] \to \Psi, *$$

Proof Let $(\Gamma[\uparrow])_n = B$, that is we have some B' such that $\Gamma_n = B'$ and $B = B'[\uparrow]$. Then from our assumption we have some m and b satisfying $\Theta \models_{\mathsf{S}} B' \sim b$, $\Sigma \vdash n \simeq m$, and $\Psi \models_{\mathsf{V}} m : b$. The corresponding PTS index is thus $\uparrow m$ with type $b[\uparrow]$ since $\Theta^{\Uparrow} \models_{\mathsf{S}} B'[\uparrow] \sim b[\uparrow]$, $\Sigma^{\uparrow} \vdash n \simeq \uparrow m$, and $\Psi, * \models_{\mathsf{V}} \uparrow m : b[\uparrow]$ all clearly hold. ∎

Lemma 4.4.28 (Left-Totality and Preservation of $\Theta; \Sigma \vdash_{\mathsf{S}} t \approx b$) For every well-typed STLC term t there exists a corresponding Ψ-term b in $\lambda_{\to}$, such that their types are also related.

$$N; \Gamma \vdash_{\mathsf{S}} t : A \implies \forall \Theta \Sigma \Psi.\ \mathsf{val}\ \Psi \implies \Theta\ \mathsf{func} \implies \Theta \Vdash_{\mathsf{S}} N \to \Psi \implies \Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi \implies \exists b a.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Theta; \Sigma \vdash_{\mathsf{S}} t \approx b \wedge \Psi \vdash_{\lambda} b : a \wedge \Psi \vdash_{\lambda} a : *$$

Proof By induction on the derivation of $N; \Gamma \vdash_{\mathsf{S}} t : A$.

Assume $N; \Gamma \vdash_{\mathsf{S}} n : A$ and therefore $\Gamma_n = A$ and $N \vdash_{\mathsf{S}} A$. From the latter, the type-level invariant and Lemma 4.4.8 it follows that there is a related PTS type a satisfying $\Theta \vdash_{\mathsf{S}} A \sim a$ and $\Psi \vdash_{\lambda} a : *$. From the term-level invariant we additionally obtain a PTS index m and another PTS type a' satisfying $\Theta \vdash_{\mathsf{S}} A \sim a'$, $\Sigma \vdash n \simeq m$ and $\Psi \vdash_{\mathsf{V}} m : a'$. Since Θ, and therefore the type relation under Θ, is functional, we know that $a = a'$, which allows us to infer $\Psi \vdash_{\lambda} m : a$. Similarly we have $\Theta; \Sigma \vdash_{\mathsf{S}} n \approx m$, which is the last missing component.

Now consider $N; \Gamma \vdash_{\mathsf{S}} s\,t : B$ with some A such that $N; \Gamma \vdash_{\mathsf{S}} s : A \to B$ and $N; \Gamma \vdash_{\mathsf{S}} t : A$. Our inductive hypotheses yield all of the following for some p, l, a and c:

$$\Theta \vdash_{\mathsf{S}} A \to B \sim p \qquad \Theta; \Sigma \vdash_{\mathsf{S}} s \approx l \qquad \Psi \vdash_{\lambda} l : p$$
$$\Theta \vdash_{\mathsf{S}} A \sim a \qquad \Theta; \Sigma \vdash_{\mathsf{S}} t \approx c \qquad \Psi \vdash_{\lambda} c : a$$

We clearly have $p = \Pi a'.\, b$ with $\Theta \vdash_{\mathsf{S}} A \sim a'$ and $\Theta^{\uparrow} \vdash_{\mathsf{S}} B \sim b$. With functionality we get $a = a'$ and Corollary 4.4.22 yields a type b' satisfying $b = b'[\uparrow]$ and $\Theta \vdash_{\mathsf{S}} B \sim b'$. The related term and type are $l\,c$ and b' respectively, since $\Theta; \Sigma \vdash_{\mathsf{S}} s\,t \approx l\,c$ clearly holds and the required type relation is given. We still have to show that we obtain the proper PTS typings, $\Psi \vdash_{\lambda} l\,c : b'$ and $\Psi \vdash_{\lambda} b' : *$, which requires further effort. From $\Psi \vdash_{\lambda} c : a$, $\Psi \vdash_{\lambda} l : \Pi a.\, b'[\uparrow]$ and Lemma 3.6.14 it follows that $\Psi \vdash_{\lambda} b'[\uparrow][c \cdot \mathsf{id}] : *$, which simplifies to $\Psi \vdash_{\lambda} b' : *$ as required. We can further derive the PTS typing $\Psi \vdash_{\lambda} l\,c : b'[\uparrow][c \cdot \mathsf{id}]$, which similarly simplifies as required. Note how in both cases the definition of the type relation ensures that the codomain of the dependent PTS function type is suitably shifted to render the resulting instantiation vacuous.

The final case is abstraction, that is we have $N; \Gamma \vdash_{\mathsf{S}} \lambda A.\, t : A \to B$, which derived from $N; \Gamma, A \vdash_{\mathsf{S}} t : B$ and $N \vdash_{\mathsf{S}} A$. With Lemma 4.4.8 it follows that there is some PTS type a satisfying $\Theta \vdash_{\mathsf{S}} A \sim a$ and $\Psi \vdash_{\lambda} a : *$. We therefore know that $\mathsf{val}\ \Psi, a$ and can also extend the type- and term-level invariants to $\Theta^{\uparrow} \Vdash_{\mathsf{S}} N \to \Psi, a$ using Lemma 4.4.6 and $\Theta^{\uparrow}; \Sigma^{\Uparrow} \Vdash_{\mathsf{S}} \Gamma, A \to \Psi, a$ using Lemma 4.4.26 respectively. This allows us to obtain the following inductive hypotheses for some b and c:

$$\Theta^{\uparrow} \vdash_{\mathsf{S}} B \sim b \qquad \Psi, a \vdash_{\lambda} b : * \qquad \Theta^{\uparrow}; \Sigma^{\Uparrow} \vdash_{\mathsf{S}} t \approx c \qquad \Psi, a \vdash_{\lambda} c : b$$

The required related term and type are $\lambda a.\, c$ and $\Pi a.\, b$ respectively. All four desired properties are now easily derivable. ∎

Definition 4.4.29 (Typing Invariant: $\lambda_{\to}$ to STLC) An STLC term variable context Γ is suitably related to a given $\lambda_{\to}$ context Ψ as mediated by relational contexts Θ and Σ, whenever the following property is satisfied.

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi \quad :\Leftrightarrow \quad \forall m a.\ \Psi \vdash_{\mathsf{V}} m : a \implies \Gamma \vdash_{\lambda} a : * \implies \exists n A.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Gamma_n = A$$

Lemma 4.4.30 (Skewing/Extension Law) Skewing of the relational type variable context Θ and extension of relational term variable context Σ preserves the invariant under context extension with a new term variable.

$$\mathsf{val}\ \Psi, a \implies \Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi \implies \Theta \vdash_{\mathsf{S}} A \sim a \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \Vdash_{\mathsf{S}} \Gamma, A \leftarrow \Psi, a$$

Proof Let $(\Psi, a) \vdash_{\mathsf{V}} m : c$ with $\Psi, a \vdash_{\lambda} c : *$.

We either have $m = 0$ and $a[\uparrow] = c$, in which case $n = 0$ with corresponding type A are the required witnesses, and all properties follow easily.

Otherwise we have $m = \uparrow m'$ and $b[\uparrow] = c$, where b satisfies $\Gamma \vdash_{\mathsf{V}} m' : b$. We strengthen $\Psi, a \vdash_{\lambda} b[\uparrow] : *$ to $\Psi \vdash_{\lambda} b : *$ using Fact 4.2.8. This allows us to use the invariant for Γ and Ψ to obtain some B at index n in Γ ($\Gamma_n = B$), such that $\Theta \vdash_{\mathsf{S}} B \sim b$ and $\Sigma \vdash n \simeq m'$. Then we clearly have $(\Gamma, A)_{\uparrow n} = B$, $\Sigma^{\Uparrow} \vdash \uparrow n \simeq \uparrow m'$ and $\Theta^{\uparrow} \vdash_{\mathsf{S}} B \sim b[\uparrow]$. ∎

Lemma 4.4.31 (Extension/Skewing Law) Extension of the relational type variable context Θ and skewing of the relational term variable context Σ preserves the invariant under context extension with a new type variable.

$$\mathsf{val}\ \Psi, * \implies \Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi \implies \Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_{\mathsf{S}} \Gamma[\uparrow] \leftarrow \Psi, *$$

Proof Let $(\Psi, *) \vdash_{\mathsf{V}} m : c$ with $\Psi, * \vdash_{\lambda} c : *$. Then clearly $m = \uparrow m'$ for otherwise $c = *$, which is not possible. As before we also have $b[\uparrow] = c$, where b satisfies $\Gamma \vdash_{\mathsf{V}} m' : b$, allowing us to strengthen $\Psi, * \vdash_{\lambda} b[\uparrow] : *$ to $\Psi \vdash_{\lambda} b : *$. We again use the invariant for Γ and Ψ to obtain some B at index n in Γ ($\Gamma_n = B$), such that $\Theta \vdash_{\mathsf{S}} B \sim b$ and $\Sigma \vdash n \simeq m'$. Now clearly $\Sigma^{\uparrow} \vdash n \simeq \uparrow m'$, so n is the required index, while the corresponding type is $B[\uparrow]$. $\Theta^{\Uparrow} \vdash_{\mathsf{S}} B[\uparrow] \sim b[\uparrow]$ is easy to verify, and $(\Gamma[\uparrow])_n = B[\uparrow]$ holds since instantiation is applied pointwise to contexts. ∎

Lemma 4.4.32 (Right-Totality and Preservation of $\Theta; \Sigma \vdash_{\mathsf{S}} t \approx b$) For every Ψ-term b in $\lambda_{\to}$ there exists a corresponding well-typed STLC term t, such that their types are also related.

$$\mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : * \implies \Psi \vdash_{\lambda} b : a \implies \forall \Theta \Sigma N \Gamma.\ \Theta\ \mathsf{inj} \implies \Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi \implies \Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi \implies \exists t A.\ \Theta \vdash_{\mathsf{S}} A \sim a \wedge \Theta; \Sigma \vdash_{\mathsf{S}} t \approx b \wedge N; \Gamma \vdash_{\mathsf{S}} t : A \wedge N \vdash_{\mathsf{S}} A$$

Proof We use Lemma 4.2.7 to perform an induction on the fact that b is a Ψ-term.

Assume $\Psi \vdash_\lambda m : a$ and therefore $\Psi \vdash_\forall m : a$ and $\Psi \vdash_\lambda a : *$. From the latter, the type-level invariant and Lemma 4.4.12 it follows that there is a related STLC type A satisfying $\Theta \models_\varsigma A \sim a$ and $N \models_\varsigma A$. From the term-level invariant we additionally obtain an STLC index n and another STLC type A' satisfying $\Theta \models_\varsigma A' \sim a$, $\Sigma \vdash n \simeq m$ and $\Gamma_n = A'$. Since Θ, and therefore the type relation under Θ, is injective, we know that $A = A'$, which allows us to infer $N; \Gamma \models_\varsigma n : A$. Similarly we have $\Theta; \Sigma \models_\varsigma n \approx m$, which is the last missing component.

Now consider $\Psi \vdash_\lambda a\, b : d'$ with some c, d such that $\Psi \vdash_\lambda a : \Pi c.\, d$, $\Psi \vdash_\lambda b : c$ and $d' = d[b \cdot \mathsf{id}]$. Our inductive hypotheses yield all of the following for some types F and C, and terms f and s:

$$\Theta \models_\varsigma F \sim \Pi c.\, d \qquad N \models_\varsigma F \qquad \Theta; \Sigma \models_\varsigma f \approx a \qquad N; \Gamma \models_\varsigma f : F$$
$$\Theta \models_\varsigma C \sim c \qquad\qquad \Theta; \Sigma \models_\varsigma s \approx b \qquad N; \Gamma \models_\varsigma s : C$$

We clearly have $F = C' \to D$ with $\Theta \models_\varsigma C' \sim c$ and $\Theta^\uparrow \models_\varsigma D \sim d$. With injectivity we get $C = C'$ and Corollary 4.4.22 yields a type d'' satisfying $d = d''[\uparrow]$ and $\Theta \models_\varsigma D \sim d''$. We therefore have $d' = d''[\uparrow][b \cdot \mathsf{id}]$, which simplifies to $d' = d''$. The related term and type are $f\, s$ and D respectively, since $\Theta; \Sigma \models_\varsigma f\, s \approx a\, b$ clearly holds and the required type relation is given. The fact that $N \models_\varsigma D$ follows trivially from $N \models_\varsigma C \to D$ and the typing $N; \Gamma \models_\varsigma f\, s : D$ can also be constructed easily.

The final case is abstraction, that is we have $\Psi \vdash_\lambda \lambda a.\, b : \Pi a.\, c$, which in particular means that we also know $\Psi \vdash_\lambda a : *$. With Lemma 4.4.12 it follows that there is some STLC type A satisfying $\Theta \models_\varsigma A \sim a$ and $N \models_\varsigma A$. We therefore know that $\mathsf{val}\ \Psi, a$ and can also extend the type and term-level invariants to $\Theta^\uparrow \Vdash_\varsigma N \leftarrow \Psi, a$ using Lemma 4.4.10 and $\Theta^\uparrow; \Sigma^{\Uparrow} \Vdash_\varsigma \Gamma, A \leftarrow \Psi, a$ using Lemma 4.4.30 respectively. This allows us to obtain the following inductive hypotheses for some type C and term s:

$$\Theta^\uparrow \models_\varsigma C \sim c \qquad N \models_\varsigma C \qquad \Theta^\uparrow; \Sigma^{\Uparrow} \models_\varsigma s \approx b \qquad N; \Gamma, A \models_\varsigma s : C$$

The required related term and type are $\lambda A.\, s$ and $A \to C$ respectively. All four desired properties are now easily derivable. ∎

As for the type level it is helpful to consider the various ground instances of the invariants used in the preceding proofs. Those for the empty initial contexts again hold vacuously and we state them without proof.

Fact 4.4.33 (Ground Invariant Instances for Typing) The following hold vacuously.

$$\Theta; \Sigma \Vdash_\varsigma \bullet \to \Psi \tag{i}$$
$$\Theta; \Sigma \Vdash_\varsigma \Gamma \leftarrow \bullet \tag{ii}$$

With respect to an initial PTS context that consists solely of type variables we also get a ground instance, but since the STLC term-variable context does not concern itself with type variables there is no symmetric result as we had in Lemma 4.4.14.

Lemma 4.4.34 (Identity Invariant Instance for Typing) The following instance holds.

$$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \mathsf{T}_N$$

Proof Let m and a be such that $\mathsf{T}_N \vdash_{\mathsf{V}} m : a$ and $\mathsf{T}_N \vdash_{\lambda} a : *$. The former implies $a = *$, which contradicts the latter, so such m and a cannot exist and the instance holds vacuously. ∎

4.4.4 Instantiation Compatibility for $\approx$

We now conclude our discussion of the relation on terms by showing that it is compatible with instantiation. We utilise a similar setup to the one for the type relation above and first consider renamings, as well as monotonicity with respect to the relational term variable context.

Lemma 4.4.35 Related terms are preserved under renamings, when the contextual information about related free variables is adjusted accordingly.

$$\Theta; \Sigma \vdash_{\mathsf{S}} s \approx b \implies \mathsf{map}\,(\mathsf{id} \times \zeta)\,\Theta; \mathsf{map}\,(\xi \times \zeta)\,\Sigma \vdash_{\mathsf{S}} s[\xi] \approx b[\zeta]$$

Proof By induction on the derivation of $\Theta; \Sigma \vdash_{\mathsf{S}} s \approx b$, using Fact 4.3.2 for the variable case. The application case is trivial, while the abstraction case requires Lemma 4.4.15 for the type annotations and Lemma 4.3.10 (both parts) to move the skewing of Θ and the extension of Σ through their respective map operations. ∎

Lemma 4.4.36 Related terms are preserved under monotone extension of the relational term variable context.

$$\Sigma_1 \sqsubseteq \Sigma_2 \implies \Theta; \Sigma_1 \vdash_{\mathsf{S}} s \approx b \implies \Theta; \Sigma_2 \vdash_{\mathsf{S}} s \approx b$$

Proof By induction on the derivation of $\Theta; \Sigma_1 \vdash_{\mathsf{S}} s \approx b$ using Fact 4.3.5 for the variable case. Application is again trivial, and for abstractions we only need to consider the body where we have the context $\Theta^{\uparrow}; \Sigma_2^{\Uparrow}$, since the type context does not change. For the body we need Lemma 4.3.7 to ensure that $\Sigma_1^{\Uparrow} \sqsubseteq \Sigma_2^{\Uparrow}$ holds. ∎

We again combine these two results to obtain the extension principle for the term relation with respect to binding a new term variable.

Lemma 4.4.37 (Preservation of $\Theta; \Sigma \vdash_{\mathsf{S}} s \approx b$ under Skewing/Extension) The following principle holds.

$$\Theta; \Sigma \vdash_{\mathsf{S}} s \approx b \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \vdash_{\mathsf{S}} s[\uparrow] \approx b[\uparrow]$$

Proof With Lemma 4.4.35 we infer

$$\Theta^{\uparrow}; \mathsf{map}\,(\uparrow \times \uparrow)\,\Sigma \vdash_{\mathsf{S}} s[\uparrow] \approx b[\uparrow],$$

and since we can easily show that $\mathsf{map}\,(\uparrow \times \uparrow)\,\Sigma \sqsubseteq \Sigma^{\Uparrow}$ holds, we can obtain the desired result with Lemma 4.4.36. ∎

We can now precisely define what it means for the term relation to be compatible with instantiation. As with the type relation we have to be able to track changes to the contextual information that are introduced through instantiation. While the corresponding invariant has to track seven related quantities, and therefore may look a bit daunting, it actually still follows the basic CML pattern. The only aspect that is slightly tricky is the fact that we have two distinct parallel substitutions on the STLC side, ρ for types and σ for terms, while for $\lambda_\to$ we only have one substitution τ that has to mirror the actions of both ρ and σ. We also have to track both the type and the term level simultaneously, which effectively incorporates the type-level invariant into the term-level invariant. The associated extension principle covers the binding of a fresh term variable and is designed to deal with the abstraction case in the subsequent CML proof.

Definition 4.4.38 (Term Relation Invariant)

$$\begin{aligned}\langle(\rho,\sigma)\sim\tau\rangle \Vdash_{\mathsf{S}} \Theta_1;\Sigma_1 \to \Theta_2;\Sigma_2 \quad :=\quad & (\forall nm.\ \Theta_1 \vdash n \simeq m \implies \Theta_2 \vdash_{\mathsf{S}} \rho\, n \sim \tau\, m) \wedge \\ & (\forall nm.\ \Sigma_1 \vdash n \simeq m \implies \Theta_2;\Sigma_2 \vdash_{\mathsf{S}} \sigma\, n \approx \tau\, m)\end{aligned}$$

Lemma 4.4.39 (Skewing/Extension Law) The invariant satisfies the following extension principle, which corresponds to the addition of a fresh term variable.

$$\langle(\rho,\sigma)\sim\tau\rangle \Vdash_{\mathsf{S}} \Theta_1;\Sigma_1 \to \Theta_2;\Sigma_2 \implies \langle(\rho,\Uparrow\sigma)\sim\Uparrow\tau\rangle \Vdash_{\mathsf{S}} \Theta_1^{\uparrow};\Sigma_1^{\Uparrow} \to \Theta_2^{\uparrow};\Sigma_2^{\Uparrow}$$

Proof The first conjunct is simply Lemma 4.4.19. For the second we use Lemma 4.3.8 to perform a case analysis on $\Sigma_1^{\Uparrow} \vdash n \simeq m$. The case for $n = m = 0$ is trivial, otherwise we use the invariant for the non-extended context and Lemma 4.4.37 to handle the resulting shifts. ∎

Note that in the current setting, we can actually fix the type substitution ρ to the identity substitution, since instantiation on STLC terms does not propagate into types. As we will see in the next chapter, this fails to hold for System F.

Lemma 4.4.40 (CML for $\Theta;\Sigma \vdash_{\mathsf{S}} s \approx b$) Related terms remain related under instantiation with related type and term substitutions.

$$\Theta_1;\Sigma_1 \vdash_{\mathsf{S}} s \approx b \implies \langle(\mathsf{id},\sigma)\sim\tau\rangle \Vdash_{\mathsf{S}} \Theta_1;\Sigma_1 \to \Theta_2;\Sigma_2 \implies \Theta_2;\Sigma_2 \vdash_{\mathsf{S}} s[\sigma] \approx b[\tau]$$

Proof By induction on the derivation of $\Theta_1;\Sigma_1 \vdash_{\mathsf{S}} s \approx b$, using the second conjunct of the invariant for the variable case. The application case is again straightforward. For abstractions we need the first conjunct of the invariant and Lemma 4.4.20 to ensure relatedness of type annotations, as well as the extension principle of Lemma 4.4.39 to instantiate the inductive hypothesis for the abstraction bodies. ∎

With this result for general substitutions we can derive compatibility with β-substitutions as a simple corollary.

Corollary 4.4.41 (Compatibility of $\Theta; \Sigma \models_{\mathsf{S}} s \approx b$ with β-Substitution) The following inference rule is admissible.

$$\frac{\Theta^{\uparrow}; \Sigma^{\Uparrow} \models_{\mathsf{S}} s \approx b \qquad \Theta; \Sigma \models_{\mathsf{S}} t \approx c}{\Theta; \Sigma \models_{\mathsf{S}} s[t \cdot \mathsf{id}] \approx b[c \cdot \mathsf{id}]}$$

Proof From the second premise we can derive a particular instance of the CML invariant, namely

$$\langle (\mathsf{id}, t \cdot \mathsf{id}) \sim c \cdot \mathsf{id} \rangle \Vdash_{\mathsf{S}} \Theta^{\uparrow}; \Sigma^{\Uparrow} \to \Theta; \Sigma,$$

which allows us to simply instantiate Lemma 4.4.40 and thereby close the proof. ∎

4.4.5 Existence of Related Contexts

When we consider the structure of Lemmas 4.4.8 and 4.4.28 as well as Lemmas 4.4.12 and 4.4.32, we notice that in each case we start from a valid initial context and a derivable judgement under said context. The results then yield corresponding terms and types of the respective other system, proofs of relatedness and derivable judgements for the related terms and types, *provided* that we can present a suitable, valid context of the target system and sufficient contextual information to connect the initial and target contexts in the form of invariants.

This raises the question of where such target contexts and invariant instances should come from. We have already seen that it is relatively easy to provide such information for the special case where the initial contexts happen to be empty, but what about the general case? As it turns out, we can always construct a matching target context, including contextual mapping information and all potentially relevant invariants. The main idea is that we can perform inductions on the initial context validity assumptions. Then, whenever a new entry is added to an initial context, we know that it is well-formed with respect to the context to which it is going to be added. With the totality and preservation results for type formation we can then in each case obtain a corresponding entry for the target context and hence build it up in lockstep. Scaling the invariants is also easy since we have already proven the respective extension principles throughout this chapter. The following two Lemmas establish this result for each of the two directions.

Lemma 4.4.42 Let $N; \Gamma$ be a valid STLC context, then there exists a valid $\lambda_{\to}$ context Ψ which relates to $N; \Gamma$ according to some relational contexts Θ and Σ, such that all the following hold.

Θ func	Σ func	$\Theta \Vdash_{\mathsf{S}} N \to \Psi$	$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi$
Θ inj	Σ inj	$\Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi$	$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi$

Proof By induction on $\mathsf{val}\ N; \Gamma$.

In the base case we consider $N; \bullet$. The corresponding valid (Fact 4.2.10) PTS context is $\Psi := \mathsf{T}_N$, with $\Theta := \mathsf{id}_{\mathsf{R}}\, N$ and $\Sigma := \bullet$ which are both functional and

injective. The required invariant instances are provide by Lemma 4.4.14 for the type level and Fact 4.4.33 and Lemma 4.4.34 for the term level.

Otherwise we have the full set of properties for the context $N; \Gamma$ and a corresponding Ψ as well as some Θ and Σ connecting them. We further have some STLC type A which satisfies $N \models_{\mathsf{S}} A$ and need to establish the various properties for the context $N; \Gamma, A$. The inductive hypotheses and the well-typedness assumption allow us to use the left-totality and preservation result for type formation (Lemma 4.4.8) and thereby obtain a PTS term a satisfying $\Theta \models_{\mathsf{S}} A \sim a$ and $\Psi \vdash_{\lambda} a : *$. The required PTS context therefore is Ψ, a, which is clearly valid, with relational contexts $\Theta^{\uparrow}$ and $\Sigma^{\Uparrow}$. We know that both skewing and extending of relational contexts preserve functionality and injectivity from Fact 4.3.11. For the invariant instances we reuse the extension principles that were developed for the totality and preservation results, namely Lemmas 4.4.6, 4.4.10, 4.4.26 and 4.4.30. ∎

Lemma 4.4.43 Let Ψ be a valid $\lambda_{\to}$ context, then there exists a valid STLC context $N; \Gamma$ which relates to Ψ according to some relational contexts Θ and Σ, such that all the following hold.

Θ func	Σ func	$\Theta \Vdash_{\mathsf{S}} N \to \Psi$	$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi$
Θ inj	Σ inj	$\Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi$	$\Theta; \Sigma \Vdash_{\mathsf{S}} \Gamma \leftarrow \Psi$

Proof By induction on $\mathsf{val}\ \Psi$.

In the base case we consider $\Psi = \bullet$. The corresponding valid STLC context is $0; \bullet$, with $\Theta := \bullet$ and $\Sigma := \bullet$ which are both functional and injective. The required invariant instances are provide by Facts 4.4.13 and 4.4.33.

Otherwise we have the full set of properties for the context Ψ and corresponding $N; \Gamma$ as well as some Θ and Σ connecting them. We further have some PTS term a which satisfies $\Psi \vdash_{\lambda} a : \mathsf{s}$ and need to establish the various properties for the context Ψ, a. At this point we need to distinguish two cases.

- We could have $\mathsf{s} = *$, meaning that a is a semantic type and the associated variable is a term variable. With Lemma 4.4.8 we obtain a corresponding STLC type A that satisfies $\Theta \models_{\mathsf{S}} A \sim a$ and $N \models_{\mathsf{S}} A$. The required STLC context is $N; \Gamma, A$ (valid by defintion), and the relational contexts are $\Theta^{\uparrow}$ and $\Sigma^{\Uparrow}$. The preservation of all relevant properties under these adjustments follows from the same results we used in the previous proof.

- Otherwise we have $\mathsf{s} = \Box$ and thus $a = *$ by Lemma 4.2.2. The extension therefore adds a new type variable, and accordingly we choose $N + 1; \Gamma[\uparrow]$ as the new STLC context, which is valid due to Lemma 4.1.6. Here, in contrast to the preceding case, we now extend the relational type variable context Θ to $\Theta^{\Uparrow}$ and skew the relational term variable context Σ to $\Sigma^{\uparrow}$. Functionality and injectivity can be justified as before but for the change in the invariants we now need Lemmas 4.4.7, 4.4.11, 4.4.27 and 4.4.31. ∎

4.4.6 Closed Correspondence

In the beginning of this chapter we introduced our desired correspondence result for the simply typed λ-calculus as a set of equivalences that hold for closed judgements. We have since argued that the closed setting, while easy to grasp, is only of marginal use. For simple scenarios, like the transfer of propagation, they may be sufficient, but results like β-substitutivity inherently rely on non-empty contexts. We now introduce the equivalences for the closed setting for the sake of completeness.

Theorem 4.4.44 (STLC Correspondence for Closed Judgements) For the special case of closed type formation and typing derivations we obtain the following equivalences.

$$N \mathrel{\mathsf{f}_{\mathsf{S}}} A \iff \exists a.\ \mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a \wedge \mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : * \tag{i}$$

$$\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : * \iff \exists A.\ \mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a \wedge N \mathrel{\mathsf{f}_{\mathsf{S}}} A \tag{ii}$$

$$N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s : A \iff \exists ba.\ \mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a \wedge \mathsf{id}_{\mathsf{R}}\, N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s \approx b \wedge \mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} b : a \wedge \mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : * \tag{iii}$$

$$\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : * \wedge \mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} b : a \iff \exists sA.\ \mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a \wedge \mathsf{id}_{\mathsf{R}}\, N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s \approx b \wedge N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s : A \wedge N \mathrel{\mathsf{f}_{\mathsf{S}}} A \tag{iv}$$

Proof Each of the equivalences can be established with the same technique. We first prove the direction from left to right using the corresponding preservation result and the corresponding ground instances for the invariant. For the inverse direction we additionally need that the correspondence relations are functional and injective. We give the proofs of (i) and (iv), the other two are analogue.

- **Equivalence (i)** Let $N \mathrel{\mathsf{f}_{\mathsf{S}}} A$ be given, then instantiate Lemma 4.4.8, using Lemma 4.4.14 to ensure $\mathsf{id}_{\mathsf{R}}\, N \Vdash_{\mathsf{S}} N \to \mathsf{T}_N$. Otherwise, let a be given such that $\mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a$ and $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : *$ hold. From the latter and Lemma 4.4.12, as well as again Lemma 4.4.14, here for the invariant instance $\mathsf{id}_{\mathsf{R}}\, N \Vdash_{\mathsf{S}} N \leftarrow \mathsf{T}_N$, it follows that there is some A' such that $\mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A' \sim a$ and $N \mathrel{\mathsf{f}_{\mathsf{S}}} A'$. Since the type relation is injective, it follows that $A = A'$ and thus $N \mathrel{\mathsf{f}_{\mathsf{S}}} A$ as required.
- **Equivalence (iv)** Let $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : *$ and $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} b : a$ be given. Since T_N is valid and $\mathsf{id}_{\mathsf{R}}\, N$ is injective, we again directly instantiate Lemma 4.4.32 to obtain the existential, using Lemma 4.4.14 to ensure $\mathsf{id}_{\mathsf{R}}\, N \Vdash_{\mathsf{S}} N \leftarrow \mathsf{T}_N$ and Lemma 4.4.34 to ensure $\mathsf{id}_{\mathsf{R}}\, N; \bullet \Vdash_{\mathsf{S}} \bullet \leftarrow \mathsf{T}_N$. Otherwise, let s and A be given such that $\mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a$, $\mathsf{id}_{\mathsf{R}}\, N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s \approx b$, $N \mathrel{\mathsf{f}_{\mathsf{S}}} A$ and $N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s : A$ hold. From the latter and Lemma 4.4.28 it follows that there are some a' and b' satisfying $\mathsf{id}_{\mathsf{R}}\, N \mathrel{\mathsf{f}_{\mathsf{S}}} A \sim a'$, $\mathsf{id}_{\mathsf{R}}\, N; \bullet \mathrel{\mathsf{f}_{\mathsf{S}}} s \approx b'$, $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} b' : a'$ and $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a' : *$. It is again easy to provide all required invariants. Both the type and the term relation are functional, hence we have $a = a'$ and $b = b'$, and therefore $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} b : a$ and $\mathsf{T}_N \mathrel{\mathsf{f}_{\lambda}} a : *$, as required. ∎

4.5 Demo: Transfer of Results

We demonstrate the use of our correspondence result with the transfer of two metatheoretic properties, namely propagation and β-substitutivity from $\lambda_{\to}$ to STLC. We start with the former.

The proof is actually quite trivial, since we had to build a form of propagation into the two term-level preservation properties. This was necessary to precisely delineate the sensible syntactic expressions on the PTS side. Hence it should come as no surprise that PTS propagation played a key role in the proofs of judgement-preservation and all we have left to do is extract this to the STLC side.

Lemma 4.5.1 (STLC Propagation) The type of a derivable STLC judgement is always well-formed.

$$\mathsf{val}\ N;\Gamma \implies N;\Gamma \vdash_{\mathsf{S}} s : A \implies N \vdash_{\mathsf{S}} A$$

Proof Since $N;\Gamma$ is valid, Lemma 4.4.42 allows us to obtain a valid PTS context Ψ and relational contexts Θ, Σ, which satisfy

$$\begin{array}{lll} \Theta\ \mathsf{func} & \Theta \Vdash_{\mathsf{S}} N \to \Psi & \Theta;\Sigma \Vdash_{\mathsf{S}} \Gamma \to \Psi \\ \Theta\ \mathsf{inj} & \Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi & \end{array}$$

From the validity of Ψ and the first three properties it follows, by Lemma 4.4.28, that there is an a which satisfies $\Theta \vdash_{\mathsf{S}} A \sim a$ and $\Psi \vdash_{\lambda} a : *$. From $\Theta \Vdash_{\mathsf{S}} N \leftarrow \Psi$ and Lemma 4.4.12 it then follows that there is an A' satisfying $\Theta \vdash_{\mathsf{S}} A' \sim a$ and $N \vdash_{\mathsf{S}} A'$. Injectivity now yields $A = A'$, which closes the proof. ∎

Lemma 4.5.2 (STLC β-Substitutivity) STLC typing is compatible with β-substitutions.

$$\mathsf{val}\ N;\Gamma \implies N;\Gamma \vdash_{\mathsf{S}} t : B \implies N;\Gamma, B \vdash_{\mathsf{S}} s : A \implies N;\Gamma \vdash_{\mathsf{S}} s[t \cdot \mathsf{id}] : A$$

Proof The proof proceeds similar to the previous result. Let Ψ be the PTS context that corresponds via Θ and Σ to $N;\Gamma$, with all the usual invariants.

We first map the typing assumption on t with Lemma 4.4.28 to $\Psi \vdash_{\lambda} c : b$ with $\Psi \vdash_{\lambda} b : *$ as well as $\Theta \vdash_{\mathsf{S}} B \sim b$ and $\Theta;\Sigma \vdash_{\mathsf{S}} t \approx c$. This allows us to extend the invariants to $N;\Gamma, B$ and subsequently enables us to use Lemma 4.4.28 again and map the typing assumption on s to $\Psi, b \vdash_{\lambda} d : a$ with $\Psi, b \vdash_{\lambda} a : *$ as well as $\Theta^{\uparrow} \vdash_{\mathsf{S}} A \sim a$ and $\Theta^{\uparrow};\Sigma^{\Uparrow} \vdash_{\mathsf{S}} s \approx d$. From Lemma 4.4.17 it also follows that there is some a' such that $a = a'[\uparrow]$ and $\Theta \vdash_{\mathsf{S}} A \sim a'$.

Now β-substitutivity of the term relation (Corollary 4.4.41) yields

$$\Theta;\Sigma \vdash_{\mathsf{S}} s[t \cdot \mathsf{id}] \approx d[c \cdot \mathsf{id}],$$

while PTS β-substitutivity (Corollary 3.6.11) clearly derives $\Psi \vdash_{\lambda} d[c \cdot \mathsf{id}] : a[c \cdot \mathsf{id}]$, or equivalently

$$\Psi \vdash_{\lambda} d[c \cdot \mathsf{id}] : a'.$$

We can also strengthen $\Psi, b \vdash_\lambda a'[\uparrow] : *$ to

$$\Psi \vdash_\lambda a' : *.$$

We now use Lemma 4.4.32 to map these PTS typings back to the STLC side, which yields $N; \Gamma \vdash_\varsigma u : A'$ for some A' and u satisfying $\Theta \vdash_\varsigma A' \sim a'$ and respectively $\Theta; \Sigma \vdash_\varsigma u \approx d[c \cdot \mathsf{id}]$. Since both the type and the term relation are injective we know that $A' = A$ as well as $u = s[t \cdot \mathsf{id}]$, and therefore $N; \Gamma \vdash_\varsigma s[t \cdot \mathsf{id}] : A$, as required. ∎

4.6 Discussion

Now that we have seen the first correspondence proof, it makes sense to highlight a few points.

The whole construction may appear somewhat heavy, or overly verbose, when considered in isolation, and it is certainly possible that shorter, more direct constructions may exist. The reason we believe this is due to the fact that STLC is somewhat simplistic. Consider for example its inability to internalise type variables. Also due to the latter, the whole discussion of context internalisation and equivalence of closed judgements, may appear slightly forced, since the concept is only partially applicable, and as mentioned, only partially useful.

The main reason why we went through all these issues in detail pertains to the fact that we want to scale the constructions to System F. Several of our definitions were therefore phrased in a way that leads to natural extensions in Chapter 5. The concept of internalisation also becomes much more interesting in that setting as well. In addition, we will also be able to drop a few auxiliary definitions, like the special PTS context T_N and the particular relational context $\mathsf{id}_\mathsf{R}\, N$. We further observe that while the inductive correspondence relation follows the type systems which are to be aligned, it does not incorporate the PTS conversion rule. The handling of the conversion cases is instead delegated to the custom induction principles for $\lambda_\to$, which encapsulate the associated complications.

Also note that the material of Section 4.3, while introduced here, is in fact not particular to the STLC setting, but fully generic, as witnessed by a self-contained Coq development.

One of the most interesting points of this chapter is the generalisation of the CML techniques developed in Section 3.6 to the inter-system setting. This step was already suggested in [KTS17], albeit with syntactic translation functions instead of the relational approach employed here. Let us briefly summarise the essence of the CML technique in its generalised form to witness the similarities. The main goal is to proof that one judgement under some initial context entails another judgement under some final context. Furthermore, this proof should proceed by structural induction on the initial judgement, which entails a number of consequences. Firstly, at each step we need a suitable constructor in the final system, which may be obtained with a recursively defined translation function (or even the identity if both judgements use the same expression language), or, as in our case, a correspondence relation. Secondly,

the presence of contexts indicates that the judgements may be open, so we have to know what we should do with variables. This is where the morphism condition comes into play, which states that for every entry in the initial context we obtain something sensible under the final context, as mediated by a suitable substitution (and potentially a syntactic mapping or translation). We have repeatedly referred to such conditions as invariants since they have to be preserved throughout the respective inductive CML proof. For cases that involve local variable binding this leads to extension lemmas which describe how the substitution(s) and/or mapping have to be adjusted whenever the connected contexts are extended. Finally it is often interesting to observe certain instances of the resulting CML based on special instances of the morphism condition/invariant. We always get that the condition is vacuously satisfied when the initial judgement is closed, since it internally quantifies over all entries in the – now empty – initial context. Taken together, these principles are surprisingly often applicable.

With respect to the final transfer of properties, we observe that for each property we required a compatibility result which shows that the defined inductive relation preserves it. We conjecture that while our relations appear to fully capture the corresponding syntactic fragments, this will hold true for further properties in some sense. If we were to consider, for example, co-reducibility, it appears very likely that we would need to show that our relation is a bisimulation.

The variant of the simply typed λ-calculus used in this section is slightly non-standard, due to the explicit type variable context and arises from a similar treatment of System F, as for example presented in [Har13]. If we want to establish the correspondence for more traditional presentations of the system, say with only a single base type, or no tracking of the type variable context, the best approach would be to connect the desired system to the one presented here (again using a relation, should the expression language change), and then rely on the transitivity of equivalence. This approach could also be used to consider a Curry-style type system. Note that we did not cover STLC metatheory in any detail. For a thorough discussion we direct the reader to [SU06, Chapter 3] and [BDS13].

5 System F

System F extends the simply typed λ-calculus with type variables over which types can universally quantify. In addition, terms are equipped with constructions to abstract over type variables and a dedicated application mechanism that instantiates them. In conjunction these additions enable **first-class polymorphism**. This allows us to express, for example, a single identity function that operates on all types. Moreover, the **impredicative** nature of polymorphism in System F admits the emulation of most common algebraic data types [BB85]. The type of lists, where A is the element type, could for example be encoded as

$$\mathsf{list}\, A \quad :=\quad \forall X.\, (A \to X \to X) \to X \to X.$$

Despite this expressive power, we still have a rather small system with a manageable semantics. For this reason, System F often serves as a prototypical programming language. It is small enough to study in detail, while still exhibiting enough features to derive interesting insights.

There is a second story. Recall that the Curry-Howard correspondence relates STLC to intuitionistic propositional logic. In a similar vein, it connects System F to the universal fragment of second-order intuitionistic logic, where universal quantification over propositions, i.e. types, is allowed.

The original form of System F was developed by Girard [Gir72, GTL89] in the context of proof theory, that is, with a focus on the logical aspects of the language. Meanwhile, Reynolds independently invented the same system with his attempts to concisely capture the notion of parametric polymorphism in programming languages [Rey74]. In [Rey94], Reynolds later gives a nice exposition on how Girard and himself, each unaware of the others efforts, approached seemingly rather distinct issues and ended up with essentially the same system.

Based on the versatility and the dual history of the system it should come as no surprise that it has since appeared in countless texts. As a consequence, there are many smaller or larger variations on the exact formulation of the system, which are all more or less implicitly assumed to coincide. As mentioned in the introduction, the fact that, apart from Geuvers' partial proof sketch in [Geu93], this correspondence is rarely made explicit nor examined rigorously, was the original motivation for this work.

We are now going to scale the constructions of the preceding chapter to System F. The stratified, two-sorted variant that extends STLC is PLC, the polymorphic λ-calculus. The concrete presentation we use here is taken from [Har13]. The corresponding PTS, on the other hand, is $\lambda 2$, which similarly extends $\lambda_{\to}$. Note that $\lambda 2$, just like $\lambda_{\to}$, is a corner of Barendregt's λ-cube (see Section 3.9).

In the following we are going to present and discuss the two systems and their de Bruijn representations with a particular focus on the changes with respect to their simply typed sub-systems. We then adapt the correspondence proof of Chapter 4 to the present setting. Since most constructions carry over we do not go over every proof in detail but instead focus on those aspects that change due to the presence of polymorphism.

One major difference is the ability of System F to abstract over type variables, which allows it to fully internalise the contexts of derivable judgements. We will briefly discuss, how this is established for our variants PLC and $\lambda 2$.

5.1 PLC

The system PLC is stratified, or two-sorted, just like STLC. That is, we syntactically distinguish between terms and types. As informally discussed above, we extend the two syntactic classes of Definition 4.1.1 with one new type constructor for universal types and two new term constructors for type abstraction and application to arrive at the following definition.

Definition 5.1.1 (PLC Syntax) The de Bruijn syntax of types and terms of the polymorphic λ-calculus is given by the following grammar.

$$\boxed{\mathsf{Ty_P}} \qquad A, B ::= n \mid A \to B \mid \forall.\, A \qquad n : \mathbb{N}$$

$$\boxed{\mathsf{Tm_P}} \qquad s, t ::= n \mid s\, t \mid \lambda A.\, s \mid s\, A \mid \Lambda.\, s \qquad n : \mathbb{N}$$

We observe that two of the three new constructors, namely $\forall.\, A$ and $\Lambda.\, s$, are binders. Both bind type variables, and since type variables do not have a type of their own, that is a *kind*, we do not get any form of kind ascription on these binders. This is where the convention of denoting the presence of the binder with a dot becomes most useful, as it makes it easy to spot the three binding constructions.

We are again dealing with two variable scopes, and in contrast to STLC, we now have binders for both. As a consequence, instantiation becomes more involved. In the following, some of the constructions need to be disambiguated by the types of occurring parallel substitutions, so we will make variable constructors explicit to aid with this task.

Let us again first consider instantiation for types, which, due to the universal types, exhibits the familiar mutual recursion with forward substitution composition. We define the operation $A[\tau]$ for $\tau : \mathbb{N} \to \mathsf{Ty_P}$ as follows.

$$\begin{aligned} \mathsf{V}_n[\tau] &:= \tau\, n & (\tau \circ \tau')\, n &:= (\tau\, n)[\tau'] \\ (A \to B)[\tau] &:= A[\tau] \to B[\tau] & & \\ (\forall.\, A)[\tau] &:= \forall.\, A[\Uparrow \tau] & \Uparrow \tau &:= \mathsf{V}_0 \cdot \tau \circ \uparrow \end{aligned}$$

As an interesting aside: observe how the recursion structure of instantiation for PLC types is identical to that of instantiation for STLC terms.

For PLC terms we are now faced with the situation that types, and therefore type variables can occur in various positions in terms and moreover, terms can abstract over such type variables. This, in turn, will lead to situations where we have to talk about terms which are instantiated with both non-trivial term and type substitutions. To facilitate this we are going to work with **vector substitutions**, which we introduced in [KSS17]. The main idea is to not only instantiate all variables of a single scope but to also simultaneously handle all occurring variable scopes. To this end, instantiation for terms s takes a vector (τ, σ) of substitutions, with one component for each variable scope, here types and terms respectively, and again recursively traverses the term. We write $s[\tau, \sigma]$ and define the operation mutually recursive with forward composition as follows.

$$
\begin{aligned}
\mathsf{v}_n[\tau,\sigma] &\coloneqq \sigma\, n & (\sigma \circ (\tau,\sigma'))\, n &\coloneqq (\sigma\, n)[\tau,\sigma'] \\
(s\, t)[\tau,\sigma] &\coloneqq s[\tau,\sigma]\, t[\tau,\sigma] & & \\
(\lambda A.\, s)[\tau,\sigma] &\coloneqq \lambda A[\tau].\, s[\Uparrow_{\mathsf{tm}}(\tau,\sigma)] & \Uparrow_{\mathsf{tm}}(\tau,\sigma) &\coloneqq (\tau \circ \mathsf{id}, \mathsf{v}_0 \cdot \sigma \circ (\mathsf{id}, \uparrow)) \\
(s\, A)[\tau,\sigma] &\coloneqq s[\tau,\sigma]\, A[\tau] & & \\
(\Lambda.\, s)[\tau,\sigma] &\coloneqq \Lambda.\, s[\Uparrow_{\mathsf{ty}}(\tau,\sigma)] & \Uparrow_{\mathsf{ty}}(\tau,\sigma) &\coloneqq (\mathsf{V}_0 \cdot \tau \circ \uparrow, \sigma \circ (\uparrow, \mathsf{id}))
\end{aligned}
$$

The definitions of $\Uparrow_{\mathsf{tm}}$ and $\Uparrow_{\mathsf{ty}}$ may look somewhat verbose but they do exhibit a rather regular pattern. Note how only the component that receives a fresh binding is accordingly extended with a consed-on 0 (of the correct scope), while all components receive a post-composed renaming to adjust for the presence of a new binder. The fact that this latter renaming happens to be id for the type component in $\Uparrow_{\mathsf{tm}}$ is an artefact of the recursion structure of the PLC grammar.

We again obtain the instantiation operations and their equational theory from the Autosubst library. Note that our development uses the old version 1 of Autosubst. For type instantiation, the definition above exactly corresponds to the library output, while for terms we have taken the liberty to present the form of instantiation that is generated by Autosubst 2 [SSK19]. The latter is, in our opinion, more principled and elegant.

However, for the sake of comparison and to keep our exposition grounded in the underlying formalisation, we will briefly present the form of term instantiation that the version 1 of Autosubst generates. Since the library is not yet aware of the notion of vector substitutions it instead employs certain ad hoc constructions to handle multi-variate syntactic sorts. For our present PLC setting we obtain two separate instantiation operations for terms, $s[\sigma]$ and $s[\![\tau]\!]$, which respectively instantiate terms and types. They commute according to

$$
s[\sigma][\![\tau]\!] = s[\![\tau]\!][\sigma \mathbin{\hat{\circ}} \tau] \qquad\qquad (\sigma \mathbin{\hat{\circ}} \tau)\, n \coloneqq (\sigma\, n)[\![\tau]\!]
$$

where $\sigma \hat{\circ} \tau$ is referred to as heterogeneous forward composition. Since we have separate instantiation operations they are each recursively defined over the term syntax. The two definitions themselves are not very instructive and it is more interesting to observe

their reduction behaviour when considered in conjunction.

$$\begin{aligned}
\mathsf{v}_n\llbracket\tau\rrbracket[\sigma] &= \sigma\, n \\
(s\, t)\llbracket\tau\rrbracket[\sigma] &= s\llbracket\tau\rrbracket[\sigma]\, t\llbracket\tau\rrbracket[\sigma] \\
(\lambda A.\, s)\llbracket\tau\rrbracket[\sigma] &= \lambda A[\tau].\, s\llbracket\tau\rrbracket[\Uparrow\sigma] \\
(s\, A)\llbracket\tau\rrbracket[\sigma] &= s\llbracket\tau\rrbracket[\sigma]\, A[\tau] \\
(\Lambda.\, s)\llbracket\tau\rrbracket[\sigma] &= \Lambda.\, s\llbracket\Uparrow\tau\rrbracket[\sigma \mathbin{\hat{\circ}} \uparrow]
\end{aligned}$$

Note that the older, heterogeneous definitions and the vector definition satisfy the following equality.

$$s\llbracket\tau\rrbracket[\sigma] = s[\tau, \sigma]$$

Let us now turn our attention to the PLC type system, which is based on the STLC type system and therefore also consists of two judgements for type formation and typing respectively. PLC type formation is slightly more interesting than its STLC counterpart, since the set of admissible type variables may now change in the course of a typing derivation.

Definition 5.1.2 (PLC Typing) The PLC type system consists of a type formation judgement $N \vDash_{\mathsf{P}} A$ and a typing judgement $N; \Gamma \vDash_{\mathsf{P}} s : A$, where N is a de Bruijn type variable context and Γ is a de Bruijn term variable context. The judgements are inductively characterised by the rules of Figure 5.1, where the three rightmost rules (set in blue) are new with respect to STLC typing (c.f. Figure 4.2).

While this definition is mostly a direct extension of the STLC type system, it makes sense to briefly take a closer look at the two abstraction rules.

$$\frac{N; \Gamma, A \vDash_{\mathsf{P}} s : B \qquad N \vDash_{\mathsf{P}} A}{N; \Gamma \vDash_{\mathsf{P}} \lambda A.\, s : A \to B} \qquad \frac{N + 1; \Gamma[\uparrow] \vDash_{\mathsf{P}} s : A}{N; \Gamma \vDash_{\mathsf{P}} \Lambda.\, s : \forall.\, A}$$

Observe the differences in how the contexts change when we descend into the body of the two abstractions. For ordinary term abstractions we simply add the annotated type A to the front of the term context, while we keep the type context stable. We recall that this also takes care of the implicit shifting of the term-level de Bruijn scope. Meanwhile, for type abstractions, we have to account for a new type variable, encoded as V_0, while all other type variables are shifted. This is reflected in the incremented type variable bound $N + 1$, as well as the shift, which is mapped over Γ. The latter is necessary to preserve the meaning of the term variable context, since it contains types, and therefore type variables, whose interpretation has just changed.

Recall that for STLC we had to derive a renaming CML for type formation for the sole purpose of establishing the preservation of context validity under extension with a new type variable. For PLC, the renaming version, as well as the full CML play a much more prominent role, since the new rule for type specialisation,

$$\frac{N; \Gamma \vDash_{\mathsf{P}} s : \forall.\, B \qquad N \vDash_{\mathsf{P}} A}{N; \Gamma \vDash_{\mathsf{P}} s\, A : B[A \cdot \mathsf{id}]}$$

$$\frac{n < N}{N \vDash_{\mathsf{P}} n} \qquad \frac{N \vDash_{\mathsf{P}} A \qquad N \vDash_{\mathsf{P}} B}{N \vDash_{\mathsf{P}} A \to B} \qquad \frac{N+1 \vDash_{\mathsf{P}} A}{N \vDash_{\mathsf{P}} \forall.\, A}$$

$$\frac{\Gamma_n = A \qquad N \vDash_{\mathsf{P}} A}{N; \Gamma \vDash_{\mathsf{P}} n : A} \qquad \frac{N; \Gamma, A \vDash_{\mathsf{P}} s : B \qquad N \vDash_{\mathsf{P}} A}{N; \Gamma \vDash_{\mathsf{P}} \lambda A.\, s : A \to B} \qquad \frac{N+1; \Gamma[\uparrow] \vDash_{\mathsf{P}} s : A}{N; \Gamma \vDash_{\mathsf{P}} \Lambda.\, s : \forall.\, A}$$

$$\frac{N; \Gamma \vDash_{\mathsf{P}} s : A \to B \qquad N; \Gamma \vDash_{\mathsf{P}} t : A}{N; \Gamma \vDash_{\mathsf{P}} s\, t : B} \qquad \frac{N; \Gamma \vDash_{\mathsf{P}} s : \forall.\, B \qquad N \vDash_{\mathsf{P}} A}{N; \Gamma \vDash_{\mathsf{P}} s\, A : B[A \cdot \mathsf{id}]}$$

Figure 5.1: The PLC de Bruijn type system.

attaches the β-substitution $A \cdot \mathsf{id}$ to the result type. Since both the renaming and the full CML for PLC type formation exactly follow the standard pattern we present them here in condensed form.

Lemma 5.1.3 (Renaming CML for PLC Type Formation) The PLC type formation judgement is compatible with instantiation with context renamings, that is

$$\frac{N \vDash_{\mathsf{P}} A \qquad \xi \Vdash_{\mathsf{P}} N \xrightarrow{r} M}{M \vDash_{\mathsf{P}} A[\xi]}$$

is admissible, where the notion of context renaming is defined as

$$\xi \Vdash_{\mathsf{P}} N \xrightarrow{r} M \quad :\equiv \quad \forall n < N.\ \xi\, n < M.$$

Proof By induction on $N \vDash_{\mathsf{P}} A$, using the following extension property for the binder case.

$$\xi \Vdash_{\mathsf{P}} N \xrightarrow{r} M \implies \Uparrow \xi \Vdash_{\mathsf{P}} N+1 \xrightarrow{r} M+1$$

This follows by case analysis on the quantified index n and basic arithmetic. ∎

Lemma 5.1.4 (Weakening for PLC Type Formation) The weakening rule

$$\frac{N \vDash_{\mathsf{P}} A}{N+1 \vDash_{\mathsf{P}} A[\uparrow]}$$

is admissible.

Proof Instantiate Lemma 5.1.3 with $\uparrow \Vdash_{\mathsf{P}} N \xrightarrow{r} N+1$, which trivially holds. ∎

Lemma 5.1.5 (Full CML for PLC Type Formation) PLC type formation is compatible with instantiation with context morphisms, that is

$$\frac{N \vDash_{\mathsf{P}} A \qquad \sigma \Vdash_{\mathsf{P}} N \to M}{M \vDash_{\mathsf{P}} A[\sigma]}$$

is admissible, where the notion of context morphism is defined as

$$\sigma \Vdash_{\mathsf{p}} N \to M \quad := \quad \forall n < N.\ M \vdash_{\mathsf{p}} \sigma\, n.$$

Proof By induction on $N \vdash_{\mathsf{p}} A$, using the following extension property for the binder case.

$$\sigma \Vdash_{\mathsf{p}} N \to M \implies \Uparrow\sigma \Vdash_{\mathsf{p}} N + 1 \to M + 1$$

This follows by case analysis on the quantified index n and Lemma 5.1.4. ∎

Lemma 5.1.6 (PLC Type Formation β-Substitutivity) PLC type formation is compatible with β-substitutions.

$$N \vdash_{\mathsf{p}} A \implies N + 1 \vdash_{\mathsf{p}} B \implies N \vdash_{\mathsf{p}} B[A \cdot \mathsf{id}]$$

Proof Instantiate Lemma 5.1.5 with $A \cdot \mathsf{id} \Vdash_{\mathsf{p}} N+1 \to N$, which holds by case analysis on the quantified index n. For the case $n = 0$ we need the assumption $N \vdash_{\mathsf{p}} A$. ∎

For context validity we use a definition that is very close to the one we used for STLC. The only difference is a finer resolution with respect to type variable contexts, which simplifies later inductions on validity.

$$\frac{}{\mathsf{val}\ 0; \bullet} \qquad \frac{\mathsf{val}\ N; \bullet}{\mathsf{val}\ N + 1; \bullet} \qquad \frac{\mathsf{val}\ N; \Gamma \qquad N \vdash_{\mathsf{p}} A}{\mathsf{val}\ N; \Gamma, A}$$

The equivalence of this definition and a direct adaptation of the STLC version is due to the following fact.

Fact 5.1.7 $\mathsf{val}\ N; \bullet$

The astute reader may have noticed a slight mismatch of our definition of validity and the possible context extensions which appear in Figure 5.1. In particular, we do not (yet) know whether the extension

$$N; \Gamma \quad \rightsquigarrow \quad N + 1; \Gamma[\uparrow],$$

which arise from the type abstraction rule

$$\frac{N + 1; \Gamma[\uparrow] \vdash_{\mathsf{p}} s : A}{N; \Gamma \vdash_{\mathsf{p}} \Lambda.\, s : \forall.\, A}$$

preserves context validity. This, however, is easy to justify. We again rely on the CML method, this time for the validity predicate.

Lemma 5.1.8 (Renaming CML for PLC Context Validity) The following rule is admissible.

$$\frac{\mathsf{val}\ N; \Gamma \qquad \xi \Vdash_{\mathsf{p}} N \xrightarrow{r} M}{\mathsf{val}\ M; \Gamma[\xi]}$$

Proof By induction on $\mathsf{val}\ N;\Gamma$. For the two cases with $\Gamma = \bullet$ we have $\bullet[\xi] = \bullet$ and can hence use Fact 5.1.7. For the third case we have $(\Gamma, A)[\xi] = \Gamma[\xi], A[\xi]$ where we use the inductive hypothesis for Γ and additionally need to infer $M \Vdash_{\mathsf{P}} A[\xi]$ from $N \Vdash_{\mathsf{P}} A$. The latter follows with Lemma 5.1.3 and $\xi \Vdash_{\mathsf{P}} N \xrightarrow{r} M$. ∎

Lemma 5.1.9 Validity is preserved under extension with a new type variable for arbitrary term variable contexts Γ.

$$\frac{\mathsf{val}\ N;\Gamma}{\mathsf{val}\ N+1;\Gamma[\uparrow]}$$

Proof Trivial instance of Lemma 5.1.8. ∎

This brings us to the notion of context internalisation, which for PLC is a more sensible concept than it was for STLC. This is due to the fact that both the term and the type variable context can be internalised with the following recursive function. Note that for technical reasons we decompose this definition in the formalisation into two functions which deal with the two contexts separately. The recursion structure is the same, though.

$$\begin{aligned}
\mathsf{intern}\,(0;\bullet)\,s\,A &:= (s, A)\\
\mathsf{intern}\,(N+1;\bullet)\,s\,A &:= \mathsf{intern}\,(N;\bullet)\,(\Lambda.\,s)\,(\forall.\,A)\\
\mathsf{intern}\,(N;\Gamma, C)\,s\,A &:= \mathsf{intern}\,(N;\Gamma)\,(\lambda C.\,s)\,(C \to A)
\end{aligned}$$

Lemma 5.1.10 (PLC Context Internalisation) The internalisation function intern yields, for a given judgement, a term and a type which allow the construction of a closed judgement, such that the new judgement and the input judgement are equi-derivable.

$$\begin{aligned}
&\mathsf{val}\ N;\Gamma \implies \mathsf{intern}\,(N;\Gamma)\,s\,A = (t, B) \implies\\
&\qquad (N \Vdash_{\mathsf{P}} A \iff 0 \Vdash_{\mathsf{P}} B) \wedge (N;\Gamma \Vdash_{\mathsf{P}} s : A \iff 0;\bullet \Vdash_{\mathsf{P}} t : B)
\end{aligned}$$

Proof By induction on the derivation of $\mathsf{val}\ N;\Gamma$. Analogue to Lemma 4.1.7. For the new internalisation of a type variable we additionally have to establish

$$N+1;\bullet \Vdash_{\mathsf{P}} s : A \iff N;\bullet \Vdash_{\mathsf{P}} \Lambda.\,s : \forall.\,A,$$

which follows from the typing rules and the fact that $\bullet[\uparrow] = \bullet$. ∎

5.2 The PTS λ2

Let $\mathcal{P}_2 = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a PTS specification with the following components.

$$\begin{array}{lcl} \mathcal{S} & \coloneqq & \{*, \Box\} \\ \mathcal{A} & \coloneqq & \{(*, \Box)\} \\ \mathcal{R} & \coloneqq & \{(*,*,*), (\Box,*,*)\} \end{array}$$

We refer to the resulting PTS as $\lambda 2$. Note that the only difference over $\mathcal{P}_\to$ is the extra rule $(\Box, *, *)$, or $(\Box, *)$ in the abbreviated notation of the λ-cube. We are still not able to construct dependent function types in $\Box$, and both universes are still only inhabited by normal terms. We recall the following basic structural results.

Fact 5.2.1 $\Psi \vdash_\lambda \Box : a \implies \bot$

Fact 5.2.2 $\Psi \vdash_\lambda * : * \implies \bot$

Fact 5.2.3 $\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : \Box \implies a = *$

Fact 5.2.4 $\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : \mathsf{s} \implies \lfloor a \rfloor$

Fact 5.2.5 $\mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : b \implies \lfloor b \rfloor$

The new rule $(\Box, *)$ admits the formation of function types of the form $\Pi *.\, a$ with corresponding abstractions $\lambda *.\, b$. That is, we now admit the binding of variables in $*$. Recall from our STLC discussion that we identified the inhabitants of $*$ as the semantic types and therefore the corresponding variables in $*$ as type variables. This suggests that $\lambda 2$ is equipped to express the type variable abstractions which constitute an integral part of System F. There is, however, one caveat: the function type, abstraction and application constructions all now serve dual purposes, which are not clearly syntactically distinguished. Separating these notions will be the key objective throughout the lifting of our STLC results to the System F setting. The first place where this concern comes into play is the construction of the induction principles for semantic type formation and semantic typing.

Lemma 5.2.6 (λ2 Induction Principle for Type Formation.) Let Q be a binary property of a PTS context and a PTS term. Then in order to prove that Q holds for all Ψ-types, it is sufficient to only consider type variables and two particular forms of function types, as these are the only possible constructions of $\lambda 2$ types. This leads to the following admissible induction principle for $\lambda 2$ types.

$$\begin{array}{ll} (\forall \Psi n.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\forall n : * \implies Q\, \Psi\, n) \implies & (\mathrm{H}_V) \\ (\forall \Psi a b.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : * \implies \Psi, a \vdash_\lambda b : * \implies & \\ \qquad Q\, \Psi\, a \implies Q\, (\Psi, a)\, b \implies Q\, \Psi\, (\Pi a.\, b)) \implies & (\mathrm{H}_\to) \\ (\forall \Psi b.\ \mathsf{val}\ \Psi \implies \Psi, * \vdash_\lambda b : * \implies Q\, (\Psi, *)\, b \implies Q\, \Psi\, (\Pi *.\, b)) \implies & (\mathrm{H}_\forall) \\ (\forall \Psi a.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : * \implies Q\, \Psi\, a) & \end{array}$$

Note that in the premise H_V, similar to the corresponding STLC result, the n in $Q\,\Psi\,n$ is a PTS term, while the n in $\Psi \vdash_{\mathsf{V}} n : *$ is a plain index.

Proof Assume H_V, $\mathrm{H}_\to$ and the new premise $\mathrm{H}_\forall$. Then by induction on the derivation of $\Psi \vdash_\lambda a : *$, where most cases are identical to the proof of Lemma 4.2.6.

The only difference occurs for the case $a = \Pi c.\, d$ with some s_1, s_2 such that $(\mathsf{s}_1, \mathsf{s}_2, *) \in \mathcal{R}$, as well as $\Psi \vdash_\lambda c : \mathsf{s}_1$ and $\Psi, c \vdash_\lambda d : \mathsf{s}_2$. We clearly have $\mathsf{s}_2 = *$ but s_1 could be either $*$ or $\Box$. In the first case we proceed as before and close the case with $\mathrm{H}_\to$. Otherwise we can use Fact 5.2.3 to infer $c = *$ and then close the case with $\mathrm{H}_\forall$, which completes the proof. ∎

Observe how we considered all rules $(\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3) \in \mathcal{R}$ that may have admitted the construction of a given function type. In each of the resulting cases, the universe of the domain of the function type is known, which in turn tells us something about the nature of the domain itself, and thus allows us to proceed with steps that are only applicable to the respective case. This form of case analysis on the universe of a function type domain will be a recurring pattern throughout the remainder of this chapter. Take for example the induction principle for typing which complements the preceding result and applies the technique twice, once for abstractions and once for applications.

Lemma 5.2.7 (λ2 Induction Principle for Typing.) Let Q be a ternary property of one PTS context and two PTS terms. Then in order to prove that Q holds for all Ψ-terms and their corresponding Ψ-types, it is sufficient to only consider term variables, as well as two particular forms of abstraction and respectively application, as these are the only possible constructions of $\lambda 2$ terms. This leads to the following admissible induction principle for $\lambda 2$ terms.

$$(\forall \Psi n a.\ \mathsf{val}\ \Psi \implies \Psi \vdash_{\mathsf{V}} n : a \implies \Psi \vdash_\lambda a : * \implies Q\,\Psi\,n\,a) \implies \tag{H_V}$$

$$(\forall \Psi a b c d d'.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : \Pi c.\, d \implies \Psi \vdash_\lambda c : * \implies \Psi, c \vdash_\lambda d : * \implies \Psi \vdash_\lambda b : c \implies Q\,\Psi\,a\,(\Pi c.\, d) \implies Q\,\Psi\,b\,c \implies d' = d[b \cdot \mathsf{id}] \implies Q\,\Psi\,(a\,b)\,d') \implies \tag{H_A}$$

$$(\forall \Psi a b c.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : * \implies \Psi, a \vdash_\lambda c : * \implies \Psi, a \vdash_\lambda b : c \implies Q\,(\Psi, a)\,b\,c \implies Q\,\Psi\,(\lambda a.\, b)\,(\Pi a.\, c)) \implies \tag{H_λ}$$

$$(\forall \Psi a b d d'.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda a : \Pi *.\, d \implies \Psi, * \vdash_\lambda d : * \implies \Psi \vdash_\lambda b : * \implies Q\,\Psi\,a\,(\Pi *.\, d) \implies d' = d[b \cdot \mathsf{id}] \implies Q\,\Psi\,(a\,b)\,d') \implies \tag{H_S}$$

$$(\forall \Psi b c.\ \mathsf{val}\ \Psi \implies \Psi, * \vdash_\lambda c : * \implies \Psi, * \vdash_\lambda b : c \implies Q\,(\Psi, *)\,b\,c \implies Q\,\Psi\,(\lambda *.\, b)\,(\Pi *.\, c)) \implies \tag{H_Λ}$$

$$(\forall \Psi a b.\ \mathsf{val}\ \Psi \implies \Psi \vdash_\lambda b : * \implies \Psi \vdash_\lambda a : b \implies Q\,\Psi\,a\,b)$$

Proof Assume H_V, H_A and H_λ, as well as the new premises H_S and H_Λ. Then by induction on the derivation of $\Psi \vdash_\lambda a : b$, where most cases are identical to the proof of Lemma 4.2.7. The application and abstraction case each require extra attention.

For the application case we know $\Psi \vdash_\lambda a : \Pi c.\, d$ and hence by Lemma 3.6.13 infer there is a rule $(\mathsf{s}_1, *, *)$ which admitted the formation of $\Pi c.\, d$. For $\mathsf{s}_1 = *$, H_A easily solves the case. For $\mathsf{s}_1 = \Box$, we again use Fact 5.2.3 to infer $c = *$ and then proceed with H_S.

Similarly, for the abstraction case we proceed with H_λ when the domain of $\lambda a.\, b$ and $\Pi a.\, c$ is a type. For $a = *$ we use H_Λ instead. ∎

Recall that all systems of the λ-cube are functional and hence admit the generic strengthening result (Theorem 3.8.8). This of course also applies to $\lambda 2$ and we can again formulate the restricted variant for Ψ-types:

Fact 5.2.8 (Strengthening for $\lambda 2$ Type Formation)

$$\mathsf{val}\ \Psi, a \implies \Psi, a \vdash_\lambda b[\uparrow] : * \implies \Psi \vdash_\lambda b : *$$

The last aspect of $\lambda 2$ prior to the full correspondence proof is the issue of context internalisation. For $\lambda_\to$ we did not give a formal implementation due to technical complications. Here, however, the construction is not only feasible, but about as elegant as one might hope for. We first define our internalisation by recursion on the PTS context Ψ without semantically distinguishing the context entries.

$$\begin{aligned} \mathsf{intern}\ \bullet\ a\, b &:= (a, b) \\ \mathsf{intern}\ (\Psi, c)\, a\, b &:= \mathsf{intern}\ \Psi\ (\lambda c.\, a)\ (\Pi c.\, b) \end{aligned}$$

Lemma 5.2.9 ($\lambda 2$ Context Internalisation) The internalisation function intern yields equi-derivable judgements.

$$\mathsf{val}\ \Psi \implies \mathsf{intern}\ \Psi\, a\, b = (c, d) \implies (\Psi \vdash_\lambda b : * \wedge \Psi \vdash_\lambda a : b \iff \bullet \vdash_\lambda d : * \wedge \bullet \vdash_\lambda c : d)$$

Proof By induction on the derivation of $\mathsf{val}\ \Psi$, where the base case is trivial.

For the step it is sufficient to show

$$\Psi, e \vdash_\lambda b : * \wedge \Psi, e \vdash_\lambda a : b \iff \Psi \vdash_\lambda \Pi e.\, b : * \wedge \Psi \vdash_\lambda \lambda e.\, a : \Pi e.\, b,$$

where we additionally know that $\Psi \vdash_\lambda e : \mathsf{s}$ for some sort s. The implication from left to right clearly follows from the PTS typing rules, since $(\mathsf{s}, *, *) \in \mathcal{R}$ for both instances of s.

For the inverse direction we can infer $\Psi, e \vdash_\lambda b : *$ from $\Psi \vdash_\lambda \lambda e.\, a : \Pi e.\, b$ and Lemma 3.6.13. From $\Psi \vdash_\lambda \Pi e.\, b : *$ and $\Psi \vdash_\lambda \lambda e.\, a : \Pi e.\, b$ we can further infer by stripping (Lemma 3.5.6), that there is some sort s' and term b', such that $\Psi, e \vdash_\lambda b : \mathsf{s}'$, $\Psi, e \vdash_\lambda a : b'$ and $\Pi e.\, b \equiv \Pi e.\, b'$. Thus by Lemma 3.3.30 we have $b \equiv b'$ and can close the case with the PTS conversion rule. ∎

5.3 Proof of Correspondence

We now prove the correspondence of PLC and $\lambda 2$ where the semantic disambiguation of certain PTS terms is the main challenge. In our opinion, the most elegant way to achieve this is to build the correct distinctions directly into the inductive correspondence relation. We reuse the treatment of related de Bruijn indices developed in Section 4.3 and then extend Definitions 4.4.1 and 4.4.2 as well as the rules of Figure 4.3 to incorporate the new syntactic constructions of System F (again highlighted in blue).

Definition 5.3.1 (System F Correspondence Relation for Types) Let Θ be a relational type variable context. Then a PLC type A and a (semantic) $\lambda 2$ type a are related whenever $\Theta \vDash_{\mathsf{P}} A \sim a$ is derivable from the rules in Figure 5.2.

Definition 5.3.2 (System F Correspondence Relation for Terms) Let Θ and Σ be relational type and term variable contexts respectively. Then a PLC term s and a (semantic) $\lambda 2$ term b are related whenever $\Theta; \Sigma \vDash_{\mathsf{P}} s \approx b$ is derivable from the rules in Figure 5.2.

We will follow the familiar proof structure and establish that the defined relations are functional and injective as well as right- and left-total and judgement-preserving for the well-formed and -typed language fragments. Some of these results are necessarily more involved due to the fact that both relations now have internal structural overlap on the PTS side. There are for example two $\sim$ rules with a dependent function type on the right. Another complication stems from the application rules: when we have $a\,b$ on the PTS side, then this either corresponds to standard application $s\,t$, provided that we can relate b to a PLC term t, or to a type application $s\,B$, when b relates to a PLC type B. It will be crucial to establish that only one of the two scenarios is possible, but not both, which in turn requires us to lift disjointness of codomains from the relational variable contexts to the type and term relations under such contexts. We would also like to point out that for STLC we only applied *skewing* to the relational type variable context and *extension* to the relational term variable context. Here we also get the inverse constructions, that is skewing of the term context and extension of the type context.

The four required preservation proofs again depend on CML-style invariants, which we introduce as we go along. We do, however, again provide a summary of these invariants in Figure 5.3 for reference. Note the structural similarities to the STLC invariants shown in Figure 4.4.

Let us now prove the four properties, first for the correspondence at the type level and then later also for the terms.

5.3.1 Four Properties for $\sim$

We start with functionality, which scales without any surprises, and injectivity which requires some added effort.

$$\frac{\Theta \vdash n \simeq m}{\Theta \models_{\mathsf{P}} n \sim m} \qquad \frac{\Theta \models_{\mathsf{P}} A \sim a \qquad \Theta^{\uparrow} \models_{\mathsf{P}} B \sim b}{\Theta \models_{\mathsf{P}} A \to B \sim \Pi a.\, b} \qquad \frac{\Theta^{\Uparrow} \models_{\mathsf{P}} B \sim b}{\Theta \models_{\mathsf{P}} \forall.\, B \sim \Pi *.\, b}$$

$$\frac{\Sigma \vdash n \simeq m}{\Theta; \Sigma \models_{\mathsf{P}} n \approx m} \qquad \frac{\Theta; \Sigma \models_{\mathsf{P}} s \approx a \qquad \Theta; \Sigma \models_{\mathsf{P}} t \approx b}{\Theta; \Sigma \models_{\mathsf{P}} s\, t \approx a\, b} \qquad \frac{\Theta \models_{\mathsf{P}} A \sim a \qquad \Theta^{\uparrow}; \Sigma^{\Uparrow} \models_{\mathsf{P}} s \approx b}{\Theta; \Sigma \models_{\mathsf{P}} \lambda A.\, s \approx \lambda a.\, b}$$

$$\frac{\Theta; \Sigma \models_{\mathsf{P}} s \approx a \qquad \Theta \models_{\mathsf{P}} B \sim b}{\Theta; \Sigma \models_{\mathsf{P}} s\, B \approx a\, b} \qquad \frac{\Theta^{\Uparrow}; \Sigma^{\uparrow} \models_{\mathsf{P}} s \approx b}{\Theta; \Sigma \models_{\mathsf{P}} \Lambda.\, s \approx \lambda *.\, b}$$

Figure 5.2: Inductive correspondence relation for System F.

$$\Theta \Vdash_{\mathsf{P}} N \to \Psi := \forall n < N.\ \exists m.\ \Theta \vdash n \simeq m \wedge \Psi \models_{\mathsf{V}} m : * \qquad \text{(Def. 5.3.5)}$$

$$\Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi := \forall m.\ \Psi \models_{\mathsf{V}} m : * \implies \exists n.\ \Theta \vdash n \simeq m \wedge n < N \qquad \text{(Def. 5.3.9)}$$

$$\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi := \forall n A.\ \Gamma_n = A \implies \exists m a.\ \Theta \models_{\mathsf{P}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Psi \models_{\mathsf{V}} m : a \qquad \text{(Def. 5.3.28)}$$

$$\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi := \forall m a.\ \Psi \models_{\mathsf{V}} m : a \implies \Gamma \models_{\lambda} a : * \implies \exists n A.\ \Theta \models_{\mathsf{P}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Gamma_n = A \qquad \text{(Def. 5.3.32)}$$

Figure 5.3: System F preservation invariants.

Lemma 5.3.3 (Functionality of $\Theta \models_{\mathsf{P}} A \sim a$) The type relation is functional, provided that Θ is a functional relation.

$$\Theta\ \mathsf{func} \implies \Theta \models_{\mathsf{P}} A \sim a_1 \implies \Theta \models_{\mathsf{P}} A \sim a_2 \implies a_1 = a_2$$

Proof By induction on the derivation of $\Theta \models_{\mathsf{P}} A \sim a_1$ and discriminating on the derivation of $\Theta \models_{\mathsf{P}} A \sim a_2$. The cases for type variables and function types are as in Lemma 4.4.3. For the quantification case we need the preservation of context functionality under extension (Fact 4.3.11). ∎

Lemma 5.3.4 (Injectivity of $\Theta \models_{\mathsf{P}} A \sim a$) The type relation is injective, provided that Θ is an injective relation.

$$\Theta\ \mathsf{inj} \implies \Theta \models_{\mathsf{P}} A_1 \sim a \implies \Theta \models_{\mathsf{P}} A_2 \sim a \implies A_1 = A_2$$

Proof We proceed as for Lemma 4.4.4 by induction on the derivation of $\Theta \models_{\mathsf{P}} A_1 \sim a$ and then discriminate on the derivation of $\Theta \models_{\mathsf{P}} A_2 \sim a$. We have to consider a total of five cases, where two are in fact impossible and caused by the internal structural overlap in the type relation. The variable case again follows from the injectivity of Θ and for the correctly aligned function type and quantification cases we simply rely on the preservation of injectivity under skewing and, respectively, extension (Fact 4.3.11).

In the remaining two cases we know that the domain a of the function type is related to some PLC type A, but we also have $a = *$. This is impossible, since there is no rule that could have derived $\Theta \vDash_{\mathsf{P}} A \sim *$, which allows us to refute these cases and close the proof. ∎

The two preservation results again each take the form of a CML, so we give for each case the respective invariant, the extension laws and the result itself. Recall that for STLC, we gave two extension principles for each invariant, but only used one of them in the respective CML proof while the other was required at a much later stage. Here, both extension principles are immediately needed in the CML proof, since the context is both skewed and extended. The invariants and the respective extension laws are essentially identical to those used for STLC, so we only state them without giving the respective proofs.

Definition 5.3.5 (Type Formation Invariant: PLC to $\lambda 2$)

$$\Theta \Vdash_{\mathsf{P}} N \to \Psi \quad :\Longleftrightarrow \quad \forall n < N.\ \exists m.\ \Theta \vdash n \simeq m \wedge \Psi \vDash_{\mathsf{V}} m : *$$

Fact 5.3.6 (Skewing Law)

$$\Theta \Vdash_{\mathsf{P}} N \to \Psi \implies \Theta^{\uparrow} \Vdash_{\mathsf{P}} N \to \Psi, a$$

Fact 5.3.7 (Extension Law)

$$\Theta \Vdash_{\mathsf{P}} N \to \Psi \implies \Theta^{\Uparrow} \Vdash_{\mathsf{P}} N + 1 \to \Psi, *$$

Lemma 5.3.8 (Left-Totality and Preservation of $\Theta \vDash_{\mathsf{P}} A \sim a$) For every well-formed PLC type A there exists a corresponding Ψ-type a in $\lambda 2$.

$$N \vDash_{\mathsf{P}} A \implies \forall \Theta \Psi.\ \Theta \Vdash_{\mathsf{P}} N \to \Psi \implies \exists a.\ \Theta \vDash_{\mathsf{P}} A \sim a \wedge \Psi \vDash_{\lambda} a : *$$

Proof By induction on the derivation of $N \vDash_{\mathsf{P}} A$, using the invariant for the variable case, Fact 5.3.6 for the implication case and Fact 5.3.7 for the quantification case. ∎

Definition 5.3.9 (Type Formation Invariant: $\lambda 2$ to PLC)

$$\Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi \quad :\Longleftrightarrow \quad \forall m.\ \Psi \vDash_{\mathsf{V}} m : * \implies \exists n.\ \Theta \vdash n \simeq m \wedge n < N$$

Fact 5.3.10 (Skewing Law)

$$\Psi \vDash_{\lambda} a : * \implies \Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi \implies \Theta^{\uparrow} \Vdash_{\mathsf{P}} N \leftarrow \Psi, a$$

Fact 5.3.11 (Extension Law)

$$\Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi \implies \Theta^{\Uparrow} \Vdash_{\mathsf{P}} N + 1 \leftarrow \Psi, *$$

Lemma 5.3.12 (Right-Totality and Preservation of $\Theta \vDash_{\mathsf{P}} A \sim a$) For every Ψ-type a in $\lambda 2$ there exists a corresponding well-formed PLC type A.

$$\mathsf{val}\ \Psi \implies \Psi \vDash_{\lambda} a : * \implies \forall \Theta N.\ \Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi \implies \exists A.\ \Theta \vDash_{\mathsf{P}} A \sim a \wedge N \vDash_{\mathsf{P}} A$$

Proof We use Lemma 5.2.6 to perform an induction on the fact that a is a Ψ-type. The variable case as well as the PTS function type case that corresponds to a PLC function type remain unchanged, using the invariant and Fact 5.3.10. For PTS function types with domain $*$, we can quickly close the case using Fact 5.3.11. ∎

Note that the main reason why the right-totality result was so straightforward to prove is due to the custom induction principle which did all the heavy lifting like the case distinction on the nature of the function type domain.

As before, we also get ground instances for our invariants, but here it is sufficient to only consider contexts which are completely empty. Instances for auxiliary constructions like the PTS context T_N which contains only type variables are not required for System F.

Fact 5.3.13 (Ground Invariant Instances for Type Formation) For empty initial contexts, the following hold vacuously.

$$\Theta \Vdash_{\mathsf{P}} 0 \to \Psi \tag{i}$$
$$\Theta \Vdash_{\mathsf{P}} N \leftarrow \bullet \tag{ii}$$

5.3.2 Instantiation Compatibility for $\sim$

We again need to consider the interaction of our type relation with instantiation and as usual, we first consider renamings and then full substitutions. The changes with respect to the STLC development are mostly minor.

Lemma 5.3.14 Related types are preserved under renaming, when the relational context is adjusted accordingly.

$$\Theta \models_{\mathsf{P}} A \sim a \implies \mathsf{map}\,(\xi \times \zeta)\,\Theta \models_{\mathsf{P}} A[\xi] \sim a[\zeta]$$

Proof By induction on the derivation of $\Theta \models_{\mathsf{P}} A \sim a$, using Fact 4.3.2 for the variable case. We use Lemma 4.3.10 for both binder cases to move the skewing, and now also the extension, on Θ through the map. ∎

Lemma 5.3.15 Related types are preserved under monotone extension of the relational context.

$$\Theta_1 \sqsubseteq \Theta_2 \implies \Theta_1 \models_{\mathsf{P}} A \sim a \implies \Theta_2 \models_{\mathsf{P}} A \sim a$$

Proof Analogue to Lemma 4.4.16 by induction on the derivation of $\Theta_1 \models_{\mathsf{P}} A \sim a$ and using Fact 4.3.5 and Lemma 4.3.6. For the quantification case we additionally need Lemma 4.3.7 which states that $\Theta_1 \sqsubseteq \Theta_2$ entails $\Theta_1^{\Uparrow} \sqsubseteq \Theta_2^{\Uparrow}$. ∎

Lemma 5.3.16 (Preservation of $\Theta \models_{\mathsf{P}} A \sim a$ under Skewing and Extension) The following principles hold.

$$\Theta \models_{\mathsf{P}} A \sim a \implies \Theta^{\uparrow} \models_{\mathsf{P}} A \sim a[\uparrow] \tag{i}$$
$$\Theta \models_{\mathsf{P}} A \sim a \implies \Theta^{\Uparrow} \models_{\mathsf{P}} A[\uparrow] \sim a[\uparrow] \tag{ii}$$

Proof Consequences of Lemmas 5.3.14 and 5.3.15. ∎

For full substitutions we also proceed as for STLC, but we now need an extension law in addition to the skewing law. We omit the proof for the latter since it is identical to the STLC version.

Definition 5.3.17 (Type Relation Invariant)

$$\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1 \to \Theta_2 \quad := \quad \forall nm.\ \Theta_1 \vdash n \simeq m \implies \Theta_2 \vdash_{\mathsf{P}} \sigma\, n \sim \tau\, m$$

Fact 5.3.18 (Skewing Law) One-sided lifting of two related type substitutions preserves the invariant under context skewing.

$$\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1 \to \Theta_2 \implies \langle \sigma \sim \Uparrow\tau \rangle \Vdash_{\mathsf{P}} \Theta_1^{\uparrow} \to \Theta_2^{\uparrow}$$

Lemma 5.3.19 (Extension Law) Full lifting of two related type substitutions preserves the invariant under context extension.

$$\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1 \to \Theta_2 \implies \langle \Uparrow\sigma \sim \Uparrow\tau \rangle \Vdash_{\mathsf{P}} \Theta_1^{\Uparrow} \to \Theta_2^{\Uparrow}$$

Proof Assume $\langle \sigma \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1 \to \Theta_2$, as well as indices n, m such that $\Theta_1^{\Uparrow} \vdash n \simeq m$. Then by Lemma 4.3.8 we either have $n = 0$ and $m = 0$ and $\Theta_2^{\Uparrow} \vdash_{\mathsf{P}} \Uparrow\sigma\, 0 \sim \Uparrow\tau\, 0$ reduces to $\Theta_2^{\Uparrow} \vdash_{\mathsf{P}} 0 \sim 0$, which clearly holds by Lemma 4.3.9. Otherwise we have some n', m' such that $n = \uparrow n'$, $m = \uparrow m'$ and $\Theta_1 \vdash n' \simeq m'$. But then our morphism assumption entails $\Theta_2 \vdash_{\mathsf{P}} \sigma\, n' \sim \tau\, m'$. With Lemma 5.3.16 we obtain $\Theta_2^{\Uparrow} \vdash_{\mathsf{P}} (\sigma\, n')[\uparrow] \sim (\tau\, m')[\uparrow]$ which is equivalent to the required $\Theta_2^{\Uparrow} \vdash_{\mathsf{P}} \Uparrow\sigma\, n \sim \Uparrow\tau\, m$. ∎

Lemma 5.3.20 (CML for $\Theta \vdash_{\mathsf{P}} A \sim a$) Related types remain related under instantiation with related type substitutions.

$$\Theta_1 \vdash_{\mathsf{P}} A \sim a \implies \langle \sigma \sim \tau \rangle \Vdash_{\mathsf{P}} \Theta_1 \to \Theta_2 \implies \Theta_2 \vdash_{\mathsf{P}} A[\sigma] \sim a[\tau]$$

Proof By induction on the derivation of $\Theta_1 \vdash_{\mathsf{P}} A \sim a$ using Fact 5.3.18 and Lemma 5.3.19 for the two binder cases. ∎

As before we are going to need the preceding result in order to establish β-substitutivity for the term relation and then construct our final transfer example. What is new, though, is that we also require β-substitutivity of the type relation before we can prove the preservation results for the term relation. This necessity arises from the fact that type applications introduce non-vacuous β-substitutions on both sides of the type relation. This is in contrast to the ordinary term application which only introduces a vacuous, and therefore eliminable, β-substitution on the PTS side. As usual, β-substitutivity follows from general substitutivity with a suitable invariant instance, namely the following.

Lemma 5.3.21 Related types admit the construction of related β-substitutions at the type level.

$$\Theta \vdash_{\mathsf{P}} B \sim b \implies \langle B \cdot \mathsf{id} \sim b \cdot \mathsf{id} \rangle \Vdash_{\mathsf{P}} \Theta^{\Uparrow} \to \Theta$$

Proof Trivial using Lemma 4.3.8. ∎

Lemma 5.3.22 (Compatibility of $\Theta \vDash_{\mathsf{p}} A \sim a$ with β-Substitutions)

$$\Theta \vDash_{\mathsf{p}} B \sim b \implies \Theta^{\Uparrow} \vDash_{\mathsf{p}} A \sim a \implies \Theta \vDash_{\mathsf{p}} A[B \cdot \mathsf{id}] \sim a[b \cdot \mathsf{id}]$$

Proof Use Lemma 5.3.21 to obtain $\langle B \cdot \mathsf{id} \sim b \cdot \mathsf{id} \rangle \Vdash_{\mathsf{p}} \Theta^{\Uparrow} \to \Theta$ and then instantiate Lemma 5.3.20. ∎

We can also still strip renamings from the type relation without any major surprises so we simply state the result for the present discussion.

Fact 5.3.23 (Stripping of Renamings)

$$\mathsf{map}\,(\xi \times \zeta)\,\Theta \vDash_{\mathsf{p}} A \sim a \implies \exists A' a'.\ A = A'[\xi] \wedge a = a'[\zeta] \wedge \Theta \vDash_{\mathsf{p}} A' \sim a'$$

Corollary 5.3.24 (Stripping of Skewing)

$$\Theta^{\uparrow} \vDash_{\mathsf{p}} A \sim a \implies \exists a'.\ a = a'[\uparrow] \wedge \Theta \vDash_{\mathsf{p}} A \sim a'$$

Proof Direct instance of Fact 5.3.23. ∎

This completes our treatment of the type-level correspondence and brings us to the proof of the four properties for the term level.

5.3.3 Four Properties for $\approx$

Functionality is again straightforward, as there is no structural overlap on the PLC side of the relation. For injectivity, however, we will obtain spurious cases due to overlap, just as we did in the injectivity proof of the type relation, and refuting them imposes an auxiliary side condition.

Lemma 5.3.25 (Functionality of $\Theta; \Sigma \vDash_{\mathsf{p}} t \approx b$) The term relation is functional, provided that both Θ and Σ are functional relations.

$$\Theta\ \mathsf{func} \implies \Sigma\ \mathsf{func} \implies \Theta; \Sigma \vDash_{\mathsf{p}} t \approx b_1 \implies \Theta; \Sigma \vDash_{\mathsf{p}} t \approx b_2 \implies b_1 = b_2$$

Proof By induction on the derivation of $\Theta; \Sigma \vDash_{\mathsf{p}} t \approx b_1$ and discriminating on the derivation of $\Theta; \Sigma \vDash_{\mathsf{p}} t \approx b_2$. Lemma 5.3.3 for the embedded type-level derivations and Fact 4.3.11 for the preservation of functionality under the various context modifications are used repeatedly. ∎

The auxiliary information that is needed for the injectivity proof is range-disjointness of the two relational contexts Θ and Σ. Under this condition we can show that a given PTS term is never related to both a PLC type and a PLC term. This in turn is useful when it comes to the disambiguation of PTS applications in the injectivity proof.

Lemma 5.3.26 (Range-Disjointness of $\Theta \vDash_{\mathsf{P}} A \sim a$ and $\Theta; \Sigma \vDash_{\mathsf{P}} t \approx b$) The type and term relations do not overlap on the PTS side, provided such an overlap is not introduced through relational assumptions on the indices.

$$\Theta \parallel \Sigma \implies \Theta \vDash_{\mathsf{P}} A \sim a \implies \Theta; \Sigma \vDash_{\mathsf{P}} s \approx a \implies \bot$$

Proof A case analysis on the two relational derivations refutes all but the variable case, where we have $\Theta \vdash n_1 \simeq m$ and $\Sigma \vdash n_2 \simeq m$. This, however, contradicts $\Theta \parallel \Sigma$ and hence closes the proof. ∎

Lemma 5.3.27 (Injectivity of $\Theta; \Sigma \vDash_{\mathsf{P}} t \approx b$) The term relation is injective, provided that both Θ and Σ are injective and range-disjoint relations.

$$\Theta \ \mathsf{inj} \implies \Sigma \ \mathsf{inj} \implies \Theta \parallel \Sigma \implies \Theta; \Sigma \vDash_{\mathsf{P}} t_1 \approx b \implies \Theta; \Sigma \vDash_{\mathsf{P}} t_2 \approx b \implies t_1 = t_2$$

Proof By induction on the derivation of $\Theta; \Sigma \vDash_{\mathsf{P}} t_1 \approx b$ and discriminating on the derivation of $\Theta; \Sigma \vDash_{\mathsf{P}} t_2 \approx b$. Due to the structural overlap we obtain a total of nine cases, five regular ones and four which are spurious and need to be refuted.

The variable case follows from the injectivity of Σ, and the various embedded type-level derivations are handled with Lemma 5.3.4 and the injectivity of Θ. For the correctly matched binder cases we of course also need the preservation of injectivity under skewing and extension (Fact 4.3.11) as well as Lemma 4.3.13 for the preservation of range-disjointness under skewing one context and extending the other.

This leaves the four spurious cases where the two different application rules, and respectively the two different abstraction rules have been incorrectly matched. For the binders we can again look at the $\lambda 2$ domains a, which by one rule are known to be related to some PLC type A ($\Theta \vDash_{\mathsf{P}} A \sim a$), while the other forces $a = *$, which together is easily refuted. For the $\lambda 2$ applications $a\,b$ we look at the argument part b which by one rule is related to a term t ($\Theta; \Sigma \vDash_{\mathsf{P}} t \approx b$) and by the other rule to a type B ($\Theta \vDash_{\mathsf{P}} B \sim b$). These cases can be discharged using $\Theta \parallel \Sigma$ and Lemma 5.3.26. ∎

We next scale the totality and preservation results for the term relation to the present setting. The required invariants and associated extension laws can be adapted directly from the STLC setting for both statements. For the inductive proofs of the respective results themselves we of course have to consider two additional cases each, plus a few minor adjustments. Observe how the type application cases rely on β-substitutivity of the type relation.

Definition 5.3.28 (Typing Invariant: PLC to $\lambda 2$)

$$\begin{aligned} \Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi \quad :=\quad & \forall n A.\ \Gamma_n = A \implies \\ & \exists m a.\ \Theta \vDash_{\mathsf{P}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Psi \vDash_{\mathsf{V}} m : a \end{aligned}$$

Lemma 5.3.29 (Skewing/Extension Law) Skewing of the relational type variable context Θ and extension of the relational term variable context Σ preserves the invariant under context extension with a new term variable.

$$\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi \implies \Theta \vDash_{\mathsf{P}} A \sim a \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \Vdash_{\mathsf{P}} \Gamma, A \to \Psi, a$$

Proof Analogue to Lemma 4.4.26. ∎

Lemma 5.3.30 (Extension/Skewing Law) Extension of the relational type variable context Θ and skewing of the relational term variable context Σ preserves the invariant under context extension with a new type variable.

$$\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi \implies \Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_{\mathsf{P}} \Gamma[\uparrow] \to \Psi, *$$

Proof Analogue to Lemma 4.4.27. ∎

Lemma 5.3.31 (Left-Totality and Preservation of $\Theta; \Sigma \vdash_{\mathsf{P}} t \approx b$) For every well-typed PLC term t there exists a corresponding Ψ-term b in $\lambda 2$, such that their types are also related.

$$\begin{aligned} N; \Gamma \vdash_{\mathsf{P}} t : A \implies \forall \Theta \Sigma \Psi.\ \mathsf{val}\ \Psi \implies \Theta\ \mathsf{func} \implies \\ \Theta \Vdash_{\mathsf{P}} N \to \Psi \implies \Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi \implies \\ \exists b a.\ \Theta \vdash_{\mathsf{P}} A \sim a \wedge \Theta; \Sigma \vdash_{\mathsf{P}} t \approx b \wedge \Psi \vdash_{\lambda} b : a \wedge \Psi \vdash_{\lambda} a : * \end{aligned}$$

Proof By induction on the derivation of $N; \Gamma \vdash_{\mathsf{P}} t : A$.

The cases for variables, term abstractions and term applications are analogue to Lemma 4.4.28. Where applicable, we of course substitute results from this chapter in place of the STLC/$\lambda_{\to}$-specific properties.

Now let the last rule of the derivation be a type application $N; \Gamma \vdash_{\mathsf{P}} s\, B : A[B \cdot \mathsf{id}]$, with $N; \Gamma \vdash_{\mathsf{P}} s : \forall. A$ and $N \vdash_{\mathsf{P}} B$. The inductive hypothesis yields the following for some p and l.

$$\Theta \vdash_{\mathsf{P}} \forall. A \sim p \qquad \Theta; \Sigma \vdash_{\mathsf{P}} s \approx l \qquad \Psi \vdash_{\lambda} l : p$$

Meanwhile, Lemma 5.3.8 yields a term b satisfying $\Theta \vdash_{\mathsf{P}} B \sim b$ and $\Psi \vdash_{\lambda} b : *$. Now clearly, p must be a dependent function type of the form $\Pi *. a$ for some a satisfying $\Theta^{\Uparrow} \vdash_{\mathsf{P}} A \sim a$. With β-substitutivity of the type relation (Lemma 5.3.22) it follows that $\Theta \vdash_{\mathsf{P}} A[B \cdot \mathsf{id}] \sim a[b \cdot \mathsf{id}]$ holds, which identifies the related type $a[b \cdot \mathsf{id}]$. The corresponding related term is therefore $l\, b$, with $\Theta; \Sigma \vdash_{\mathsf{P}} s\, B \approx l\, b$ being clearly derivable. It remains to show that $\Psi \vdash_{\lambda} l\, b : a[b \cdot \mathsf{id}]$ and $\Psi \vdash_{\lambda} a[b \cdot \mathsf{id}] : *$ hold. Since we have $\Psi \vdash_{\lambda} l : \Pi *. a$ and $\Psi \vdash_{\lambda} b : *$, we get the former with the PTS application rule while the latter follows from Lemma 3.6.14 and the fact that all $\lambda 2$ rules in $\mathcal{R}$ have $*$ as their second component.

For the final case, let the last derivation step be a type abstraction $N; \Gamma \vdash_{\mathsf{P}} \Lambda. s : \forall. A$ with $N + 1; \Gamma[\uparrow] \vdash_{\mathsf{P}} s : A$. We have $\mathsf{val}\ \Psi$, and therefore also $\mathsf{val}\ \Psi, *$ and similarly, using Fact 4.3.11, $\Theta^{\Uparrow}\ \mathsf{func}$. Additionally, we have $\Theta \Vdash_{\mathsf{P}} N \to \Psi$ and $\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi$, and hence from Fact 5.3.7 and Lemma 5.3.30 also $\Theta^{\Uparrow} \Vdash_{\mathsf{P}} N + 1 \to \Psi, *$ and $\Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_{\mathsf{P}} \Gamma[\uparrow] \to \Psi, *$. This allows us to instantiate the inductive hypothesis and obtain terms a and b which satisfy the following.

$$\Theta^{\Uparrow} \vdash_{\mathsf{P}} A \sim a \qquad \Theta^{\Uparrow}; \Sigma^{\uparrow} \vdash_{\mathsf{P}} s \approx b \qquad \Psi, * \vdash_{\lambda} b : a \qquad \Psi, * \vdash_{\lambda} a : *$$

We therefore clearly have $\Theta \vDash_{\mathsf{P}} \forall.\, A \sim \Pi *.\, a$ and $\Theta; \Sigma \vDash_{\mathsf{P}} \Lambda.\, s \approx \lambda *.\, b$ and the two typing claims $\Psi \vdash_{\lambda} \lambda *.\, b : \Pi *.\, a$ and $\Psi \vdash_{\lambda} \Pi *.\, a : *$ also follow easily (the universe that types $*$ is obviously $\Box$). ∎

Definition 5.3.32 (Typing Invariant: λ2 to PLC)

$$\begin{aligned}\Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi \quad :=\quad & \forall m a.\ \Psi \vdash_{\mathsf{V}} m : a \implies \Gamma \vdash_{\lambda} a : * \implies \\ & \exists n A.\ \Theta \vDash_{\mathsf{P}} A \sim a \wedge \Sigma \vdash n \simeq m \wedge \Gamma_n = A\end{aligned}$$

Lemma 5.3.33 (Skewing/Extension Law) Skewing of the relational type variable context Θ and extension of the relational term variable context Σ preserves the invariant under context extension with a new term variable.

$$\mathsf{val}\ \Psi, a \implies \Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi \implies \Theta \vDash_{\mathsf{P}} A \sim a \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \Vdash_{\mathsf{P}} \Gamma, A \leftarrow \Psi, a$$

Proof Analogue to Lemma 4.4.30, using Fact 5.2.8 for the required strengthening step. ∎

Lemma 5.3.34 (Extension/Skewing Law) Extension of the relational type variable context Θ and skewing of the relational term variable context Σ preserves the invariant under context extension with a new type variable.

$$\mathsf{val}\ \Psi, * \implies \Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi \implies \Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_{\mathsf{P}} \Gamma[\uparrow] \leftarrow \Psi, *$$

Proof Analogue to Lemma 4.4.31, using Fact 5.2.8 for the required strengthening step. ∎

Lemma 5.3.35 (Right-Totality and Preservation of $\Theta; \Sigma \vDash_{\mathsf{P}} t \approx b$) For every Ψ-term b in $\lambda 2$ there exists a corresponding well-typed PLC term t, such that their types are also related.

$$\begin{aligned}\mathsf{val}\ \Psi \implies \Psi \vdash_{\lambda} a : * \implies \Psi \vdash_{\lambda} b : a \implies \\ \forall \Theta \Sigma N \Gamma.\ \Theta\ \mathsf{inj} \implies \Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi \implies \Theta; \Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi \implies \\ \exists t A.\ \Theta \vDash_{\mathsf{P}} A \sim a \wedge \Theta; \Sigma \vDash_{\mathsf{P}} t \approx b \wedge N; \Gamma \vDash_{\mathsf{P}} t : A \wedge N \vDash_{\mathsf{P}} A\end{aligned}$$

Proof We use Lemma 5.2.7 to perform an induction on the fact that b is a Ψ-term.

The cases for variables and term abstractions are analogue to Lemma 4.4.32. The case for term applications is also mostly analogue, but recall that we had $\Psi \vdash_{\lambda} a : \Pi c.\, d$ and by induction some PLC type F such that $\Theta \vDash_{\mathsf{P}} F \sim \Pi c.\, d$. In the STLC setting we could immediately infer that $F = C \to D$ for some C and D which are suitably related to the $\lambda 2$ components and proceed from there. Here we could also have $F = \forall.\, D$ for some D related to d but only when $c = *$. We do however also know that $\Psi \vdash_{\lambda} c : *$, so Fact 5.2.2 allows us to discard this spurious case.

This leaves the two new cases for type abstraction and application.

Let the last rule of the derivation be $\Psi \vdash_\lambda a\, b : d[b \cdot \mathsf{id}]$, which derived from $\Psi \vdash_\lambda a : \Pi *.\, d$ and $\Psi \vdash_\lambda b : *$. From the latter and Lemma 5.3.12 we obtain some PLC type B which satisfies $\Theta \vdash_\mathsf{P} B \sim b$ and $N \vdash_\mathsf{P} B$. Our inductive hypothesis additionally yields the following for some term f and type F.

$$\Theta \vdash_\mathsf{S} F \sim \Pi *.\, d \qquad N \vdash_\mathsf{S} F \qquad \Theta; \Sigma \vdash_\mathsf{S} f \approx a \qquad N; \Gamma \vdash_\mathsf{S} f : F$$

From the first it follows that $F = \forall.\, D$, since $*$ is not type-related to anything. Moreover we have $\Theta^{\Uparrow} \vdash_\mathsf{P} D \sim d$ and thus by β-substitutivity of the type relation (Lemma 5.3.22) $\Theta \vdash_\mathsf{P} D[B \cdot \mathsf{id}] \sim d[b \cdot \mathsf{id}]$. The corresponding related term is $f\, B$ since $\Theta; \Sigma \vdash_\mathsf{P} f\, B \approx a\, b$ is easy to derive. The typing $N; \Gamma \vdash_\mathsf{P} f\, B : D[B \cdot \mathsf{id}]$ follows with the PLC type application rule while $N \vdash_\mathsf{P} D[B \cdot \mathsf{id}]$ relies on the fact that $N + 1 \vdash_\mathsf{P} D$ and $N \vdash_\mathsf{P} B$ as well as the compatibility of PLC type formation with β-substitutions (Lemma 5.1.6).

Finally, let $\Psi \vdash_\lambda \lambda *.\, b : \Pi *.\, c$ derive from $\Psi, * \vdash_\lambda b : c$. We use Fact 4.3.11, Fact 5.3.11 and Lemma 5.3.34 to adjust our invariants respectively to the following.

$$\Theta^{\Uparrow}\ \mathsf{inj} \qquad \Theta^{\Uparrow} \Vdash_\mathsf{P} N + 1 \leftarrow \Psi, * \qquad \Theta^{\Uparrow}; \Sigma^{\uparrow} \Vdash_\mathsf{P} \Gamma[\uparrow] \leftarrow \Psi, *$$

This allows us to instantiate the inductive hypothesis and obtain a term s and type C which satisfy the following.

$$\Theta^{\Uparrow} \vdash_\mathsf{P} C \sim c \qquad N + 1 \vdash_\mathsf{P} C \qquad \Theta^{\Uparrow}; \Sigma^{\uparrow} \vdash_\mathsf{S} s \approx b \qquad N + 1; \Gamma[\uparrow] \vdash_\mathsf{S} s : C$$

Let $\Lambda.\, s$ and $\forall.\, C$ be the related term and type. All four claims are easily derivable for these witnesses. ∎

We complete our discussion of the four properties of the term relation with the corresponding ground invariant instances for empty contexts.

Fact 5.3.36 (Ground Invariant Instances for Typing) The following hold vacuously.

$$\Theta; \Sigma \Vdash_\mathsf{P} \bullet \to \Psi \tag{i}$$
$$\Theta; \Sigma \Vdash_\mathsf{P} \Gamma \leftarrow \bullet \tag{ii}$$

5.3.4 Instantiation Compatibility for $\approx$

Just as for STLC, we can now obtain compatibility with β-substitutions for the term relation. Let us consider, what changes we need to make in order to derive the result also for the extended System F setting. As usual we decompose the problem into a preliminary step for renamings, followed by the result for generic substitutions and then instantiate that to the particular case of β-substitutions.

The first major difference results from the fact that we now also deal with terms that are instantiated with type substitutions. Recall the formulation of the renaming compatibility statement for STLC (Lemma 4.4.35).

$$\Theta; \Sigma \vdash_\mathsf{S} s \approx b \implies \mathsf{map}\,(\mathsf{id} \times \zeta)\,\Theta; \mathsf{map}\,(\xi \times \zeta)\,\Sigma \vdash_\mathsf{S} s[\xi] \approx b[\zeta]$$

Observe how the PTS renaming ζ affects both type and term indices, while ξ only operates on STLC term indices. The id-renaming applied to Θ reflects that STLC type indices remain unchanged. For the present setting, we introduce a third renaming ρ that is going to act on the type indices. The resulting statement then becomes the following.

Lemma 5.3.37 Related terms are preserved under renaming, when the relational contexts are adjusted accordingly.

$$\Theta; \Sigma \models_{\mathsf{P}} s \approx b \implies \mathsf{map}\,(\rho \times \zeta)\,\Theta; \mathsf{map}\,(\xi \times \zeta)\,\Sigma \models_{\mathsf{S}} s[\rho, \xi] \approx b[\zeta]$$

Proof By induction on the derivation of $\Theta; \Sigma \models_{\mathsf{P}} s \approx b$. The cases for variables, term abstraction and term application are analogue to Lemma 4.4.35.

The case for type application needs Lemma 5.3.14 for the embedded derivation of a type relation and type abstraction again relies on both parts of Lemma 4.3.10 to move the extension of Θ and the skewing of Σ through the mappings. All occurring substitution equalities are easily handled by Autosubst. ∎

We also generalise our concept of monotonicity to incorporate changes in the relational type variable context (c.f. Lemma 4.4.36).

Lemma 5.3.38 Related terms are preserved under monotone extension of the relational contexts.

$$\Theta_1 \sqsubseteq \Theta_2 \implies \Sigma_1 \sqsubseteq \Sigma_2 \implies \Theta_1; \Sigma_1 \models_{\mathsf{P}} s \approx b \implies \Theta_2; \Sigma_2 \models_{\mathsf{P}} s \approx b$$

Proof By induction on the derivation of $\Theta_1; \Sigma_1 \models_{\mathsf{P}} s \approx b$. Since both contexts are now subject to modifications we additionally require monotonicity of the type relation (Lemma 5.3.15) and monotonicity of mapping (Lemma 4.3.6). ∎

A consequence of these two results are two extension/skewing laws that correspond to the addition of type and term variables. The first for term variables is a direct adaptation of the STLC result, while the second is new and the dual result for type variables.

Lemma 5.3.39 (Preservation of $\Theta; \Sigma \models_{\mathsf{P}} s \approx b$ under Skewing/Extension) The following principles holds.

$$\Theta; \Sigma \models_{\mathsf{S}} s \approx b \implies \Theta^{\uparrow}; \Sigma^{\Uparrow} \models_{\mathsf{S}} s[\mathsf{id}, \uparrow] \approx b[\uparrow] \quad \text{(i)}$$

$$\Theta; \Sigma \models_{\mathsf{S}} s \approx b \implies \Theta^{\Uparrow}; \Sigma^{\uparrow} \models_{\mathsf{S}} s[\uparrow, \mathsf{id}] \approx b[\uparrow] \quad \text{(ii)}$$

Proof Both principles follow from Lemmas 5.3.37 and 5.3.38. Part (i) is analogue to Lemma 4.4.37, part (ii) is dual with the roles of Θ and Σ interchanged. ∎

For full substitutions we now adapt the STLC invariant and provide the missing dual extension principle to handle type abstraction. Note also the use of vector substitutions, where we recall the two *up*-operations $\Uparrow_{\mathsf{tm}}$ and $\Uparrow_{\mathsf{ty}}$.

$$\Uparrow_{\mathsf{tm}}(\rho, \sigma) = (\rho, 0 \cdot \sigma \circ (\mathsf{id}, \uparrow))$$
$$\Uparrow_{\mathsf{ty}}(\rho, \sigma) = (0 \cdot \rho \circ \uparrow, \sigma \circ (\uparrow, \mathsf{id}))$$

Definition 5.3.40 (Term Relation Invariant)

$$\langle(\rho, \sigma) \sim \tau\rangle \Vdash_{\mathsf{P}} \Theta_1; \Sigma_1 \to \Theta_2; \Sigma_2 \quad := \quad (\forall nm.\ \Theta_1 \vdash n \simeq m \implies \Theta_2 \vdash_{\mathsf{P}} \rho\, n \sim \tau\, m) \wedge (\forall nm.\ \Sigma_1 \vdash n \simeq m \implies \Theta_2; \Sigma_2 \vdash_{\mathsf{P}} \sigma\, n \approx \tau\, m)$$

Lemma 5.3.41 (Skewing/Extension Law) The invariant satisfies the following extension principle, which corresponds to the addition of a new term variable.

$$\langle(\rho, \sigma) \sim \tau\rangle \Vdash_{\mathsf{P}} \Theta_1; \Sigma_1 \to \Theta_2; \Sigma_2 \implies \langle\Uparrow_{\mathsf{tm}}(\rho, \sigma) \sim \Uparrow\tau\rangle \Vdash_{\mathsf{P}} \Theta_1^{\uparrow}; \Sigma_1^{\Uparrow} \to \Theta_2^{\uparrow}; \Sigma_2^{\Uparrow}$$

Proof Analogue to Lemma 4.4.39. ∎

Lemma 5.3.42 (Extension/Skewing Law) The invariant satisfies the following extension principle, which corresponds to the addition of a new type variable.

$$\langle(\rho, \sigma) \sim \tau\rangle \Vdash_{\mathsf{P}} \Theta_1; \Sigma_1 \to \Theta_2; \Sigma_2 \implies \langle\Uparrow_{\mathsf{ty}}(\rho, \sigma) \sim \Uparrow\tau\rangle \Vdash_{\mathsf{P}} \Theta_1^{\Uparrow}; \Sigma_1^{\uparrow} \to \Theta_2^{\Uparrow}; \Sigma_2^{\uparrow}$$

Proof The first conjunct is simply Lemma 5.3.19. For the second we use Lemma 4.3.8 on the assumption $\Sigma_1^{\uparrow} \vdash n \simeq m$ to obtain an m' such that $m = \uparrow m'$ and $\Sigma_1 \vdash n \simeq m'$. Now $\Theta_2; \Sigma_2 \vdash_{\mathsf{P}} \sigma\, n \approx \tau\, m'$ follows from the premise and part (ii) of Lemma 5.3.39 yields $\Theta_2^{\Uparrow}; \Sigma_2^{\uparrow} \vdash_{\mathsf{P}} (\sigma\, n)[\uparrow, \mathsf{id}] \approx (\tau\, m')[\uparrow]$. The latter is equivalent to

$$\Theta_2^{\Uparrow}; \Sigma_2^{\uparrow} \vdash_{\mathsf{P}} (\sigma \circ (\uparrow, \mathsf{id}))\, n \approx \Uparrow\tau\, m,$$

as required. ∎

Lemma 5.3.43 (CML for $\Theta; \Sigma \vdash_{\mathsf{P}} s \approx b$) Related terms remain related under instantiation with related type and term substitutions.

$$\Theta_1; \Sigma_1 \vdash_{\mathsf{P}} s \approx b \implies \langle(\rho, \sigma) \sim \tau\rangle \Vdash_{\mathsf{P}} \Theta_1; \Sigma_1 \to \Theta_2; \Sigma_2 \implies \Theta_2; \Sigma_2 \vdash_{\mathsf{P}} s[\rho, \sigma] \approx b[\tau]$$

Proof Analogue to Lemma 4.4.40 by induction on the derivation of $\Theta_1; \Sigma_1 \vdash_{\mathsf{P}} s \approx b$. The additional type application case needs Lemma 5.3.20, while type abstraction follows with Lemma 5.3.42. ∎

Corollary 5.3.44 (Compatibility of $\Theta; \Sigma \vdash_{\mathsf{P}} s \approx b$ with β-Substitution) The following inference rule is admissible.

$$\frac{\Theta^{\uparrow}; \Sigma^{\Uparrow} \vdash_{\mathsf{P}} s \approx b \qquad \Theta; \Sigma \vdash_{\mathsf{P}} t \approx c}{\Theta; \Sigma \vdash_{\mathsf{P}} s[\mathsf{id}, t \cdot \mathsf{id}] \approx b[c \cdot \mathsf{id}]}$$

Proof From the second premise we can derive a particular instance of the CML invariant, namely

$$\langle(\mathsf{id}, t \cdot \mathsf{id}) \sim c \cdot \mathsf{id}\rangle \Vdash_{\mathsf{P}} \Theta^{\uparrow}; \Sigma^{\Uparrow} \to \Theta; \Sigma,$$

which allows us to simply instantiate Lemma 5.3.43 and thereby close the proof. ∎

5.3.5 Existence of Related Contexts

The existence of related contexts can be scaled similarly. Here we have to incorporate two adjustments.

First we have to establish an extra invariant, namely that the two relational contexts Θ and Σ, which connect the two valid typing contexts, are range-disjoint. For otherwise we do not know whether the term relation under these contexts is injective (Lemma 5.3.27).

Secondly, for the construction of a $\lambda 2$ context from a PLC context we also have to consider the addition of type variables.

Since we have seen the construction for STLC as well as all required extension principles throughout this chapter, we only give the existence results and do not reiterate their proofs.

Fact 5.3.45 Let $N;\Gamma$ be a valid PLC context, then there exists a valid $\lambda 2$ context Ψ which relates to $N;\Gamma$ according to some Θ and Σ, such that $\Theta \parallel \Sigma$ and all the following hold.

Θ func	Σ func	$\Theta \Vdash_{\mathsf{P}} N \to \Psi$	$\Theta;\Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi$
Θ inj	Σ inj	$\Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi$	$\Theta;\Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi$

Fact 5.3.46 Let Ψ be a valid $\lambda 2$ context, then there exists a valid PLC context $N;\Gamma$ which relates to Ψ according to some Θ and Σ, such that $\Theta \parallel \Sigma$ and all the following hold.

Θ func	Σ func	$\Theta \Vdash_{\mathsf{P}} N \to \Psi$	$\Theta;\Sigma \Vdash_{\mathsf{P}} \Gamma \to \Psi$
Θ inj	Σ inj	$\Theta \Vdash_{\mathsf{P}} N \leftarrow \Psi$	$\Theta;\Sigma \Vdash_{\mathsf{P}} \Gamma \leftarrow \Psi$

5.3.6 Closed Correspondence

We can of course also formulate the correspondence result for closed contexts. Note in particular, that we can work with completely empty contexts, in contrast to Theorem 4.4.44.

Theorem 5.3.47 (System F Correspondence for Closed Judgements) The following equivalences hold for closed instances of type formation, (i) and (ii), and typing, (iii) and (iv).

$$0 \Vdash_{\mathsf{P}} A \iff \exists a.\ \bullet \Vdash_{\mathsf{P}} A \sim a \wedge \bullet \vdash_{\lambda} a : * \tag{i}$$

$$\bullet \vdash_{\lambda} a : * \iff \exists A.\ \bullet \Vdash_{\mathsf{P}} A \sim a \wedge 0 \Vdash_{\mathsf{P}} A \tag{ii}$$

$$0;\bullet \Vdash_{\mathsf{P}} s : A \iff \exists ba.\ \bullet \Vdash_{\mathsf{P}} A \sim a \wedge \bullet;\bullet \Vdash_{\mathsf{P}} s \approx b \wedge \bullet \vdash_{\lambda} b : a \wedge \bullet \vdash_{\lambda} a : * \tag{iii}$$

$$\bullet \vdash_{\lambda} a : * \wedge \bullet \vdash_{\lambda} b : a \iff \exists sA.\ \bullet \Vdash_{\mathsf{P}} A \sim a \wedge \bullet;\bullet \Vdash_{\mathsf{P}} s \approx b \wedge 0;\bullet \Vdash_{\mathsf{P}} s : A \wedge 0 \Vdash_{\mathsf{P}} A \tag{iv}$$

Proof Each equivalence is proven analogue to the corresponding part of Theorem 4.4.44. The ground invariant instances are of course chosen from Fact 5.3.13 and Fact 5.3.36 to match the respective structure of the given contexts. ∎

We have shown all results which are necessary to exactly replay the two property transfers of propagation and β-substitutivity from $\lambda 2$ to PLC. That is, we can now prove analogue results to Lemma 4.5.1 and Lemma 4.5.2. The only difference over the STLC results occurs with the use of term-level injectivity in the proof of β-substitutivity. Here we rely on the non-overlapping codomains of the relational contexts. This fact is in turn additionally provided by the utilised context existence result, namely Fact 5.3.45. The detailed transfer results are given in the formalisation but omitted here to avoid repetition.

5.4 Discussion

With Theorem 5.3.47 as well as the correctness results for context internalisation (Lemmas 5.1.10 and 5.2.9), we have finally come to the point where we can well and truly claim that our two rather distinct presentations of System F amount to *the same system*, that is we have a bidirectional reduction of type formation and typing.

We again point out that while Theorem 5.3.47 allows for a reasonably concise presentation of the state of affairs, it is usually much easier to use the properties of the correspondence for open judgements directly, in particular since any valid context is known to have a suitably related counterpart (Facts 5.3.45 and 5.3.46). Our example property transfers bear testimony to this observation.

Now that we have seen the correspondence proofs for both the simply typed λ-calculus as well as its polymorphic extension to System F, it is interesting to note how the overall effort is distributed across the two developments.

We originally approached the stripped-down STLC proof under the assumption that this would significantly simplify matters. This turned out to not be the case. The reason for this is, however, easily explained. The biggest complication of the presented result is the mismatch of the encoded variable scopes between the formal systems which are being related. And this challenge requires the full machinery for relating de Bruijn indices, already for the simply typed scenario. Hence Section 4.3 carries equal relevance for the proofs in both Chapters 4 and 5. The CML proof pattern then makes the remainder of the proofs, while tedious, mostly mechanical, given that the treatment of variables is under control. The only real complications which are particular to the polymorphic case are the need for full β-substitutivity of PLC type formation, the structural overlap in the two PLC correspondence relations, and therefore the need to lift range-disjointness from variable relations to the defined correspondences.

There are certain aspects of the presented constructions that could be improved upon, in particular if we were to scale the construction further to say System F_ω, but we postpone this discussion for now and come back to it in Section 8.2.2.

6 Higher-Order Abstract Syntax

In this chapter we are going to discuss the higher-order abstract syntax (HOAS) [PE88] representation of our systems as an alternative to the first-order de Bruijn approach. We first introduce the basics of HOAS in general and then apply them to our System F correspondence proof. Since HOAS structures are fundamentally incompatible with the underlying type theory of the Coq proof assistant, we instead provide two HOAS formalisations of the equivalence result in the proof systems Abella [Gac08, Gac09, BCG+14, ABL] and Beluga [PD10, PC15, BEL]. Both are designed to natively support HOAS reasoning.

6.1 Basic HOAS

Let us recall Section 2.2, where we discussed the intended meaning of variables. We in particular identified variable occurrences as placeholders which reference a corresponding binder (or entry in a suitable context) and illustrated the situation as

where v is some subterm not referencing the considered binder. Let us take a closer look at the body of this abstraction, that is the application $\square\, v$. It is perfectly reasonable to consider this as an *expression with a hole*, which should eventually be filled with another expression. It is easy to imagine that a particular hole may occur repeatedly and the concept further extends to multiple binders, which would each have their own set of associated holes.

The idea of having formal expressions with fillable holes is familiar to anybody who is acquainted with basic type theory or functional programming. There, it arises in the form of abstraction and application, which corresponds to the creation and filling of such holes, respectively. The concept of **higher-order abstract syntax** (HOAS) is based on these observations. The main idea is to exploit the abstraction and application mechanisms of the underlying host theory to implement the object language in question. This turns object-level expressions with holes, like the body of an abstraction, into host-level functions. Such expressions with holes have functional, and therefore **higher-order**, types. The host-level application of such a functional expression to another expression is then an easy way to provide capture-avoiding instantiation.

To illustrate the idea, let us assume some host language with a notion of functions and function types, as well as a notion of application. Let us further assume that we

can define new types[1] and new term constructors, also referred to as constants. We consider ULC as an example and recall its named syntax.

$$\boxed{\mathsf{Tm}_{\mathsf{U}}^{\mathsf{nmd}}} \qquad u, v ::= x \mid u\,v \mid \lambda x.\, v \qquad x \in \mathcal{V}$$

A HOAS signature for this syntactic language would look as follows.

$$\mathsf{T} \;:\; \mathbf{Type}$$

$$\begin{aligned} \mathsf{app} &: \mathsf{T} \to \mathsf{T} \to \mathsf{T} \\ \mathsf{lam} &: (\mathsf{T} \to \mathsf{T}) \to \mathsf{T} \end{aligned}$$

Here we declare a new object type T with two term constructors app and lam. The host type signature of the term constructor for applications (app) is not very surprising: it simply constructs a term from two subterms. The term constructor for abstractions (lam) is more interesting, though. Observe how it takes a host-level function of type $\mathsf{T} \to \mathsf{T}$ as its sole argument. This function exactly implements the abstraction body as a term with holes, in the sense of the intuition given above. We can express our initial example in this formalism as

$$\mathsf{lam}\,(\lambda \mathsf{x}{:}\mathsf{T} \Rightarrow \mathsf{app}\,\mathsf{x}\,v).$$

Note that the variable x is a variable of the host language which implements the object-level variable. This also illustrates why there is no term constructor for object-level variables; it is simply not needed. When we do not explicitly analyse the nature of the HOAS constructors themselves, we will use a symbolic, nameless presentation of terms which is similar to the one introduced for our de Bruijn development. For concrete examples, where bound variables do occur, we ascribe the corresponding name to the relevant binder for clarity. We would thus write the above expression simply as $\lambda^x.\, x\,v$.

Another useful intuition in this context is to understand the type T as the syntactic class of *closed* ULC expressions. The syntactic classes for *open* expressions are then formed as host language function types. That is, the type $\mathsf{T} \to \mathsf{T}$ represents the syntactic class of ULC expressions with one free variable of type T. Similarly, $\mathsf{T} \to \mathsf{T} \to \mathsf{T}$ represents the class of terms with two distinct free variables, while $(\mathsf{T} \to \mathsf{T}) \to \mathsf{T}$ captures the class of single-variable binders. Take for example $f : \mathsf{T} \to \mathsf{T}$, which has one free variable that is subsequently bound in $\lambda.\, f$. The latter is a closed term of type T.

To complete the basic picture, let us also illustrate the contraction of β-redices in the HOAS setting. We take the concrete abstraction $\lambda^x.\, x\,v$ from above (where we again assume that x is not free in v) and apply it to a term u. In our high-level symbolic notation we expect the following contraction.

$$(\lambda^x.\, x\,v)\,u \succ (x\,v)[u/x] = u\,v$$

[1] These new types are sometimes referred to as *sorts*, but we avoid this here to prevent confusion with our notion of PTS sorts at the object level.

To see how the contraction and in particular the variable instantiation are facilitated in HOAS, it is helpful to unfold the notation and look at the underlying higher-order term constructors lam and app. Without the mathematical notation, our example contraction looks as follows,

$$\mathsf{app}\,(\mathsf{lam}\,(\lambda \mathsf{x}{:}\mathsf{T} \Rightarrow \mathsf{app}\,\mathsf{x}\,v))\,u \succ (\lambda \mathsf{x}{:}\mathsf{T} \Rightarrow \mathsf{app}\,\mathsf{x}\,v)\langle u\rangle = \mathsf{app}\,u\,v$$

where $f\langle u\rangle$ denotes host-level application. Observe how the reduction simply applies the body of the abstraction, that is the function $\lambda \mathsf{x}{:}\mathsf{T} \Rightarrow \mathsf{app}\,\mathsf{x}\,v$, to the term that is being substituted, here u. The key point here is the absence of a user-defined capture-avoiding substitution operation, or *instantiation*, for our object-level language. We simply inherit its implementation with all associated properties from the host language. In other words, object-level application and abstraction are realised directly in terms of their host-level counterparts.

The present exposition might suggest that the HOAS approach is wholly superior to our first-order de Bruijn encodings, but there are some major drawbacks.

First of all, the HOAS term type T is not an inductive type in the strict sense, due to the negative occurrence of T in the term constructor lam. It is still possible to inductively reason about such definitions, but only if the host-level abstraction $f : \mathsf{T} \to \mathsf{T}$ in $\mathsf{lam}\,f$ can be structurally analysed. Practically, this forces $\mathsf{T} \to \mathsf{T}$ to be an **intensional** function type with a rather weak notion of equality (essentially structural equality modulo $\alpha\beta\eta$-conversion) [Hof99]. Note that most functional programming languages and type theories are, however, based on **extensional** function types, including the Calculus of (co)inductive Constructions (CiC). We recall that the latter is the underlying theory of the Coq proof assistant, which consequentially has no native support for HOAS reasoning. Since CiC is a rather powerful formalism, it is of course possible to envision some form of deep embedding of intensional function types, which could then be used to indirectly support HOAS definitions. We discuss these ideas further in Section 8.2.4.

Systems that do support HOAS definitions usually handle these complications with a so-called **two-level logic approach** [GMN12]. The key idea is to, more or less explicitly, stratify the host theory into a generic reasoning layer and a restricted specification layer with suitable function types. The specification layer is used to express the object language, while its metatheory is developed in the reasoning layer. The latter is able to inspect the structure of terms, formulas and derivations of the specification layer.

We will now take a closer look at the nature of such a specification layer in the context of our main HOAS system definitions and later see how the constructions are implemented in the proof systems Abella [Gac08] and Beluga [PD10].

6.2 HOAS Representations of PLC and λ2

Let us now consider how our two variants of System F, that is PLC and $\lambda 2$, are expressed using HOAS definitions. As mentioned above, these definitions will be

expressed in a certain specification language. Our initial exposition assumed that our specification language admits the definition of new type families with corresponding constants. These new type families live in the host universe **Type**.

Let us first consider PLC, which is two-sorted and thus requires two new type families, $\mathsf{Ty_P}$ and $\mathsf{Tm_P}$, for object-level types and, respectively, terms. Its HOAS signature is shown in Figure 6.1, where we give the symbolic, nameless notation for reference on the right. Observe how the signature makes it easy to identify the three binding constructors ($\mathsf{all_P}, \mathsf{lam_P}, \mathsf{tylam_P}$), which all contain functions as subexpressions. Take for example $\Lambda.\, M$, where the subexpression M is a function of type $\mathsf{Ty_P} \to \mathsf{Tm_P}$. The body of the abstraction is thus a term with one free type variable that is captured by the $\mathsf{tylam_P}$ term constructor.

The HOAS signature for $\lambda 2$ is shown in Figure 6.2, again annotated with symbolic notation. Observe how we only introduce a single type family Tm_λ for terms, in accordance with the single-sorted nature of the PTS framework. The two binders (prod_λ and lam_λ) are again clearly discernible from the constructor types.

Next we consider judgements over our three new type families. A derivable judgement instance is a formula at the specification level and lives in the special type of propositions, $\mathbf{o}$[2]. Thus in HOAS, judgements and similar predicates are simply constructors with target type $\mathbf{o}$. Note that $\mathbf{o}$ is a specification-level type and sits logically at the same level as, for example, Tm_λ. It should, in particular, not be confused with the universe of reasoning-level propositions, where we will later place statements *about* our syntactic systems.

We encode the typing disciplines of our two variants of System F with the following specification-level predicates, or formula constructors.

$$\begin{array}{rcll} \mathsf{isty_P} & : & \mathsf{Ty_P} \to \mathbf{o} & A\ \mathbf{ty} \\ \mathsf{of_P} & : & \mathsf{Tm_P} \to \mathsf{Ty_P} \to \mathbf{o} & M :_{\mathsf{P}} A \\ & & & \\ \mathsf{univ}_\lambda & : & \mathsf{Tm}_\lambda \to \mathbf{o} & \mathcal{U}\ S \\ \mathsf{of}_\lambda & : & \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathbf{o} & S :_\lambda T \end{array}$$

The predicates $A\ \mathbf{ty}$ and $M :_{\mathsf{P}} A$ provide PLC type formation and, respectively, typing, while $S :_\lambda T$ represents $\lambda 2$ typing. The auxiliary predicate $\mathcal{U}\ S$ is used to recognise PTS universes, that is the PTS sorts $*$ and $\Box$.

We have introduced the predicates $\mathsf{isty_P}$, $\mathsf{of_P}$, univ_λ and of_λ as the HOAS judgements of our two object languages. A closer look at the respective specification-level types reveals a surprising difference with respect to our earlier de Bruijn judgements: the constructions presented here do not carry typing contexts. Moreover, we have not even defined a notion of "HOAS typing context" for the two languages . The reason for this omission is tied to a key feature that is often found in two-level logic systems, namely the implicit tracking of contextual information. More precisely, the management of

[2] This derives from the Greek letter 'omicron', which was used by Church [Chu40] as the type of propositions.

$$
\begin{array}{rcll}
\mathsf{Ty_P}, \mathsf{Tm_P} & : & \mathbf{Type} & \\
 & & & \\
\mathsf{arr_P} & : & \mathsf{Ty_P} \to \mathsf{Ty_P} \to \mathsf{Ty_P} & A \to B \\
\mathsf{all_P} & : & (\mathsf{Ty_P} \to \mathsf{Ty_P}) \to \mathsf{Ty_P} & \forall.\, A \\
 & & & \\
\mathsf{app_P} & : & \mathsf{Tm_P} \to \mathsf{Tm_P} \to \mathsf{Tm_P} & M\, N \\
\mathsf{tyapp_P} & : & \mathsf{Tm_P} \to \mathsf{Ty_P} \to \mathsf{Tm_P} & M\, A \\
\mathsf{lam_P} & : & \mathsf{Ty_P} \to (\mathsf{Tm_P} \to \mathsf{Tm_P}) \to \mathsf{Tm_P} & \lambda A.\, M \\
\mathsf{tylam_P} & : & (\mathsf{Ty_P} \to \mathsf{Tm_P}) \to \mathsf{Tm_P} & \Lambda.\, M
\end{array}
$$

Figure 6.1: HOAS signature for PLC, the two-sorted variant of System F.

$$
\begin{array}{rcll}
\mathsf{Tm}_\lambda & : & \mathbf{Type} & \\
 & & & \\
\mathsf{star}_\lambda, \mathsf{box}_\lambda & : & \mathsf{Tm}_\lambda & *, \Box \\
\mathsf{prod}_\lambda & : & \mathsf{Tm}_\lambda \to (\mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda) \to \mathsf{Tm}_\lambda & \Pi S.\, T \\
\mathsf{app}_\lambda & : & \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda & S\, T \\
\mathsf{lam}_\lambda & : & \mathsf{Tm}_\lambda \to (\mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda) \to \mathsf{Tm}_\lambda & \lambda S.\, T
\end{array}
$$

Figure 6.2: HOAS signature for the single-sorted PTS λ2.

contextual information is usually handled by the implementation of the specification layer. The exact details do, however, crucially depend on the system in question. For now, we simply assume that our host language somehow keeps track of the set of predicate instances that hold at any given point of a derivation. We refer to this dynamic set of assumptions as the **ambient reasoning context**.

The specification layer allows the user to provide declarations and thus specify how new valid predicate instances can be derived from existing instances in the ambient context. We will again use inference rules to express such declarations. So for $P, Q, R : \mathbf{o}$ we would write

$$\frac{P \quad Q}{R}$$

to declare, that if P and Q are known to hold, then it can be inferred that R also holds.

In order to be able to declare our typing disciplines in terms of inference declarations we require two further features of our specification layer. The first is the notion of **hypothetical premises**, which we write $P \Rightarrow Q$. A declaration of the form

$$\frac{P \qquad Q_1 \Rightarrow\!\blacktriangleright Q_2}{R}$$

should be understood as follows: R can be inferred, whenever P and Q_2 hold or can be inferred; for the inference of Q_2, Q_1 is added to the ambient set of assumptions. That is, hypothetical premises enable the controlled extension of the ambient derivation context.

The last missing ingredient is the ability to **locally quantify** in a given premise over individuals of the known type families, that is in our case, syntactic expressions. We express such quantifications as $\mathbf{\Pi}x.Q[x]$, where $Q[x]$ denotes a premise Q with a potential free occurrence of the variable x.

We can now put all these ingredients together and express the typing discipline of PLC with the set of declarations shown in Figure 6.3. The process of derivation in this formalism is best illustrated with an example. Let us therefore consider in detail, how the well-formedness of a universally quantified type, that is the proposition $(\forall.\, A)$ **ty**, is derived. Recall that A has type $\mathsf{Ty_P} \to \mathsf{Ty_P}$. The main idea is that $\forall.\, A$ is a well-formed type, whenever $A\langle B\rangle$ is well-formed for some well-formed type B, where $A\langle B\rangle$ again denotes specification layer application. The declaration $\mathsf{I}^{\forall}_{\mathsf{P}}$ now combines local quantification to abstract over the substituted type B as x and a hypothetical premise to ensure the well-formedness of the abstracted type x. Note that this rule has only a single premise. The two rules for type and term abstraction, $\mathsf{T}^{\Lambda}_{\mathsf{P}}$ and $\mathsf{T}^{\lambda}_{\mathsf{P}}$, work similarly, but it should be noted that for term abstraction the local quantification is over a term x (of type $\mathsf{Tm_P}$) for which a derivable (object-level) typing $x :_{\mathsf{P}} A$ is assumed in the hypothetical premise. Observe also, how specification layer application is used to implement the instantiation of the polymorphic type in the type application rule $\mathsf{T}^{\mathsf{tyapp}}_{\mathsf{P}}$ as $B\langle A\rangle$.

The process we have just described is often referred to as **mobility of binders** [MN12, Section 7.3]. We convert an object-level binder $(\mathsf{all_P}, \mathsf{lam_P}, \mathsf{tylam_P})$ into a universally quantified specification layer binder $(\mathbf{\Pi}x.Q[x])$. As we will see shortly this process continues further, once we start reasoning about our definitions, where specification layer quantifiers transform into reasoning layer quantifiers, and subsequently into premises of reasoning contexts.

We can proceed similarly to define the $\lambda 2$ typing discipline. First of all it is straightforward to fix the two universes like this:

$$\frac{}{\mathcal{U}\ *}\ \mathsf{U}^{*}_{\lambda} \qquad \frac{}{\mathcal{U}\ \Box}\ \mathsf{U}^{\Box}_{\lambda}$$

Note that we could have equivalently moved the two constants $*$ and $\Box$ to a separate type family and embedded them with a dedicated sort-constructor into Tm_{λ}, with negligible effect on the subsequent development.

The actual typing discipline is then encoded with the set of declarations given in Figure 6.4.

$$\frac{A\ \mathbf{ty} \qquad B\ \mathbf{ty}}{(A \to B)\ \mathbf{ty}}\ \mathsf{I}^{\to}_{\mathsf{P}} \qquad \frac{\mathbf{\Pi} x.\ x\ \mathbf{ty} \Rightarrow A\langle x\rangle\ \mathbf{ty}}{(\forall.\, A)\ \mathbf{ty}}\ \mathsf{I}^{\forall}_{\mathsf{P}}$$

$$\frac{M :_{\mathsf{P}} A \to B \qquad N :_{\mathsf{P}} A}{M\,N :_{\mathsf{P}} B}\ \mathsf{T}^{\mathsf{app}}_{\mathsf{P}} \qquad \frac{M :_{\mathsf{P}} \forall.\, B \qquad A\ \mathbf{ty}}{M\,A :_{\mathsf{P}} B\langle A\rangle}\ \mathsf{T}^{\mathsf{tyapp}}_{\mathsf{P}}$$

$$\frac{A\ \mathbf{ty} \qquad \mathbf{\Pi} x.\ x :_{\mathsf{P}} A \Rightarrow M\langle x\rangle :_{\mathsf{P}} B}{\lambda A.\, M :_{\mathsf{P}} A \to B}\ \mathsf{T}^{\lambda}_{\mathsf{P}} \qquad \frac{\mathbf{\Pi} x.\ x\ \mathbf{ty} \Rightarrow M\langle x\rangle :_{\mathsf{P}} A\langle x\rangle}{\Lambda.\, M :_{\mathsf{P}} \forall.\, A}\ \mathsf{T}^{\Lambda}_{\mathsf{P}}$$

Figure 6.3: Type system of PLC using hypothetical and locally quantified premises.

$$\frac{}{* :_{\lambda} \Box}\ \mathsf{T}^{\mathsf{ax}}_{\lambda} \qquad \frac{S :_{\lambda} \Pi U.\, V \qquad T :_{\lambda} U}{S\,T :_{\lambda} V\langle T\rangle}\ \mathsf{T}^{\mathsf{app}}_{\lambda} \qquad \frac{S :_{\lambda} T \qquad \mathcal{U}\ T \qquad \mathbf{\Pi} x.\ x :_{\lambda} S \Rightarrow U\langle x\rangle :_{\lambda} *}{\Pi S.\, U :_{\lambda} *}\ \mathsf{T}^{\Pi}_{\lambda}$$

$$\frac{S :_{\lambda} T \qquad \mathcal{U}\ T \qquad \mathbf{\Pi} x.\ x :_{\lambda} S \Rightarrow V\langle x\rangle :_{\lambda} * \qquad \mathbf{\Pi} x.\ x :_{\lambda} S \Rightarrow U\langle x\rangle :_{\lambda} V\langle x\rangle}{\lambda S.\, U :_{\lambda} \Pi S.\, V}\ \mathsf{T}^{\lambda}_{\lambda}$$

Figure 6.4: Type system of λ2 using hypothetical and locally quantified premises.

At this point we have to make two remarks. First, we do not implement the PTS framework generically, as we did for our de Bruijn development, but directly define the concrete PTS λ2. Second, and more important, we consider a simplified PTS definition without conversion, which is justified by the normality of System F types. For a discussion of both points we refer the reader to the end of this chapter.

We define our type- and term-level correspondence relations, similar to the type systems above, as specification-level predicates, with corresponding inference declarations. The concrete definitions are shown in Figure 6.5. Observe how we again leave the tracking of contextual information implicit and instead rely on hypothetical premises and local quantification. Let us consider the rules in detail to see how the various features interact.

The easiest are probably the two declarations $\mathsf{R}^{\mathsf{app}}_{\approx}$ and $\mathsf{R}^{\mathsf{tyapp}}_{\approx}$ for related applications. Note, though, how the relatedness of a λ2 term T, either to a PLC term N or a PLC type A determines whether the λ2 application $S\,T$ relates to a PLC term- or type-application, respectively. As before we will eventually have to ensure that these alternatives are mutually exclusive. All other rules involve binders at some point and therefore make use of local quantification to instantiate the embedded host functions with abstract terms and types. The rules $\mathsf{R}^{\lambda}_{\approx}$ and $\mathsf{R}^{\Lambda}_{\approx}$ which relate abstractions and the rule $\mathsf{R}^{\forall}_{\sim}$ for universal quantification are all similar in that they quantify over two entities which are related at some level, before being passed into the respective bodies of the involved binders. Take for example $\mathsf{R}^{\Lambda}_{\approx}$, the correspondence

$$\begin{array}{rcl} \mathsf{tyrel} & : & \mathsf{Ty_P} \to \mathsf{Tm}_\lambda \to \mathbf{o} \\ \mathsf{tmrel} & : & \mathsf{Tm_P} \to \mathsf{Tm}_\lambda \to \mathbf{o} \end{array} \qquad \begin{array}{l} A \sim S \\ M \approx S \end{array}$$

$$\frac{A \sim S \qquad \mathbf{\Pi} x.\ B \sim T\langle x\rangle}{A \to B \sim \Pi S.\, T}\ \mathsf{R}^{\to}_{\sim} \qquad \frac{\mathbf{\Pi} xy.\ x \sim y \Rrightarrow A\langle x\rangle \sim S\langle y\rangle}{\forall.\, A \sim \Pi *.\, S}\ \mathsf{R}^{\forall}_{\sim}$$

$$\frac{M \approx S \qquad N \approx T}{M\ N \approx S\ T}\ \mathsf{R}^{\mathsf{app}}_{\approx} \qquad \frac{M \approx S \qquad A \sim T}{M\ A \approx S\ T}\ \mathsf{R}^{\mathsf{tyapp}}_{\approx}$$

$$\frac{A \sim S \qquad \mathbf{\Pi} xy.\ x \approx y \Rrightarrow M\langle x\rangle \approx T\langle y\rangle}{\lambda A.\, M \approx \lambda S.\, T}\ \mathsf{R}^{\lambda}_{\approx} \qquad \frac{\mathbf{\Pi} xy.\ x \sim y \Rrightarrow M\langle x\rangle \approx S\langle y\rangle}{\Lambda.\, M \approx \lambda *.\, S}\ \mathsf{R}^{\Lambda}_{\approx}$$

Figure 6.5: HOAS variant of the correspondence relation.

rule for type abstraction, where the bodies M and S should correspond as terms whenever we apply them to related types x and y. The rule $\mathsf{R}^{\to}_{\sim}$ for implications is somewhat subtle. The first premise is straightforward and simply requires that the domains A and S of the function types are related as types. For the codomains it is helpful to recall the involved types. We have $B : \mathsf{Ty_P}$ and $T : \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda$, where the latter is in principle dependent. Since we do however require that $B \sim T\langle x\rangle$ for an arbitrary $\lambda 2$ term x, it should be intuitively clear that T has to be a vacuous host-level abstraction. One might ask, why we did not make this vacuous dependence more explicit and formulate the rule instead as follows.

$$\frac{A \sim S \qquad B \sim T}{A \to B \sim \Pi S.\, (\lambda x \Rightarrow T)}\ \hat{\mathsf{R}}^{\to}_{\sim}$$

The short answer is, that the correspondence relation is designed to act as a bridge between two distinct type systems. And as such it should structurally fit to both systems as close as possible. The proposed alternative rule $\hat{\mathsf{R}}^{\to}_{\sim}$ is undesirable since it exhibits a subtle mismatch with the PTS typing discipline of $\lambda 2$. We need some further background to fully grasp the involved complication so we defer an in-depth discussion to the end of this chapter.

Also note in general, how the two correspondence relations distinguish between semantic $\lambda 2$ types and terms, which are both elements of the syntactic type Tm_λ. That is, when $\_ \sim S$ holds, we are looking at a type, while $\_ \approx S$ labels S as a semantic term. It is easy to see that both $\lambda 2$ universes, $*$ and $\Box$, appear in neither of the two relations (since they are neither semantic types nor semantic terms).

The final claim is of course based on the premise that the ambient, implicit reasoning context does not contain assumptions of, e.g., the form $A \sim *$. As we will see shortly, controlling the exact composition of the ambient reasoning context is one of the main challenges of a HOAS development.

6.3 Subordination

The way in which we have defined new types and their constructors in the preceding sections originates from the school of formalisms known as logical frameworks (LF) [HHP87]. When we work with such LF types, it is often useful to know whether certain terms may or may not appear as subterms within other terms. These subterm occurrence dependencies can be captured with a partial order among the defined type families, known as **subordination**. The concept was introduced in [Vir96, Vir99] as a crucial ingredient of type-safe LF-rewriting. A more recent exposition can be found in [HL07]. We recap the core ideas as far as they apply to our present work.

Let T_A and T_B be two LF type families. When T_A is subordinate to T_B, written $T_A \preceq T_B$, then *terms of type T_A can appear in terms of type T_B, or in indices of the type family T_B*. Conversely $T_A \not\preceq T_B$ indicates that terms of type T_A cannot appear in such positions. As an example, let us recall Figure 6.1 which introduces the two type families $\mathsf{Ty_P}$ and $\mathsf{Tm_P}$. The given set of constants admits the following subordination information.

$$\mathsf{Ty_P} \preceq \mathsf{Ty_P} \qquad\qquad \mathsf{Tm_P} \not\preceq \mathsf{Ty_P}$$
$$\mathsf{Ty_P}, \mathsf{Tm_P} \preceq \mathsf{Tm_P}$$

The fact that $\mathsf{Tm_P} \not\preceq \mathsf{Ty_P}$ can now be exploited in two ways.

Firstly, consider the derivation of a judgement A **ty**. The subordination order tells us that A may potentially contain subterms of type $\mathsf{Ty_P}$ but definitely not subterms of type $\mathsf{Tm_P}$. We can therefore safely remove any occurrences of terms of type $\mathsf{Tm_P}$ from the reasoning context under which A **ty** is to be established. In other words, negated subordination justifies certain forms of context strengthening.

The second use relates to the formation of abstractions. More precisely, it allows us to identify certain abstractions which are guaranteed to be vacuous, solely based on their type. Consider for example the abstraction $\lambda x{:}\mathsf{Tm_P} \Rightarrow A : \mathsf{Tm_P} \to \mathsf{Ty_P}$. Any instance of $(\lambda x{:}\mathsf{Tm_P} \Rightarrow A)\langle M \rangle$, can safely be replaced by just A, even without knowing A's internal structure, since the abstraction is necessarily vacuous. We will revisit this particular scenario when we discuss Abella's usage of *raising* in Section 6.4.1.

Note that is easy to compute, exhaustively, all subordinates of a given type family as soon as all constants of that family are known. This is immediate for systems that operate on a closed-world assumption, like Beluga. Meanwhile open-world systems, like Abella, allow the delayed addition of new constants to any given type family, so they require the user to inform the system from where on a type family should be considered closed.[3] Both Abella and Beluga are equipped to automatically compute the subordination order for all eligible type families, and also tacitly exploit it in the two ways outlined above.

We will now proceed to the actual formalisations of the presented setup, which as before establishes that the two correspondence relations are functional and injective,

[3] Abella does not allow the closing of the formula type **o**.

as well as total and judgement-preserving on the respective well-formed and -typed fragments.

6.4 Abella Implementation

As mentioned above, the proof system Abella [BCG+14] is a two-level logic system. For its specification layer, it utilises the declarative logical programming language λProlog [NM88, FGH+88], which is a higher-order extension of Prolog [CR93, Bra13]. Since Abella's reasoning layer is able to inspect (and interact with) the operational behaviour of the specification layer it is useful to understand how λProlog works in isolation. To this end, we take a small detour into logical programming.

Logical programs consist of a set of declarations, which each essentially amount to an inference rule, as well as a single goal clause. Both declarations and the goal clause may use logical variables. The underlying execution model is a resolution strategy that tries to find a derivation of the goal clause from the set of declarations. The procedure is unification-based to deal with the variables and computes a most general unifier, if the clause is at all derivable. One of the key resolution steps is **backchaining**, which is the process of unifying an atomic goal with the head of a declaration. This may discharge the respective goal, if the chosen rule is a fact (i.e. a rule without premises) or it may generate new subgoals.

Prolog logical programs restrict declarations to Horn clauses, that is, inference rules of the form

$$\frac{P_1 \qquad P_2 \qquad \dots \qquad P_n}{R}$$

where the P_i and R are atomic. Such Horn clauses are, however, unable to express the inference rules which we introduced in Section 6.2.

A specification language that is capable of implementing the required definitions is the **higher-order logic of hereditary Harrop formulas** (HOHH) [MN12]. It is based on the foundational notion of uniform proofs [MNPS91] and constitutes a higher-order extension of Horn clauses with hypothetical premises and local quantification. Before we proceed we should clarify that here 'higher-order' means that types like $(\mathsf{T} \to \mathsf{T}) \to \mathsf{T}$ may be formed and quantified over. Quantification over types involving $\mathbf{o}$ is, however, not permitted, which disallows quantification over predicates. For a more detailed discussion of this subtlety, see [MN12, Section I.3].

The programming language λProlog is based on HOHH formulas just like Prolog is based on Horn clauses. The higher-order nature of clauses does of course affect the resolution algorithm, which now has to employ **higher-order pattern unification** [Mil91, Qia93, Nip93] to obtain most general unifiers. The Teyjus system [NM99] is an implementation of λProlog which allows the execution of HOHH-based logical programs.

It is straightforward to transcribe our various HOAS definitions from Section 6.2 into a λProlog logical program. We could then use a system like Teyjus to *animate* our

definitions. When we, for example, want to compute the PTS term that corresponds to the PLC polymorphic identity function $\Lambda^a.\, \lambda^x a.\, x$, then we would enter the below query (first line), where K is a logical variable. The system would respond with the second line and yield the most general unification for K, which happens to be the corresponding, related PTS term.

$$?- \qquad \Lambda^a.\, \lambda^x a.\, x \approx K$$

$$- \qquad K := \lambda^y *.\, \lambda^z y.\, z$$

While it is interesting to experiment with a set of definitions in this fashion, and helpful to see under which conditions a derivation exists, we are here more interested in proofs *about* such derivations and unifications. This brings us to the heart of Abella, namely its reasoning layer and the connection between the two levels, which we discuss next.

6.4.1 The Abella Theorem Prover

Abella's reasoning layer is based on the logic $\mathcal{G}$ [GMN08, GMN11], which derives from the intuitionistic, predicative fragment of Church's simple theory of types [Chu40]. Notable (and conscious) omissions are the axioms of extensionality, infinity and choice. The built-in notion of equality is structural modulo $\alpha\beta\eta$-conversion. $\mathcal{G}$ admits the definition of inductive and coinductive predicates, but forbids custom inductive types and recursive functions. Inductive definitions with negative occurrences of the defined predicate are accepted with a warning, but the authors of Abella stress that the consistency of the system is likely compromised after adding such a definition [BCG$^+$14, Section 4.1].

The last, and probably most significant, extensions are the related notions of **nominal constants** (n_i) and **generic quantification** ($\nabla x.t$) [MT05]. The binder ∇ is pronounced 'nabla'. Since both nominal constants and generic quantification play a major role in the treatment of object-level variables in the subsequent proofs they warrant further study.

Nominal Constants

In $\mathcal{G}$, every type is inhabited by countably infinite nominal constants n_i, which are special syntactic forms that are provably distinct from each other and distinct from any other constants. We can for example easily prove

$$\mathsf{n}_2 \neq \mathsf{n}_5$$

and, recalling our HOAS encoding of ULC, also

$$\mathsf{n}_3 \neq \mathsf{lam}\ f.$$

Nominal constants are designed to represent, at the reasoning level, free variables of specification-level types with binding constructors and thus allow reasoning about open terms of such types.

Generic Quantification

Generic quantification is used to abstract over nominal constants. It is in some aspects similar to universal quantification, but it also exhibits some rather unusual properties.

A proof of $\nabla x.P$, where x may appear free in P, should be understood as choosing a fresh constant n_i and then proving $P[\mathsf{n}_i/x]$. In order to support the intended freshness semantics, $\mathcal{G}$ is equipped with a number of axioms that govern the behaviour of the ∇ binder. First of all we have *strengthening*, that is $\nabla x.P$ is equivalent to P, when x is not free in P, and *exchange*, which entails that $\nabla x.\nabla y.P$ and $\nabla y.\nabla x.P$ are equivalent. Additionally, ∇ *distributes* over propositional connectives of $\mathcal{G}$. Based on our earlier observation we clearly obtain $\nabla xy.x \neq y$ as a theorem, since opening the ∇ will select provably distinct nominal constants for x and y. Compare this to $\forall xy.x \neq y$, which is not generally provable, since x and y could very well be instantiated with the same value. Conversely, while $(\forall xy.p\,x\,y) \implies (\forall z.p\,z\,z)$ is easily provable, we do not obtain $(\nabla xy.p\,x\,y) \implies (\nabla z.p\,z\,z)$ as a theorem in $\mathcal{G}$. These two examples should provide an idea of the differences of universal and generic quantification in $\mathcal{G}$.

Note that the relative ordering of universal and generic quantifiers is highly relevant. To see why, let us consider the expression $\forall L.\nabla x.L$. The freshness guarantees which follow from the axioms governing ∇ ensure that x does not occur in L. More generally, a ∇-bound identifier is guaranteed to be fresh for everything that is bound above the respective ∇. This includes, in particular, those identifiers in Abella theorems without any explicit binder, as these are assumed to be *universally* bound at the top level of the theorem. In contrast to this, the term L in the body of $\nabla x.\forall L.L$, may have free occurrences of x. If we want to change the order of the quantifiers without changing the meaning of the expression, we have to employ a technique known as *raising*, which can be seen as dual to Skolemisation [MN12, cf. Section 4.4.1]. For the present example, this yields $\forall L.\nabla x.L\,x$. Since L is now syntactically prevented from having free occurrences of x, it has to take x as an explicit argument. Note that raising changes the type of L to incorporate this added dependency. One says that L *is raised over the type of* x. It may of course be the case that the type of x is not a subordinate of the type of L. In this case, no (vacuous) abstraction is formed and we simply obtain $\forall L.\nabla x.L$, without any changes to the type of L.

Logical Embedding

Now that we have a good idea of both the reasoning and the specification layer, let us consider how these are connected. In Abella, this is facilitated with a logical embedding that takes the form of a dedicated inductive predicate. Let $J : \mathbf{o}$ be a λProlog formula, and let us further assume that J has a λProlog derivation from the (implicit) set of premises $S = \{I_0, \ldots I_n\}$. Then $\{L^S \vdash J\} : \mathsf{prop}$ is a $\mathcal{G}$-proposition where $L^S : \mathsf{olist}$ is an *explicit* list representation of the *implicit* assumption set S. We have

$$\{L^S \vdash J\} \text{ holds in } \mathcal{G} \iff S \vdash J \text{ is derivable in } \lambda\text{Prolog}$$

When $S = \emptyset$ and respectively $L^{\emptyset} = \bullet$, we simply write $\{J\}$. Also note that we usually consider derivations in their embedded form and therefore drop the superscript on the assumption list and write $\{L \vdash J\}$.

Recall that λProlog supports hypothetical $(J_1 \Rightarrow J_2)$ and locally quantified $(\mathbf{\Pi} x.J[x])$ premises. The embedding treats them as follows.

$$\{L \vdash J_1 \Rightarrow J_2\} \rightsquigarrow \{L, J_1 \vdash J_2\} \qquad \{L \vdash \mathbf{\Pi} x.\ J[x]\} \rightsquigarrow \nabla x.\ \{L \vdash J[x]\}$$

The first simply reflects how λProlog treats hypothetical premises in the course of a derivation, that is by moving them to the current set of assumptions. The second is more interesting. Note how the λProlog binder $\mathbf{\Pi}$ is turned into the $\mathcal{G}$-binder ∇. A case analysis on the final expression at the reasoning level then produces $\{L \vdash J[\mathsf{n}_i]\}$, where n_i is automatically chosen to be sufficiently fresh. Note in particular that in most practical scenarios, the assumption list L is implicitly universally bound at the top level and therefore guaranteed to not contain the new constant n_i, in accordance with the usual freshness conditions on the underlying object-level typing rules. This shows, how free object-level variables appear as nominal constants at the reasoning level and completes the notion of binder mobility introduced above.

One of the core ideas that make Abella's two-level logic approach work is the fact that the metatheory of the specification logic, including monotonicity, instantiation and cut, can be established once, abstractly, and then exploited for the encoding of logical systems. These metatheoretical statements appear in Abella as properties of the logical embedding. They are exposed to the user via built-in tactics, since they are not expressible in $\mathcal{G}$. We present the ones which are of immediate relevance for us. For a more detailed treatment of this aspect, see [BCG+14, Section 8.4].

Fact 6.4.1 (Properties of $\{L \vdash J\}$) The logical embedding $\{L \vdash J\}$ satisfies cut and a nominal instantiation principle:

$$\{L \vdash I\} \implies \{L, I \vdash J\} \implies \{L \vdash J\} \qquad \text{(cut)}$$
$$\forall t{:}\mathsf{typeof}\ \mathsf{n}_i.\ \{L[\mathsf{n}_i] \vdash J[\mathsf{n}_i]\} \implies \{L[t] \vdash J[t]\} \qquad \text{(inst)}$$

We will make heavy use of these properties throughout our Abella proof in the following section, for example to quickly obtain substitutivity results for our object languages.

6.4.2 The Correspondence Proof

We now consider the actual correspondence proof in our Abella development. All λProlog definitions of our two syntactic systems are implemented exactly as shown in Section 6.2.[4]

Let us start by recalling that a significant portion of our de Bruijn development involved the proofs of various substitutivity properties. Here, these properties are

[4] For relevant files, see: `http://www.ps.uni-saarland.de/static/kaiser-diss/index.php`

trivial consequences of Fact 6.4.1, and in particular do not require the generalisation of a context morphism lemma. We demonstrate this with the substitution lemma for PLC type formation, and focus on the usage of generic quantification and nominal constants. Note that the presentation is intended to illustrate how the two layers of the system interact in practice and therefore somewhat lengthy. The underlying formal proof is a one-liner.

Lemma 6.4.2 (β-Substitutivity of PLC Type Formation)

$$\forall LAB.\ \nabla x.\ \{L, x\ \mathbf{ty} \vdash A\langle x\rangle\ \mathbf{ty}\} \implies \{L \vdash B\ \mathbf{ty}\} \implies \{L \vdash A\langle B\rangle\ \mathbf{ty}\}$$

Proof Due to the quantifier ordering, x is fresh for L, A and B, thus $L, x\ \mathbf{ty}$ is a reasonable typing context. Since the initial derivation under this context should have potential occurrences of x, we have to make this dependence explicit for A, which has type $\mathsf{Ty_P} \to \mathsf{Ty_P}$. To prove the result we have to show $\{L \vdash A\langle B\rangle\ \mathbf{ty}\}$ from the following set of assumptions, where n_1 is the freshly chosen nominal constant that results from opening the ∇.

$$\{L, \mathsf{n}_1\ \mathbf{ty} \vdash A\langle \mathsf{n}_1\rangle\ \mathbf{ty}\} \tag{H1}$$
$$\{L \vdash B\ \mathbf{ty}\} \tag{H2}$$

We use Fact 6.4.1 to instantiate $\mathsf{n}_1 \coloneqq B$ in (H1) and obtain

$$\{L, B\ \mathbf{ty} \vdash A\langle B\rangle\ \mathbf{ty}\} \tag{H3}$$

and then simply cut (H3) with (H2) to obtain the desired result. ∎

The β-substitutivity result for typing is established similarly.

When we now start to prove structural results about our definitions, say by inversion on a logically embedded derivation, we are quickly faced with a problem. Consider for example the following inversion principle for well-formed arrow types in PLC, which we could reasonably expect to hold.

$$\{L \vdash (A \to B)\ \mathbf{ty}\} \implies \{L \vdash A\ \mathbf{ty}\} \wedge \{L \vdash B\ \mathbf{ty}\}.$$

The problem with this statement is that the premise may hold not only due to structural reasons, as claimed, but also due to backchaining and the fact that $(A \to B)\ \mathbf{ty} \in L$. We recall that L is a list of specification-level propositions and as such may a priori contain arbitrary facts, which may not even be related to typing or type formation. It is hence clear that an arbitrary $\mathsf{olist}\ L$ does not faithfully reflect our understanding of a typing context.

To fix this mismatch we are going to define inductive **context predicates** that will constrain the shape of a given assumption list to one that constitutes a context of the kind we are interested in. For PLC this means that we only want judgements $J \in L$ to be of the form $\mathsf{n}_i\ \mathbf{ty}$, which indicates that n_i is a type variable, or of the form $\mathsf{n}_i :_\mathsf{P} A$, where n_i is a term variable and A is a well-formed type. Moreover, the

various nominal constants, that is variables, should be distinct in a given L. These ideas are captured in the following inductive definition of $\mathbb{C}_{\mathsf{P}}(L)$.

$$\frac{}{\mathbb{C}_{\mathsf{P}}(\bullet)} \qquad \frac{\mathbb{C}_{\mathsf{P}}(L)}{\mathbb{C}_{\mathsf{P}}(L, x\ \mathbf{ty})}\ x \notin L \qquad \frac{\mathbb{C}_{\mathsf{P}}(L) \quad \{L \vdash A\ \mathbf{ty}\}}{\mathbb{C}_{\mathsf{P}}(L, x :_{\mathsf{P}} A)}\ x \notin L, A$$

The freshness side-conditions are again implemented using generic quantification. That is, we have the following definition in Abella.

$$\begin{array}{ll} \mathsf{Define} & \mathbb{C}_{\mathsf{P}}(-) : \mathsf{olist} \to \mathsf{prop} \quad \mathsf{by} \\ & \mathbb{C}_{\mathsf{P}}(\bullet); \\ & \nabla x.\ \mathbb{C}_{\mathsf{P}}(L, x\ \mathbf{ty}) \coloneqq \mathbb{C}_{\mathsf{P}}(L); \\ & \nabla x.\ \mathbb{C}_{\mathsf{P}}(L, x :_{\mathsf{P}} A) \coloneqq \mathbb{C}_{\mathsf{P}}(L) \wedge \{L \vdash A\ \mathbf{ty}\}. \end{array}$$

Observe how the new variable x in the second and, respectively, the third clause is in each case locally quantified with ∇, while the parameters L and A are implicitly universally quantified at the top level of each clause. This ensures the required freshness. In the following we will only give the rule-based definition of such context predicates.

To make use of this definition, we require further auxiliary structures to recognise and isolate nominal constants as well as a **lookup lemma**. We define the following.

$$\begin{array}{lll} \langle\!\langle - \rangle\!\rangle_{\mathsf{P}} : \mathsf{Ty}_{\mathsf{P}} \to \mathsf{prop} & \coloneqq & \nabla x.\ \langle\!\langle x \rangle\!\rangle_{\mathsf{P}} \\ \langle\!\langle - \rangle\!\rangle_{\mathsf{P}} : \mathsf{Tm}_{\mathsf{P}} \to \mathsf{prop} & \coloneqq & \nabla x.\ \langle\!\langle x \rangle\!\rangle_{\mathsf{P}} \end{array}$$

Due to the generic quantification of x in each case, we know that $\langle\!\langle C \rangle\!\rangle_{\mathsf{P}}$ ensures $C = \mathsf{n}_i$, and similarly for $\langle\!\langle M \rangle\!\rangle_{\mathsf{P}}$. The overlapping notation is disambiguated by the type of the argument which will always be clear from the context. With these we can establish the following lookup inversion result.

Lemma 6.4.3 (PLC Context Lookup)

$$\begin{array}{l} \forall LK.\ \mathbb{C}_{\mathsf{P}}(L) \implies K \in L \implies \\ \qquad (\exists A.\ K = A\ \mathbf{ty} \wedge \langle\!\langle A \rangle\!\rangle_{\mathsf{P}}) \vee \\ \qquad (\exists MA.\ K = M :_{\mathsf{P}} A \wedge \langle\!\langle M \rangle\!\rangle_{\mathsf{P}} \wedge \{L \vdash A\ \mathbf{ty}\}) \end{array}$$

Proof The proof is by induction on $K \in L$ and we thus have $L = L', S$.

We start with the case where K unifies with S and discriminate on $\mathbb{C}_{\mathsf{P}}(L', K)$. This generates a nominal constant n_1 and either unifies K with $\mathsf{n}_1\ \mathbf{ty}$ or with $\mathsf{n}_1 :_{\mathsf{P}} A$ for some A satisfying $\{L' \vdash A\ \mathbf{ty}\}$. We can respectively close the left or right disjunct of the claim.

Otherwise, we consider $K \in L'$. From $\mathbb{C}_{\mathsf{P}}(L', S)$ we clearly obtain $\mathbb{C}_{\mathsf{P}}(L')$ and can thus use the inductive hypothesis, which yields all required pieces of information. Note though, that Abella is able to automatically employ a weakening result of the logical embedding to infer $\{L', S \vdash A\ \mathbf{ty}\}$ from $\{L' \vdash A\ \mathbf{ty}\}$. ∎

We now have all the ingredients to actually prove our inversion example. Observe that we have added the premise $\mathbb{C}_{\mathsf{P}}(L)$ to enforce the well-formedness of the context.

Lemma 6.4.4 (Inversion for Well-Formed PLC Arrow Types)

$$\forall LAB.\ \mathbb{C}_{\mathsf{P}}(L) \implies \{L \vdash (A \to B)\ \mathbf{ty}\} \implies \{L \vdash A\ \mathbf{ty}\} \wedge \{L \vdash B\ \mathbf{ty}\}$$

Proof The proof is as expected by discriminating on $\{L \vdash (A \to B)\ \mathbf{ty}\}$ and the structural case is trivial.

For the problematic backchaining case we obtain the following assumptions.

$$\mathbb{C}_{\mathsf{P}}(L) \tag{H1}$$
$$\{L \mid K \vdash (A \to B)\ \mathbf{ty}\} \tag{H2}$$
$$K \in L \tag{H3}$$

The notation $\{L \mid K \vdash J\}$ indicates that K is the focused element of L which was used for backchaining. Before we can proceed we have to determine the concrete nature of K. To this end, we can use Lemma 6.4.3 with (H1) and (H3).

This either unifies K with $X\ \mathbf{ty}$ such that $\langle\!\langle X \rangle\!\rangle_{\mathsf{P}}$ holds, or with $X :_{\mathsf{P}} A$. In the latter case, unification can close the goal, since $X :_{\mathsf{P}} A$ and $(A \to B)\ \mathbf{ty}$ have distinct head symbols and are therefore impossible to unify. In the other case the head symbols do match, but $\langle\!\langle X \rangle\!\rangle_{\mathsf{P}}$ tells us that $X = \mathsf{n}_1$. Since $\mathsf{n}_1 \neq A \to B$ (recall that nominal constants are distinct from declared constants) we can again discriminate on (H2) to close the case via unification failure. ∎

In the previous proof we had one case, where backchaining was attempted on a proposition with a head symbol that did not match the head symbol of the judgement under consideration. Such spurious side cases stem from the fact that our lookup lemma has a disjunctive conclusion to cover the various possible context entries. To prevent this from cluttering our future proofs we establish the following result.

Lemma 6.4.5 (Backchaining Inversion for PLC Type Formation)

$$\forall LKA.\ \mathbb{C}_{\mathsf{P}}(L) \implies K \in L \implies \{L \mid K \vdash A\ \mathbf{ty}\} \implies K = A\ \mathbf{ty} \wedge \langle\!\langle A \rangle\!\rangle_{\mathsf{P}}$$

Proof We use Lemma 6.4.3 to either obtain the desired goal, or a unification for K which makes the third premise impossible. ∎

To demonstrate how this simplifies our proofs, let us consider inversion for well-formed universal types as an analogue to the arrow types (Lemma 6.4.4).

Lemma 6.4.6 (Inversion for Well-Formed PLC Universal Types)

$$\forall LA.\ \mathbb{C}_{\mathsf{P}}(L) \implies \{L \vdash (\forall.\, A)\ \mathbf{ty}\} \implies \nabla x.\ \{L, x\ \mathbf{ty} \vdash A\langle x \rangle\ \mathbf{ty}\}$$

Proof The proof is by discriminating on $\{L \vdash (\forall.\, A)\ \mathbf{ty}\}$ and the structural case is again immediate. For the problematic backchaining case we use Lemma 6.4.5 to infer $\langle\!\langle \forall.\, A \rangle\!\rangle_{\mathsf{P}}$. The latter is impossible, since $\forall.\, A$ cannot be unified with a nominal constant, thus closing the case. ∎

Note that we have defined a single context predicate for PLC contexts which admits both type formation and typing assumptions. However, for judgements like

$$\{L, \mathsf{n}_1 :_{\mathsf{P}} A \vdash B\ \mathbf{ty}\}$$

a careful study of the PLC type formation rules reveals that the typing assumption for n_1 could not have been used in the derivation. Proving this formally is somewhat intricate, again due to the backchaining rule.

Lemma 6.4.7 (Context Strengthening for PLC Type Formation)

$$\forall LXAB.\ \mathbb{C}_{\mathsf{P}}(L) \implies \{L, X :_{\mathsf{P}} A \vdash B\ \mathbf{ty}\} \implies \{L \vdash B\ \mathbf{ty}\}$$

Proof By induction on $\{L, X :_{\mathsf{P}} A \vdash B\ \mathbf{ty}\}$. The arrow case is a straightforward consequence of the inductive hypothesis. The case for universal quantification is also easy, since the logical embedding is equipped with an exchange rule. When we instantiate the inductive hypothesis, it is implicitly used to change the given assumption $\{L, X :_{\mathsf{P}} A, \mathsf{n}_1\ \mathbf{ty} \vdash B\ \mathbf{ty}\}$ into $\{L, \mathsf{n}_1\ \mathbf{ty}, X :_{\mathsf{P}} A \vdash B\ \mathbf{ty}\}$ prior to the instantiation.

For the backchaining case we discriminate on $K \in L, X :_{\mathsf{P}} A$. The case $K = X :_{\mathsf{P}} A$ is clearly refutable, so let us consider $K \in L$. We apply Lemma 6.4.3 and only have to consider the case $K = B\ \mathbf{ty}$ (since $K = \mathsf{n}_1 :_{\mathsf{P}} A'$ is again easily refutable). Thus from $B\ \mathbf{ty} \in L$, we obtain $\{L \vdash B\ \mathbf{ty}\}$ via backchaining. ∎

We will later, mostly in the context of our injectivity and functionality proofs, encounter scenarios where we have to explicitly exploit the violation of freshness assumptions to refute certain proof branches. Technically, these refutations arise from the freshness assumptions as given through the ordering of quantifiers on the one hand and context occurrence assumptions in the form of olist membership on the other. A crucial ingredient is the following lemma, which establishes the vacuity of the abstraction K in $K\langle x \rangle \in L$ since x is fresh for L and K.

Lemma 6.4.8 (PLC Type Dependency Pruning)

$$\forall LK.\ \nabla x{:}\mathsf{Ty}_{\mathsf{P}}.\ K\langle x \rangle \in L \implies \exists R.\ K = (\lambda y \Rightarrow R)$$

Proof By induction on $K\langle \mathsf{n}_1 \rangle \in L$. We have $L = L', S$ and first consider unifying S with $K\langle \mathsf{n}_1 \rangle$, which, due to freshness assumptions, yields $K = (\lambda z \Rightarrow S)$. We can thus instantiate $R := S$ and have $(\lambda z \Rightarrow S) = (\lambda y \Rightarrow S)$, modulo α-equivalence. Otherwise we have $K\langle \mathsf{n}_1 \rangle \in L'$ and can apply the inductive hypothesis to yield $K = (\lambda z \Rightarrow S')$ for some S'. Clearly $R := S'$ closes the case. ∎

This result is primarily used in scenarios where K is a non-vacuous abstraction, to introduce impossible equalities like $(\lambda x \Rightarrow x) = (\lambda x \Rightarrow y)$ as a consequence of freshness violations. These equalities are then picked up by the unification algorithm to subsequently discharge the respective case. Note that we have an identical pruning lemma for PLC terms, where the generic quantification is over $\mathsf{Tm_P}$ instead. Abella does, unfortunately, not allow us to abstract over the concrete syntax type, so we cannot collapse the two results (as well as an analogue one for $\lambda 2$ terms) into a single statement.

For $\lambda 2$ we develop a similar set of definitions, including β-substitutivity and dependency pruning. The requisite context predicate is inductively defined as

$$\frac{}{\mathbb{C}_\lambda(\bullet)} \qquad \frac{\mathbb{C}_\lambda(L) \quad \exists U.\ \{L \vdash S :_\lambda U\} \wedge \{L \vdash \mathcal{U}\ U\}}{\mathbb{C}_\lambda(L, x :_\lambda S)}\ x \notin L, S$$

The corresponding recognition predicate for variables is

$$\langle\!\langle - \rangle\!\rangle_\lambda : \mathsf{Tm}_\lambda \to \mathsf{prop} \quad := \quad \nabla x.\ \langle\!\langle x \rangle\!\rangle_\lambda$$

and we formulate the following lookup lemma.

Fact 6.4.9 ($\lambda 2$ Context Lookup)

$$\begin{aligned}&\forall LK.\ \mathbb{C}_\lambda(L) \implies K \in L \implies \\ &\qquad (\exists TSU.\ K = T :_\lambda S \wedge \langle\!\langle T \rangle\!\rangle_\lambda \wedge \{L \vdash S :_\lambda U\}) \wedge \{L \vdash \mathcal{U}\ U\}\end{aligned}$$

Note that since proper $\lambda 2$ contexts only contain a single form of judgement, there is no need to establish an analogue to Lemma 6.4.5. We do, however, establish two simple and easy to prove results for $\lambda 2$ which arise frequently. The first is the fact that $*$ does not inhabit itself, namely

Fact 6.4.10 $\quad \mathbb{C}_\lambda(L) \implies \{L \vdash * :_\lambda *\} \implies \bot$

and the second is a corollary of β-substitutivity that provides a typing for the instantiated codomain of a dependent function type body.

Fact 6.4.11

$$\mathbb{C}_\lambda(L) \implies \{L \vdash (\Pi S.\, T) :_\lambda U\} \implies \{L \vdash V :_\lambda S\} \implies \{L \vdash T\langle V \rangle :_\lambda *\}$$

Now that we have settled various inversion principles and fixed the notions of well-formed typing contexts, we can turn our attention to the proofs of the four properties for $\sim$ and $\approx$. Here we will slightly change the order (with respect to our de Bruijn developments) in which we prove the results. We first consider injectivity and functionality for both the type and the term relation. The second part will then cover the totality and judgement-preservation results.

Part I, Functionality and Injectivity

We start by observing that the substitutivity of the correspondence relation for types is again an immediate consequence of Abella's logical embedding.

Lemma 6.4.12 (β-Substitutivity of the Type Relation)

$$\forall LASBT.\ \nabla xy.\ \{L, x \sim y \vdash A\langle x\rangle \sim S\langle y\rangle\} \implies \{L \vdash B \sim T\} \implies \{L \vdash A\langle B\rangle \sim S\langle T\rangle\}$$

Proof Analogue to Lemma 6.4.2. ∎

A more interesting point, however, is the way in which we track which free type and term variables are considered as related, when we deal with the correspondence of open expressions. From our de Bruijn development we recall, that we had to introduce quite a number of auxiliary structures with associated properties to maintain this kind of information and dedicated a full section (Section 4.3) to the topic. Here we instead capture the required structural invariants with another simple inductive context predicate $\mathbb{C}^{\sim}_{\approx}(-)$.

$$\frac{}{\mathbb{C}^{\sim}_{\approx}(\bullet)} \qquad \frac{\mathbb{C}^{\sim}_{\approx}(L)}{\mathbb{C}^{\sim}_{\approx}(L, x \sim y)}\ x, y \notin L \qquad \frac{\mathbb{C}^{\sim}_{\approx}(L)}{\mathbb{C}^{\sim}_{\approx}(L, x \approx y)}\ x, y \notin L$$

This definition is structurally similar to $\mathbb{C}_{\mathsf{P}}(-)$, in that it restricts the list of a priori arbitrary propositions to only two possible forms, namely two nominals related as types ($\mathsf{n}_1 \sim \mathsf{n}_2$) or as terms ($\mathsf{n}_1 \approx \mathsf{n}_2$). It is in fact a bit simpler, since neither of the rules imposes any additional side conditions beyond the obvious freshness constraints. Analogue to earlier constructions in this chapter, we again formulate recognition predicates which capture pairs of nominals, both at the level of types and of terms.

$$\begin{aligned} \langle\!\langle -, - \rangle\!\rangle_{\sim} : \mathsf{Ty}_{\mathsf{P}} \to \mathsf{Tm}_{\lambda} \to \mathsf{prop} &\quad := \quad \nabla x, y.\ \langle\!\langle x, y \rangle\!\rangle_{\sim} \\ \langle\!\langle -, - \rangle\!\rangle_{\approx} : \mathsf{Tm}_{\mathsf{P}} \to \mathsf{Tm}_{\lambda} \to \mathsf{prop} &\quad := \quad \nabla x, y.\ \langle\!\langle x, y \rangle\!\rangle_{\approx} \end{aligned}$$

With these definitions in place we can now formulate a context lookup lemma for our notion of contexts of related variables.

Fact 6.4.13 (Relational Context Lookup)

$$\begin{aligned} \forall LK.\ \mathbb{C}^{\sim}_{\approx}(L) \implies K \in L \implies \\ (\exists AS.\ K = A \sim S \wedge \langle\!\langle A, S \rangle\!\rangle_{\sim}) \vee \\ (\exists MT.\ K = M \approx T \wedge \langle\!\langle M, T \rangle\!\rangle_{\approx}) \end{aligned}$$

Note that the conclusion is disjunctive, similar to PLC context lookup (Lemma 6.4.3), due to the fact that relational contexts carry assumptions about both related type variables as well as term variables. We therefore establish two inversion

lemmas that ensure that only matching entries are considered during backchaining. We also have a strengthening result, similar to Lemma 6.4.7, which allows us to ignore assumptions about term variables for derivations of related types. Since their statements and proofs use principles we have already seen repeatedly we do not go into any further detail here.

From our de Bruijn proofs of functionality and injectivity we recall that, for the variable cases, we had to ensure that the relational context in question was also functional, or respectively, injective. Here, the use of generic quantification (∇) in the context predicate $\mathbb{C}^{\sim}_{\approx}(-)$ ensures that we only ever consider functional and injective contexts. We demonstrate this with functionality.

Lemma 6.4.14 (Functionality of $\mathbb{C}^{\sim}_{\approx}(L)$ for Types)

$$\forall LAST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies A \sim S \in L \implies A \sim T \in L \implies S = T$$

Proof By induction on $A \sim S \in L$. We consider $L = L', J$ and clearly have $\mathbb{C}^{\sim}_{\approx}(L')$.

In the base case we have $J = A \sim S$ and discriminate on $A \sim T \in L', A \sim S$, which either unifies S and T as required or we have $A \sim T \in L'$. We do, however, also have $\mathbb{C}^{\sim}_{\approx}(L', A \sim S)$, which, by the definition of $\mathbb{C}^{\sim}_{\approx}(-)$, forces $A = \mathsf{n}_1$ and $S = \mathsf{n}_2$, where the nominal constants are fresh for L'. We are hence considering $\mathsf{n}_1 \sim T'\langle \mathsf{n}_2 \rangle \in L'$, where T' is T raised over the new potential dependency on $\mathsf{n}_2 : \mathsf{Tm}_\lambda$. Note that T is not raised over $\mathsf{n}_1 : \mathsf{Ty}_\mathsf{P}$ due to the available subordination information, concretely $\mathsf{Ty}_\mathsf{P} \not\preceq \mathsf{Tm}_\lambda$. We can now abstract over n_1, which yields $(\lambda x \Rightarrow x \sim T'\langle \mathsf{n}_2 \rangle)\langle \mathsf{n}_1 \rangle \in L'$, and then apply Lemma 6.4.8. This in turn forces the unification of the non-vacuous abstraction $\lambda x \Rightarrow x \sim T'\langle \mathsf{n}_2 \rangle$ with a vacuous abstraction. This always fails and thus discharges the case.

For the inductive case we have $A \sim S \in L'$ and again discriminate on $A \sim T \in L', J$. For $J = A \sim T$ we again obtain a violation of freshness which is dual to the one we covered in the base case and also refuted with Lemma 6.4.8. This leaves the case $A \sim T \in L'$, which finally admits the application of the inductive hypothesis to close the proof. ∎

The following three results are established with similar proofs. The only difference among them are the respectively employed pruning lemmas (essentially analogues of Lemma 6.4.8) for the involved syntax types to handle the various freshness violations.

Fact 6.4.15 $\forall LMST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies M \approx S \in L \implies M \approx T \in L \implies S = T$

Fact 6.4.16 $\forall LABS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies A \sim S \in L \implies B \sim S \in L \implies A = B$

Fact 6.4.17 $\forall LMNS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies M \approx S \in L \implies N \approx S \in L \implies M = N$

Before we can now lift these to the actual correspondence relations, we need two further ingredients.

The first is the disjointness of codomains, also known as a *no-clash theorem*, for the type and term relation. As for the de Bruijn case, we first establish this for

contexts, where the result follows from the built-in freshness guarantees of generic quantification, and then lift it to the correspondence relations. Note that generic quantification ensures freshness, irrespective of the type of the quantified parameter. Thus, in the two extension rules of our definition of $\mathbb{C}^{\sim}_{\approx}(-)$, the new $\lambda 2$ variable y is distinct from all preexisting variables, regardless of the semantic scope. Now, since all $\lambda 2$ variables in a well-formed context are distinct, it should be intuitively clear, that a clash on the $\lambda 2$ side (or any clash for that matter) is impossible. The required reasoning to establish this formally is similar to what we did above for the functionality and injectivity of well-formed contexts.

Lemma 6.4.18 (No-Clash for $\mathbb{C}^{\sim}_{\approx}(L)$)

$$\forall LAMS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies A \sim S \in L \implies M \approx S \in L \implies \bot$$

Proof By induction on $A \sim S \in L$. We consider $L = L', J$.

In the base case we have $J = A \sim S$. From $\mathbb{C}^{\sim}_{\approx}(L', A \sim S)$ it follows that $A = \mathsf{n}_1$ and $S = \mathsf{n}_2$ where the nominal constants are fresh for L'. Now $M'\langle \mathsf{n}_1 \rangle \approx \mathsf{n}_2 \in L', \mathsf{n}_1 \sim \mathsf{n}_2$ simplifies to $M'\langle \mathsf{n}_1 \rangle \approx \mathsf{n}_2 \in L'$, since $\approx$ and $\sim$ do not unify. We can abstract over n_2 and employ the pruning lemma for $\lambda 2$ terms to refute the case.

For the inductive case we discriminate on $\mathbb{C}^{\sim}_{\approx}(L', J)$ which unifies J either with $\mathsf{n}_1 \sim \mathsf{n}_2$ or with $\mathsf{n}_1 \approx \mathsf{n}_2$. In the former case everything simplifies so that we can directly use the inductive hypothesis. In the latter case we need to further analyse the membership assumption for the term relation which either produces another freshness violation or again a setting where the inductive hypothesis is applicable. ∎

Lemma 6.4.19 (No-Clash for $\sim$ and $\approx$) We lift the previous result to the correspondence relations.

$$\forall LAMS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim S\} \implies \{L \vdash M \approx S\} \implies \bot$$

Proof We first discriminate on $\{L \vdash A \sim S\}$, which yields two structural and one backchaining case.

For the two structural cases we then discriminate on $\{L \vdash M \approx S\}$, where only backchaining is possible since no structural $\approx$-rule has arrow types or universal types as their right-hand side. In each case we can then use the backchaining inversion lemma for $\approx$ to infer $\langle\langle M, S \rangle\rangle_{\approx}$, which is impossible since S has an actual constructor in head position and is therefore not a nominal.

This leaves the case $\{L \mid K \vdash A \sim S\}$, with $K \in L$. The backchaining inversion lemma for $\sim$ now tells us that $A = \mathsf{n}_1$, and more importantly $S = \mathsf{n}_2$. Therefore $\{L'\langle \mathsf{n}_1 \rangle\langle \mathsf{n}_2 \rangle \vdash M'\langle \mathsf{n}_1 \rangle \approx \mathsf{n}_2\}$ could only have been obtained via backchaining as well, where L' and M' are the suitably raised versions of L and M. But then we have $\mathbb{C}^{\sim}_{\approx}(L'\langle \mathsf{n}_1 \rangle\langle \mathsf{n}_2 \rangle)$, $\mathsf{n}_1 \sim \mathsf{n}_2 \in L'\langle \mathsf{n}_1 \rangle\langle \mathsf{n}_2 \rangle$ and $M'\langle \mathsf{n}_1 \rangle \approx \mathsf{n}_2 \in L'\langle \mathsf{n}_1 \rangle\langle \mathsf{n}_2 \rangle$, which together admit a refutation via Lemma 6.4.18.

The second missing ingredient is a rather extensive list of inversion principles for the logical embedding of relational derivations, where the top structural constant of

one of the involved terms or types is known. The primary purpose of each of these results is to rule out impossible backchaining derivations. In addition to that, the inversion principles for the cases where the head constant on the $\lambda 2$ side is known yield the necessary disambiguation of the unified $\lambda 2$ syntax. Since the latter is crucial for the injectivity results, we look at the $\lambda 2$ inversion principles in detail while we omit the less interesting PLC results for the sake of brevity.

First of all, let us establish the fact that the $\lambda 2$ sort $*$ is never related to a PLC type.

Fact 6.4.20 $\forall LA.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim *\} \implies \bot$

With this first principle we can now tackle the remaining inversion results and start with the relation for types where the $\lambda 2$ term is a dependent function type. Observe, how information about the domain of the function type is used as an auxiliary premise to disambiguate the situation and identify a PLC arrow type as the correct counterpart.

Lemma 6.4.21

$$\begin{aligned}\forall LABST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim \Pi S.\, T\} \implies \{L \vdash B \sim S\} \implies \\ \exists CD.\ \nabla x.\ A = (C \to D) \wedge \{L \vdash C \sim S\} \wedge \{L \vdash D \sim T\langle x\rangle\}\end{aligned}$$

Proof By discriminating on $\{L \vdash A \sim \Pi S.\, T\}$, which yields three cases. The correct case unifies A with some PLC arrow type and yields all the results to instantiate the existential.

Otherwise A is unified with a universally quantified type, but this also sets $S = *$. We therefore obtain $\{L \vdash B \sim *\}$, which is impossible due to Fact 6.4.20.

Lastly we have the backchaining case, but since a suitable backchaining inversion lemma fails to unify $\Pi S.\, T$ with a nominal constant we are again done. ∎

Note that in the previous result we could not (yet) have equated the PLC types B and C, even though they will eventually turn out to be the same. Their equality relies on the injectivity of $\sim$, which we have not yet established. More precisely, the inversion results we are currently developing are used in the proof of injectivity and can therefore not themselves depend on the property, lest we produce a circular argument. We will encounter the same issue also at the term level.

Let us continue and establish the dual to the preceding result, that is inversion for dependent function types that correspond to universally quantified PLC types instead. Here the $\lambda 2$ domain is fixed to $*$, which obviates the need for an auxiliary premise.

Lemma 6.4.22

$$\begin{aligned}\forall LAS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim \Pi *.\, S\} \implies \\ \exists B.\ \nabla xy.\ A = (\forall.\, B) \wedge \{L, x \sim y \vdash B\langle x\rangle \sim S\langle y\rangle\}\end{aligned}$$

Proof By discriminating on $\{L \vdash A \sim \Pi *.\, S\}$, which again yields three cases. The backchaining case is discharged as before, and for the unification of A with some $\forall.\, B$ the goal is easy to prove. Meanwhile, the case where $A = (C \to D)$ also forces $\{L \vdash C \sim *\}$, which again conflicts with Fact 6.4.20. ∎

For abstractions at the term level we obtain two inversion principles that are very similar to the two results we have established for function types. We therefore only give the statements. The same reasoning about syntax disambiguation applies.

Fact 6.4.23

$$\forall LMAST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx \lambda S.\, T\} \implies \{L \vdash A \sim S\} \implies \\ \exists NB.\ \nabla xy.\ M = (\lambda B.\, N) \wedge \{L \vdash B \sim S\} \wedge \{L, x \approx y \vdash N\langle x\rangle \approx T\langle y\rangle\}$$

Fact 6.4.24

$$\forall LMS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx \lambda *.\, S\} \implies \\ \exists N.\ \nabla xy.\ M = (\Lambda.\, N) \wedge \{L, x \sim y \vdash N\langle x\rangle \approx S\langle y\rangle\}$$

For the disambiguation of $\lambda 2$ applications we need to modify our strategy somewhat in order to discharge the respective problematic structural case. The main idea is to determine the semantic type of the $\lambda 2$ argument part of the application with an auxiliary premise and then exploit the fact that our two relations have disjoint ranges.

Lemma 6.4.25

$$\forall LMKST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx S\,T\} \implies \{L \vdash K \approx T\} \implies \\ \exists NO.\ M = N\,O \wedge \{L \vdash N \approx S\} \wedge \{L \vdash O \approx T\}$$

Proof By discriminating on $\{L \vdash M \approx S\,T\}$, which yields three cases. The correct structural case and the problematic backchaining case are trivial, as usual.

For the impossible structural case we have $M = N\,A$ for some PLC term N and PLC type A, together with the assumption $\{L \vdash A \sim T\}$. Combining the latter with $\{L \vdash K \approx T\}$ and Lemma 6.4.19 yields the required contradiction. ∎

Lemma 6.4.26

$$\forall LMAST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx S\,T\} \implies \{L \vdash A \sim T\} \implies \\ \exists NB.\ M = N\,B \wedge \{L \vdash N \approx S\} \wedge \{L \vdash B \sim T\}$$

Proof Dual to Lemma 6.4.25. ∎

At this point we have everything to lift the injectivity and functionality of proper relational contexts to the correspondence relation. We give the lifting of injectivity to the type relation in detail, but keep the other three proofs brief, since the differences are minor.

Theorem 6.4.27 (Injectivity of $\sim$)

$$\forall LABS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim S\} \implies \{L \vdash B \sim S\} \implies A = B$$

Proof By induction on $\{L \vdash A \sim S\}$. For the two structural cases we use Lemma 6.4.21, and respectively Lemma 6.4.22, to then invert $\{L \vdash B \sim S\}$. The inductive hypotheses close the goal in each case. In the backchaining case we know that $A = \mathsf{n}_1$ and $S = \mathsf{n}_2$, hence $\{L \vdash B'\langle \mathsf{n}_1 \rangle \sim \mathsf{n}_2\}$ has to follow from backchaining as well. We consequentially have $\mathsf{n}_1 \sim \mathsf{n}_2 \in L'\langle \mathsf{n}_1 \rangle \langle \mathsf{n}_2 \rangle$ and $B'\langle \mathsf{n}_1 \rangle \sim \mathsf{n}_2 \in L'\langle \mathsf{n}_1 \rangle \langle \mathsf{n}_2 \rangle$, which due to Fact 6.4.16 yields $\mathsf{n}_1 = B'\langle \mathsf{n}_1 \rangle$, as required. ∎

Theorem 6.4.28 (Injectivity of $\approx$)

$$\forall LMNS.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx S\} \implies \{L \vdash N \approx S\} \implies M = N$$

Proof By induction on $\{L \vdash M \approx S\}$ and inversion on $\{L \vdash N \approx S\}$. We use Theorem 6.4.27 for the embedded type-level correspondence derivations. ∎

Theorem 6.4.29 (Functionality of $\sim$)

$$\forall LAST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash A \sim S\} \implies \{L \vdash A \sim T\} \implies S = T$$

Proof Dual to Theorem 6.4.27 by induction on $\{L \vdash A \sim S\}$ and inversion on $\{L \vdash A \sim T\}$. The latter relies on a number of suitable inversion principles which were mentioned above but not shown in detail. ∎

Theorem 6.4.30 (Functionality of $\approx$)

$$\forall LMST.\ \mathbb{C}^{\sim}_{\approx}(L) \implies \{L \vdash M \approx S\} \implies \{L \vdash M \approx T\} \implies S = T$$

Proof Dual to Theorem 6.4.28. Embedded type-level derivations are handled with Theorem 6.4.29. ∎

Part II, Totality and Preservation of Judgements

We now proceed with the second half of our correspondence proof and establish that our correspondence relations are suitably total and preserve judgements. The latter naturally entails that we have statements which involve typing (or type formation) judgements from both systems, as well as a judgement about relatedness. So in total, each statement mentions three classes of judgements. All the results we have seen up to this point only ever considered one of these classes. The reason why this matters is tied to the notion of correctness predicates for contexts. Recall that we defined $\mathbb{C}_{\mathsf{P}}(L_{\mathsf{P}})$, $\mathbb{C}^{\sim}_{\approx}(L_{\approx})$, and $\mathbb{C}_{\lambda}(L_{\lambda})$, respectively for each of the three classes, where L_{P} tracks type formation and typing assumptions for PLC, L_{λ} tracks information for $\lambda 2$ typing, and $L_{\approx}$ carries assumptions about related variables. The context predicates each ensured local well-formedness of their respective context.

This is no longer sufficient for our present purposes. Consider the scenario where we want to connect the well-formedness of a PLC type variable, $\{L_{\mathsf{P}} \vdash \mathsf{n}_1\ \mathbf{ty}\}$, to a corresponding $\lambda 2$ judgement $\{L_\lambda \vdash \mathsf{n}_2 :_\lambda *\}$, because we have a derivation of $\{L_\approx \vdash \mathsf{n}_1 \sim \mathsf{n}_2\}$. All three judgements can only hold because each proposition was extracted from its respective context. That is we must have had $\mathsf{n}_1\ \mathbf{ty} \in L_{\mathsf{P}}$, $\mathsf{n}_1 \sim \mathsf{n}_2 \in L_\approx$ and $\mathsf{n}_2 :_\lambda * \in L_\lambda$. Note how the same nominal constants n_1 and n_2 are spread over three statements. The situation becomes even more involved when we consider matching term variables, which come with type ascriptions and we only want a matching if the ascribed types are also related. The only way to achieve this in the light of generic quantification and nominal reasoning, is when all assumptions are *simultaneously* added to their respective contexts, while potential side conditions are also checked. We can enforce this with a new combined context predicate, which is again inductively defined.

Definition 6.4.31 (Combined Context Predicate)

$$\frac{}{\mathbb{C}(\bullet \mid \bullet \mid \bullet)} \qquad \frac{\mathbb{C}(L_{\mathsf{P}} \mid L_\approx \mid L_\lambda)}{\mathbb{C}(L_{\mathsf{P}}, x\ \mathbf{ty} \mid L_\approx, x \sim y \mid L_\lambda, y :_\lambda *)}\ x, y \notin L_i$$

$$\frac{\{L_{\mathsf{P}} \vdash A\ \mathbf{ty}\} \quad \mathbb{C}(L_{\mathsf{P}} \mid L_\approx \mid L_\lambda) \quad \{L_\approx \vdash A \sim S\} \quad \{L_\lambda \vdash S :_\lambda *\}}{\mathbb{C}(L_{\mathsf{P}}, x :_{\mathsf{P}} A \mid L_\approx, x \approx y \mid L_\lambda, y :_\lambda S)}\ x, y \notin L_i, A, S$$

Note that $\mathbb{C}(L_{\mathsf{P}} \mid L_\approx \mid L_\lambda)$ entails the local well-formedness of each of the three contexts, as witnessed by the following projection lemma.

Lemma 6.4.32 (Context Projection)

$$\forall L_{\mathsf{P}} L_\approx L_\lambda.\ \mathbb{C}(L_{\mathsf{P}} \mid L_\approx \mid L_\lambda) \implies \mathbb{C}_{\mathsf{P}}(L_{\mathsf{P}}) \wedge \mathbb{C}^{\sim}_{\approx}(L_\approx) \wedge \mathbb{C}_\lambda(L_\lambda)$$

Proof By induction on $\mathbb{C}(L_{\mathsf{P}} \mid L_\approx \mid L_\lambda)$. ∎

As before, we require a number of inversion principles to make full use of this context predicate. We start with two very basic results, which state that whenever one of the two outer contexts is non-empty, then so are the other two contexts as well. Technically, there is a similar result for a non-empty relational context, but this one is not needed for our present development.

Lemma 6.4.33 (Left Context Inversion)

$$\forall L_{\mathsf{P}} L_\approx L_\lambda J.\ \mathbb{C}(L_{\mathsf{P}}, J \mid L_\approx \mid L_\lambda) \implies \exists J' J'' L'_\approx L'_\lambda.\ L_\approx = L'_\approx, J' \wedge L_\lambda = L'_\lambda, J'' \wedge \mathbb{C}(L_{\mathsf{P}} \mid L'_\approx \mid L'_\lambda)$$

Proof By discriminating on $\mathbb{C}(L_{\mathsf{P}}, J \mid L_\approx \mid L_\lambda)$. In both cases, the required context suffixes are given, and the missing judgements are easy to construct from the available assumptions. ∎

Lemma 6.4.34 (Right Context Inversion)

$$\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} J.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}, J) \implies \\ \exists J' J'' L'_{\mathsf{P}} L'_{\approx}.\ L_{\mathsf{P}} = L'_{\mathsf{P}}, J'' \wedge L_{\approx} = L'_{\approx}, J' \wedge \mathbb{C}(L'_{\mathsf{P}} \mid L'_{\approx} \mid L_{\lambda})$$

Proof By discriminating on $\mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}, J)$. ∎

Next on our list are the lookup lemmas, again one for having an element in the PLC context and another one for having an entry from the $\lambda 2$ context. Due to the shape of our context predicate, both have disjunctive conclusions, and a relatively large set of facts that follow from a lookup. The two results may therefore look slightly intimidating, but the underlying structure is in fact rather simple and a straightforward adaptation of techniques we have already employed.

Lemma 6.4.35 (Left Context Lookup)

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) &\implies K \in L_{\mathsf{P}} \implies \\ &(\exists A S.\ K = A\ \mathbf{ty} \wedge \langle\!\langle A \rangle\!\rangle_{\mathsf{P}} \wedge \\ &\quad A \sim S \in L_{\approx} \wedge \langle\!\langle A, S \rangle\!\rangle_{\sim} \wedge \\ &\quad S :_{\lambda} * \in L_{\lambda} \wedge \langle\!\langle S \rangle\!\rangle_{\lambda}) \vee \\ &(\exists M A T S.\ K = M :_{\mathsf{P}} A \wedge \langle\!\langle M \rangle\!\rangle_{\mathsf{P}} \wedge \{L_{\mathsf{P}} \vdash A\ \mathbf{ty}\} \wedge \\ &\quad M \approx T \in L_{\approx} \wedge \langle\!\langle M, T \rangle\!\rangle_{\approx} \wedge \{L_{\approx} \vdash A \sim S\} \wedge \\ &\quad T :_{\lambda} S \in L_{\lambda} \wedge \langle\!\langle T \rangle\!\rangle_{\lambda} \wedge \{L_{\lambda} \vdash S :_{\lambda} *\})\end{aligned}$$

Proof By induction on $K \in L_{\mathsf{P}}$, with $L_{\mathsf{P}} = L'_{\mathsf{P}}, J$.

In the base case we have $K = J$ and discriminate on $\mathbb{C}(L'_{\mathsf{P}}, K \mid L_{\approx} \mid L_{\lambda})$, which produces two cases. Each of these has enough information to establish one of the two disjuncts.

For the inductive case we have $K \in L'_{\mathsf{P}}$ and use Lemma 6.4.33 to obtain $L'_{\approx}$ and L'_{λ} which satisfy $\mathbb{C}(L'_{\mathsf{P}} \mid L'_{\approx} \mid L'_{\lambda})$. This allows us to instantiate the inductive hypothesis and obtain two cases which again match the two disjuncts of the conclusion. ∎

Lemma 6.4.36 (Right Context Lookup)

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) &\implies K \in L_{\lambda} \implies \\ &(\exists A S.\ K = S :_{\lambda} * \wedge \langle\!\langle S \rangle\!\rangle_{\lambda} \wedge \\ &\quad A \sim S \in L_{\approx} \wedge \langle\!\langle A, S \rangle\!\rangle_{\sim} \wedge \\ &\quad A\ \mathbf{ty} \in L_{\mathsf{P}} \wedge \langle\!\langle A \rangle\!\rangle_{\mathsf{P}}) \vee \\ &(\exists M A T S.\ K = T :_{\lambda} S \wedge \langle\!\langle T \rangle\!\rangle_{\lambda} \wedge \{L_{\lambda} \vdash S :_{\lambda} *\} \wedge \\ &\quad M \approx T \in L_{\approx} \wedge \langle\!\langle M, T \rangle\!\rangle_{\approx} \wedge \{L_{\approx} \vdash A \sim S\} \wedge \\ &\quad M :_{\mathsf{P}} A \in L_{\mathsf{P}} \wedge \langle\!\langle M \rangle\!\rangle_{\mathsf{P}} \wedge \{L_{\mathsf{P}} \vdash A\ \mathbf{ty}\})\end{aligned}$$

Proof Analogue to Lemma 6.4.35. ∎

The two preceding lookup lemmas are both similar to PLC context lookup (Lemma 6.4.3), in that the result is a disjunction of two existentials. To increase the usability of that result we introduced separate backchaining inversion lemmas, like Lemma 6.4.5, which deal with the two possible outcomes in isolation, based on additional information. We adopt a similar pattern here and introduce, for each of the two lookup lemmas, two separate backchaining inversion lemmas. The two results for PLC lookup in this setting are relatively straightforward, while the two for $\lambda 2$ lookup are slightly more involved. We therefore keep the former two brief and instead focus on the latter.

Lemma 6.4.37

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K A.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) \implies K \in L_{\mathsf{P}} \implies \{L_{\mathsf{P}} \mid K \vdash A\ \mathbf{ty}\} \implies \\ K = A\ \mathbf{ty} \wedge \langle\!\langle A \rangle\!\rangle_{\mathsf{P}} \wedge \exists S.\ \{L_{\approx} \vdash A \sim S\} \wedge \{L_{\lambda} \vdash S :_{\lambda} *\}\end{aligned}$$

Proof We apply Lemma 6.4.35 to $\mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda})$ and $K \in L_{\mathsf{P}}$. We either obtain the required properties, or a unification failure, and thus close the proof. ∎

Lemma 6.4.38

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K M A.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) \implies K \in L_{\mathsf{P}} \implies \{L_{\mathsf{P}} \mid K \vdash M :_{\mathsf{P}} A\} \implies \\ K = M :_{\mathsf{P}} A \wedge \langle\!\langle M \rangle\!\rangle_{\mathsf{P}} \wedge \exists S T.\ \{L_{\approx} \vdash M \approx T\} \wedge \\ \{L_{\approx} \vdash A \sim S\} \wedge \{L_{\lambda} \vdash T :_{\lambda} S\} \wedge \{L_{\lambda} \vdash S :_{\lambda} *\}\end{aligned}$$

Proof Analogue to Lemma 6.4.37. ∎

Lemma 6.4.39

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K S.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) \implies K \in L_{\lambda} \implies \{L_{\lambda} \mid K \vdash S :_{\lambda} *\} \implies \\ K = S :_{\lambda} * \wedge \langle\!\langle S \rangle\!\rangle_{\lambda} \wedge \exists A.\ \{L_{\approx} \vdash A \sim S\} \wedge \{L_{\mathsf{P}} \vdash A\ \mathbf{ty}\}\end{aligned}$$

Proof As a first step we apply Lemma 6.4.36 to the assumptions $\mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda})$ and $K \in L_{\lambda}$, which yields two cases.

The case corresponding to related type variables is straightforward.

For the impossible case of related term variables we obtain, by unification, $W = *$ and therefore $\{L_{\approx} \vdash Z \sim *\}$. We obtain $\mathbb{C}^{\approx}_{\approx}(L_{\approx})$ via context projection and then apply Fact 6.4.20 to discharge this case. ∎

Lemma 6.4.40

$$\begin{aligned}\forall L_{\mathsf{P}} L_{\approx} L_{\lambda} K T S.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda}) \implies K \in L_{\lambda} \implies \\ \{L_{\lambda} \mid K \vdash T :_{\lambda} S\} \implies \{L_{\lambda} \vdash S :_{\lambda} *\} \implies \\ K = T :_{\lambda} S \wedge \langle\!\langle T \rangle\!\rangle_{\lambda} \wedge \exists M A.\ \{L_{\approx} \vdash M \approx T\} \wedge \\ \{L_{\approx} \vdash A \sim S\} \wedge \{L_{\mathsf{P}} \vdash M :_{\mathsf{P}} A\} \wedge \{L_{\mathsf{P}} \vdash A\ \mathbf{ty}\}\end{aligned}$$

Proof We again apply Lemma 6.4.36 to the first two premises and consider the resulting cases.

Here the case for related term variables is trivial.

For the impossible case of related type variables, unification yields $\{L_\lambda \vdash * :_\lambda *\}$, which can be discharged with Fact 6.4.10. The latter of course relies on projecting the local well-formedness of L_λ, that is $\mathbb{C}_\lambda(L_\lambda)$. ∎

We are now in a position to prove the (easier) totality and preservation result in the direction from PLC to $\lambda 2$. Since the inverse direction requires further properties of the $\lambda 2$ typing relation we postpone the respective proofs for a moment.

Theorem 6.4.41 (Preservation of PLC Type Formation Under $\sim$)

$$\begin{aligned}&\forall L_\mathsf{P} A.\ \{L_\mathsf{P} \vdash A\ \mathbf{ty}\} \implies\\ &\quad \forall L_\approx L_\lambda.\ \mathbb{C}(L_\mathsf{P} \mid L_\approx \mid L_\lambda) \implies\\ &\quad\quad \exists S.\ \{L_\approx \vdash A \sim S\} \wedge \{L_\lambda \vdash S :_\lambda *\}\end{aligned}$$

Proof By induction on $\{L_\mathsf{P} \vdash A\ \mathbf{ty}\}$.

Let $A = B_1 \to B_2$, then from our inductive hypothesis we have some S_1 and S_2 satisfying the following.

$$\begin{aligned}&\{L_\approx \vdash B_1 \sim S_1\} && \{L_\lambda \vdash S_1 :_\lambda *\}\\ &\{L_\approx \vdash B_2 \sim S_2\} && \{L_\lambda \vdash S_2 :_\lambda *\}\end{aligned}$$

The required $\lambda 2$ term is $S \coloneqq \Pi S_1.\,(\lambda x \Rightarrow S_2)$, for which both required judgements are easily derivable.

Now consider $A = \forall.\, B$, with $\{L_\mathsf{P}, \mathsf{n}_1\ \mathbf{ty} \vdash B\langle \mathsf{n}_1\rangle\ \mathbf{ty}\}$. In addition we have $\mathbb{C}(L_\mathsf{P} \mid L_\approx \mid L_\lambda)$ and therefore can form $\mathbb{C}(L_\mathsf{P}, \mathsf{n}_1\ \mathbf{ty} \mid L_\approx, \mathsf{n}_1 \sim \mathsf{n}_2 \mid L_\lambda, \mathsf{n}_2 :_\lambda *)$, where n_1 and n_2 are guaranteed to be sufficiently fresh for the three contexts. Hence by induction we have some $S' : \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda$ satisfying $\{L_\approx, \mathsf{n}_1 \sim \mathsf{n}_2, \vdash B\langle \mathsf{n}_1\rangle \sim S'\langle \mathsf{n}_2\rangle\}$ and $\{L_\lambda, \mathsf{n}_2 :_\lambda * \vdash S'\langle \mathsf{n}_2\rangle :_\lambda *\}$, where S' is obtained by raising S over n_2. The required term is now clearly $S \coloneqq \Pi *.\, S'$ and the two judgements again follow.

Finally we have the backchaining case with $\{L_\mathsf{P} \mid K \vdash A\ \mathbf{ty}\}$ and $K \in L_\mathsf{P}$. The required $\lambda 2$ term and judgements are directly obtained from Lemma 6.4.37. ∎

Theorem 6.4.42 (Preservation of PLC Typing Under $\approx$)

$$\begin{aligned}&\forall L_\mathsf{P} M A.\ \{L_\mathsf{P} \vdash M :_\mathsf{P} A\} \implies\\ &\quad \forall L_\approx L_\lambda.\ \mathbb{C}(L_\mathsf{P} \mid L_\approx \mid L_\lambda) \implies\\ &\quad\quad \exists S T.\ \{L_\approx \vdash M \approx T\} \wedge \{L_\approx \vdash A \sim S\} \wedge \{L_\lambda \vdash T :_\lambda S\} \wedge \{L_\lambda \vdash S :_\lambda *\}\end{aligned}$$

Proof By induction on $\{L_\mathsf{P} \vdash M :_\mathsf{P} A\}$.

Let $M = N_1\, N_2$, with $\{L_{\mathsf{P}} \vdash N_1 :_{\mathsf{P}} A' \to A\}$ and $\{L_{\mathsf{P}} \vdash N_2 :_{\mathsf{P}} A'\}$. By induction we have all the following for some S_1, S_2, T_1 and T_2.

$$\begin{array}{ll}
\{L_{\approx} \vdash N_1 \approx T_1\} & \{L_{\approx} \vdash N_2 \approx T_2\} \\
\{L_{\approx} \vdash A' \to A \sim S_1\} & \{L_{\approx} \vdash A' \sim S_2\} \\
\{L_{\lambda} \vdash T_1 :_{\lambda} S_1\} & \{L_{\lambda} \vdash T_2 :_{\lambda} S_2\} \\
\{L_{\lambda} \vdash S_1 :_{\lambda} *\} & \{L_{\lambda} \vdash S_2 :_{\lambda} *\}
\end{array}$$

With context projection we also have $\mathbb{C}^{\sim}_{\approx}(L_{\approx})$ and $\mathbb{C}_{\lambda}(L_{\lambda})$. Now by inversion we have $S_1 = \Pi S_1'.\, S_1''$, with $\{L_{\approx} \vdash A' \sim S_1'\}$ and $\{L_{\approx} \vdash A \sim S_1''\langle \mathsf{n}_1 \rangle\}$. We instantiate the latter to $\{L_{\approx} \vdash A \sim S_1''\langle T_2 \rangle\}$. Since $\sim$ is functional we also have $S_2 = S_1'$ and from $\{L_{\lambda} \vdash T_1 :_{\lambda} \Pi S_2.\, S_1''\}$ and $\{L_{\lambda} \vdash T_2 :_{\lambda} S_2\}$ we can infer $\{L_{\lambda} \vdash S_1''\langle T_2 \rangle :_{\lambda} *\}$ using Fact 6.4.11. The required $\lambda 2$ terms are now $S \coloneqq S_1''\langle T_2 \rangle$ and $T \coloneqq T_1\, T_2$.

Let $M = \lambda B_1.\, N$ and $A = B_1 \to B_2$ which additionally satisfy $\{L_{\approx} \vdash B_1\ \mathbf{ty}\}$ and $\{L_{\mathsf{P}}, \mathsf{n}_1 :_{\mathsf{P}} B_1 \vdash N\langle \mathsf{n}_1 \rangle :_{\mathsf{P}} B_2\}$. From the former and Theorem 6.4.41 we obtain some S_1 which satisfies $\{L_{\approx} \vdash B_1 \sim S_1\}$ and $\{L_{\lambda} \vdash S_1 :_{\lambda} *\}$. We can now derive $\mathbb{C}(L_{\mathsf{P}}, \mathsf{n}_1 :_{\mathsf{P}} B_1 \mid L_{\approx}, \mathsf{n}_1 \approx \mathsf{n}_2 \mid L_{\lambda}, \mathsf{n}_2 :_{\lambda} S_1)$ and apply the inductive hypothesis, to obtain the following for some T' and S_2 (which are both raised over n_2).

$$\begin{array}{l}
\{L_{\approx}, \mathsf{n}_1 \approx \mathsf{n}_2 \vdash N\langle \mathsf{n}_1 \rangle \approx T'\langle \mathsf{n}_2 \rangle\} \\
\{L_{\approx}, \mathsf{n}_1 \approx \mathsf{n}_2 \vdash B_2 \sim S_2\langle \mathsf{n}_2 \rangle\} \\
\{L_{\lambda}, \mathsf{n}_2 :_{\lambda} S_1 \vdash T'\langle \mathsf{n}_2 \rangle :_{\lambda} S_2\langle \mathsf{n}_2 \rangle\} \\
\{L_{\lambda}, \mathsf{n}_2 :_{\lambda} S_1 \vdash S_2\langle \mathsf{n}_2 \rangle :_{\lambda} *\}
\end{array}$$

With context strengthening we also have $\{L_{\approx} \vdash B_2 \sim S_2\langle \mathsf{n}_2 \rangle\}$ and we set $S \coloneqq \Pi S_1.\, S_2$ and $T \coloneqq \lambda S_1.\, T'$ to close the case.

Let $M = N\, B$ and $A = A'\langle B \rangle$, with $\{L_{\mathsf{P}} \vdash B\ \mathbf{ty}\}$ and $\{L_{\mathsf{P}} \vdash N :_{\mathsf{P}} \forall.\, A'\}$. By induction and Theorem 6.4.41 we have all the following for some S_1, S_2 and T'.

$$\begin{array}{ll}
\{L_{\approx} \vdash N \approx T'\} & \\
\{L_{\approx} \vdash \forall.\, A' \sim S_1\} & \{L_{\approx} \vdash B \sim S_2\} \\
\{L_{\lambda} \vdash T' :_{\lambda} S_1\} & \\
\{L_{\lambda} \vdash S_1 :_{\lambda} *\} & \{L_{\lambda} \vdash S_2 :_{\lambda} *\}
\end{array}$$

By inversion we have $S_1 = \Pi *.\, S_1'$, with $\{L_{\approx}, \mathsf{n}_1 \sim \mathsf{n}_2 \vdash A'\langle \mathsf{n}_1 \rangle \sim S_1'\langle \mathsf{n}_2 \rangle\}$ and hence by substitutivity $\{L_{\approx} \vdash A'\langle B \rangle \sim S_1'\langle S_2 \rangle\}$. From $\{L_{\lambda} \vdash T' :_{\lambda} \Pi *.\, S_1'\}$ and $\{L_{\lambda} \vdash S_2 :_{\lambda} *\}$ we can further infer $\{L_{\lambda} \vdash S_1'\langle S_2 \rangle :_{\lambda} *\}$ using Fact 6.4.11. The required $\lambda 2$ terms for the type application case are then $S \coloneqq S_1'\langle S_2 \rangle$ and $T \coloneqq T'\, S_2$.

Let $M = \Lambda.\, N$ and $A = \forall.\, A'$ with $\{L_{\mathsf{P}}, \mathsf{n}_1\ \mathbf{ty} \vdash N\langle \mathsf{n}_1 \rangle :_{\mathsf{P}} A'\langle \mathsf{n}_1 \rangle\}$. We derive $\mathbb{C}(L_{\mathsf{P}}, \mathsf{n}_1\ \mathbf{ty} \mid L_{\approx}, \mathsf{n}_1 \sim \mathsf{n}_2 \mid L_{\lambda}, \mathsf{n}_2 :_{\lambda} *)$ and apply the inductive hypothesis, to obtain

the following for some T' and S' (which are both raised over n_2).

$$\{L_{\approx}, \mathsf{n}_1 \sim \mathsf{n}_2 \vdash N\langle \mathsf{n}_1 \rangle \approx T'\langle \mathsf{n}_2 \rangle\}$$
$$\{L_{\approx}, \mathsf{n}_1 \sim \mathsf{n}_2 \vdash A'\langle \mathsf{n}_1 \rangle \sim S'\langle \mathsf{n}_2 \rangle\}$$
$$\{L_{\lambda}, \mathsf{n}_2 :_\lambda * \vdash T'\langle \mathsf{n}_2 \rangle :_\lambda S'\langle \mathsf{n}_2 \rangle\}$$
$$\{L_{\lambda}, \mathsf{n}_2 :_\lambda * \vdash S'\langle \mathsf{n}_2 \rangle :_\lambda *\}$$

Here we set $S \coloneqq \Pi *.\, S'$ and $T \coloneqq \lambda *.\, T'$ to close the case.

The last open case is again due to backchaining with $\{L_{\mathsf{P}} \mid K \vdash M :_{\mathsf{P}} A\}$ and $K \in L_{\mathsf{P}}$. Here we close the case with a straightforward application of Lemma 6.4.38. ∎

As mentioned above, we require some additional metatheoretical results before we can tackle the two preservation results for $\lambda 2$ typings. The required results deal with the internal structure of universes, like the degeneracy of $\Box$, or the fact that dependent function types are always inhabited by abstractions. In addition we require propagation for $\lambda 2$. We keep the proofs brief and elide the handling of impossible backchaining cases.

Fact 6.4.43 $\forall LS.\ \mathbb{C}_\lambda(L) \implies \{L \vdash \mathcal{U}\ S\} \implies S = * \vee S = \Box$

Fact 6.4.44 $\forall LS.\ \mathbb{C}_\lambda(L) \implies \{L \vdash \Box :_\lambda S\} \implies \bot$

Lemma 6.4.45 (Stripping for Dependent Function Types)

$$\forall LSTU.\ \mathbb{C}_\lambda(L) \implies \{L \vdash \Pi S.\, T :_\lambda U\} \implies$$
$$U = * \wedge \exists V.\ \{L \vdash \mathcal{U}\ V\} \wedge \{L \vdash S :_\lambda V\} \wedge \nabla x.\ \{L, x :_\lambda A \vdash T\langle x \rangle :_\lambda *\}$$

Proof Straightforward by discriminating on $\{L \vdash \Pi S.\, T :_\lambda U\}$. ∎

Lemma 6.4.46 (Propagation for $\lambda 2$)

$$\forall LST.\ \mathbb{C}_\lambda(L) \implies \{L \vdash S :_\lambda T\} \implies T = \Box \vee \exists U.\ \{L \vdash \mathcal{U}\ U\} \wedge \{L \vdash T :_\lambda U\}$$

Proof By induction on $\{L \vdash S :_\lambda T\}$. The axiom case as well as the cases for dependent function types and abstractions are all trivial, and the variable case follows with Lemma 6.4.9.

For the application case we have $S = S_1\, S_2$ and $T = T'\langle S_2 \rangle$ and additionally know that $\{L \vdash S_1 :_\lambda \Pi U.\, T'\}$ and $\{L \vdash S_2 :_\lambda U\}$. By the inductive hypothesis there is some V such that $\{L \vdash \Pi U.\, T' :_\lambda V\}$ and from Lemma 6.4.45 it therefore follows that $\{L, \mathsf{n}_1 :_\lambda U \vdash T'\langle x \rangle :_\lambda *\}$ holds. With substitutivity we then obtain $\{L \vdash T'\langle S_2 \rangle :_\lambda *\}$, which closes the proof. ∎

Lemma 6.4.47

$$\forall LSTUV.\ \mathbb{C}_\lambda(L) \implies \{L \vdash S :_\lambda \Pi T.\, U\} \implies \{L \vdash V :_\lambda S\} \implies$$
$$\{L \vdash U\langle V \rangle :_\lambda *\} \wedge \{L \vdash \Pi T.\, U :_\lambda *\} \wedge \exists V'.\ \{L \vdash \mathcal{U}\ V'\} \wedge \{L \vdash T :_\lambda V'\}$$

Proof Straightforward using Lemma 6.4.46 on $\{L \vdash S :_\lambda \Pi T.\, U\}$. The various goals follow from Fact 6.4.45 and substitutivity. ∎

Lemma 6.4.48 (Degeneracy of $\Box$) $\forall LS.\ \mathbb{C}_\lambda(L) \implies \{L \vdash S :_\lambda \Box\} \implies S = *$

Proof By discriminating on $\{L \vdash S :_\lambda \Box\}$. The axiom case is immediate and for the backchaining case, Lemma 6.4.9 yields some U with $\{L \vdash \Box :_\lambda U\}$ which in turn contradicts Fact 6.4.44.

For the application case we have $S = S_1\, S_2$ with $\{L \vdash S_1 :_\lambda \Pi T.\, U\}$ as well as $\{L \vdash S_2 :_\lambda T\}$ and we also know $U\langle S_2\rangle = \Box$. From Lemma 6.4.47 we obtain $\{L \vdash U\langle S_2\rangle :_\lambda *\}$, which again conflicts with Fact 6.4.44. ∎

Lemma 6.4.49

$$\begin{aligned}\forall LST.\ \mathbb{C}_\lambda(L) \implies \{L \vdash S :_\lambda T\} \implies \{L \vdash \mathcal{U}\ T\} \implies \\ T = * \vee (T = \Box \wedge S = *)\end{aligned}$$

Proof Trivial consequence of Fact 6.4.43 and Lemma 6.4.48. ∎

The benefit of having the last result arises from the behaviour of unification for the application of a lemma with equality conclusions, since equalities are automatically substituted into the given proof state.

This brings us to our final preservation results. The proofs turn out to be rather technical since they have to deal with two issues that are in principle completely orthogonal. The first is the preservation property itself, and the second is the disambiguation of semantic terms and types. Recall that in our de Bruijn development we dealt with the latter up front by establishing two custom induction principles for the $\lambda 2$ typing relation in the presence of certain side conditions. This technique is not applicable here, since Abella's reasoning logic $\mathcal{G}$ is not powerful enough to directly express said induction principles. We are therefore inlining the handling of semantic disambiguation into the two preservation proofs.

Theorem 6.4.50 (Preservation of $\lambda 2$ Type Formation Under $\sim$)

$$\begin{aligned}\forall L_\lambda S.\ \{L_\lambda \vdash S :_\lambda *\} \implies \\ \forall L_\mathsf{P} L_\approx.\ \mathbb{C}(L_\mathsf{P} \mid L_\approx \mid L_\lambda) \implies \\ \exists A.\ \{L_\approx \vdash A \sim S\} \wedge \{L_\mathsf{P} \vdash A\ \mathbf{ty}\}\end{aligned}$$

Proof By induction on $\{L_\lambda \vdash S :_\lambda *\}$, which yields three cases. The easiest is the backchaining case, which is covered by Lemma 6.4.39.

The main case has $S = \Pi U.\, T$ and some V, for which the following hold.

$$\begin{aligned}&\{L_\lambda \vdash U :_\lambda V\} \\ &\{L_\lambda \vdash \mathcal{U}\ V\} \\ &\{L_\lambda, \mathsf{n}_1 :_\lambda U \vdash T\langle \mathsf{n}_1\rangle :_\lambda *\}\end{aligned}$$

We use Lemma 6.4.49 to consider two subcases. Let $V = *$. Then we can use the inductive hypothesis to first obtain an A_1 with $\{L_\approx \vdash A_1 \sim U\}$ and $\{L_\mathsf{P} \vdash A_1\ \mathbf{ty}\}$. This allows us to extend the context to $\mathbb{C}(L_\mathsf{P}, \mathsf{n}_2 :_\mathsf{P} A_1 \mid L_\approx, \mathsf{n}_2 \approx \mathsf{n}_1 \mid L_\lambda, \mathsf{n}_1 :_\lambda U)$ and then use the inductive hypothesis again for a second PLC type A_2 that satisfies $\{L_\approx, \mathsf{n}_2 \approx \mathsf{n}_1 \vdash A_2 \sim T\langle \mathsf{n}_1 \rangle\}$ and $\{L_\mathsf{P}, \mathsf{n}_2 :_\mathsf{P} A_1 \vdash A_2\ \mathbf{ty}\}$. We can strengthen the latter two to $\{L_\approx \vdash A_2 \sim T\langle \mathsf{n}_1 \rangle\}$ and respectively $\{L_\mathsf{P} \vdash A_2\ \mathbf{ty}\}$ and then close the subcase with $A \coloneqq A_1 \to A_2$. Now let $V = \square$ and $U = *$. This time we extend the context with new type variables to $\mathbb{C}(L_\mathsf{P}, \mathsf{n}_2\ \mathbf{ty} \mid L_\approx, \mathsf{n}_2 \sim \mathsf{n}_1 \mid L_\lambda, \mathsf{n}_1 :_\lambda *)$ and then apply the inductive hypothesis to $\{L_\lambda, \mathsf{n}_1 :_\lambda * \vdash T\langle \mathsf{n}_1 \rangle :_\lambda *\}$ to obtain an $A' : \mathsf{Ty}_\mathsf{P} \to \mathsf{Ty}_\mathsf{P}$ which satisfies $\{L_\approx, \mathsf{n}_2 \sim \mathsf{n}_1 \vdash A'\langle \mathsf{n}_2 \rangle \sim T\langle \mathsf{n}_1 \rangle\}$ and $\{L_\mathsf{P}, \mathsf{n}_2\ \mathbf{ty} \vdash A'\langle \mathsf{n}_2 \rangle\ \mathbf{ty}\}$. This subcase is closed by setting $A \coloneqq \forall.\, A'$.

This leaves a problematic application case with $S = S_1\, S_2$ and $\{L_\lambda \vdash S_1 :_\lambda \Pi U.\, T\}$ as well as $\{L_\lambda \vdash S_2 :_\lambda U\}$ and also $T\langle S_2 \rangle = *$. With Lemma 6.4.47 we infer $\{L_\lambda \vdash T\langle S_2 \rangle :_\lambda *\}$, which contradicts Fact 6.4.10 and thus completes the proof. ∎

Theorem 6.4.51 (Preservation of λ2 Typing Under ≈)

$$\begin{aligned}
&\forall L_\lambda S T.\ \{L_\lambda \vdash T :_\lambda S\} \implies \{L_\lambda \vdash S :_\lambda *\} \implies \\
&\qquad \forall L_\mathsf{P} L_\approx.\ \mathbb{C}(L_\mathsf{P} \mid L_\approx \mid L_\lambda) \implies \\
&\qquad\quad \exists M A.\ \{L_\approx \vdash M \approx T\} \wedge \{L_\approx \vdash A \sim S\} \wedge \{L_\mathsf{P} \vdash M :_\mathsf{P} A\} \wedge \{L_\mathsf{P} \vdash A\ \mathbf{ty}\}
\end{aligned}$$

Proof By induction on $\{L_\lambda \vdash T :_\lambda S\}$. This yields two impossible cases, two actual cases for abstractions and applications, and the backchaining case which is covered by Lemma 6.4.40. The problematic axiom and function type cases set the second premise to $\{L_\lambda \vdash \square :_\lambda *\}$ and respectively to $\{L_\lambda \vdash * :_\lambda *\}$. The former can be discharged with Fact 6.4.44 and the latter with Fact 6.4.10.

Now let $T = \lambda S_1.\, T'$ and $S = \Pi S_1.\, S_2$ with the typing deriving from the following.

$$\begin{aligned}
&\{L_\lambda \vdash \mathcal{U}\ U\} && \{L_\lambda, \mathsf{n}_1 :_\lambda S_1 \vdash T'\langle \mathsf{n}_1 \rangle :_\lambda S_2\langle \mathsf{n}_1 \rangle\} \\
&\{L_\lambda \vdash S_1 :_\lambda U\} && \{L_\lambda, \mathsf{n}_1 :_\lambda S_1 \vdash S_2\langle \mathsf{n}_1 \rangle :_\lambda *\}
\end{aligned}$$

We use Lemma 6.4.49 to distinguish two subcases and first consider $U = *$. Then from $\{L_\lambda \vdash S_1 :_\lambda *\}$ and Theorem 6.4.50 it follows that there is some A_1 with $\{L_\approx \vdash A_1 \sim S_1\}$ and $\{L_\mathsf{P} \vdash A_1\ \mathbf{ty}\}$. Thus $\mathbb{C}(L_\mathsf{P}, \mathsf{n}_2 :_\mathsf{P} A_1 \mid L_\approx, \mathsf{n}_2 \approx \mathsf{n}_1 \mid L_\lambda, \mathsf{n}_1 :_\lambda S_1)$ is well-formed and we can apply the inductive hypothesis to obtain M' and A_2 such that the following hold.

$$\begin{aligned}
&\{L_\approx, \mathsf{n}_2 \approx \mathsf{n}_1 \vdash M'\langle \mathsf{n}_2 \rangle \approx T'\langle \mathsf{n}_1 \rangle\} && \{L_\mathsf{P}, \mathsf{n}_2 :_\mathsf{P} A_1 \vdash M'\langle \mathsf{n}_2 \rangle :_\mathsf{P} A_2\} \\
&\{L_\approx, \mathsf{n}_2 \approx \mathsf{n}_1 \vdash A_2 \sim S_2\langle \mathsf{n}_1 \rangle\} && \{L_\mathsf{P}, \mathsf{n}_2 :_\mathsf{P} A_1 \vdash A_2\ \mathbf{ty}\}
\end{aligned}$$

Two applications of strengthening yield $\{L_\approx \vdash A_2 \sim S_2\langle \mathsf{n}_1 \rangle\}$ and $\{L_\mathsf{P} \vdash A_2\ \mathbf{ty}\}$ and setting $M \coloneqq \lambda A_1.\, M'$ and $A \coloneqq A_1 \to A_2$ closes the subcase. For the other subcase we have $U = \square$ and $S_1 = *$, and thus $\mathbb{C}(L_\mathsf{P}, \mathsf{n}_2\ \mathbf{ty} \mid L_\approx, \mathsf{n}_2 \sim \mathsf{n}_1 \mid L_\lambda, \mathsf{n}_1 :_\lambda *)$ as the

extended context. Induction yields M' and A' such that the following are given.

$$\{L_{\approx}, \mathsf{n}_2 \sim \mathsf{n}_1 \vdash M'\langle \mathsf{n}_2\rangle \approx T'\langle \mathsf{n}_1\rangle\} \qquad \{L_{\mathsf{P}}, \mathsf{n}_2\ \mathbf{ty} \vdash M'\langle \mathsf{n}_2\rangle :_{\mathsf{P}} A'\langle \mathsf{n}_2\rangle\}$$
$$\{L_{\approx}, \mathsf{n}_2 \sim \mathsf{n}_1 \vdash A'\langle \mathsf{n}_2\rangle \sim S_2\langle \mathsf{n}_1\rangle\} \qquad \{L_{\mathsf{P}}, \mathsf{n}_2\ \mathbf{ty} \vdash A'\langle \mathsf{n}_2\rangle\ \mathbf{ty}\}$$

Observe how A' has been raised over n_2 in contrast to the previous subcase, since n_2 is now a PLC type. Setting $M \coloneqq \Lambda.\, M'$ and $A \coloneqq \forall.\, A'$ closes the second subcase.

Finally, we consider application with $T = T'\,U$ and $S = S_2\langle U\rangle$, and have the assumptions $\{L_\lambda \vdash T' :_\lambda \Pi S_1.\, S_2\}$ and $\{L_\lambda \vdash U :_\lambda S_1\}$. Thus Lemma 6.4.47 yields the following for some V.

$$\{L_\lambda \vdash S_2\langle U\rangle :_\lambda *\} \qquad \{L_\lambda \vdash \mathcal{U}\ V\}$$
$$\{L_\lambda \vdash \Pi S_1.\, S_2 :_\lambda *\} \qquad \{L_\lambda \vdash S_1 :_\lambda V\}$$

We again employ Lemma 6.4.49 to distinguish two subcases and start with $V = *$. From two applications of our inductive hypothesis we obtain all the following.

$$\{L_{\approx} \vdash M' \approx T'\} \qquad \{L_{\approx} \vdash N' \approx U\}$$
$$\{L_{\approx} \vdash A' \sim \Pi S_1.\, S_2\} \qquad \{L_{\approx} \vdash A_1 \sim S_1\}$$
$$\{L_{\mathsf{P}} \vdash M' :_{\mathsf{P}} A'\} \qquad \{L_{\mathsf{P}} \vdash N' :_{\mathsf{P}} A_1\}$$
$$\{L_{\mathsf{P}} \vdash A'\ \mathbf{ty}\} \qquad \{L_{\mathsf{P}} \vdash A_1\ \mathbf{ty}\}$$

Since S_1 is related as a type we can invoke Lemma 6.4.21 to infer $A' = A'_1 \to A_2$ with $\{L_{\approx} \vdash A'_1 \sim S_1\}$ and $\{L_{\approx} \vdash A_2 \sim S_2\langle \mathsf{n}_1\rangle\}$. The injectivity of $\sim$ yields $A_1 = A'_1$ and we also set $\mathsf{n}_1 \coloneqq U$ and invert $\{L_{\mathsf{P}} \vdash (A_1 \to A_2)\ \mathbf{ty}\}$ to obtain $\{L_{\mathsf{P}} \vdash A_2\ \mathbf{ty}\}$. We close the first subcase with $M \coloneqq M'\,N'$ and $A \coloneqq A_2$. This leaves the subcase for $V = \Box$ and $S_1 = *$. We proceed as before and have the inductive assumptions for T', while for $\{L_\lambda \vdash U :_\lambda *\}$ we rely on Theorem 6.4.50 to establish the following facts.

$$\{L_{\approx} \vdash M' \approx T'\}$$
$$\{L_{\approx} \vdash A' \sim \Pi *.\, S_2\} \qquad \{L_{\approx} \vdash A_1 \sim U\}$$
$$\{L_{\mathsf{P}} \vdash M' :_{\mathsf{P}} A'\}$$
$$\{L_{\mathsf{P}} \vdash A'\ \mathbf{ty}\} \qquad \{L_{\mathsf{P}} \vdash A_1\ \mathbf{ty}\}$$

From Lemma 6.4.22 it follows that $A' = \forall.\, A_2$ with $\{L_{\approx}, \mathsf{n}_1 \sim \mathsf{n}_2 \vdash A_2\langle \mathsf{n}_1\rangle \sim S_2\langle \mathsf{n}_2\rangle\}$. Inverting $\{L_{\mathsf{P}} \vdash (\forall.\, A_2)\ \mathbf{ty}\}$ yields $\{L_{\mathsf{P}}, \mathsf{n}_1\ \mathbf{ty} \vdash A_2\langle \mathsf{n}_1\rangle\ \mathbf{ty}\}$. Thus substitutivity gives $\{L_{\approx} \vdash A_2\langle A_1\rangle \sim S_2\langle U\rangle\}$ and $\{L_{\mathsf{P}} \vdash A_2\langle A_1\rangle\ \mathbf{ty}\}$. The PLC term and type to close this final case are $M \coloneqq M'\,A_1$ and $A \coloneqq A_2\langle A_1\rangle$. ∎

At this point we are almost finished with the Abella proof. What is left is the existence result for related, well-formed contexts, and we also want to formulate the actual equivalence results. Let us first deal with context existence.

Lemma 6.4.52 (Context Existence)

$$\forall L_{\mathsf{P}}.\ \mathbb{C}_{\mathsf{P}}(L_{\mathsf{P}}) \implies \exists L_{\approx} L_{\lambda}.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda})$$
$$\forall L_{\lambda}.\ \mathbb{C}_{\lambda}(L_{\lambda}) \implies \exists L_{\mathsf{P}} L_{\approx}.\ \mathbb{C}(L_{\mathsf{P}} \mid L_{\approx} \mid L_{\lambda})$$

Proof The first is by induction on $\mathbb{C}_{\mathsf{P}}(L_{\mathsf{P}})$. For the extension with a well-typed term variable $\mathsf{n}_1 :_{\mathsf{P}} A$, Theorem 6.4.41 is used to obtain the corresponding $\lambda 2$ type S.

The second is by induction on $\mathbb{C}_{\lambda}(L_{\lambda})$ where the extension case has $L_{\lambda} = L'_{\lambda}, \mathsf{n}_1 :_{\lambda} S$ with $\{L'_{\lambda} \vdash S :_{\lambda} U\}$ and $\{L'_{\lambda} \vdash \mathcal{U}\ U\}$. We use Lemma 6.4.49 to distinguish the two possible means of extension. The case for $U = \Box$ and $S = *$ is straightforward. For $U = *$ we use Theorem 6.4.50 to obtain a corresponding PLC type A. ∎

For the final correspondence statement we restrict ourselves to closed judgements and fix the following corollary for our preservation results.

Corollary 6.4.53 (Preservation for Closed Judgements)

$$\forall A.\ \{A\ \mathbf{ty}\} \implies \exists S.\ \{A \sim S\} \wedge \{S :_{\lambda} *\} \tag{i}$$
$$\forall S.\ \{S :_{\lambda} *\} \implies \exists A.\ \{A \sim S\} \wedge \{A\ \mathbf{ty}\} \tag{ii}$$
$$\forall M A.\ \{M :_{\mathsf{P}} A\} \implies \exists S T.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{T :_{\lambda} S\} \wedge \{S :_{\lambda} *\} \tag{iii}$$
$$\forall S T.\ \{T :_{\lambda} S\} \implies \{S :_{\lambda} *\} \implies \exists M A.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{M :_{\mathsf{P}} A\} \wedge \{A\ \mathbf{ty}\} \tag{iv}$$

Proof Trivial. ∎

The correspondence theorem is, unfortunately, somewhat bulky, since Abella does not have a connective for logical equivalence. We therefore express equivalence as the conjunction of bidirectional implications.

Theorem 6.4.54 (Equivalence of PLC and $\lambda 2$)

$$\forall A.\ (\{A\ \mathbf{ty}\} \implies \exists S.\ \{A \sim S\} \wedge \{S :_{\lambda} *\}) \wedge ((\exists S.\ \{A \sim S\} \wedge \{S :_{\lambda} *\}) \implies \{A\ \mathbf{ty}\}) \tag{i}$$
$$\forall S.\ (\{S :_{\lambda} *\} \implies \exists A.\ \{A \sim S\} \wedge \{A\ \mathbf{ty}\}) \wedge ((\exists A.\ \{A \sim S\} \wedge \{A\ \mathbf{ty}\}) \implies \{S :_{\lambda} *\}) \tag{ii}$$
$$\forall M A.\ (\{M :_{\mathsf{P}} A\} \implies \exists S T.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{T :_{\lambda} S\} \wedge \{S :_{\lambda} *\}) \wedge ((\exists S T.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{T :_{\lambda} S\} \wedge \{S :_{\lambda} *\}) \implies \{M :_{\mathsf{P}} A\}) \tag{iii}$$
$$\forall S T.\ (\{T :_{\lambda} S\} \wedge \{S :_{\lambda} *\} \implies \exists M A.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{M :_{\mathsf{P}} A\} \wedge \{A\ \mathbf{ty}\}) \wedge ((\exists M A.\ \{M \approx T\} \wedge \{A \sim S\} \wedge \{M :_{\mathsf{P}} A\} \wedge \{A\ \mathbf{ty}\}) \implies \{T :_{\lambda} S\} \wedge \{S :_{\lambda} *\}) \tag{iv}$$

Proof The first implication of each part is simply the corresponding result of Corollary 6.4.53.

For the inverse implications, we again heavily rely on Corollary 6.4.53. Let C(i) through C(iv) denote the parts of that result.

- For Part (i) we use C(ii) on $\{S :_\lambda *\}$ and injectivity of $\sim$.
- For Part (ii) we use C(i) on $\{A\ \mathbf{ty}\}$ and functionality of $\sim$.
- For Part (iii) we use C(iv) on $\{T :_\lambda S\}$ and $\{S :_\lambda *\}$. We then need injectivity of both $\sim$ and $\approx$.
- For Part (iv) we use C(iii) on $\{M :_\mathsf{P} A\}$. We then need functionality of both $\sim$ and $\approx$. ∎

This completes our presentation of the Abella version of the correspondence result, though we give further comparative remarks in Section 7.1. For now we turn our focus to the third proof assistant, namely Beluga, which takes a slightly different approach to the management of contextual information in a HOAS setting like the one we have just covered.

6.5 Beluga Implementation

The programming and proof environment Beluga [PD10, PC15] is another two-level system that supports HOAS.

In contrast to Abella's use of λProlog as its specification layer, Beluga directly employs the logical framework LF [HHP87] for this purpose. Object languages, including expressions and judgements about them, are encoded as LF types. The respective constructors admit hypothetical and locally quantified (or *parametric* in LF terminology) premises, so that we can express the definitions of Section 6.2 without issues. We consider the PTS definitions as an example.

Definition 6.5.1 (λ2 in Beluga) We define an LF type Tm_λ : **Type** with the following grammar.

$$\boxed{\mathsf{Tm}_\lambda} \qquad\qquad S, T ::= * \mid \Box \mid \Pi S.\, T \mid S\, T \mid \lambda S.\, T$$

The constructor type signatures are given in Figure 6.2. Note that due to the HOAS encoding, no variable case is specified. We further define two type families $\mathcal{U}\ S$ and $S :_\lambda T$ which capture universe recognition and, respectively, typing. Their signatures are obtained from those in Section 6.2 when we replace **o** by **Type**.

$$\begin{aligned} \mathsf{univ}_\lambda &: \mathsf{Tm}_\lambda \to \mathbf{Type} && \mathcal{U}\ S \\ \mathsf{of}_\lambda &: \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathbf{Type} && S :_\lambda T \end{aligned}$$

Their constructors are defined as in Section 6.2, and in particular Figure 6.4.

The LF-encodings of the syntax and the typing relation look as follows.

$$
\begin{aligned}
&\mathsf{LF}\quad \mathsf{Tm}_\lambda : \mathbf{Type} \quad = \\
&\quad | \ \mathsf{app}_\lambda \ : \ \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \\
&\quad | \ \mathsf{lam}_\lambda \ : \ \mathsf{Tm}_\lambda \to (\mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda) \to \mathsf{Tm}_\lambda \\
&\quad | \ \mathsf{box}_\lambda \ : \ \mathsf{Tm}_\lambda \\
&\quad | \ \mathsf{star}_\lambda \ : \ \mathsf{Tm}_\lambda \\
&\quad | \ \mathsf{prod}_\lambda \ : \ \mathsf{Tm}_\lambda \to (\mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda) \to \mathsf{Tm}_\lambda ;
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{LF}\quad \mathsf{of}_\lambda : \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathbf{Type} \quad = \\
&\quad | \ \mathsf{T}_\lambda^{\mathsf{ax}} \ : \ \mathsf{of}_\lambda \, \mathsf{star}_\lambda \, \mathsf{box}_\lambda \\
&\quad | \ \mathsf{T}_\lambda^{\Pi} \ : \ \mathsf{of}_\lambda \, A \, U \to \mathsf{univ}_\lambda \, U \\
&\qquad\qquad \to (\{a{:}\mathsf{Tm}_\lambda\} \, \mathsf{of}_\lambda \, a \, A \to \mathsf{of}_\lambda \, (B \, a) \, \mathsf{star}_\lambda) \\
&\qquad\qquad \to \mathsf{of}_\lambda \, (\mathsf{prod}_\lambda \, A \, B) \, \mathsf{star}_\lambda \\
&\quad | \ \mathsf{T}_\lambda^{\lambda} \ : \ \mathsf{of}_\lambda \, A \, U \to \mathsf{univ}_\lambda \, U \\
&\qquad\qquad \to (\{x{:}\mathsf{Tm}_\lambda\} \, \mathsf{of}_\lambda \, x \, A \to \mathsf{of}_\lambda \, (M \, x) \, (B \, x)) \\
&\qquad\qquad \to (\{a{:}\mathsf{Tm}_\lambda\} \, \mathsf{of}_\lambda \, a \, A \to \mathsf{of}_\lambda \, (B \, a) \, \mathsf{star}_\lambda) \\
&\qquad\qquad \to \mathsf{of}_\lambda \, (\mathsf{lam}_\lambda \, A \, M) \, (\mathsf{prod}_\lambda \, A \, B) \\
&\quad | \ \mathsf{T}_\lambda^{\mathsf{app}} \ : \ \mathsf{of}_\lambda \, M \, (\mathsf{prod}_\lambda \, A \, B) \to \mathsf{of}_\lambda \, N \, A \\
&\qquad\qquad \to \mathsf{of}_\lambda \, (\mathsf{app}_\lambda \, M \, N) \, (B \, N)
\end{aligned}
$$

Note that for the type relation of_λ, we are reusing the names of the corresponding inference rules from Figure 6.4 as the names of the constants of the defined LF type family. In the following we will also again prefer the introduced symbolic notation for the various defined constants. That is, we will write $M \, N :_\lambda B\langle N \rangle$ in place of $\mathsf{of}_\lambda \, (\mathsf{app}_\lambda \, M \, N) \, (B \, N)$.

Proofs about the encoded object languages are expressed as total programs in contextual modal type theory (CMTT), which constitutes Beluga's reasoning logic. The programs analyse LF derivation trees, which appear as contextual objects, using pattern matching and higher-order unification. As usual, recursive functions capture inductive reasoning.

Note that Beluga does not provide a tactic language for proof construction, so proof terms have to be given explicitly. The Beluga compiler is, however, able to work with explicitly marked holes in incomplete proof terms, for which type information in the form of a reasoning context is inferred and provided to the user. A brief, high-level comparison of Abella and Beluga is shown in Figure 6.6.

	Abella	**Beluga**
Specification Layer	λProlog	LF
Reasoning Layer	$\mathcal{G}$	CMTT
... connected by	Logical Embedding	Contextual Modality/Objects
Predicates	Dedicated Universe **o**	Standard LF Types
Induction	Only on Predicates	Full Inductive Types
Contexts	Encoded as Lists	First Class Citizens
Proof Construction	Tactics	Dependently Typed Programs

Figure 6.6: High-level comparison of Abella and Beluga.

6.5.1 CMTT and Contextual Objects

Contextual modal type theory (CMTT) [NPP08], and the derived notion of a contextual object form the foundation of Beluga. We therefore start with a brief survey of CMTT and refer to the aforementioned publication for a more in-depth treatment.

CMTT is based on the *intuitionistic modal logic of necessity* (IMLN), which in turn extends basic intuitionistic logic with a judgemental notion of *categorical truth*. This means that there are propositions which are understood to be true in every reasoning context, that is, in *every world* in the language of modal logic. Such propositions are usually referred to as *valid* and a modal operator of necessity, $\Box A$, captures this universal truth. The type system of the theory tracks two contexts, one for global assumptions and one for local assumptions (i.e. the *current world*), and describes the interaction of validity and local truth. The most interesting aspect is that a valid proposition A can be assumed to hold under every local reasoning context Γ.

The main idea of CMTT is to relativise the notion of unconditional validity ($\Box A$) to *contextual validity* ($\Box_\Psi A$). We thus have propositions which hold under some but not necessarily all reasoning contexts. To this end, the modality is indexed with an explicit context of assumptions Ψ, where $\Box_\Psi A$ should be understood as "A holds in all reasoning contexts which contain Ψ" or alternatively "A is valid under Ψ". Note however, that $\Box_\Psi A$ is still independent of local reasoning contexts Γ. We can recover the unconditional IMLN validity of A as $\Box_\bullet A$, since $\bullet$ denotes the empty context, which is contained in any given context. When we want to use the truth of a proposition A, which is contextually valid under Ψ, under a local reasoning context Γ, then we have to ensure that $\Psi \subseteq \Gamma$.

Another reading of the contextual modality $\Box_\Psi A$ yields the notion of a **contextual object**, that is a potentially open object paired with a context in which it is meaningful [Pie08]. In Beluga, contextual objects are expressed as $[\gamma \vdash K]$, where K is any LF entity (term/type/context/...). Since judgements are expressed as LF types, this captures derivations as well. Observe how K is a potentially open entity at the specification level, while $[\gamma \vdash K]$ lives at the reasoning level and is closed.

In this setup we quickly reach a point where we consider $[\gamma \vdash M]$, where M is a CMTT eigenvariable that we wish to unify with some LF expression K. Now for

$[\gamma \vdash K]$ to be well-formed, the free references in K have to be tied to γ, such that we again end up with a closed and meaningful expression. In Beluga this is achieved with first-class substitutions, and powerful mechanisms that infer and apply such substitutions tacitly, though manual substitution handling is possible at any given point (and sometimes necessary). These inference features, like most of Beluga's implementation details, derive from the CMTT metatheory.

Note that contextual objects provide a form of containment with respect to LF-level binders. Let $[\gamma \vdash K]$ be a contextual object where K has a binder at the outermost position, with body K'. When we open this binder, we introduce a new variable into the scope. As a consequence, γ is automatically extended to γ' to account for this scope change. The result is that $[\gamma' \vdash K']$ is again a closed and well-formed contextual object. This prevents object-level assumptions from escaping into the reasoning-level context and hence admits inductive reasoning over contextual objects.

6.5.2 Contexts in Beluga

The presence of contextual objects at the reasoning level of Beluga turns contexts γ into first-class citizens. Before we can dive into our correspondence proof, it is therefore necessary to survey their structure and associated principles.

Contexts in Beluga are best understood as **ordered sequences of named records**, or blocks, e.g.

$$\gamma := \boxed{}_p, \boxed{}_q, \boxed{}_r.$$

The least amount of information that such a record can contain is the existence of a certain LF variable. If we assume that we have defined some LF type $\mathsf{t} : \mathbf{Type}$, then we could have a block

$$\boxed{x{:}\mathsf{t}}_p.$$

Note that a concrete LF-entity under a context containing p would reference this variable as $p.x$, which highlights that the name x is tied to the record/block.

We can also consider more complex records which illustrates their dependent nature. Consider for example that we have formulated some LF equality type $\mathsf{eq} : \mathsf{t} \to \mathsf{t} \to \mathbf{Type}$, and that we are in the process of proving reflexivity of said relation. Then in the variable case we would need the reflexivity assumption for the variable in question. Beluga allows us to package this additional information together with the argument, that is we can have records of the form

$$\boxed{x{:}\mathsf{t}, e : \mathsf{eq}\, x\, x}_p.$$

In a proof, this record would allow us to infer $p.e : \mathsf{eq}\, p.x\, p.x$ in the variable case. Note that in the following we prefer symbolic notion and would instead write $p.e : p.x = p.x$.

Furthermore, the setup allows us to package multiple variables into a single record. We could for example have blocks like

$$\boxed{x{:}\mathsf{t}, y{:}\mathsf{t}, e : \mathsf{eq}\, x\, y}_{p},$$

which capture pairs of equal variables. As above, we package the proof of equality together with the two arguments themselves. We will later employ similar techniques to deal with the issue of tracking pairs of corresponding variables.

The packaging of variables with auxiliary information is already quite useful, but Beluga admits a further degree of dependence, namely the existence of certain LF terms, prior to the formation of a new record. Recall that we claimed that contexts are *ordered*. This is relevant for the binding of variables in such exterior terms.

To illustrate this concept, let us turn to the familiar notion of type assignment. Assume we have defined two LF types for object terms and types respectively, namely $\mathsf{tm} : \mathbf{Type}$ and $\mathsf{ty} : \mathbf{Type}$, together with a typing predicate $\mathsf{of} : \mathsf{tm} \to \mathsf{ty} \to \mathbf{Type}$. Now consider a typing context γ, that we wish to extend with a new *well-typed* term variable. In Beluga we express this as

$$\gamma' \coloneqq \gamma, \boxed{x{:}\mathsf{tm}, h : \mathsf{of}\, x\, A}_{p}^{(A:\mathsf{ty})}.$$

Here A can depend on variables in γ, $p.x$ is the new variable and $p.h$ is a proof that $p.x$ has type A.

To control all of this, Beluga provides a dedicated typing mechanism, called **schema ascription**, where **context schemas** are the types of contexts [Sch00]. They can be used to describe the structure of blocks that may occur in a well-typed context. Let S be some schema and γ some Beluga context, i.e. a sequence of records. Then context ascription is expressed as $\gamma : \mathsf{S}$, which states that each record $p \in \gamma$ matches S.

For our typing example from above we could define the following schema to capture typing contexts.

$$\mathsf{S}_{\mathsf{ty}} \coloneqq \mathtt{some}\, [A{:}\mathsf{ty}]\, \mathtt{block}\, x{:}\mathsf{tm}, h : \mathsf{of}\, x\, A;$$

Note that conceptually the keyword `some` in this setting yields an existential quantification, while a `block` amounts to a dependent sum type. Both constructs are particular to schema definitions and not general connectives of Beluga's reasoning logic CMTT.

Since our correspondence proof heavily relies on the correct choice of schema definitions, and since these are much more intricate than the one we just presented, it seems prudent to adopt a more concise, mathematical notation. In the following we would phrase the above definition as

$$\mathsf{S}_{\mathsf{ty}} \coloneqq [x{:}\mathsf{tm}, h : \mathsf{of}\, x\, A]$$

where we dropped Beluga's keywords and the existential quantification over A is left implicit. We will also insert parentheses where it clarifies the meaning of a block or schema.

To see how schema ascription is used in conjunction with contextual objects, consider the following formulation of type uniqueness, where we assume a suitable LF equality predicate $\mathsf{eq} : \mathsf{ty} \to \mathsf{ty} \to \mathbf{Type}$.

$$u : \forall \gamma{:}\mathsf{S}_{\mathsf{ty}}.\ [\gamma \vdash \mathsf{of}\ M\ A] \implies [\gamma \vdash \mathsf{of}\ M\ B] \implies [\gamma \vdash \mathsf{eq}\ A\ B]$$

Note that this is a reasoning-level typing statement, and any total Beluga function u that implements the given type constitutes a proof of type uniqueness. The eigenvariables M, A and B are implicitly universally quantified.

In addition, Beluga contexts may consist of records of varying shapes. A schema definition for such heterogeneous contexts looks as follows.

$$\mathsf{S} \coloneqq \mathsf{S}_1 + \mathsf{S}_2 + \ldots + \mathsf{S}_n$$

Here each S_i corresponds to a separate block definition, where implicitly existentially quantified parameters are local to each block, and each $p \in \gamma : \mathsf{S}$ has to match one of the S_i.

Finally, Beluga comes with built-in notions of automatic **schema weakening** and **schema strengthening**. As usual they refer to the admissible addition or removal of redundant contextual information, respectively. In our present setting, context modifications can happen in two positions. Firstly, we can add or remove individual records. Secondly we can add, modify or remove complete record shapes from a given schema, which affects all records of this shape. These techniques allow us to use contexts $\gamma : \mathsf{S}_A$ in positions where contexts of schema S_B are needed, given that Beluga can automatically find a suitable weakening or strengthening (depending on the direction of reasoning) from S_A to S_B. While it is trivial to find a concrete context weakening, the inverse direction is more involved. For the latter Beluga looks at the contextual objects under the context in question and consults the subordination order for the relevant type families to identify those pieces of information which certainly do not contribute to the well-formedness of the object. To illustrate these principles, we recall the schema S_{ty} and formulate a variant $\mathsf{S}'_{\mathsf{ty}}$ where each record carries an added reflexive equality assumption.

$$\mathsf{S}_{\mathsf{ty}} \coloneqq [x{:}\mathsf{tm}, h : \mathsf{of}\ x\ A]$$
$$\mathsf{S}'_{\mathsf{ty}} \coloneqq [x{:}\mathsf{tm}, h : \mathsf{of}\ x\ A, e : x = x]$$

Now consider that for some reason we have $\gamma' : \mathsf{S}'_{\mathsf{ty}}$, as well as $[\gamma' \vdash \mathsf{of}\ M\ A]$ and $[\gamma' \vdash \mathsf{of}\ M\ B]$. Say we wish to infer $[\gamma' \vdash \mathsf{eq}\ A\ B]$ by instantiating the uniqueness function u defined above. Unfortunately, the context schemas do not match. Beluga does, however, known that $= \not\trianglelefteq \mathsf{of}$. This in turn makes it safe to strengthen γ' to $\gamma : \mathsf{S}_{\mathsf{ty}}$ in our two premises, by stripping the equality assumptions from all records, which yields $[\gamma \vdash \mathsf{of}\ M\ A]$ and $[\gamma \vdash \mathsf{of}\ M\ B]$. We can now apply u, which in turn yields $[\gamma \vdash \mathsf{eq}\ A\ B]$ and thus by weakening $[\gamma' \vdash \mathsf{eq}\ A\ B]$. In practice, we would directly apply the function u to our premises, while the necessary strengthening and weakening steps occur behind the scenes.

6.5.3 The Correspondence Proof

We now consider the Beluga variant of our correspondence proof. Due to the outlined aspects of Beluga's metatheory and the fact that proof terms have to be formulated explicitly, its proofs may appear somewhat unintuitive at first. Let us therefore start slowly and first consider **propagation** for $\lambda 2$, which allows us to study the idiosyncrasies of Beluga proofs without the added complexity of multiple systems.

We recall Definition 6.5.1 and consider the following informal presentation of the propagation lemma.

$$S :_\lambda T \implies T = \Box \vee \exists U.\ T :_\lambda U \wedge \mathcal{U}\ U$$

When we translate this into the language of CMTT, we face three main challenges. The first two relate to the handling of the LF eigenvariables S, T and U, and the treatment of open expressions. Both are integral aspects of the logical embedding that, in Beluga, is mediated through contextual objects and context schemas, and therefore closely coupled. The third complication arises from the set of logical connectives in the conclusion of the lemma, since none of them are built into CMTT. We will now discuss how we deal with each of these concerns before we formally establish the lemma.

We start with the parameters S, T and U, which are eigenvariables at the LF level. So in order to deal with them at the reasoning level, we have to wrap statements about them as contextual objects. At this point we recall that the eventual argument will proceed by induction on the given typing derivation. This will include two cases where binders are opened and contexts are extended accordingly. So to have any hope of completing the proof, we should formulate the statement in such a way that the eigenvariables are allowed to be open. This means that we need to quantify over a context, which in turn requires a schema. To sum it up, we will deal with objects like $[\gamma \vdash S :_\lambda T]$ and have to somehow control the shape of γ, which brings us to our second challenge.

The changes in scope throughout the proof will all arise from the induction on $S :_\lambda T$, so it seems prudent to take a closer look at the underlying definition in Figure 6.4. When we focus on the occurrences of hypothetical and locally quantified premises, which are the constructions that result in context extensions, then we might come up with the following schema.

$$\mathsf{S}_\lambda \coloneqq [x{:}\mathsf{Tm}_\lambda, h : (x :_\lambda S), h_S : (S :_\lambda T), j_T : \mathcal{U}\ T]$$

The schema S_λ is **canonical** in that its shape is based purely on the structure of the involved LF type families. It has been occasionally conjectured, that it should in principle be possible to infer canonical schemas automatically.

Unfortunately, S_λ is not good enough for our propagation proof, since it fails to capture the semantic distinction between type and term variables. We can build this distinction into our contexts by manually refining S_λ to the following.

$$\overline{\mathsf{S}_\lambda} \coloneqq [x{:}\mathsf{Tm}_\lambda, h : (x :_\lambda *)] + [x{:}\mathsf{Tm}_\lambda, h : (x :_\lambda S), j : (S :_\lambda *)]$$

Here we have exploited semantic domain knowledge, like the fact that $S :_\lambda *$ represents semantic type formation or that $S :_\lambda \Box$ should entail $S = *$, to determine which auxiliary typing information we package with each block. The first block variant now represents a semantic type variable, while the second represents a semantic term variable. We use the bar annotation to indicate that this schema definition is based on information which is not readily available from the underlying LF type families, that is, it is *not canonical*. The design of the presented refinement is guided by the following key observation: it is usually easy to determine the nature of a variable at the binding site, while this information is mainly needed when the bound variable actually occurs. In other words, we have the required information available, when we add a variable to a context, but we need it when we extract the variable. With the refined schema $\overline{\mathsf{S}_\lambda}$, contexts turn into conduits for the transfer of this auxiliary information from binding sites to occurrences.

The third and final challenge, as mentioned above, is the absence of various logical connectives in CMTT. For our purposes we must find a way to express $\wedge$, $\vee$ and $\exists$, as well as a notion of equality for Tm_λ. The trick here is to define a custom LF type family that exactly captures the logical structure of our lemma conclusion. We observe that the result is a property of the type $T : \mathsf{Tm}_\lambda$, so we define a predicate $\mathsf{isty}_\lambda : \mathsf{Tm}_\lambda \to \mathbf{Type}$ with two constructors that reflect the top-level disjunction.

$$\frac{}{\mathsf{isty}_\lambda\ \Box}\ \mathsf{I}_\lambda^{\Box} \qquad\qquad \frac{T :_\lambda U \qquad \mathcal{U}\ U}{\mathsf{isty}_\lambda\ T}\ \mathsf{I}_\lambda^{\mathsf{type}}$$

Observe how the non-occurrence of U in the conclusion of the second rule provides the existential quantification, and the presence of multiple premises captures conjunction. Note also that due to unification of T with $\Box$ in the first rule, we did not have to handle equality. Since we do however later need it for other purposes, we can express it in the same way and define the LF predicate $\mathsf{eq}_\lambda : \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda \to \mathbf{Type}$ with only a single reflexivity constructor.

$$\frac{}{\mathsf{eq}_\lambda\ S\ S}\ \mathsf{E}_\lambda^{\mathsf{refl}}$$

We can finally state and prove our Beluga version of $\lambda 2$ propagation. We give the proof in full detail, including the final proof term, to illustrate the inner workings of Beluga. Later on we will stick to a more mathematical level of presentation.

Lemma 6.5.2 (Propagation for $\lambda 2$) There exists a total function k that satisfies the following typing.

$$k : \forall \gamma{:}\overline{\mathsf{S}_\lambda}.\ [\gamma \vdash S :_\lambda T] \implies [\gamma \vdash \mathsf{isty}_\lambda\ T]$$

Proof We recall that the proof proceeds by induction on the typing derivation, so we require a total recursive function. Thus our basic setup looks like this:

$$\begin{aligned}&\mathsf{rec}\, k : \forall \gamma{:}\overline{\mathsf{S}_\lambda}.\ [\gamma \vdash S :_\lambda T] \implies [\gamma \vdash \mathsf{isty}_\lambda\, T] = \\ &\quad / \,\mathsf{total}\, \mathcal{D}\, (k \_\_\_ \mathcal{D}) \,/ \\ &\quad \lambda \mathcal{D} \Rightarrow ?\end{aligned}$$

There are several things to note.

Firstly, the type is universally quantified over a context γ and two contextual objects $S, T : [\gamma \vdash \mathsf{Tm}_\lambda]$. We had to give the context explicitly since we want to specify the schema, but all three are treated as implicit arguments when it comes to applying k. Together, these three arguments constitute the **metacontext** at the program point denoted by (?). In contrast to this, the derivation $\mathcal{D} : [\gamma \vdash S :_\lambda T]$ is introduced into the **local context** with an explicit abstraction, and we also have k in the local context for recursive calls. Observe how universal quantification and implication are treated separately.

Secondly, recall that LF eigenvariables are always paired with substitutions when they are placed inside contextual objects, to tie their free references to the context of the contextual object. With this in mind, the full type of k is actually

$$\forall \gamma{:}\overline{\mathsf{S}_\lambda}.\ [\gamma \vdash S[..] :_\lambda T[..]] \implies [\gamma \vdash \mathsf{isty}_\lambda\, T[..]]$$

where $(..)$ denotes the identity substitution with respect to the currently ambient context, here γ. Since the required substitution is very often $(..)$, Beluga implicitly assumes it if we do not specify the substitution explicitly.

Finally, the totality annotation at the start of the function instructs Beluga's type checker to verify that our function is recursive in the derivation $\mathcal{D}$, which is the fourth argument when we disregard implicitness of arguments. This amounts to checks that all recursive calls are structurally decreasing in the fourth argument and additionally that the preceding case analysis on $\mathcal{D}$ is exhaustive.

So let us now consider all possible forms of $\mathcal{D}$. In total, we will have to consider seven cases: four structural ones and three for context lookup, due to our detailed schema. We cover the structural cases, which correspond to the rules in Figure 6.4, first.

The easiest case has $\mathcal{D} = [\gamma \vdash \mathsf{T}^{\mathsf{ax}}_\lambda] : [\gamma \vdash * :_\lambda \Box]$ and we need to provide a contextual object of type $[\gamma \vdash \mathsf{isty}_\lambda \Box]$. Observe how unification has fixed S and T to $*$ and respectively $\Box$. The witness is of course $[\gamma \vdash \mathsf{I}^{\Box}_\lambda]$.

The next case we want to consider is the formation of function types which is still relatively straightforward. We have $\mathcal{D} : [\gamma \vdash \Pi S.\, (\lambda x \Rightarrow T[.., x]) :_\lambda *]$, and unification turns our goal into $[\gamma \vdash \mathsf{isty}_\lambda *]$. The required contextual object is constructed as $[\gamma \vdash \mathsf{I}^{\mathsf{type}}_\lambda\, \mathsf{T}^{\mathsf{ax}}_\lambda\, \mathsf{U}^{\Box}_\lambda]$.

The third structural case deals with abstractions, which have dependent function types as types, which in turn are typed with $*$. In order to ascertain that the function type in question is in fact a well-formed type we need to appeal to the corresponding

typing rule, which in turn demands a number of subderivations. To see how we access these subderivations in Beluga, it is instructive to carefully study the respective matching pattern and the resulting goal state. We have

$$\begin{aligned}\mathcal{D} =&[\gamma \vdash \mathsf{T}^{\lambda}_{\lambda}\, \mathcal{D}_S\, \mathcal{D}_U\, (\lambda x \mathcal{E} \Rightarrow \mathcal{D}_V[.., x, \mathcal{E}])\, (\lambda x \mathcal{E} \Rightarrow \mathcal{D}_T[.., x, \mathcal{E}])] \\ &: [\gamma \vdash \lambda S.\, (\lambda x \Rightarrow V[.., x]) :_{\lambda} \Pi S.\, (\lambda x \Rightarrow T[.., x])]\end{aligned}$$

and our metacontext contains $\gamma : \overline{\mathsf{S}_{\lambda}}$ as well as the following assumptions, where we explicitly spell out all closing substitutions.

$$\begin{array}{ll} S : [\gamma \vdash \mathsf{Tm}_{\lambda}] & \mathcal{D}_S : [\gamma \vdash S[..] :_{\lambda} U[..]] \\ U : [\gamma \vdash \mathsf{Tm}_{\lambda}] & \mathcal{D}_U : [\gamma \vdash \mathcal{U}\ U[..]] \\ V : [\gamma, x{:}\mathsf{Tm}_{\lambda} \vdash \mathsf{Tm}_{\lambda}] & \mathcal{D}_V : [\gamma, x{:}\mathsf{Tm}_{\lambda}, \mathcal{E} : x :_{\lambda} S[..] \vdash V[.., x] :_{\lambda} T[.., x]] \\ T : [\gamma, x{:}\mathsf{Tm}_{\lambda} \vdash \mathsf{Tm}_{\lambda}] & \mathcal{D}_T : [\gamma, x{:}\mathsf{Tm}_{\lambda}, \mathcal{E} : x :_{\lambda} S[..] \vdash T[.., x] :_{\lambda} *] \end{array}$$

Let us consider V and $\mathcal{D}_V$ in detail. The contextual type of V indicates that V may have $|\gamma| + 1$ free references. Observe how V is placed into the contextual type of the original $\mathcal{D}$, where the context has only $|\gamma|$ entries, using an LF-level abstraction and substitution with the abstracted variable, viz $(\lambda x \Rightarrow V[.., x])$. Similarly, the derivation $\mathcal{D}_V$ expects two additional assumptions over γ, a term x, and a derivation $\mathcal{E}$, which shows that x has type $S[..]$. Again, S may have up to $|\gamma|$ many free references which can be closed with the identity for the present context γ. To close the case we have to provide an object of type $[\gamma \vdash \mathsf{isty}_{\lambda}\, (\Pi S.\, (\lambda x \Rightarrow T[.., x]))]$, which we construct as follows.

$$[\gamma \vdash \mathsf{I}^{\mathsf{type}}_{\lambda}\, (\mathsf{T}^{\Pi}_{\lambda}\, \mathcal{D}_S\, \mathcal{D}_U\, (\lambda x \mathcal{E} \Rightarrow \mathcal{D}_T[.., x, \mathcal{E}]))\, \mathsf{U}^{*}_{\lambda}].$$

This brings us to the last structural case, which is application. The initial match is relatively simple with $\mathcal{D} = [\gamma \vdash \mathsf{T}^{\mathsf{app}}_{\lambda}\, \mathcal{D}_S\, \mathcal{D}_T] : [\gamma \vdash S[..]\, T[..] :_{\lambda} V[.., T[..]]]$. We observe that $\mathcal{D}_S : [\gamma \vdash S[..] :_{\lambda} \Pi U[..].\, (\lambda x \Rightarrow V[.., x])]$ and now employ induction to infer that the function type is well-formed. Since function types do not unify with $\Box$, there is only one possible object that a recursive call to k can return. That is we have

$$k\, [\gamma \vdash \mathcal{D}_S] = [\gamma \vdash \mathsf{I}^{\mathsf{type}}_{\lambda}\, \mathcal{D}_P\, \mathcal{D}_W],$$

where the two derivations $\mathcal{D}_P$ and $\mathcal{D}_W$ have, for some term W, the types

$$\mathcal{D}_P : [\gamma \vdash \Pi U.\, (\lambda x \Rightarrow V[.., x]) :_{\lambda} W[..]] \qquad\qquad \mathcal{D}_W : [\gamma \vdash \mathcal{U}\ W[..]]$$

and a subsequent inversion on $\mathcal{D}_P$ unifies W with $*$ and also reveals a derivation

$$\mathcal{D}_V : [\gamma, x{:}\mathsf{Tm}_{\lambda}, \mathcal{E} : x :_{\lambda} U[..] \vdash V[.., x] :_{\lambda} *].$$

At this point we can close the case with

$$[\gamma \vdash \mathsf{I}^{\mathsf{type}}_{\lambda}\, \mathcal{D}_V[.., \_, \mathcal{D}_T]\, \mathsf{U}^{*}_{\lambda}] : [\gamma \vdash \mathsf{isty}_{\lambda}\, (V[.., T[..]])].$$

Observe how native substitution into parametric subderivations is used to fit $\mathcal{D}_V$ under the context γ, by patching in the arguments T and its typing derivation $\mathcal{D}_T$. The parameter T is inferred, and therefore provided with the wildcard (_).

For the variable cases, let us recall our schema $\overline{\mathsf{S}_\lambda}$. There are three positions where typing derivations may appear in context blocks, and each arises as a case in our proof. We consider them in turn.

First, assume a context entry of the form $p : [\gamma \vdash [x{:}\mathsf{Tm}_\lambda, h : x :_\lambda *]]$ where we have matched $\mathcal{D} = [\gamma \vdash p.h] : [\gamma \vdash p.x :_\lambda *]$. As before, we close with $[\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathsf{T}_\lambda^{\mathsf{ax}}\, \mathsf{U}_\lambda^{\Box}]$.

Now let us instead assume that we have $p : [\gamma \vdash [x{:}\mathsf{Tm}_\lambda, h : x :_\lambda S[..], j : S[..] :_\lambda *]]$. For the case $\mathcal{D} = [\gamma \vdash p.j] : [\gamma \vdash S[..] :_\lambda *]$, we again close with $[\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathsf{T}_\lambda^{\mathsf{ax}}\, \mathsf{U}_\lambda^{\Box}]$. We could however also have $\mathcal{D} = [\gamma \vdash p.h] : [\gamma \vdash p.x :_\lambda S[..]]$. At this point we finally exploit the fact that we packaged additional typing information into the context. We thus conclude with $[\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, (p.j)\, \mathsf{U}_\lambda^{*}]$.

To finalise the proof we put everything together to obtain the following function.

$$
\begin{array}{l}
\mathsf{rec}\ \ k : \forall\gamma{:}\overline{\mathsf{S}_\lambda}.\ [\gamma \vdash S :_\lambda T] \implies [\gamma \vdash \mathsf{isty}_\lambda\, T] = \\
\quad /\ \mathsf{total}\ \ \mathcal{D}\ \ (k\ \_\ \_\ \_\ \mathcal{D})\ / \\
\quad \lambda\mathcal{D} \Rightarrow \mathsf{case}\ \ \mathcal{D}\ \ \mathsf{of} \\
\qquad \mid [\gamma \vdash \mathsf{T}_\lambda^{\mathsf{ax}}] \Rightarrow [\gamma \vdash \mathsf{I}_\lambda^{\Box}] \\
\qquad \mid [\gamma \vdash \mathsf{T}_\lambda^{\Pi}\ \_\ \_\ (\lambda x\mathcal{E} \Rightarrow \_)] \Rightarrow [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathsf{T}_\lambda^{\mathsf{ax}}\, \mathsf{U}_\lambda^{\Box}] \\
\qquad \mid [\gamma \vdash \mathsf{T}_\lambda^{\lambda}\, \mathcal{D}_S\, \mathcal{D}_U\, (\lambda x\mathcal{E} \Rightarrow \_)\, (\lambda x\mathcal{E} \Rightarrow \mathcal{D}_T[.., x, \mathcal{E}])] \Rightarrow \\
\qquad\quad [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, (\mathsf{T}_\lambda^{\Pi}\, \mathcal{D}_S\, \mathcal{D}_U\, (\lambda x\mathcal{E} \Rightarrow \mathcal{D}_T[.., x, \mathcal{E}]))\, \mathsf{U}_\lambda^{*}] \\
\qquad \mid [\gamma \vdash \mathsf{T}_\lambda^{\mathsf{app}}\, \mathcal{D}_S\, \mathcal{D}_T] \Rightarrow \\
\qquad\quad \mathsf{let}\ \ [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathcal{D}_P\, \mathcal{D}_W] = k\, [\gamma \vdash \mathcal{D}_S]\ \ \mathsf{in} \\
\qquad\quad \mathsf{let}\ \ [\gamma \vdash \mathsf{T}_\lambda^{\Pi}\ \_\ \_\ (\lambda x\mathcal{E} \Rightarrow \mathcal{D}_V[.., x, \mathcal{E}])] = [\gamma \vdash \mathcal{D}_P]\ \ \mathsf{in}\ \ [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathcal{D}_V[.., \_, \mathcal{D}_T]\, \mathsf{U}_\lambda^{*}] \\
\qquad \mid \forall p : [\gamma \vdash [x{:}\mathsf{Tm}_\lambda, h : x :_\lambda *]].\ [\gamma \vdash p.h] \Rightarrow [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathsf{T}_\lambda^{\mathsf{ax}}\, \mathsf{U}_\lambda^{\Box}] \\
\qquad \mid \forall p : [\gamma \vdash [x{:}\mathsf{Tm}_\lambda, h : x :_\lambda S[..], j : S[..] :_\lambda *]].\ [\gamma \vdash p.j] \Rightarrow [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, \mathsf{T}_\lambda^{\mathsf{ax}}\, \mathsf{U}_\lambda^{\Box}] \\
\qquad \mid \forall p : [\gamma \vdash [x{:}\mathsf{Tm}_\lambda, h : x :_\lambda S[..], j : S[..] :_\lambda *]].\ [\gamma \vdash p.h] \Rightarrow [\gamma \vdash \mathsf{I}_\lambda^{\mathsf{type}}\, (p.j)\, \mathsf{U}_\lambda^{*}]
\end{array}
$$

This function definition completes the proof of propagation for $\lambda 2$ in Beluga. ∎

Let us next consider the definition of PLC in Beluga. There are some notable differences with respect to the Coq and Abella versions that arise from the use of contextual objects. Recall that up until now we had a separate PLC type formation judgement which ensured that the type variable context covered the free type variables. At this point it is interesting to observe that Coq and Abella type variable contexts simply recorded the existence of a type variable, nothing else. Now consider that, in Beluga, contextual objects are always closed at the reasoning level, since object-level expressions are *always* packaged with closing contexts. In Beluga it is thus impossible to express a PLC type that is not well-formed because of an insufficient context. In

our other two proof systems, such ill-formed objects can be expressed and reasoned about. There we can for example show that assuming the derivability of an ill-formed PLC type entails absurdity. In Beluga, on the other hand, a corresponding ill-formed contextual object is rejected by the type checker.

Now since our notion of type formation is superseded by Beluga's host-level type checking, there is no point in manually implementing it. This is reflected in the following definition. A further consequence is that induction over well-formed types reduces to plain induction on types.

Definition 6.5.3 (PLC in Beluga) We define two LF types $\mathsf{Ty_P} : \mathbf{Type}$ and $\mathsf{Tm_P} : \mathbf{Type}$ with the following grammar.

$$\boxed{\mathsf{Ty_P}} \qquad A, B ::= A \to B \mid \forall.\, A$$

$$\boxed{\mathsf{Tm_P}} \qquad M, N ::= M\, N \mid \lambda A.\, M \mid M\, A \mid \Lambda.\, M$$

The constructor type signatures are given in Figure 6.1. Note that due to the HOAS encoding, no variable cases are specified. We only define a single type family $M :_{\mathsf{P}} A$ to capture typing. Its signature is again derived from the one in Section 6.2 where we replaced **o** by **Type**.

$$\mathsf{of_P} \;:\; \mathsf{Tm_P} \to \mathsf{Ty_P} \to \mathbf{Type} \qquad\qquad M :_{\mathsf{P}} A$$

The constructors are defined in Figure 6.7. They are derived from the inference rules in Figure 6.3 by removing all premises that relate to type formation, for the aforementioned reason. We annotate the two instances of local quantification with type information to avoid confusion.

We also define equality predicates (with obvious signatures) for our two new LF types, as before each with a single reflexivity constructor.

$$\frac{}{\mathsf{tyeq_P}\; A\; A}\;\mathsf{TE}^{\mathsf{refl}}_{\mathsf{P}} \qquad\qquad \frac{}{\mathsf{eq_P}\; M\; M}\;\mathsf{E}^{\mathsf{refl}}_{\mathsf{P}}$$

The correspondence relations are also defined as LF types with constructors outlined in Figure 6.5, though with the usual shift from **o** to **Type**. The setup therefore coincides exactly with the Abella version in this regard. For the two relations, Beluga derives the following subordination information.

$$\mathsf{Ty_P}, \mathsf{Tm}_\lambda, \sim \;\preceq\; \sim \qquad\qquad \mathsf{Ty_P}, \mathsf{Tm_P}, \mathsf{Tm}_\lambda, \sim, \approx \;\preceq\; \approx$$

Based on this and the rules for the two relations we can capture the tightest context invariants with the following two canonical schemas.

$$\mathsf{S}_\sim \coloneqq [x{:}\mathsf{Ty_P}, y{:}\mathsf{Tm}_\lambda, h : x \sim y] + [y{:}\mathsf{Tm}_\lambda]$$
$$\mathsf{S}_\approx \coloneqq [x{:}\mathsf{Ty_P}, y{:}\mathsf{Tm}_\lambda, h : x \sim y] + [x{:}\mathsf{Tm_P}, y{:}\mathsf{Tm}_\lambda, h : x \approx y]$$

Note that subordination admits the strengthening of $\mathsf{S}_\approx$ contexts to $\mathsf{S}_\sim$ contexts in $\sim$ derivations. These contexts allow us to establish the injectivity and functionality results, which we consider next.

$$\frac{M :_{\mathsf{P}} A \to B \qquad N :_{\mathsf{P}} A}{M\ N :_{\mathsf{P}} B}\ \overline{\mathsf{T}}^{\mathsf{app}}_{\mathsf{P}} \qquad \frac{M :_{\mathsf{P}} \forall.\, B}{M\ A :_{\mathsf{P}} B\langle A\rangle}\ \overline{\mathsf{T}}^{\mathsf{tyapp}}_{\mathsf{P}}$$

$$\frac{\mathbf{\Pi} x{:}\mathsf{Tm}_{\mathsf{P}}.\ x :_{\mathsf{P}} A \blacktriangleright M\langle x\rangle :_{\mathsf{P}} B}{\lambda A.\, M :_{\mathsf{P}} A \to B}\ \overline{\mathsf{T}}^{\lambda}_{\mathsf{P}} \qquad \frac{\mathbf{\Pi} x{:}\mathsf{Ty}_{\mathsf{P}}.\ M\langle x\rangle :_{\mathsf{P}} A\langle x\rangle}{\Lambda.\, M :_{\mathsf{P}} \forall.\, A}\ \overline{\mathsf{T}}^{\Lambda}_{\mathsf{P}}$$

Figure 6.7: PLC type system in Beluga.

Theorem 6.5.4 (Functionality of $\sim$ and $\approx$) There exist total functions $f_\sim$ and $f_\approx$ that satisfy the following typings.

$$f_\sim : \forall\gamma{:}\mathsf{S}_\sim.\ [\gamma \vdash A \sim S] \implies [\gamma \vdash A \sim S'] \implies [\gamma \vdash \mathsf{eq}_\lambda\, S\, S']$$
$$f_\approx : \forall\gamma{:}\mathsf{S}_\approx.\ [\gamma \vdash M \approx S] \implies [\gamma \vdash M \approx S'] \implies [\gamma \vdash \mathsf{eq}_\lambda\, S\, S']$$

Proof Both are implemented as recursive functions that structurally decrease on the respective first derivation and then invert the second. The constructions are mostly straightforward. We focus on a number of interesting cases.

We first consider the type-level correspondence where we match on $\mathcal{D} : [\gamma \vdash A \sim S]$ and focus on the case for universal quantification, where we have

$$\begin{aligned}\mathcal{D} = [\gamma \vdash \mathsf{R}^{\forall}_{\sim}\ (\lambda x y \mathcal{E} \Rightarrow \mathcal{D}_{BT}[.., x, y, \mathcal{E}])]\\ : [\gamma \vdash \forall.\, (\lambda x \Rightarrow B[.., x]) \sim \Pi*.\, (\lambda x \Rightarrow T[.., x])]\end{aligned}$$

and inversion on $\mathcal{C} : [\gamma \vdash \forall.\, (\lambda x \Rightarrow B[.., x]) \sim S']$ yields

$$\begin{aligned}\mathcal{C} = [\gamma \vdash \mathsf{R}^{\forall}_{\sim}\ (\lambda x y \mathcal{E} \Rightarrow \mathcal{D}_{BT'}[.., x, y, \mathcal{E}])]\\ : [\gamma \vdash \forall.\, (\lambda x \Rightarrow B[.., x]) \sim \Pi*.\, (\lambda x \Rightarrow T'[.., x])].\end{aligned}$$

Let us take a closer look at the subderivations $\mathcal{D}_{BT}$ and $\mathcal{D}_{BT'}$, which have the following types.

$$\mathcal{D}_{BT} : [\gamma, x{:}\mathsf{Ty}_{\mathsf{P}}, y{:}\mathsf{Tm}_\lambda, \mathcal{E} : x \sim y \vdash B[.., x] \sim T[.., y]]$$
$$\mathcal{D}_{BT'} : [\gamma, x{:}\mathsf{Ty}_{\mathsf{P}}, y{:}\mathsf{Tm}_\lambda, \mathcal{E} : x \sim y \vdash B[.., x] \sim T'[.., y]]$$

Note that the respective contexts do not conform to the schema $\mathsf{S}_\sim$, since γ is in each case extended with three separate assumptions, rather than with a single block. We thus cannot yet apply our inductive hypothesis to infer the equality of T and T'. The trick is to transform the two subderivations into derivations under a suitable context before passing them into the recursive call.

$$\mathcal{D}_{BT}[.., p.x, p.y, p.\mathcal{E}] : [\gamma, p \vdash B[.., p.x] \sim T[.., p.y]]$$
$$\mathcal{D}_{BT'}[.., q.x, q.y, q.\mathcal{E}] : [\gamma, q \vdash B[.., q.x] \sim T'[.., q.y]]$$

Beluga considers these new derivations to still be smaller for purposes of termination checking [PA15], so we can go into recursion and match the result against the reflexivity constructor to unify T with T'.

$$f_{\sim}\,[\gamma, p \vdash \mathcal{D}_{BT}[.., p.x, p.y, p.\mathcal{E}]]\,[\gamma, q \vdash \mathcal{D}_{BT'}[.., q.x, q.y, q.\mathcal{E}]] = [\gamma, r \vdash \mathsf{E}^{\mathsf{refl}}_{\lambda}]$$

Note that the recursive call and the subsequent matching also unified the abstract blocks p, q and r since the context is the same across all three contextual objects. In the full implementation, both p and r have to be annotated with the correct block type from the schema to placate the matching and unification algorithms. The case is closed with $[\gamma \vdash \mathsf{E}^{\mathsf{refl}}_{\lambda}] : [\gamma \vdash \mathsf{eq}_{\lambda}\,(\Pi *.\,(\lambda x \Rightarrow T[.., x]))\,(\Pi *.\,(\lambda x \Rightarrow T[.., x]))]$.

The other interesting case is context lookup. Since the employed schema has only a single block type with a relational assumption we only get the case

$$\mathcal{D} = [\gamma \vdash p.h] : [\gamma \vdash p.x \sim p.y]$$

and since x and y are local to the block p, inversion on $\mathcal{C} : [\gamma \vdash p.x \sim S]$ can only yield

$$\mathcal{C} = [\gamma \vdash p.h] : [\gamma \vdash p.x \sim p.y].$$

At this point $[\gamma \vdash \mathsf{E}^{\mathsf{refl}}_{\lambda}] : [\gamma \vdash \mathsf{eq}_{\lambda}\, p.y\, p.y]$ closes the variable case.

The construction of $f_{\approx}$ is completely analogue. The cases for ordinary abstraction ($\mathsf{R}^{\lambda}_{\approx}$) and type application ($\mathsf{R}^{\mathsf{tyapp}}_{\approx}$) do of course contain $\sim$ subderivations, for which $f_{\sim}$ is invoked instead of a recursive call to $f_{\approx}$. For the two abstraction cases we use the same repackaging trick from above. That is we add a fresh block p to the context and then bind the extra assumptions with projections from said block, prior to the recursive call on the abstraction bodies. Care has to be taken that for the ordinary abstraction case ($\mathsf{R}^{\lambda}_{\approx}$), we must force $p : [x{:}\mathsf{Tm}_{\mathsf{P}}, y{:}\mathsf{Tm}_{\lambda}, h : x \approx y]$, while for type abstraction ($\mathsf{R}^{\Lambda}_{\approx}$) we use $p : [x{:}\mathsf{Ty}_{\mathsf{P}}, y{:}\mathsf{Tm}_{\lambda}, h : x \sim y]$ instead. The variable case is again straightforward since only the projection $p.h$ of context blocks $p : [x{:}\mathsf{Tm}_{\mathsf{P}}, y{:}\mathsf{Tm}_{\lambda}, h : x \approx y]$ can be matched, which is automatically determined by unification. From there we proceed as we did for $f_{\sim}$. ∎

For injectivity we again require a no-clash theorem with respect to the two relations and the unified PTS syntax. To be precise, we only require it for the injectivity of $\approx$. Let false be an LF type with no constructors, which therefore has no derivations and thus encodes absurdity. Then we have the following result, where we again make use of the schema $\mathsf{S}_{\approx}$.

Lemma 6.5.5 (Disjointness of Codomains) There exists a total function $d^{\sim}_{\approx}$ that satisfies the following typing.

$$d^{\sim}_{\approx} : \forall \gamma{:}\mathsf{S}_{\approx}.\ [\gamma \vdash M \approx S] \implies [\gamma \vdash A \sim S] \implies [\gamma \vdash \mathsf{false}]$$

Proof We define $d^{\sim}_{\approx}$ by structural recursion on the derivation $\mathcal{C} : [\gamma \vdash A \sim S]$. In each of the three cases, unification is able to discover that the assumed derivation $\mathcal{D} : [\gamma \vdash M \approx S]$ cannot exist. We never have to give a derivation of $[\gamma \vdash \mathsf{false}]$. ∎

Theorem 6.5.6 (Injectivity of $\sim$ and $\approx$) There exist total functions $i_\sim$ and $i_\approx$ that satisfy the following typings.

$$i_\sim : \forall\gamma{:}\mathsf{S}_\sim.\ [\gamma \vdash A \sim S] \implies [\gamma \vdash A' \sim S] \implies [\gamma \vdash \mathsf{tyeq_P}\ A\ A']$$
$$i_\approx : \forall\gamma{:}\mathsf{S}_\approx.\ [\gamma \vdash M \approx S] \implies [\gamma \vdash M' \approx S] \implies [\gamma \vdash \mathsf{eq_P}\ M\ M']$$

Proof We define $i_\sim$ and $i_\approx$ by structural recursion on the first derivation and subsequent discrimination on the second derivation. The function $i_\approx$ of course invokes $i_\sim$ for subderivations involving $\sim$. The definitions are almost identical to those of $f_\sim$ and $f_\approx$ which is actually quite surprising. Recall that both Coq and Abella had to handle problematic cases where the inversion erroneously chose the wrong relational derivation rule which led to goals with impossible equality claims like $A \to B = \forall.\, C$. These had to be manually discharged with further inversion lemmas. Beluga, on the other hand, detects and discards most of these cases automatically. Only the disambiguation of the two $\lambda 2$ applications to their respective PLC type and term applications requires manual intervention with calls to $d^{\sim}_{\approx}$ to discard the two spurious matches. ∎

Now in order to obtain the preservation results we again have to consider schemas which are not canonical. While the two canonical schemas $\mathsf{S}_\sim$ and $\mathsf{S}_\approx$ already disambiguated type and term variables, we now also require additional information about the types of the variables as well as the fact that related term variables do in fact have related types. In total we need three new schemas. Two of these are used for the two preservation proofs at the type level, while the third covers both preservation proofs at the term level. The three new schemas are the following.

$$\begin{aligned}
\overline{\mathsf{S}_{\sim,\mathsf{P}}} &\coloneqq [x{:}\mathsf{Ty_P}, y{:}\mathsf{Tm}_\lambda, h : (x \sim y), t_y : (y :_\lambda *)] \\
&\quad + [y{:}\mathsf{Tm}_\lambda, t_y : (y :_\lambda S)] \\
\overline{\mathsf{S}_{\sim,\lambda}} &\coloneqq [x{:}\mathsf{Ty_P}, y{:}\mathsf{Tm}_\lambda, h : (x \sim y), t_y : (y :_\lambda *)] \\
&\quad + [y{:}\mathsf{Tm}_\lambda, t_y : (y :_\lambda S), h_{AS} : (A \sim S), t_S : (S :_\lambda *)] \\
\overline{\mathsf{S}_{\approx}} &\coloneqq [x{:}\mathsf{Ty_P}, y{:}\mathsf{Tm}_\lambda, h : (x \sim y), t_y : (y :_\lambda *)] \\
&\quad + [x{:}\mathsf{Tm_P}, y{:}\mathsf{Tm}_\lambda, h : (x \approx y), t_x : (x :_\lambda A), t_y : (y :_\lambda S), h_{AS} : (A \sim S)]
\end{aligned}$$

Note how they all utilise the same block type for type-level variables, while progressively more information is tracked for related term variables. The reason we need to track term variable information for the type-level proofs arises from the fact that in $\lambda 2$ arrow types are represented by dependent function types, which, while vacuous, are binders that still change the scope. Meanwhile, the reason why we require two separate schemas for the two type-level preservation proofs, arises from the fact that the preservation of type formation from PLC to $\lambda 2$ is an induction directly on a PLC type, rather than on a judgement (which, as we recall, does not exists in Beluga).

We also require further custom LF predicates which capture the various existential conclusions of our preservation theorems. We introduce these together with their respective theorems.

Let us first consider the preservation of type formation from PLC to $\lambda 2$. Due to our current setup, we have to state that for all PLC types A, there exists a $\lambda 2$ term S in the universe $*$. We capture the latter with the following predicate.

$$\frac{A \sim S \qquad S :_\lambda *}{\mathsf{tyex}_\lambda\, A}\, \mathsf{X}^{\sim}_{\lambda}$$

Theorem 6.5.7 (Preservation of PLC Type Formation under $\sim$) There is a total function $p_{\sim,\mathsf{P}}$ that satisfies the following typing.

$$p_{\sim,\mathsf{P}} : \forall \gamma{:}\overline{\mathsf{S}_{\sim,\mathsf{P}}}.\ \forall A{:}[\gamma \vdash \mathsf{Ty}_\mathsf{P}].\ [\gamma \vdash \mathsf{tyex}_\lambda\, A]$$

Proof We define $p_{\sim,\mathsf{P}}$ by recursion on the structure of $A : [\gamma \vdash \mathsf{Ty}_\mathsf{P}]$. Here it is necessary to maintain both γ and A as explicit parameters, since the adjustment of contexts becomes more involved for the recursive cases.

Consider the case $A = [\gamma \vdash A_1 \to A_2]$ and recall rule $(\mathsf{R}^{\to}_{\sim})$ for relating arrow types. Note how the codomain A_2 has to be related to a term S_2 under a context that will carry an additional $\lambda 2$ variable x, even though A_2 has no added dependencies. We need to keep this in mind for our recursive call for A_2. Before we can do that, however, we need to obtain a matching term for A_1. By recursion we get

$$p_{\sim,\mathsf{P}}\,[\gamma]\,[\gamma \vdash A_1] = [\gamma \vdash \mathsf{X}^{\sim}_{\lambda}\, S_1\, \mathcal{E}_1\, \mathcal{T}_1]$$

where $S_1 : [\gamma \vdash \mathsf{Tm}_\lambda]$, $\mathcal{E}_1 : [\gamma \vdash A_1[..] \sim S_1[..]]$ and $\mathcal{T}_1 : [\gamma \vdash S_1[..] :_\lambda *]$. We can now consider an extended context with a fresh block, namely $[\gamma, q : [y{:}\mathsf{Tm}_\lambda, t_y : (y :_\lambda S_1[..])]]$, and use that to recurse on A_2.

$$p_{\sim,\mathsf{P}}\,[\gamma, q]\,[\gamma, q \vdash A_2[..]] = [\gamma, q \vdash \mathsf{X}^{\sim}_{\lambda}\, S_2[..]\, \mathcal{E}_2[.., q.y]\, \mathcal{T}_2[.., q.y, q.t_y]]$$

Here it is interesting to consider the resulting contextual types after matching on the existential. First of all we have $S_2 : [\gamma \vdash \mathsf{Tm}_\lambda]$, which yields the vacuity of the binder. Then for the derivations we have $\mathcal{E}_2 : [\gamma, y{:}\mathsf{Tm}_\lambda \vdash A_2[..] \sim S_2[..]]$ and $\mathcal{T}_2 : [\gamma, y{:}\mathsf{Tm}_\lambda, \mathcal{T} : y :_\lambda S_1[..] \vdash S_2[..]]$. Note that these two have their respective context extensions separately, i.e. not packaged as a block. We close the case with

$$\begin{aligned}&[\gamma \vdash \mathsf{X}^{\sim}_{\lambda}\ \_\,(\mathsf{R}^{\to}_{\sim}\, \mathcal{E}_1\, (\lambda y \Rightarrow \mathcal{E}_2[.., y]))\, (\mathsf{T}^{\Pi}_{\lambda}\, \mathcal{T}_1\, \mathsf{U}^{*}_{\lambda}\, (\lambda y \mathcal{T} \Rightarrow \mathcal{T}_2[.., y, \mathcal{T}]))]\\ &\quad : [\gamma \vdash \mathsf{tyex}_\lambda\, (A_1[..] \to A_2[..])].\end{aligned}$$

The case for $A = [\gamma \vdash \forall.\, A']$ proceeds similarly, and the variable case is also straightforward, since there is only one occurrence of a Ty_P in the schema $\overline{\mathsf{S}_{\sim,\mathsf{P}}}$. ∎

At this point we are faced with a peculiarity. So far we always considered the inverse of each preservation result to establish the desired equivalence. For the theorem we have just established, this does not make any sense. If we were to assume $[\gamma \vdash \mathsf{tyex}_\lambda\, A]$

we would need to show that A is an LF term of type $\mathsf{Ty_P}$, but this already holds by virtue of Beluga's type checker when it accepts the contextual object $[\gamma \vdash \mathsf{tyex}_\lambda\, A]$ as well-formed. Hence formulating an inverse variant of the proven preservation result is meaningless. For the other three preservation results this issue does not arise and sensible inverses can be formulated and proven.

By now we have also seen all technical tricks that are needed to properly construct the various required proof terms in Beluga. We therefore keep the remaining presentations brief. For the final three preservation results we require the following LF predicates,

$$\frac{A \sim S}{\mathsf{tyex_P}\, S}\ \mathsf{X}^{\sim}_{\mathsf{P}} \qquad \frac{\begin{array}{cc} M \approx S & A \sim T \\ S :_\lambda T & T :_\lambda * \end{array}}{\mathsf{tmex}_\lambda\, M\, A}\ \mathsf{X}^{\approx}_{\lambda} \qquad \frac{\begin{array}{cc} M \approx S & A \sim T \\ \multicolumn{2}{c}{M :_{\mathsf{P}} A} \end{array}}{\mathsf{tmex_P}\, S\, T}\ \mathsf{X}^{\approx}_{\mathsf{P}}$$

which capture the following logical expressions, respectively.

$$\begin{aligned} \mathsf{tyex_P}\, S &\iff \exists A{:}\mathsf{Ty_P}.\ A \sim S \\ \mathsf{tmex}_\lambda\, M\, A &\iff \exists S, T{:}\mathsf{Tm}_\lambda.\ M \approx S \wedge A \sim T \wedge S :_\lambda T \wedge T :_\lambda * \\ \mathsf{tmex_P}\, S\, T &\iff \exists M{:}\mathsf{Tm_P}, A{:}\mathsf{Ty_P}.\ M \approx S \wedge A \sim T \wedge M :_{\mathsf{P}} A \end{aligned}$$

Theorem 6.5.8 (Preservation of λ2 Type Formation under ∼) There exists a total function $p_{\sim,\lambda}$ that satisfies the following typing.

$$p_{\sim,\lambda} : \forall \gamma{:}\overline{\mathsf{S}_{\sim,\lambda}}.\ [\gamma \vdash S :_\lambda *] \implies [\gamma \vdash \mathsf{tyex_P}\, S]$$

Proof We define $p_{\sim,\lambda}$ by structural recursion on the derivation $\mathcal{D} : [\gamma \vdash S :_\lambda *]$. We get two structural cases for dependent function types, which split on the universe of the domain being $*$ or $\Box$. All other structural cases are discharged by unification.

For variables we have to consider three cases, since there are three occurrences of the discriminated typing in $\overline{\mathsf{S}_{\sim,\lambda}}$. For the one occurrence in the type variable block the result is straightforward. If we instead match the *second* typing in the term variable block $(S :_\lambda *)$ we can still close the goal, because we also packaged a related PLC type A. For the case where we match the first typing, S is unified with $*$, after which we can discharge the case because the packaged derivation of $A \sim *$ cannot exist. ∎

Corollary 6.5.9 There is a total function $\widehat{p_{\sim,\lambda}}$, which forms an equivalence with $p_{\sim,\lambda}$. That is we have

$$\widehat{p_{\sim,\lambda}} : \forall \gamma{:}\overline{\mathsf{S}_{\sim,\lambda}}.\ [\gamma \vdash \mathsf{tyex_P}\, S] \implies [\gamma \vdash S :_\lambda *].$$

Proof We proceed analogue to the corresponding results in Coq and Abella. We first unpack the existential and then apply $p_{\sim,\mathsf{P}}$ to the unpacked PLC type A. At this point we have $[\gamma \vdash A \sim S]$ from the first unpacking and $[\gamma \vdash A \sim S']$ from the function call, which also yields $[\gamma \vdash S' :_\lambda *]$. We unify S and S' with $f_\sim$. ∎

Theorem 6.5.10 (Preservation of Typing under ≈) We can define total functions $p_{\approx,\mathsf{P}}$ and $p_{\approx,\lambda}$ that satisfy the following typings.

$$p_{\approx,\mathsf{P}} : \forall\gamma{:}\overline{\mathsf{S}_{\approx}}.\ [\gamma \vdash M :_{\mathsf{P}} A] \implies [\gamma \vdash \mathsf{tmex}_{\lambda}\, M\, A]$$
$$p_{\approx,\lambda} : \forall\gamma{:}\overline{\mathsf{S}_{\approx}}.\ [\gamma \vdash S :_{\lambda} T] \implies [\gamma \vdash T :_{\lambda} *] \implies [\gamma \vdash \mathsf{tmex}_{\mathsf{P}}\, S\, T]$$

Proof We define both functions by structural recursion on the respective first derivation.

For $p_{\approx,\mathsf{P}}$, most cases are straightforward, with abstraction and type application utilising $p_{\sim,\mathsf{P}}$. The variable case also needs $p_{\sim,\mathsf{P}}$ as well as $f_{\sim}$.

For $p_{\approx,\lambda}$ we discriminate on the derivation $\mathcal{C} : [\gamma \vdash T :_{\lambda} *]$ to discharge the impossible axiom and function type case. The application and abstraction case are both subdivided, based on the derivation of the relatedness of the involved dependent function types. For applications this of course needs a call to k ($\lambda 2$ propagation) and both also rely on $p_{\sim,\lambda}$ to handle type-level subderivations and $i_{\sim}$ (injectivity) to equate diverging PLC types. ∎

Corollary 6.5.11 There are total functions $\widehat{p_{\approx,\mathsf{P}}}$ and $\widehat{p_{\approx,\lambda}}$, which form equivalences with $p_{\approx,\mathsf{P}}$ and respectively $p_{\approx,\lambda}$. That is we have

$$\widehat{p_{\approx,\mathsf{P}}} : \forall\gamma{:}\overline{\mathsf{S}_{\sim,\lambda}}.\ [\gamma \vdash \mathsf{tmex}_{\lambda}\, M\, A] \implies [\gamma \vdash M :_{\mathsf{P}} A]$$
$$\widehat{p_{\approx,\lambda}} : \forall\gamma{:}\overline{\mathsf{S}_{\sim,\lambda}}.\ [\gamma \vdash \mathsf{tmex}_{\mathsf{P}}\, S\, T] \implies [\gamma \vdash \mathsf{tmty}_{\lambda}\, S\, T]$$

where the LF type $\mathsf{tmty}_{\lambda}\, S\, T$ simply forms the conjunct of the two $\lambda 2$ typing derivations.

$$\frac{S :_{\lambda} T \qquad T :_{\lambda} *}{\mathsf{tmty}_{\lambda}\, S\, T}$$

Proof Analogue to Corollary 6.5.9.

The construction of $\widehat{p_{\approx,\mathsf{P}}}$ needs $p_{\approx,\lambda}$ as well as $i_{\sim}$ and $i_{\approx}$.

The construction of $\widehat{p_{\approx,\lambda}}$ needs $p_{\approx,\mathsf{P}}$ as well as $f_{\sim}$ and $f_{\approx}$. ∎

Note that we do not have context existence statements like we had for our Coq and Abella proofs. This is similar to the discussion about PLC type formation and again due to the fact that in Beluga contexts are first-class rather than encoded structures. The context existence statements deal, among other aspects, with issues of malformed contexts, which are simply non-existent in Beluga. Thus with our four preservation results, as well as their inverse statements, in so far as they are applicable in the current setting, we have completed the Beluga equivalence proof.

The most interesting part of the Beluga development appears to be the particularly rich structure and interdependencies of the various schemas. We used several schemas that were not canonical and it is not clear to us if the same results could be obtained if we were to restrict ourselves to only using canonical schemas. The reason why this is interesting and somewhat surprising is that it runs contrary to a common

belief in the LF community that canonical schemas should be sufficient for all (or at least most) purposes. Consider for example [BC14], where this belief appears to be the underlying motivation for the development of an Abella plugin for schema inference. The described techniques would likely fail to obtain suitable invariants for our purposes. Note, however, that our results merely indicate that non-canonical schemas are useful, not that they are necessary

The schemas we have introduced can be arranged in a hierarchy (Figure 6.8), where we recall that a bar indicates a non-canonical schema. The solid lines denote a weakening/strengthening relationship between contexts of the connected schemas, where the less informative schema is placed higher in the figure. While weakening can be applied indiscriminately, strengthening of course has to respect subordination constraints among judgements under these contexts. The relationship of S_λ and $\overline{\mathsf{S}_\lambda}$ is not weakening/strengthening in the strict sense. It is better described as a refinement, though the vertical arrangement with respect to informativeness still applies.

6.6 Remarks on Adequacy

In the context of formalised mathematics we are at some point faced with the question, whether our representation of mathematical objects is **adequate**. That is, we have to argue why our formal definitions correspond to our intuitive understanding of the mathematical objects under consideration. Note though, that **adequacy** is a metatheoretical property about mechanisation frameworks, not something that is established within a given system.

For the de Bruijn setup we have used in Coq, the question does not really arise, as it is a well-understood first-order encoding. The representation of the syntax is comparable to any other inductive datatype, like natural numbers or lists. It is our understanding that a de Bruijn encoding is the canonical implementation of the *Barendregt convention*. The fact that all three proof assistants used here are internally implemented using de Bruijn supports this belief.

The situation is quite different for our two HOAS encodings, as they borrow their function types and instantiation mechanisms directly from their host environment. When HOAS was first introduced it was not at all clear that this would yield sensible syntactic structures. Thus since at least the 1990s a lot of techniques were developed to argue that such definitions are faithful (see [Pfe97, HL07]). Take for example [AHMP92], where the authors prove the adequacy of an encoding of ULC as an LF type using HOAS. Since our HOAS language definitions are close to the ones for which adequacy has been established, we deem them reasonably trustworthy.

When it comes to the HOAS type systems and proofs about them, the situation becomes less clear, as both Abella and Beluga go beyond basic λ-tree syntax and exploit subordination to justify the inductiveness of certain proofs. While we do not have adequacy proofs for these systems and our encodings, we can at least resort to the following line of reasoning.

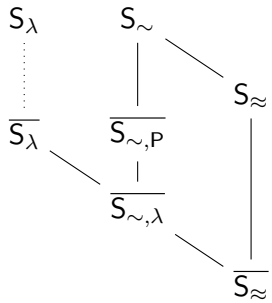


Figure 6.8: Hierarchy of context schemas.

First, we have proven that in each case two variants of intuitively the same mathematical system do in fact behave the same and the encodings also admit all the expected properties. Second, we were able to replay the same overall proof structure that worked for the de Bruijn approach on the HOAS encodings. Taken together, this stability within and across proof systems allows us to assume the adequacy of representations until evidence to the contrary is provided.

6.7 Discussion

Now that we have completed both HOAS variants of our equivalence proof it is worth to consider a few things. We defer a full comparison of our developments to Section 7.1.

The most crucial point to make is probably the fact that the two developments in this chapter follow two different philosophies with respect to how contextual information is organised in a multi-system setting. For Abella we use separate contexts for the two systems as well as the correspondence relation and then tie them together with a special predicate. In terms of data structures, this resembles a *pair of lists* (we disregard arity to some degree in this analogy). In the Beluga development we instead work with something that is much closer to a *list of pairs*. That is, we have a single contextual structure where each record carries information for all involved type families. Now the data structures of our analogy are closely related since they come with structure-preserving translation functions (zip and unzip, respectively). Depending on the application context, one or the other may, however, be more suitable. This reasoning carries over to our proof settings to some extent. While both systems do not force the exact representation of contexts, we found that each system exhibits a natural preference for the respective choice we have made here.

We also want to point out that both the Abella and the Beluga proof work on a stripped-down version of the PTS $\lambda 2$ where we have removed the conversion rule. We have justified this by the fact that all $\lambda 2$ types are normal, which we formally established for the Coq development in Fact 5.2.4. The main advantage of this decision was that we did not have to define reduction and its associated properties. Recall from Section 3.3 that, while being a standard result, significant effort went into a formal proof of confluence. Note, however, that a treatment of reduction and confluence is

certainly possible in both Abella and Beluga, which both ship with confluence proofs for STLC in their example sections. We believe that our survey would not have gained a lot from simply scaling these examples to the PTS setting.

For similar reasons we also did not consider the STLC variant of the proof in Abella and Beluga. Recall our observation at the end of Chapter 5 that the difference in proof complexity between STLC and System F is surprisingly small. Thus we went straight to System F for our HOAS developments.

We also still have to settle one subtle point regarding the exact shape of our correspondence relations. Recall from Section 6.2 the following correspondence rule for PLC arrow types (left) and an alternative that makes the vacuity of the body of the dependent function type on the $\lambda 2$ side more explicit (right).

$$\frac{A \sim S \qquad \mathbf{\Pi} x.\ B \sim T\langle x\rangle}{A \to B \sim \Pi S.\, T}\ \mathsf{R}^{\to}_{\sim} \qquad\qquad \frac{A \sim S \qquad B \sim T}{A \to B \sim \Pi S.\, (\lambda x \Rightarrow T)}\ \hat{\mathsf{R}}^{\to}_{\sim}$$

We favoured the rule $\mathsf{R}^{\to}_{\sim}$ because it fits more closely to the PTS type system of $\lambda 2$. Now why is that? To get an idea, let us put the two involved typing rules side by side. Here we use the variants of the Abella proof, but the argument mostly applies to the Beluga case as well.

$$\frac{A\ \mathbf{ty} \qquad B\ \mathbf{ty}}{(A \to B)\ \mathbf{ty}}\ \mathsf{I}^{\to}_{\mathsf{P}} \qquad\qquad \frac{\begin{array}{c} S :_\lambda T \qquad \mathcal{U}\ T \\ \mathbf{\Pi} x.\ x :_\lambda S \blacktriangleright U\langle x\rangle :_\lambda * \end{array}}{\Pi S.\, U :_\lambda *}\ \mathsf{T}^{\Pi}_{\lambda}$$

By now we know that the rule $\mathsf{T}^{\Pi}_{\lambda}$ captures two cases, so let us focus on the case with $T = *$, which simplifies to the following instance.

$$\frac{S :_\lambda * \qquad \mathbf{\Pi} x.\ x :_\lambda S \blacktriangleright U\langle x\rangle :_\lambda *}{\Pi S.\, U :_\lambda *}$$

At this point it should be apparent from just looking at the rules that $\mathsf{R}^{\to}_{\sim}$ is the more suitable candidate.

The choice of rules has, however, more than just aesthetic consequences. Recall one of our core results, namely that PLC typing is preserved under $\approx$ (Theorem 6.4.42). The statement involves several inductive structures: typings on both the PLC and the $\lambda 2$ side, as well as relatedness statements at the term and the type level. The proof itself proceeds by induction on the PLC typing statement. Of particular interest is the case for abstractions $\lambda B_1.\, N : B_1 \to B_2$, where we have to construct corresponding PTS terms T and S which relate, respectively, to the given PLC term and type. The construction of the abstraction T is relatively straightforward, but the dependent function type S is more involved. The inductive hypothesis gives us the following premise: $\{L_{\approx}, \mathsf{n}_1 \approx \mathsf{n}_2 \vdash B_2 \sim S_2\langle \mathsf{n}_2\rangle\}$. The nominals are terms, and on the PLC side subordination came to our help by *not* raising B_2 over n_1. On the $\lambda 2$ side we were not so lucky (due to single-sortedness) and now have a $S_2 : \mathsf{Tm}_\lambda \to \mathsf{Tm}_\lambda$ raised over

$\mathsf{n}_2 : \mathsf{Tm}_\lambda$. Subordination allows us to simplify the hypothesis to $\{L_\approx \vdash B_2 \sim S_2\langle \mathsf{n}_2 \rangle\}$ but we are still stuck with the potential dependence of S_2 on n_2. At this point, rule $\mathsf{R}^{\rightarrow}_{\sim}$ quickly gets us to the desired goal. With rule $\hat{\mathsf{R}}^{\rightarrow}_{\sim}$ we are instead faced with the obligation to show that there is some S_2' such that $S_2 = (\lambda x \Rightarrow S_2')$, that is, we have to explicitly show the vacuity of S_2. While this is true, and therefore likely also provable, it poses a significant and unnecessary overhead to our proof effort. More generally speaking, the characteristic of $\lambda 2$ that is causing issues here is the fact that a PTS can have variables which are syntactically in scope, but impossible to appear in certain expressions for semantic reasons.

Interestingly, the problem we just discussed is not exclusive to the HOAS setting but originates from the internal structure of the uniform and single-sorted PTS syntax. Recall Section 3.4 where we introduced the predicate all as a means to precisely but indirectly express that certain de Bruijn indices do *not* occur freely in a given expression. We used it heavily in Section 3.8 to establish a strengthening result. There we have drawn the conclusion that it is usually a bad idea to directly talk about non-occurrence of variables and vacuity of abstractions. Here we have seen that those lessons apply to the HOAS setting as well.

The rationale behind the preceding arguments can be taken as a more general lesson: the design of a structure should follow its intended use. While the correspondence relation exhibits a certain beauty in its own right, this was not the driving force behind its chosen formulation. The most beautiful bridge is pointless, when it does not reach the shores of the river it is supposed to span. In our case the *shores* were the two type systems we considered as fixed and the relation is the *bridge* that has to accommodate the various correspondence proofs.

7 Remarks on the Formalisation

In this chapter we want to give some remarks that are of a more technical nature. We begin with a qualitative comparison of our three developments and also include a short history of how the results that are presented in this work where originally reached. We then give a quantitative overview of the proof efforts and conclude with a few remarks on the employed theorem provers.

7.1 De Bruijn vs HOAS

Let us step back and briefly compare our three developments. The core topic of all three developments was the treatment of contextual information, that is invariants attached to reasoning contexts for various forms of object-level derivations.

In our de Bruijn development in Coq we had to manually implement the full context machinery, which imposed a significant overhead on the equivalence proof. This involved the explicit representation of contexts as lists, paired with lookup mechanisms that respect dependencies, as well as involved proofs of weakening and substitutivity results. For the various substitutivity-like results we found that a proof pattern inspired by **context morphism lemmas** was useful in defining the required inductive invariants and subsequently establishing the corresponding results. The latter allowed us to structure the required overhead in a meaningful way, but it still constitutes a sizeable portion of the development. Note further that this effort sits on top of the structures that were generated by the Autosubst library, like the instantiation operations and the associated equational theory.

The support for HOAS syntax and two-level logic reasoning in Abella significantly improved the situation for our second formalisation. The substitutivity properties come essentially for free, including context weakening, contraction and exchange. The concept of subordination also turns out to be a powerful tool. In addition, nominal constants, ∇-quantification and the principle of binder mobility lead to an elegant treatment of object-level variables and binders. The point where Abella becomes cumbersome are the actual entries in a given context. These have to be controlled with well-crafted context predicates and large collections of lookup and inversion lemmas since Abella's logical embedding a priori admits backchaining at every position in the proof. Note that the context predicates are tied very closely to the context morphism definitions that where used in the Coq proof. Take for example the single predicate that was used in Abella as a premise for all four preservation results (Definition 6.4.31). It encodes exactly the information that was encapsulated in the four different invariants used for the same four preservation proofs in Coq (see Figure 5.3). We further recall

that in the Coq development we established custom PTS induction principles which exploit semantic type formation information. We were not able to directly express these in Abella's reasoning logic $\mathcal{G}$, though it might be possible to capture them with an inductive predicate instead. We did not investigate this avenue further and instead inlined the structure into the affected preservation proofs.

Finally, in Beluga we have all the theoretical benefits of the Abella setting plus very fine-grained control over contextual information. Its contextual objects are similar to Abella's logical embedding, though there are notable differences. First of all, since contextual objects do not distinguish between object-level types and predicates, induction is possible over both, where Abella only allowed induction over the latter. Second, in Abella, the embedding was just a certain inductive predicate with special syntax and tactics that expose its properties, like cut and substitutivity. Meanwhile in Beluga, contextual objects are an integral part of the metatheory, up to the point that the type formation judgement of PLC becomes redundant. Beluga also gains a lot of power from the fact that context schemas, which replace the inductive context predicates of Abella, are baked into the system. This allows Beluga's type checker to validate the well-formedness of contextual objects. We thus have native and dedicated support for our inductive invariants.

Over the course of the three developments it became apparent that the contextual structure of our correspondence results is quite rich. In Coq this manifests as a considerable collection of CML-like invariants, in Abella it shows up in the form of non-trivial predicates over multiple specification-level context lists, and in Beluga we had to come up with context schemas that are not canonical. What all of these have in common are two things. Firstly, semantic information about the problem domain has to be injected into a structure that is often considered to be of a purely syntactical nature. The three developments differ in how tightly the syntactic and the semantic aspects of the contextual setup are tied together. Secondly, the contexts for our purpose exhibit a two-fold dependency structure. On the one hand, syntactic entities like variables have to be tied to their semantic aspects, and on the other we have the well known dependency of later items in a context depending on earlier entries. The latter was apparent in all three developments, while the former is most explicit in Beluga's context schemas and first-class contexts.

It is also interesting to note that working with all three systems led to some cross-pollination. This can be seen best, when we arrange the various developments in their order of appearance.

The first solution, which is *not* presented in this thesis, was executed in Coq for System F and the PTS $\lambda 2$ without conversion. It already relied on the CML techniques, but the mapping of syntactical expressions was facilitated by translation functions. This led to rather involved cancellation properties for round-trip translations and sever complications in the area of context validity. The resulting proof was hard to find and harder to explain. We direct the interested reader to [KTS17] for a presentation of this result.

Due to this we turned our attention to Abella, which was promoted as an ideal system for reasoning about programming language metatheory. Transporting our de Bruijn language definitions and type systems to the HOAS setting was trivial, but the fact that Abella made the conscious choice to not support the definition of functions on our language representations forced us to completely rethink our approach. The result of this was the notion of an inductively defined correspondence relation that precisely connects the meaningful parts of the involved languages. At this point the four properties (injectivity, functionality, totality and preservation in both directions) began to take shape and we also devised how they would interact to obtain the desired equivalence result. While the role of injectivity and functionality was clear from the start we first formulated the totality and preservation results each in two forms. One where the inner involved contexts were existentially quantified and one where universal quantification was used instead. In the developments in this thesis only the latter survived, but more on that later. While the resulting proof in Abella was already much cleaner we were somewhat unhappy with the copious amounts of required inversion Lemmas to discard spurious backchaining cases.

This is what led us to Beluga, where context schemas and first-class contexts promised to provide much finer control over what is placed in and extracted from a context. Transporting the various definitions from λProlog syntax in Abella to the LF types used in Beluga was again trivial. But the need to now construct explicit proof terms forced us to carefully think about our proofs where before we had worked with rather indiscriminate automation techniques. As one might expect, this uncovered a number of unnecessary detours.

At this point we returned to Coq to exploit our newly gained insights. We replaced the old proof with Abella's relational approach and improved the various proof scripts since we now knew what the underlying proof terms should look like. This is when the convoluted nature of the preservation results with the existentially quantified contexts became apparent. They were consequentially removed and replaced with the stand-alone context existence results for valid contexts. A second motivation at this point was the discovery that the equivalence results for closed judgements were only of marginal use, while the variants for open judgements relied on the ability to construct relational correspondence information.

The final steps happened mostly in parallel. We transported the improvements developed in Coq to Abella and Beluga (which allowed us to drop several redundant lemmas) to arrive at the respective variants in their present form. We also finally scaled the Coq development to a full, stand-alone PTS representation, including the conversion rule, in order to recycle common parts for our study of the STLC case. As pointed out earlier, we found that the restriction to STLC did not simplify the correspondence proof as much as we would have expected. Which is why we did not transport the STLC proofs to Abella and Beluga.

We hope that this brief history highlights, how much can be gained from tackling the same problem in a variety of formalisms.

7.2 Comparison of Effort

We give an overview of the concrete formalisation effort. A breakdown of line numbers is shown in Table 7.1 (excluding blank lines and comments). Note that the numbers do not allow for a fair, quantitative comparison, due to stylistic aspects like the formatting and layout of the code as well as more inherent aspects like the usage of tactics or proof terms. Thus they are mostly meant to illustrate the approximate scope of the present work. We briefly comment on each of the counts.

For the Coq development, we have omitted a small utility library from the count, as it is only partially applicable to our present development. We also have not included the Autosubst library, which takes care of some of the heavy lifting for the de Bruijn definitions. Also note that the PTS development contains several results that are not strictly necessary to obtain the system correspondence. Finally, the two correspondence proofs include the instantiations of the PTS framework to $\lambda_{\to}$ and respectively $\lambda 2$.

For the Abella development it is crucial to remember that we work with a simplified PTS variant without a conversion rule. Additionally, the PTS $\lambda 2$ is defined directly, rather than as an instance of the general framework. Both aspects simplify the proof obligations and account for some portion of the shorter development. On the other hand, the absence of certain higher-order quantification abilities in its reasoning logic $\mathcal{G}$ forces us to duplicate several definitions and proofs. Note that we also do not demonstrate the transfer of properties as we did in the de Bruijn setting, which further obfuscates the comparison. We have separated the definitions in $\mathcal{G}$ into those that constrain specification level context lists into meaningful shapes (since they play a comparable role to the schemas of Beluga), from those used to recognise nominals, as well as other definitions like those that close certain types and trigger the computation of the subordination order.

While the Beluga development is structurally close to the Abella version, given that both are based on almost identical HOAS definitions, it is even harder to quantitatively compare to the other two developments. The reason for this is the fact that while Coq and Abella facilitate a tactic language which allows the implicit construction of proof terms, Beluga requires the explicit formulation of such proof terms. In most cases, a tactic-based proof script is considerably shorter than the proof term it generates. In this light it is quite surprising that Beluga still comes out ahead in terms of plain line counts. The *LF Helper Definitions* refer to equality definitions for the three syntactic sorts, while the *CMTT Logic Definitions* reflect the fact that Beluga is missing various native logical connectives that we had to manually encode.

7.3 Technical Comparison of Provers

At this point we would like give a brief account of our experience of working with three different provers, with a focus on usability. We also observe to what extent certain system aspects affected our proof development in particular. We should point out that we have been using Coq extensively, not only for the results presented in this work,

						LoC
Coq (de Bruijn, tactics) – count according to `coqwc` tool –						
			spec		*total*	
Abstract Reduction Systems			69		167	
Pure Type Systems			237		614	
2-sorted STLC			69		126	
2-sorted PLC			77		138	
De Bruijn Variable Relation			45		98	
STLC $\approx \lambda_{\rightarrow}$			142		451	
PLC $\approx \lambda 2$			152		550	
Σ			791		2144	**2144**
Abella (HOAS, tactics) – manual count –						
Specification Layer (λProlog)		*Reasoning Layer (𝒢)*				
PLC	17	Context Predicate Defs.	15	***Proofs***		
$\lambda 2$	16	Nominal Predicate Defs.	5	*spec*	77	
Correspondence Relation	14	Other Defs. / Instructions	3	*script*	303	
	47		23		380	**450**
Beluga (HOAS, proof terms) – manual count –						
Specification Layer (LF)		*Reasoning Layer (CMTT)*				
PLC	20	Context Schema Defs.	13	***Proofs***		
$\lambda 2$	25	Logic Defs.	10	*fun. heads*	26	
Correspondence Relation	23	Other Defs.	3	*fun. bodies*	211	
Helper Defs.	6					
	74		26		237	**337**

Table 7.1: Comparison of line counts.

while Abella and Beluga are more recent additions to our set of tools. The following evaluation is therefore clearly subjective and potentially biased, but we believe that our experiences may help future generations in choosing the correct tool for their own formalisation developments.

Let us start with Coq, which is a mature system and based on the reasonably well understood Calculus of (co)inductive Constructions. The trusted kernel is small and consistency issues are rare and quickly fixed, though they do occur from time to time. The system has a large user base, an extensive standard library (though its quality varies) and many useful tools. The integration into Emacs [EMC] using the ProofGeneral plugin [PRG] provides for an advanced development environment. The ability to construct proofs using tactics as well as the ability to extend the tactic language are powerful reasoning tools. On the downside, since Coq is a general-purpose theorem prover, we have to rely on libraries for domain specific features or manually build them. This can lead to comparatively large developments, in contrast to systems with a narrower application focus. Coq is also the oldest of the three systems used.

Let us next turn to Abella, which is a system primarily geared towards proof search and relational specifications. It is considerably younger than Coq and appeared in 2008 [Gac08], in the wake of the PoplMark challenge [ABF+05]. It is structured as a two-level logic system, and both the reasoning layer and the specification layer are interesting choices. The latter is verbatim λProlog which enables us to use a single system description for both reasoning about it and animating it. The reasoning layer $\mathcal{G}$ is a double-edged sword. On the one side, generic quantification, nominal constants and inductive predicates are powerful tools for metatheoretical reasoning, but especially the former two are sometimes hard to grasp. Take for example Lemma 6.4.8, which is used to discharge proof branches where an assumption states that x occurs in L while simultaneously being fresh for L. In addition, the inability to quantify over propositions, and the mostly first-order nature of $\mathcal{G}$ lead to significant proof script duplication in various places. From a usability point it is also worth mentioning that a ProofGeneral fork for Abella exists, but that, at the point of writing, has not been merged into the upstream development of ProofGeneral. As a consequence, we had to maintain two mostly identical Emacs setups to be able to work on both Coq and Abella code. We also dearly missed Coq-style bullets in proof scripts. A particular problem of Abella is that all hypotheses are simply referred to as H_n with n incrementing as needed and that additionally most tactic invocations are a case analysis on some hypothesis H_i. As a consequence proof scripts are tremendously unstable during development, when definitions are changed, since many tactic invocations may still work even though the script has gone wrong at a much earlier point. Backtracking from the point of breakage to the actual error in the proof tree was time consuming and tedious. Bullets would have helped to contain such problems and simplified the proof development noticeably.

Finally we have Beluga, which appeared in 2010 [PD10] and is thus the youngest of the three. It is based on the Twelf logical framework [PS99]. Working in Beluga feels much closer to dependently typed programming, since all proofs have to be given

as explicit proof terms. Its theoretical foundation is contextual modal type theory, which is rather unusual in that contexts are internalised as first-class entities. A basic integration with Emacs exists, though it does not go much beyond syntax highlighting and the ability to compile the program and view the result in a separate buffer. One useful feature is the support for incomplete proof terms with explicit holes, for which the compiler reports the respective typing contexts. These contexts are also presented to the user. It is, unfortunately, not possible to query for such contexts at arbitrary points of code (apart from replacing subterms with holes and recompiling). If it were provided, this mode of operation could simplify the analysis of existing code and serve as a substitute for the ability to "step" through proofs as in the tactic based systems Coq and Abella. Lastly, a word of caution. Beluga is still under active development, and there appear to be bugs which affect both soundness and completeness of the system. At the time of writing, there appear to be several open issues connected to the coverage checker, which forms an integral part of ensuring that recursive functions are total and terminating. These properties are necessary to accept such functions as proofs. The known issues include cases where obviously total functions are rejected, as well as those where non-total functions are accepted as total. We started our development on the latest stable release (0.8.2 from July 2015), but encountered one of the latter issues. Thereafter, we switched to the development version at a particular git commit[1] where our particular issue was fixed.

[1] The exact Beluga commit for our development was: `e7d538a4`. We have a locally built Beluga binary from this commit, which cleanly accepts the accompanying Beluga script. At the time of writing, this binary does not appear to be buildable any longer. We have however managed to find a functioning build environment for the current master branch of the Beluga project (commit: `285dd31c`). The resulting Beluga binary again cleanly checks our proof script. For details, see the `README.md` packaged with the Beluga proof script on the project web page.

8 Conclusion

This brings us to the end of our exposition. So let us briefly recap the main results, consider the lessons learned and also ponder on the value that we hope our work adds to the field of formalised metatheory. Then, before we finally conclude, we will reflect on a number of directions in which this work could be extended.

8.1 Summary of Results

We have studied in detail the connection between two variants of the well-known formalism System F, also known as the polymorphic λ-calculus, and formally established that these variants are co-typeable, both in Coq and using a first-order de Bruijn representation of the involved systems as well as in Abella and Beluga where we employed higher-order abstract syntax (HOAS). By co-typeable we mean, that each derivable typing judgement in one of the variants has a unique derivable counterpart in the respective other variant. The value of this result is manyfold. First of all, we have confirmed what was often tacitly assumed, namely that the two variants are really, at least with respect to typeability, the same system. Secondly, and as an immediate consequence, we are no longer limited to reason within a single variant to determine, whether a given statement is derivable. We can just as well translate the statement along the established correspondence relation and answer the question of derivability in the second variant. The outcome will hold for both variants by virtue of the underlying reduction. Moreover, assume that we have some non-trivial properties proven for one variant. Then, instead of starting from scratch, we can transfer such results along the correspondence directly to the other variant, given that the properties in question are compatible with the correspondence. We have demonstrated this latter use with propagation (or type-correctness) and β-substitutivity. The developed structures and identified principles are sufficiently general to apply to other questions of system equivalence, but more on that later. Finally, with our equivalence proof we have produced a case study that exercises a variety of aspects of frameworks for metatheoretical reasoning and as such presents itself as an interesting benchmark. We have used our results in this way to compare the Coq/de Bruijn approach to the HOAS solutions in Abella and Beluga.

The two variants that we considered are reasonably canonical representatives of the multitude of existing formulations of System F. On the one hand we had a traditional two-sorted presentation where terms and types are syntactically separated, while the other was the pure type system (PTS) $\lambda 2$. The major complications were (a) the alignment of the two-sorted syntactic language of System F with the uniform,

single-sorted PTS syntax in general, (b) the alignment of binding disciplines for local variables in particular and (c) the handling of complex contextual information when reasoning about open expressions.

Our first formalisation was executed in the proof assistant Coq, where we used de Bruijn syntax to encode our object languages and then enlisted the Autosubst library to handle the automatic definition of instantiation operations. The library also provided us with an equational theory and decision procedure for substitution expressions, which was a crucial component in numerous lemmas and theorems. We made the decision to first consider a simplified equivalence problem and take polymorphic/universal types out of the equation. This left us with the simply typed λ-calculus on the two-sorted side as well as the PTS $\lambda_{\to}$. Since we now had to deal with two PTSs, and moreover two instances that are both members of Barendregt's λ-cube, we resolved to develop a general and principled PTS library to avoid duplication of work. One of the key challenges of the latter was the adaptation of a strengthening proof from the literature to the formal context of our work. The difficulty arose from the fact that the literature version was using a named representation and single-point substitutions, while we were working with nameless de Bruijn syntax and parallel substitutions. Quite surprisingly, the shift from the named to the de Bruijn setting was in this particular case rather intricate and involved concepts like partial context renaming and the uniqueness of terms modulo partial renaming. This stands in contrast to the majority of other metatheoretical reasoning scenarios where the adaptation and necessary adjustments are mostly straightforward.

The equivalence proof for the simply typed setting starts with the notion of semantically related free variables, a concept that permeates all subsequent proofs (including to some extent the HOAS developments as well) and the first instance of non-standard context information. Related free variables were then extended to related types and related terms, as captured by inductively defined correspondence relations. To provide a foundation for our desired reduction it was then crucial to establish a set of four key properties of the defined relations, namely injectivity and functionality, as well as totality and preservation of judgements on the well-typed fragments (proven in conjunction) going from one system to the other and vice versa. The resulting inductive structure can be seen as the careful merging of the two involved type systems, such that semantic alignment is ensured. A crucial technical device throughout the two preservation proofs was the notion of context morphism lemmas, which were originally devised as a principled way to prove weakening and β-substitutivity for a given type system. The key insight is to maintain, for a given judgement, an inductive proof invariant over three quantities, namely an original context, a new context and a substitution that connects the two. We managed to generalise this principle, such that not only two judgements from the same system, but also judgements from distinct systems, with possibly distinct syntactic languages, can be connected. We moreover demonstrated that the principle does not only apply to typing judgements but other judgemental structures as well.

Scaling the whole development to System F and $\lambda 2$ was then technical, though surprisingly straightforward. In hindsight, this does not come as much of a surprise, since most of the key complications already arose in the simply typed setting. The additional typing rules that had to be aligned for the full System F equivalence did expand the set of possible inference rule pairings somewhat, so that more possible, as well as impossible, cases needed to be dealt with across a number of proofs. No extra conceptual insights were necessary, though.

Based on the observation that the simply typed and the polymorphic case were largely similar in terms of complexity, we made the decision to only consider the polymorphic case for our exploration of HOAS solutions to our problem at hand. Additionally, we simplified the PTS $\lambda 2$ in our HOAS setups by removing the conversion rule. This was justified since all System F types in derivable judgements are normal by construction, rendering conversion obsolete here.

Our first candidate framework for a HOAS solution was the Abella theorem prover, a two-level system inspired by logic programming and equipped with dedicated features for reasoning about object-language variables, like nominals and generic quantification (∇). Abella is being advertised as the ideal framework for formalised metatheory, and our experience with the system supports this claim to a reasonable extent. The only conceptual flaw we found with the system design is the extensive number of required inversion lemmas to deal with spurious applications of the backchaining rule of the underlying theory. On the plus side, Abella's dedication to relational reasoning was a major guidepost towards our eventual inductive correspondence relation, and also forced us to take a good look at the various forms of contextual information involved in our proofs. The fact that Abella is a much younger system than Coq becomes apparent in the area of basic proof engineering, where we found that the Abella scripts were less stable than their Coq counterparts during the proof development, for example due to the absence of structuring elements like proof script bullets.

The second HOAS candidate, Beluga, is not a proof assistant as such (e.g. there are no proof tactics) but a dependently typed programming language based on contextual modal type theory. The latter, however, did position it as an ideal framework for our equivalence proofs, since it provides very fine-grained control over the handling of contexts, which, in Beluga, are first-class structures. Our particular setup was a good test of this aspect, since we had to work with contexts and context schemas that were not initially envisioned to occur by the designers of the language. Despite this, the language appears to be robust enough to handle them regardless. One of the major benefits we take away from our foray into Beluga is that writing explicit proof terms is both challenging as well as enlightening. Challenging, since we were often faced with the problem that we could not really verify partial solutions. The mental distance between states which are marked off as correct by the system is considerably greater than in frameworks, where tactics enable an incremental exploration of the proof state. The work was enlightening for essentially the same reason. On the one hand we gained first hand experience of the actual complexity of the problem, and on

the other we were able to cut away unnecessary detours that careless guessing with tactics had introduced in the other two systems.

So what have we learned along our journey? Mainly three things. Firstly, local variable binding is tough and so far there does not appear to be a silver bullet solution. It is not impossible to handle, but it is easy get bogged down in irrelevant detail and requires a lot of care to get it sufficiently right. This involves, among other things the right choice of the level of abstraction, which brings us to our second point. Both the first-order de Bruijn approach as well as HOAS provide abstractions for reasoning about syntax with variable binding, and they sit at noticeably different levels. Here HOAS appears to be much closer to the level at which we intuitively understand variables, while de Bruijn is, in sense, closer to the metal, i.e. it exposes implementation details. The latter may actually be a good thing, depending on what one wants to achieve. One point where the difference in the level of abstraction becomes blatantly apparent is the length of the respective proof scripts, where the two HOAS solutions turn out to be noticeably shorter. The difference is more than what can be attributed to the way in which we simplified the problem (e.g. no conversion). There are of course more than the two abstraction levels considered here, with the locally nameless representation sitting prominently in between, and they warrant further investigation, as set out below. Lastly, to understand a problem, it is instructive to formalise it in a proof system to uncover subtle complications like the ones encountered in our PTS strengthening proof. Moreover, to **really** understand a problem it is tremendously helpful to formalise it more than once in different systems, since it allows us to separate inherent from incidental complexity.

8.2 Open Questions and Challenges

While we are pleased with the point to which we have pushed the present project, there are of course several potential directions for further investigation and we want to consider a few of them.

8.2.1 Relating Reduction

Throughout this work we have exclusively discussed co-typeability of our two variants, which allowed us to demonstrate the transfer of certain metatheoretic properties. What would it take to also enable the transfer of properties related to the reduction behaviour of our two variants? Here subject reduction (also known as preservation) as well as normalisation immediately come to mind. What we are looking for is co-reducibility, that is the property, that whenever a term in one variant can take a step, then there is a unique corresponding term in the other variant that can mirror this step. In essence we are looking for a correspondence relation that constitutes a bisimulation. It is clear that the required relation would need to be restricted to the reduction behaviour of well-typed terms, since without typing it is trivial to construct a reducible PTS expression that has no matching counterpart on the two-sorted side,

e.g. by simply applying an abstraction to an argument of the wrong syntactic class. In two-sorted System F, such an expression cannot be constructed, while in $\lambda 2$ it can; it simply won't be typeable. It seems likely that the need to track some level of typing information will lead to a reduction correspondence relation that has subject reduction baked in.

8.2.2 Scaling to F_ω

We have considered two corners of the λ-cube, $\lambda_\to$ and $\lambda 2$. The obvious next step would be $\lambda\omega$, also known as F_ω, which can still be expressed as a stratified system. This step immediately introduces two challenges.

Firstly, the universe of kinds $\Box$ is no longer degenerate. It now contains kinds other then $*$, like for example unary type constructors of kind $* \to *$. Our proofs for $\lambda_\to$ and $\lambda 2$ heavily rely on the degeneracy result, so major adjustments to the proof structure appear to be in order.

Secondly, the existence of non-trivial kinds in $\Box$ leads to the existence of β-redices in the universe of types $*$. This affects our universe normality result (Lemma 3.7.8), where we will be unable to satisfy the premise. It should be noted though that the universe of kinds $\Box$ is still only inhabited by normal terms. Lemma 3.7.8 states that *all* universes only contain normal terms, which for $\lambda\omega$ clearly is not the case. Since the normality of kinds will be relevant throughout the correspondence proof, we likely need a replacement for Lemma 3.7.8, namely one with a premise that is sufficient to yield the normality of a *particular* universe. Further investigation in this area appears to be necessary.

8.2.3 Further Correspondences

We have seen a co-typeability result for two particular variants of System F (and STLC) and it is only natural to ask which other variants or systems could also be taken into consideration. Further such correspondence problems can loosely be grouped into two classes

The first class concerns scenarios where the two respective systems share the same syntactic language and only exhibit variations in their typing disciplines. We could for example take our PTS variant presented in Section 5.1 and connect it to the system where context validity is build into the typing judgement, like the type system used in Adams' formalisation [Ada04] and Barendregt's original exposition [Bar91]. On the other side, we could take our two-sorted variant and connect it to one where the type variable context is kept implicit. For cases like these, it appears likely that the full relational machinery developed in this thesis is not necessary. We do imagine, though, that context morphism style lemmas will still be useful.

The second class of problems is comparable to the concrete setting we have discussed in this thesis, namely the one where the syntactic languages differ. For these, the relational technique provides a feasible angle of approach. Recall, that we have exclusively formulated our systems in Church-style, that is with type/domain

annotations on abstractions. We could now, for example, try to relate our variant to a Curry-style formulation as in [KSW16], where such annotations are omitted. Alternatively, we could go back to Adams' development [Ada04], where the PTS syntax is well-scoped, i.e. it is represented by a $\mathbb{N}$-indexed family of types, and relate it to our version of the PTS formalism. Note that the handling of well-scoped syntax falls outside the scope of Autosubst 1. We would need to either switch to Autosubst 2, which does support well-scoped syntax, or we would need to find another framework or custom solution to handle the first-order de Bruijn syntax.

8.2.4 A Benchmark Problem for Contextual Information

One of the major milestones in the field of formal metatheory was the PoplMark challenge [ABF+05]. It covered the issue of formally representing and reasoning about syntactic systems that involve variable binding and served as a benchmark for a sizeable number of contenders.

It appears to be time to widen our scope and include another piece of the puzzle into the benchmarking process, namely the handling of contextual information. The need for such considerations was raised by the ORBI project [FMP15a] and a number of small context problems and associated solutions in various proof systems are presented in [FMP15b]. We would like to propose our present work as a more large-scale candidate to test a systems ability for contextual reasoning. As we have seen, our contexts are rather involved multi-dimensional dependency structures, which go beyond what is covered by the ORBI project so far. In Coq this was mostly implicitly hidden in the various inductive invariants while the higher-order solutions clearly showcased the underlying complexity. We did, in particular, witness various design and implementation issues and contrasted them throughout the text. Recall, for example, that in Abella, our contexts were akin to tuples of lists, while in Beluga we structured them as lists of blocks, and in each case the choice appeared more natural in the respective framework. Running these alternative solutions through the Twelf system [PS99] would certainly be informative.

There are in fact quite a number of further candidates for which we would love see implementations of our correspondence proofs to complement the comparison.

On the first-order side, the locally nameless approach of [ACP+08] immediately comes to mind. It is an encoding that distinguishes between free variables, which are named, and bound variables which are encoded as de Bruijn indices. So as an abstraction level, it can be placed somewhere between the pure de Bruijn approach and the HOAS layer. The process of descending underneath a binder and thereby transferring a new variable with a freshly chosen name into the context is referred to as *opening*. A major advantage of this approach is the fact that open expressions are not sensitive to context reorderings in the sense we have witnessed for a pure de Bruijn setup. On the down side, the necessary bookkeeping to keep track of the frequent opening and closing of terms, as well as the need to always be able to generate fresh names place a heavy burden on the proof development. In addition, during early

experiments, we got the impression that while the provided level of abstraction feels rather natural it does not appear to be sufficiently stable. We were repeatedly dropped to a level where we had to deal with the supposedly internal de Bruijn representation, albeit without suitable support. This was one of the reasons we eventually settled for the pure de Bruijn approach supported by Autosubst. Still, it would be interesting to see how an expert for the LN approach would cope with our presented challenge.

Another promising framework that could be tested against our case is Hybrid [FM12]. It is an attempt to enable HOAS reasoning in Coq along the lines we have seen in Abella. That is, a HOAS interface is presented to the user, and provided definitions are mapped onto an internal de Bruijn encoded λ-calculus with metatheoretic properties that carry over to the defined object languages. The design effectively constructs a deep embedding of an intensional function space into Coq's extensional landscape. While the concept appears promising it should be noted that one of the authors[1] raised concerns about a certain lack of features. There does, at the moment not appear to be a construction that would mirror Abella's ∇ or Beluga's contextual types. An attempt to tackle our challenge with Hybrid, even a failed one, would therefore likely provide valuable insights into which aspects of the framework would benefit most from further improvements.

A further candidate for investigation is the set of induction principles proposed in [UBN07] which employ nominal techniques to formally handle the underlying principles of Barendregt's variable convention. The basic idea of the nominal approach is to make α-equivalence renamings explicit through dedicated swapping operations. See [Pit03] for an introduction to nominal reasoning techniques.

Finally, we would love to see other theorem provers in the mix, like Lean [dMKA+15], Agda [Nor08] or Isabelle [WPN08]. For the latter, both nominal libraries [Urb08], as well as the original version of Hybrid [Fel10], present themselves as potentially suitable frameworks.

8.2.5 Autosubst 2

A major portion of the formalisation that accompanies this work was executed in Coq and had to deal with first-order de Bruijn syntax. The original Autosubst 1 library was a crucial component that made the whole project feasible in the first place. We did, however, come across a few aspects, where better tool support would have been desirable, and took the opportunity to let these findings guide our design and ongoing development of the second iteration of Autosubst.

The first interesting challenge was the two-sorted variant of System F, where Autosubst 1 implements the required instantiation mechanisms for the two variable sorts with so-called heterogeneous substitutions. And in contrast to the native and elegant parallel substitutions of the de Bruijn world, these heterogeneous constructions always felt somewhat ad hoc. Additionally, while they provably work for the special case of System F they do not scale to more complex syntactic systems. We therefore

[1] Amy P. Felty – received in private communication.

made the decision to prefer a single instantiation operation which, in the multi-sorted case, takes a vector of parallel substitutions, one for each relevant variable scope, and replaces all variables, across all scopes, simultaneously. Recall that we already notationally adopted vector instantiations in this text, while the underlying proof scripts still deal with the heterogeneous structures.

The second interesting observation was the fact that we were in essence dealing with a variety of so-called syntax traversals [ACMM17]. The key issue here is that our first-order syntax comes with a binding discipline that is not explicitly expressed in the underlying inductive types. The induction and recursion principles which are automatically associated with these syntax types are therefore also not aware of the intended binding behaviour. Thus many problems with binders and free variables throughout inductions over syntax can be traced back to the usage of unsuitable induction and recursion principles. Traversals attempt to answer this mismatch and provide correct, binder-aware, recursion principles. The prime example is the recursive definition of instantiation itself. Moreover, many of the context morphism lemma proofs also exhibit traversal structures. We employ these ideas in Autosubst 2 to provide a more principled generation of the vector instantiation operation for a large and now well-defined class of possible syntactic systems. In addition we experimented with ways to automatically generate not only the instantiation mechanisms and associated equational theory, but also basic substitutivity results for given judgemental structures (e.g. the type system).

A third avenue of improvements arose from our work with HOAS and the intent to benchmark multiple systems against each other. We asked ourselves, if it would be possible to free the Autosubst project from the confines of the Coq ecosystem. The result was a complete redesign of the framework, including the user workflow. In Autosubst 1 we would write down a sufficiently annotated inductive type in Coq and then delegate all the heavy lifting to Coq's ltac system and type class inference. In Autosubst 2 we now ask the user to express his syntactic system in a Twelf-like second-order HOAS language. A compiler and analyser then take this system specification, determine the dependencies among the syntactic sorts and construct an internal, prover-agnostic representation of the required structures and proofs. We then provide the ability to attach different backends which produce prover-specific source files and libraries that yield the required syntax infrastructure.

The decoupling of the system specification from the final, prover-specific realisation via an internal, abstract representation also allowed us to optionally generate well-scoped syntax. The choice of well-scopedness turns out to just be an implementation detail for the generated prover scripts.

The whole development of Autosubst 2 is captured in [KSS17, KSS18, SSK19]. It would be interesting to see how the switch to version 2 would affect the formalisations of this work.

Bibliography

[Abe08] Andreas Abel. Weak beta-theta-normalization and normalization by evaluation for System F. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2008.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[ABL] Abella. Available from `http://abella-prover.org/`.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACMM17] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Bertot and Vafeiadis [BV17], pages 195–207.

[ACP+08] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In Necula and Wadler [NW08], pages 3–15.

[Ada04] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.

[AHMP92] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

[AMP17] Andreas Abel, Alberto Momigliano, and Brigitte Pientka. POPLMARK reloaded. In Miculan and Rabe [MR17].

[AW10] Brian E. Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania Department of Computer and Information Science, June 2010.

[Bar84] Hendrik Pieter (Henk) Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, revised edition, 1984.

[Bar91] Hendrik Pieter (Henk) Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BC14] Olivier Savary Bélanger and Kaustuv Chaudhuri. Automatically deriving schematic theorems for dynamic contexts. In Amy P. Felty and Brigitte Pientka, editors, *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP '14, Vienna, Austria, July 17, 2014*, pages 9:1–9:8. ACM, 2014.

[BCG+14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[BD01] Martin W. Bunder and Wil Dekkers. Pure type systems with more liberal rules. *Journal of Symbolic Logic*, 66(4):1561–1580, 2001.

[BDS13] Hendrik Pieter (Henk) Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[BEL] Beluga. Available from `http://complogic.cs.mcgill.ca/beluga/`.

[Ber90a] Stefano Berardi. Girard normalization proof in LEGO. In Gérard P. Huet and Gordon David Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks, Antibes, May 1990*, pages 67–78, 1990.

[Ber90b] Stefano Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, Department of Computer Science, Carnegie Mellon University,

Pittsburgh, Pennsylvania (USA) and Dipartimento di Informatica, Torino University, Torino, Italy, 1990.

[BHS97] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. A notion of classical pure type system. *Electronic Notes in Theoretical Computer Science*, 6:4–59, 1997.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[Bra13] Max Bramer. *Logic Programming with Prolog.* Springer, 2013.

[BV17] Yves Bertot and Viktor Vafeiadis, editors. *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017.* ACM, 2017.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[COQ] The Coq Proof Assistant. Available from `http://coq.inria.fr/`.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 37–52. ACM, 1993.

[dB72] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[dMKA+15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Felty and Middeldorp [FM15], pages 378–388.

[EMC] GNU Emacs. Available from `https://www.gnu.org/software/emacs/`.

[Fel10] Amy P. Felty. Hybrid: Reasoning with higher-order abstract syntax in Coq and Isabelle. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 1–2. ACM, 2010.

[FGH+88] Amy P. Felty, Elsa L. Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda-Prolog: An extended logic programming language. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*, pages 754–755. Springer, 1988.

[FM12] Amy P. Felty and Alberto Momigliano. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[FM15] Amy P. Felty and Aart Middeldorp, editors. *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*. Springer, 2015.

[FMP15a] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1–A common infrastructure for benchmarks. *arXiv e-prints*, 2015, arXiv:1503.06095.

[FMP15b] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2–A survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015.

[Gac08] Andrew Gacek. The Abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.

[Gac09] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, September 2009.

[Gan99] Harald Ganzinger, editor. *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*. Springer, 1999.

[Geu93] Jan Herman Geuvers. Logics and type systems. Proefschrift, Katholieke Universiteit Nijmegen, 1993.

[Gir72] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII, 1972.

[GM97] Healfdene Goguen and James McKinna. Candidates for substitution. Technical Report ECS-LFCS-97-358, School of Informatics, University of Edinburgh, Edinburgh, UK, 1997.

[GMN08] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 33–44. IEEE Computer Society, 2008.

[GMN11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

[GMN12] Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, 2012.

[GN91] Jan 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.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Har13] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

[HHP87] Robert Harper, Furio Honsell, and Gordon David Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 194–204. IEEE Computer Society, 1987.

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999.

[How80] William Alvin Howard. The formulae-as-types notion of construction. In Hindley and Seldin [HS80]. The original version was circulated privately in 1969.

[HS80] J. Roger Hindley and Jonathan P. Seldin, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, New York, NY, USA, 2nd edition, 2008.

[Hue80] Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[JM97] Simon Peyton Jones and Erik Meijer. Henk: A typed intermediate language. In *Proceedings of the First International Workshop on Types in Compilation*, 1997.

[KPS17] Jonas Kaiser, Brigitte Pientka, and Gert Smolka. Relating System F and $\lambda 2$: A case study in Coq, Abella and Beluga. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[KSS17] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Autosubst 2: Towards reasoning with multi-sorted de Bruijn terms and vector substitutions. In Miculan and Rabe [MR17], pages 10–14.

[KSS18] Jonas Kaiser, Steven Schäfer, and Kathrin Stark. Binder aware recursion over well-scoped de Bruijn syntax. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 293–306. ACM, 2018.

[KSW16] Fairouz Kamareddine, Jonathan P. Seldin, and J. B. Wells. Bridging Curry and Church's typing style. *Journal of Applied Logic*, 18:42–70, 2016.

[KTS17] Jonas Kaiser, Tobias Tebbi, and Gert Smolka. Equivalence of System F and $\lambda 2$ in Coq based on context morphism lemmas. In Bertot and Vafeiadis [BV17], pages 222–234.

[KWS16] Steven Keuchel, Stephanie Weirich, and Tom Schrijvers. Needle & Knot: Binder boilerplate tied up. In Peter Thiemann, editor, *Programming*

Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, volume 9632 of *Lecture Notes in Computer Science*, pages 419–445. Springer, 2016.

[Lam80] Joachim Lambek. From λ-calculus to cartesian closed categories. In Hindley and Seldin [HS80], pages 375–402.

[LdSOCY12] Gyesik Lee, Bruno C. d. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. GMeta: A generic formal metatheory framework for first-order representations. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 436–455. Springer, 2012.

[Luo90] Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, UK, 1990.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1):125–157, 1991.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3-4):373–409, 1999.

[MR17] Marino Miculan and Florian Rabe, editors. *LFMTP '17: Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, New York, NY, USA, 2017. ACM.

[MT05] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 64–74. IEEE Computer Society, 1993.

[NM88] Gopalan Nadathur and Dale Miller. An overview of Lambda-Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 810–827. MIT Press, 1988.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus - A compiler and abstract machine based implementation of Lambda-Prolog. In Ganzinger [Gan99], pages 287–291.

[Nor08] Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):23:1–23:49, 2008.

[NW08] George C. Necula and Philip Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008.

[PA15] Brigitte Pientka and Andreas Abel. Well-founded recursion over contextual objects. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 273–287. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[PC15] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In Felty and Middeldorp [FM15], pages 272–281.

[PD10] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988.

[Pfe97] Frank Pfenning. Computation and deduction. Lecture Notes, Carnegie Mellon University, April 1997. Available from `https://www.cs.cmu.edu/~twelf/notes/cd.pdf`.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Pie08] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In Necula and Wadler [NW08], pages 371–382.

[Pie15] Brigitte Pientka. Mechanizing meta-theory in Beluga. Conference Tutorial Slides at Automated Deduction - CADE-25, Berlin, Germany, August 2015.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[Pit17] Andrew M. Pitts. Brief notes on the category theoretic semantics of simply typed lambda calculus. Lecture Notes for CST Part III, Category Theory and Logic, Michaelmas term 16/17, Computer Laboratory, Cambridge University, UK, January 2017. Available from `http://www.cl.cam.ac.uk/teaching/1617/L108/catl-notes.pdf`.

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[PRG] Proof General. Available from `https://proofgeneral.github.io/`.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - A meta-logical framework for deductive systems. In Ganzinger [Gan99], pages 202–206.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*, volume 668 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 1993.

[Rey74] John Charles Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[Rey94] John Charles Reynolds. An introduction to the polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1994.

[Roo00] Jan-Willem Roorda. Pure type systems for functional programming. Master's thesis, University of Utrecht, 2000. INF/SCR-00-13.

[Sch00] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.

[SH12] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012.

[Smo15a] Gert Smolka. Confluence and normalization in reduction systems. Lecture Notes for Semantics WS15, Programming Systems Lab, Saarland University, Germany, December 2015. Available from `https://www.ps.uni-saarland.de/courses/sem-ws15/ars.pdf`.

[Smo15b] Gert Smolka. Library ARS. Coq Development for Semantics WS15, Programming Systems Lab, Saarland University, Germany, December 2015. Available from `https://www.ps.uni-saarland.de/courses/sem-ws15/html/ARS.html`.

[SSK19] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de Bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019.

[SST15] Steven Schäfer, Gert Smolka, and Tobias Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 67–73. ACM, 2015.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27,*

2015, Proceedings, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.

[Tak95] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.

[UBN07] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.

[Urb08] Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[vBJ93] Lambert S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

[vD80] D.T. van Daalen. *The language theory of Automath*. PhD thesis, TUE : Department of Mathematics and Computer Science, 1980.

[Vir96] Roberto Virga. Higher-order superposition for dependent types. In Harald Ganzinger, editor, *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1996.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. CMU-CS-99-167.

[Vou12] Jérôme Vouillon. A solution to the POPLMARK challenge based on de Bruijn indices. *Journal of Automated Reasoning*, 49(3):327–362, 2012.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.