Semi-Automation of Meta-Theoretic Proofs

Johanna Schwartzentruber

School of Computer Science

McGill University, Montréal

August 2023

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Abstract

BELUGA is a proof assistant designed for the mechanization of programming languages and other formal systems. An interactive tactic-based prover HARPOON was recently deployed, and though it was designed to ease usability for BELUGA's users, much human interaction is still required for its proof developments.

We develop a theoretical foundation for a semi-automated proof search procedure within BELUGA in the form of a two-level focusing calculus and implement it through the tactic `auto-invert-solve`. The focusing calculus is sound and complete with respect to the sequent calculus for the fragment that we are automating. Once a case analysis has been conducted, `auto-invert-solve` searches for a focused uniform proof of a subgoal via a bounded depth-first search by using all available assumptions. Upon completion of a proof, `auto-invert-solve` produces a proof witness in the form of a program that is independently type-checked against the subgoal and subsequently spliced into the proof script.

Our aim is to automate the tedious cases of proof development, leaving only the interesting cases to the user. We have utilized `auto-invert-solve` to simplify several common theorems including type preservation and value soundness for MiniML, weak-head normalization for the simply-typed lambda-calculus, and the Church-Rosser theorem for the untyped lambda-calculus. In these case studies, we demonstrate that `auto-invert-solve` reduces the amount of user interaction needed to complete large and complex proofs by automatically solving simple subgoals and helper lemmas.

Résumé

Beluga est un assistant de preuve conçu pour la mécanisation de languages de programmation et autres systèmes formels. Récemment, un prouveur interactif basé sur les tactiques Harpoon a été déployé, et malgré qu'il ait été conçu pour faciliter l'utilisation de Beluga, il nécessite tout de même beaucoup d'interaction de la part de l'utilisateur afin de produire des preuves.

Nous développons un fondement théorique pour une procédure de recherche de preuve semi-automatisée dans Beluga sous la forme d'un calcul à deux niveaux avec focus, et nous l'implémentons en tant que la tactique `auto-invert-solve`. Ce calcul avec focus est correct et complet par rapport au fragment du calcul des séquents que nous automatisons. Lorsqu'une analyse par cas est effectuée, `auto-invert-solve` cherche une preuve uniforme par le focusing pour un sous-objectif au moyen d'un parcours en profondeur limitée qui utilise toutes les hypothèses disponibles. Lorsque la preuve est terminée, `auto-invert-solve` produit une démonstration sous la forme d'un programme dont le type est vérifié indépendamment avec celui du sous-objectif, puis le programme est inséré dans la preuve.

Notre objectif est d'automatiser les cas fastidieux du développement de preuves, et de ne laisser que les cas intéressants pour l'utilisateur. Nous avons utilisé `auto-invert-solve` pour simplifier plusieurs théorèmes communs, incluant la préservation des types et la cohérence des valuers dans MiniML, la normalisation weak-head du lambda-calcul à types simples, et le théorème de Church-Rosser pour le lambda-calcul sans types. Par ces études de cas, nous démontrons que `auto-invert-solve` réduit la quantité d'interactions nécessaires de la part de l'utilisateur pour compléter des preuves larges et complèxes en résolvant automatiquement des sous-objectifs simples ainsi que des lemmes auxiliaires.

Contents

List of Figures

Abbreviations

CMTT contextual modal type theory. 24, 25

HOAS higher-order abstract syntax. 4, 12, 13, 19, 21, 23, 34–38, 42

HOL higher-order logic. 38

ITP interactive theorem proving. 12, 30

LF Edinburgh Logical Framework. 5, 6, 12, 14, 15, 20–28, 40–45, 47–51, 61–64, 66, 70, 85–91, 95–97, 101–103

PL programming language. 11, 14, 18, 20, 23, 104

STLC simply typed lambda-calculus. 6, 17, 21, 26, 89, 99, 100

Preface

Acknowledgements

I would like to first thank my supervisor Prof. Brigitte Pientka. She taught me what an impactful scientist looks like and has done so patiently during uncertain times. Both her intelligence and humility are part of what makes her an inspiring role model for anyone in STEM and it was an honour to work with her. I too want to thank the students working under Prof. Pientka for always begin inclusive and willing to answer any questions I had. I wish you all the best.

Thank you also to Prof. William M. Farmer for the support and for all the great discussions. You made me believe I could do research, and have fun while doing it. I would also like to thank Prof. Wolfram Kahl for creating CalcCheck, the first proof assistant I was introduced to. It is the reason for my continuation in computer science.

Finally I would like to thank those closest to me. To my brother who ventured off to the unknown land of Montréal with me- thank you for your company during the pandemic. I hope I was a decent roommate. Thank you always to my parents for their never-ending love and support. Last but not least, I owe much thanks to my partner Omario. You have made it fun.

Funding

This work was funded by Canada Graduate Scholarships - Master's (CGS M) and my supervisor, Brigitte Pientka. I am grateful for the financial support I have received and recognize the advantages it has given me.

Chapter 1

Introduction

Since the first personal computer came to market in the early seventies, society has shifted to incorporate technology in every sector possible. Among those include government, education, law-enforcement, banking, entertainment, and health care. Recently, the COVID-19 pandemic forced millions of students to participate in e-learning, with some as old as five having to learn how to operate a computer.

Our dependency on technology then makes it crucial that the software we employ performs as intended. To ensure this, we count on various forms of software verification, one of which is formal verification. Formal verification is the process of proving that a system upholds some property using formal methods. One common method is theorem proving. Often this involves modelling the system of interest in some logic, called the *specification logic* and proving statements about the behaviour of the system within another logic, called the *reasoning logic*. It is common practice to construct these formal proofs within computer programs specifically designed to aid users in constructing mathematical proofs, i.e. proof assistants. The reason for their use is to minimize human errors made during verification, which is essential when working to ensure trust in software.

Formal verification has been used to verify the correctness of the behaviour of a number of software systems. One notable example is the CompCert C verified compiler: a high-assurance compiler for nearly all of the C language [Leroy, 2009]. It is mechanized in the Coq proof assistant [Bertot and Castéran, 2004], making it the first verified compiler for a realistic language. Its ingenuity was put to the test by researchers Yang et al. who found that it produced no middle-end bugs, which were found in every other tested C compiler [Yang et al., 2011]. Having a verified compiler eliminates one of the sources of bugs in code, bringing us closer to writing error-free programs.

There are numerous processes to verify within the software layer of computer systems. One of interest is the *programming language* (PL) itself that one uses to write programs in. There are certain properties programming languages can uphold which guarantees that programs written in them perform as they should. For example, in the context of statically-typed languages there is a notion of type safety which ensures that any well-typed program cannot "go wrong". In other words, any well-typed expression must evaluate to a value of the appropriate type and not get stuck. The concepts of "value", "evaluation", and "well-typed" are formally defined by the syntax and semantics specific to each PL.

When developing proofs *about* a theory we are performing meta-reasoning, which differs from developing proofs *within* a theory. Constructing meta-theoretic proofs about PLs in particular is tricky as it involves careful consideration about proof infrastructure such as: how to represent variables, substitutions, contexts, and derivations. At present, there is still no canonical way to support these concepts in proof assistants. In 2005, a set of PL researchers became determined to bring together the PL and automated proof assistant communities. With the desire to make proof assistants commonplace in PL research, these scientists created the POPLmark challenge [Aydemir et al., 2005]: a set of benchmark problems intended to investigate the current state of the art, as well as stimulate innovation within the field. These problems were designed to highlight some of the known difficulties that arise in PL theory proofs. Despite its potential, the POPLmark challenge did not push existing systems to their limits. All systems already had enough infrastructure to be able to complete the tasks

[Aydemir et al., 2012]. One in particular was Beluga [Pientka and Dunfield, 2010].

Beluga is a proof assistant designed for the mechanization of programming languages whose sophisticated infrastructure makes for simple and concise formalizations. Some features of Beluga include its first-class support for substitutions and contexts, its use of contextual objects, and employing *higher-order abstract syntax* (HOAS) in its specification logic, an implementation of the logical framework LF [Harper et al., 1993]. These features reduce the amount of infrastructure one needs to build manually and allow for more concise and elegant mechanizations. Users of Beluga formalize their systems within LF and subsequently prove properties about them in Beluga's reasoning logic: a dependently typed first-order logic. Since reasoning about a formal system often involves also reasoning within one, proof search in Beluga requires proof search over *two* logics. Beluga represents theories and meta-theories using the *propositions-as-types* perspective, that is, propositions are encoded as types of the (meta-)theory, and their proofs as objects or programs within that theory. We focus our attention on proofs by structural induction. Those proofs are implemented as recursive dependently-typed programs and checking the correctness of them reduces to type checking the corresponding program. In order to assist with proof development, an interactive tactic-based prover was recently deployed for Beluga, called Harpoon [Errington et al., 2021]. Conducting proofs in Beluga now requires one to apply tactics to a goal until no subgoals remain. These tactics are meant to mimic the natural large steps one usually takes in written proofs making proof development more familiar for users which in turn improves their experience. While proving a theorem interactively, Harpoon simultaneously builds a proof script that, upon completion of the proof, gets translated to a Beluga program and presented to the user. Therefore Beluga proofs are *verifiable*.

Even with the help of *interactive theorem proving* (ITP), developing formal proofs often still requires much assistance from a user with a high level of domain specific knowledge and familiarity with the proof assistant. Many formal theorems have numerous helper lemmas and theorems that precede them, many of which are tedious to construct and offer no real substance to the main proof. Even main theorems often only include a small number of interesting cases that are worth investigating, with the rest being simple, almost trivial. These are examples of cases and theorems we would like to prove automatically. This would allow users to focus their energy on the parts of the proofs that offer insight.

Currently, Twelf is the only HOAS-based proof assistant designed for mechanizing programming languages which is fully automated [Pfenning and Schürmann, 1999]. Why bother then providing more automation to Beluga? There are several shortcomings of Twelf that are addressed by features of Beluga. Most notably, Beluga's logic is more expressive: unlike Twelf, Beluga supports inductive and co-inductive types allowing for encodings of recursive definitions about LF objects. Further, Twelf does not produce proof witnesses, providing no way to verify the correctness of a proof. It is therefore of interest to extend Beluga's automatic proof search capabilities which would make it an ideal candidate for formal system verification.

In this thesis I present the theoretical foundation and implementation of the tactics `auto-invert-solve` and `inductive-auto-solve` for Harpoon. The goal of these tactics is to ease proof development for Beluga users by allowing them to by-pass many of the simpler proofs. `auto-invert-solve` is the implementation of a two-level focusing calculus. The tactic is meant to be employed once all variable splits for a subgoal have been made. `auto-invert-solve` either finishes a proof or fails, in which case the user is asked for another tactic. If `auto-invert-solve` finishes a proof for a subgoal, the program that is generated is spliced into the proof script, and Harpoon continues to the next subgoal if any remain. The tactic `inductive-auto-solve` is an extension of `auto-invert-solve`, which when called automatically performs induction on the user-specified argument and calls `auto-invert-solve` on each produced subgoal. We demonstrate the usefulness of these tactics by employing them on a number of notable case studies such as type preservation and value soundness for MiniML, weak-head normalization for the simply typed lambda calculus, and the Church-Rosser theorem for the untyped lambda calculus. We show that they allow for automatic completion of many of the simpler lemmas and subcases of these theorems, so long as the proof state falls within our subset of interest. This subset includes all of Beluga's logic excluding substitution and parameter variables, and block context schemas in the specification logic, and inductive types, (automatic) recursion, and pattern matching in the reasoning logic.

1.1 Contributions

The contribution of this work is three-fold. First, we develop a two-layer focusing calculus designed for proof search over a subset of Beluga's logic. We prove it sound and complete with respect to the corresponding sequent calculus in Chapter 3. Second, we implement this focusing calculus in the form of two tactics, `auto-invert-solve` and `inductive-auto-solve` in Chapter 4. Third, we demonstrate the effectiveness of our work using a number of popular case studies within PL theory (see also Chapter 4).

1.2 Contribution of Authors

This thesis is based in part on the previously published paper accepted at the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, cited here [Schwartzentruber and Pientka, 2023]. I have permission from my co-author to use the work in my thesis.

The literature review presented in Chapter 2 is of my own work. The sequent calculus for contextual LF presented in Chapter 3.2 is an extension of that presented in Nanevski et al. [2008] with the addition of explicit substitutions, context variables, and universal statements. The sequent calculus for the computation level also presented in Chapter 3.2 is my own creation and also based off the work in Nanevski et al. [2008], especially the $\Box R$ and $\Box L$ rules. The admissibility of cut and contextual cut for the computation level was proven by myself.

The focusing cacluli presented in Chapter 3.3 was my own work and the soundness and completeness theorems and proofs were all proven by myself.

The theorem prover for contextual LF in Beluga was implemented by Jacob Errington which I then extended with proof search over substitutions. I implemented the meta-theorem prover and tested it on a number of interesting examples which are illustrated in Chapter 4.3.

Chapter 2

Background

As theorem proving relies on encodings, we begin in Chapter 2.1 with a discussion of the different approaches used to encode languages, and in particular, on the sophisticated approach taken by Beluga. In Chapter 2.2 we introduce Beluga by providing an overview of its interesting aspects which impact our proof search algorithm and motivate our work by showcasing the large amount of user interaction needed to interactively build proofs using Harpoon. Finally, to situate our work, we survey similar proof assistants along with their automation statuses, comparing them to Beluga.

2.1 Techniques to Encode Formal Systems

Meta-reasoning is limited to how effective an encoding is. Stronger meta-languages allow for more direct encodings, and the more direct the encoding, the easier it is reason about. Therefore it is crucial for proof assistant designers to think carefully about how they plan to implement encodings of object-languages.

2.1.1 Encoding Binders

Reasoning about languages poses a unique challenge that does not arise when reasoning about other inductive data structures, such as lists or trees. Languages, like the simply typed lambda-calculus (STLC), incorporate a special concept, namely *bindings*.

A STLC is a small formal system that can be used to model computation based on two simple ideas: function abstraction and application. These concepts appear everywhere in computation, and as such, this core calculus has been used extensively as the logical basis for studying programming languages. Therefore, we use it as the running example throughout this thesis.

Its syntax is simple: consisting of a set of types and terms. There is a finite set of type constants from which new types are created using the function-type constructor. Terms are either variables, constants, or constructed through function application and abstraction.

$$\begin{array}{lll l} \text{Base Types} & P & ::= & \mathbf{b} \\ \text{Types} & A, B & ::= & P \mid A \to B \\ \text{Terms} & M, N & ::= & x \mid \mathbf{c} \mid \lambda x : A.\, M \mid M N \end{array}$$

Figure 2.1: Grammar of a STLC.

Seemingly simple, this small calculus poses quite the challenge for proof assistant designers. A reoccurring issue when building a proof assistant for PL theory is determining how to represent variable binders, like the abstraction operator, in an implementation. In the expression $\lambda x : A.M$, the λ operator *binds* the variable x of type A in the term M. There are several ways to encode the concept of binding, each with their own benefits and implications.

First-Order Approaches

These techniques restrict the quantifier to quantifying solely over individuals, as below. In the implementation of the untyped lambda-calculus below, variables are represented as strings and the λ operator is implemented as the constructor `abs`.

```
                                        term =
var = string                              | abs : var → term → term
                                          | app : term → term → term
```

Often using these approaches make it easy to directly formalize systems but at the cost of providing manual support for keeping track of and renaming bound variables and avoiding variable capture. Such processes are used when applying substitutions and deciding if terms are α-equivalent, which happens regularly when reasoning about PL theory.

A few common first-order approaches include nominal [Pitts, 2001], de Bruijn [de Bruijn, 1972], and locally nameless representations, also conceived by de Bruijn. Nominal representations are most similar to what we write on paper; bound variables are given a name and terms are equivalent based on α-conversion, making it simple to read. The downside to using this method is that much infrastructure is needed to deal with substitution.

A more straightforward technique to use in implementations are De Bruijn encodings. Variables are represented as positions indicating their distance from their binding constructor. Take for instance the term $\lambda x.\lambda y.xy$. Using de Bruijn indices this would be expressed as $\lambda.\lambda.2\ 1$. The benefit to this approach is that there is no need to handle α-conversions since each term has a unique representation. On the other hand, these formulations can be difficult to read for humans and since terms and their indices are highly dependent on the context, much consideration is needed when moving terms between different contexts.

Locally nameless encodings provide the best of both worlds [de Bruijn, 1972]. Bound variables are represented by de Bruijn indices and free variables by explicit names. Therefore α-equivalence classes do not exist and there is no need to shift indices when introducing free variables. Despite these benefits, much overhead is still required to construct proofs. In fact, Xavier Leroy's locally-nameless solution to the POPLmark challenge was *more* verbose than a pure de Bruijn solution [Leroy, 2007].

Higher-Order Abstract Syntax

The higher-order approach implements the object-level λ operator as a higher-order function that takes as input some function `f` which maps lambda-terms to lambda-terms. Abstractions may be implemented as: `abs f`. Now the argument is viewed as a function at the meta-level, that is, object-level variables and binders are represented by variables and abstractions in the meta-language.

```
term =
  | abs : (term → term) → term
  | app : term → term → term
```

Issues of renaming, α-conversion, and substitution are also passed to the meta-language, which already has designated infrastructure to handle these operations. The drawback to this method is that the implementation logic must be able to express higher-order types, which not all can. Nevertheless, by reusing the infrastructure in the meta-language, HOAS allows for more elegant and concise formalizations.

2.1.2 Logical Frameworks

Logical frameworks are (often simple) meta-logics used to present or define deductive systems in a uniform way by encoding them into a signature. There are several properties a logic should posses in order to be a "good" logical framework. First, it must be possible for an implementation to validate proofs within a specification of a deductive system. Second, it should be possible to specify *adequate* encodings to ensure the derivations within the specification are correct. An adequate encoding is one that offers a safe translation between object-level expressions and their encodings. This ensures that after we operate on meta-level expressions, we may translate the results back into the object-language for correct interpretation. Further, for HOAS-based systems, adequacy actually guarantees a bijection between object-level expressions and their encodings.

We discuss Beluga's logical framework of choice, the Edinburgh Logical Framework (LF) [Harper et al., 1993], and its logic programming interpretation, hereditary Harrop formulas [Miller et al., 1991].

Hereditary Harrop Formulas

Higher-order hereditary Harrop formulas are the higher-order extension of hereditary Harrop formulas [Miller et al., 1991]. These sets of formulas in intuitonistic logic are generalizations of Horn clause logic [Emden and Kowalski, 1976] which makes them better suited for encoding PLs. These subsets are interesting because they possess a special property which makes them ideal candidates as logical frameworks.

$$\begin{array}{lcl} G & ::= & \top \mid A \mid G \wedge G \mid G \vee G \mid D \rightarrow G \mid \forall x : \tau . G \mid \exists x : \tau . G \\ D & ::= & A \mid G \rightarrow D \mid D \wedge D \mid \forall x : \tau . D \end{array}$$

Figure 2.2: Grammar of hereditary Harrop formulas.

The logic consists of two mutual recursively defined sets of formulas. A *program* (or later referred to as an environment) is a set closed of D-formulas and a *goal* is a closed G-formula. In this setting, proof search corresponds to deducing a given goal G from a given program P.

Constructing proofs in intuitonistic logic can be highly non-deterministic. Hereditary Harrop formulas possess a special property in which proof construction in their theories is more directed. In particular, these classes exhibit the existence of *uniform proofs* for all provable goal formulas [Miller et al., 1991]. Uniform proofs are discussed in more detail in Chapter 3.3, but for now it is enough to know they are proofs whose structure obeys a set of conditions. Restrictions on proof structure help to simplify the search procedure which is especially important for automating proof search.

The Edinburgh Logical Framework

The *Edinburgh Logical Framework* (LF) is a simple meta-logic created by Robert Harper, Furio Honsell, and Gordon Plotkin in the nineties [Harper et al., 1993]. LF is based on a dependently-typed lambda-calculus with support for higher-order types thereby allowing for encodings using HOAS. There are three levels of terms in LF: objects, types, and kinds. By viewing hereditary Harrop formulas as the logic programming interpretation of LF types, specifications are encoded using the *propositions-as-types* perspective. Therefore proofs are validated by type checking the corresponding proof object. Since Beluga has an implementation of LF which it uses as its specification language, we discuss it here in more detail.

To demonstrate how theories are encoded in LF (and similarly in Beluga), we encode a STLC with one base type, making use of HOAS encodings which LF permits. We choose an intrinsically-typed representation to simplify our discussion.

```
LF tp : type =
| b : tp
| arr : tp → tp → tp
;
```

```
LF term : tp → type =
| app : term (arr A B) → term A → term B
| abs : tp → (term A → term B) → term (arr A B)
| c : term b
;
```

Note that the types for the constructors `app` and `abs` contain free variables in them (`A`, `B`). We consider such variables as being implicitly universally quantified over. We continue to omit inferable universal abstractions in LF declarations for better readability.

We use the keyword **`LF`** to specify LF encodings. LF kinds, types, and objects are printed in black, pink, and blue respectively. In the above encoding, `tp` is an LF type which has the LF kind **`type`**. `tm` is an LF type which has the LF kind Πx:`tp`. **`type`**. This type has three constructors. The LF object `c` is a nullary constructor, and represents an object with type `term b`. The LF object `app` is a binary constructor which when applied to two LF objects of type `term (arr A B)` and `term A`, outputs an LF object of type `term B`. As an example, if $\lambda x : A.\ M$ is a term of type $A \to B$ and N is some term of type A then we implement the function application $(\lambda x : A.\ M)N$ as `app (abs A λx. M) N` which is an LF object of type `term B.`

The reductions of lambda terms define their operational semantics. Reduction is often formalized as a set of possible steps a term can take. We define this semantics using the relation `step`:

```
LF step : term A → term A → type =
| beta : step (app (abs A M) N) (M N)
| stepapp : step M M' → step (app M N) (app M' N)
;
```

The relation `step` is an LF type which relates two LF objects of the same type. Viewing this from the *propositions-as-types* perspective, we have for any two encodings of lambda terms `M` and `N` the type `step M N` if the term M steps or *reduces* to the term N. This type has two constructors. As an example, the LF object `beta` has type:

`ΠA: tp. B: tp. M: term A → term B. N: term A. step (app (abs A M) N) (M N)`

Viewing this from the proofs-as-objects perspective, we treat `beta` as a proof of the proposition encoded as its type, which we translate to mean: for any types A and B, and any terms M and N of types $A \to B$ and A respectively (i.e. $M = \lambda x : A.\ M'$), the lambda term $(\lambda x : A.\ M')N$ steps to the term $[N/x]M'$. This is exactly beta-reduction, where object-level substitution is modelled by LF application.

We use these LF constructors to construct lemmas *within* the logic. For example, if $\mathbf{c}$ is a constant in our object language with base type $\mathbf{b}$, the proposition stating that the term $((\lambda x : \mathbf{b} \to \mathbf{b}.(\lambda y : \mathbf{b}.\ x\ y))\ (\lambda w : \mathbf{b}.\ w))\ \mathbf{c}$ reduces to $([(\lambda w : \mathbf{b}.\ w)/x](\lambda y : \mathbf{b}.(x\ y)))\ \mathbf{c}$ is encoded as:

```
step (app (app (abs (arr b b) (λx. abs b (λy. app x y)))
   (abs b (λw. w))) c) (app ((λx. abs b (λy. app x y)) (abs b (λw. w))) c)
```

In order to "prove" this statement, we need to find an LF object with this type. After some thought, we can see that the object:

```
stepapp b b (app (abs (arr b b) (λx. abs b (λy. app x y)))
   (abs b (λw. w))) ((λx. abs b (λy. app x y)) (abs b (λw. w))) c
```

```
(beta (arr b b) (arr b b) (λx. abs b (λy. app x y)) (abs b (λw. w)))
```

inhabits this type. Therefore this object represents a proof of the statement $((\lambda x : \mathbf{b} \to \mathbf{b}.(\lambda y : \mathbf{b}.\ x\ y))\ (\lambda w : \mathbf{b}.\ w))\ \mathbf{c}$ reduces to $([(\lambda w : \mathbf{b}.\ w)/x](\lambda y : \mathbf{b}.(x\ y)))\ \mathbf{c}$. In fact this is the only proof of this fact within our encoding, but this is not always the case.

2.2 Beluga System Overview

Proofs in PL theory are usually syntactic. The difficulty of these proofs comes from the extensive amount of infrastructure that is required to formalize and reason about formal systems.

In the previous section we explored different ways to encode formal systems. First-order approaches require additional mechanisms (written in the object-level) to deal with variable renaming and applying substitutions. These operations must be manually programmed for each new object language encoding. On the other hand, using HOAS encodings allow us to pass these issues off to the meta-level (specification logic), which already has the required infrastructure to support these operations.

Beluga is a proof assistant designed for mechanizing the meta-theory of formal systems [Pientka and Dunfield, 2010]. It includes an implementation of LF which users can use to encode their system of interest. Reasoning *about* these LF encodings takes place in Beluga's meta-logic for LF, which we call the reasoning logic. In order to perform meta-reasoning, we embed these LF objects within the reasoning logic using a modal box (necessity) operator $\Box$ [Pientka, 2008]. Consequently, this reasoning logic exists on a different level *above* the specification logic LF.

We explore this notion in more detail as well as other key Beluga features. We then showcase how meta-theorems are formalized and proven in Beluga as well as interactively using Harpoon. This provides an informal description of the reasoning logic of Beluga.

2.2.1 Beluga: Contextual Modal Type Theory Extended

There are two levels of logic within the Beluga system. Beluga's specification logic is a generalization of LF called contextual LF and its reasoning logic is analogous to a dependently-typed first-order logic [Pientka, 2008]. This two-level logic is based on dependent *contextual modal type theory* (CMTT) [Nanevski et al., 2008] and resembles the constructive modal logic S4. To simplify our discussion, we only focus here on simple types.

In traditional CMTT we maintain notions of *truth* and *validity*. We consider a proposition to be valid, represented by the judgment C *valid*, if its truth does not depend on the truth of other propositions. This concept is generalized to *contextual validity*, written with *box*-syntax: $[\Psi \vdash C]$ and represented by the judgment C *valid* Ψ where Ψ is short for C_1 *true*$, \ldots, C_n$ *true*. We take this to mean that the truth of C may depend only on the truth of assumptions in Ψ (and contextually valid assumptions which are assumed to always be true). Hypothetical judgments contain two contexts: a *modal* context Δ which contains contextually valid assumptions and a local context (Γ or Ψ) which contains ordinary bound variables. The general judgment takes the form:

$$A_1 \textit{ valid } \Psi_1, \ldots, A_n \textit{ valid } \Psi_n; B_1 \textit{ true}, \ldots, B_m \textit{ true} \Vdash C \textit{ true}$$

To prove contextual validity $[\Psi \vdash C]$ is to prove C using only assumptions from Ψ and Δ.

$$\frac{\Delta; \Psi \Vdash C \textit{ true}}{\Delta; \Gamma \Vdash [\Psi \vdash C] \textit{ true}}$$

Figure 2.3: Definition of contextual validity.

We call the assumptions in the modal context *contextual* or *meta* assumptions, and we denote them as u. We may use such an assumption C *valid* Ψ to deduce C *true*, but only if we verify Ψ. Later, we see this is achieved by finding a substitution from Ψ into the current local context in which the contextual assumption is to be used. As a consequence, we pair contextual variables with an explicit postponed substitution, representing a closure. We then apply the substitution once we know what the meta-variable should stand for. We use the judgment $\Delta; \Gamma \Vdash \Psi$ to denote that all the propositions in Ψ are true using only assumptions from Δ and Γ.

$$\frac{\Delta, u :: C\ valid\ \Psi; \Gamma \Vdash \Psi}{\Delta, u :: C\ valid\ \Psi; \Gamma \Vdash C\ true}$$

In BELUGA's application of CMTT, two different languages are used for terms inside and outside a box, unlike the single one used in our discussion of CMTT thus far. In BELUGA, contextual LF is the language used inside the box and is the one in which users encode their formal systems. Meta-reasoning about such *contextual objects* is then done in BELUGA's reasoning logic, the language used outside the box.

Now, we use Ψ to denote LF (specification-level) contexts when reasoning within LF and Γ to denote local contexts in our reasoning or computation logic. We continue to use Δ as before. There are now two judgments to consider: we take $\Delta; \Psi \vdash A$ to mean LF type A is provable using assumptions in Δ and Ψ, and $\Delta; \Gamma \Vdash \tau$ to mean computation type τ is provable using assumptions in Δ and Γ. Our new definition of contextual validity then becomes:

$$\frac{\Delta; \Psi \vdash A}{\Delta; \Gamma \Vdash [\Psi \vdash A]}$$

Figure 2.4: Definition of contextual validity in BELUGA.

We can then conclude that if in our specification logic we derive the type A under assumptions from Δ and Ψ we then have a derivation in our reasoning logic of the contextual type $[\Psi \vdash A]$ under assumptions from Δ and *any* computation-level context we choose. We then may use these contextual types to construct properties about such LF derivations.

When we turn this definition into an inference rule in our proof system and apply it in bottom-up search, we must remember that none of the assumptions from Γ may be used to prove A. These assumptions reside on a different level/logic and thus make no sense in LF.

We discuss Beluga's specification logic in more detail. Contextual LF extends LF by providing explicit support for contextual objects and first-class contexts. These concepts allow us to concisely formalize and reason about *open* LF objects which depend on assumptions. As a result, we can encode an entire (closed) hypothetical judgment within a contextual object. Extensions made to Beluga also allow inductive reasoning over such contextual objects and contexts [Pientka and Abel, 2015].

Support for first-class contexts gives us the ability to quantify and reason abstractly over them. This is done through the use of *context variables*. Quantification over contexts is required for many meta-theorems, and being able to do so explicitly gives the user more control over proof development. Further, having this direct support for contexts and hypothetical derivations allows for more compact proofs as it eliminates the need to build and maintain contexts explicitly and it gives us substitution lemmas for "free".

2.2.2 Specifying Meta-Theories

We focus here on constructing two lemmas needed to prove weak-head normalization for the STLC to highlight key aspects of Beluga. For simplicity we do not reduce inside abstractions. Interested readers may check out [Cave and Pientka, 2013] for the full mechanization of the theorem.

We build on the specification started in Section 2.1.2, all of which may be implemented in Beluga. We extend our semantics with a multi-step relation, `steps`. We say `steps M M'` if `M` steps to `M'` in some finite number of steps.

```
LF steps : term A → term A → type =
| id   : steps M M
| sstep : step M M' → steps M' M'' → steps M M''
;
```

In Beluga, users need not provide arguments for parameters that are implicitly universally quantified.

We denote which terms in our language are values. Values are considered to be those terms which do not step to a syntactically different term, in other words, they cannot be reduced. We also define what it means for a term to halt, that is, it steps to a value.

```
LF val : term A → type =
| val/c : val c
| val/abs : val (abs A M)
;
```

```
LF halts : term A → type =
| halts/m : steps M M' → val M' → halts M
;
```

Finally, we encode the notion of reducibility using a *logical predicate* or in other words a (unary) *logical relation* [Tait, 1967]. Proofs using logical relations are a common technique used when typical induction, say over the structure of the type, does not provide a strong enough induction hypothesis. We formalize the predicate in Beluga's computation logic, as it requires a strong, computational function space unlike the weak function space of LF. It is represented using an indexed recursive type which allows us to define inductive properties about contextual objects, contextual types, and contexts [Cave and Pientka, 2012]. Our type is stratified by its index `tp`.

```
stratified Reduce : {A:[ ⊢ tp]}{M:[ ⊢ term A]} ctype =
| I   : [ ⊢ halts M] → Reduce [ ⊢ b ] [ ⊢ M]
| Arr : [ ⊢ halts M]
          → ({N:[ ⊢ term A]} Reduce [ ⊢ A] [ ⊢ N] → Reduce [ ⊢ B ] [ ⊢ app M N])
            → Reduce [ ⊢ arr A B ] [ ⊢ M]
;
```

Notation wise, we denote computation-level types and terms in red and teal coloured font respectively. Our predicate `Reduce` acts on closed terms (note the empty LF context in the description of `M`). The constructor `I` states that a term or base type reduces to base type if it halts. The constructor `Arr` states that `M` reduces to the function type `arr A B` if it halts, and for every term `N` that reduces to type `A`, the application of `M` to `N` reduces to type `B`. The keyword **`stratified`** indicates that our type is recursive but not strictly positive.

There are two types of recursive datatype definitions used in Beluga [Pientka and Cave, 2015]. Typical inductive definitions, defined using the keyword **`inductive`**, must follow a positivity condition, that is, no inductive occurrences may appear to the left of an implication. Other recursive datatypes that do not obey this rule are defined with the keyword **`stratified`**. In order to be a valid stratified datatype, there must be a smaller index argument that decreases in each recursive occurrence of the definition, as in our definition of `Reduce` (`A` and `B` are smaller types than `arr A B`).

Writing Proofs as Programs

Next, we present a trivial result that will aid us in the construction of our main helper lemma.

```
rec halts_step : [ ⊢ step M M’] → [ ⊢ halts M’] → [ ⊢ halts M] =
fn s, h =>
  let [ ⊢ halts/m MS V] = h in let [ ⊢ S] = s in [ ⊢ halts/m (sstep S MS) V]
;
```

We leave the contextual variables `M`, `M'`, and `M''` implicitly universally quantified as Beluga is able to reconstruct their type. We use Beluga's simple function space to formalize our implication statement. The proof is straightforward and presented above. Recall proofs are programs in Beluga, therefore proof development proceeds in a functional manner.

We begin by stating the theorem name and statement, prefixed with the keyword **`rec`**. The proof starts by peeling off the implication antecedents (**`fn`** `s, h =>`). Working backwards, we know we must use `halts/m` to construct our desired term as it is currently the only constructor for terms of type `halts`, therefore we must solve its subgoals, namely that there is a value that `M` steps to. We first *invert* assumption `h` as it has only one possible constructor. This reveals that it is actually the contextual object `[⊢ halts/m MS V]` where `MS` and `V` are LF terms of type `steps M' N` and `val N` (for some implicit meta-variable `N`) respectively. It may appear we have every piece of the puzzle required to solve our goal: we have a value that our term `M` steps too. Recall however that once we transition to the LF level to build our LF proof term, we do not have access to our computation-level context, in which `s` resides. Therefore, we must first *unbox* said assumption.

We are now ready to prove that reducibility is closed under expansion. In other words, if we have that term `M` steps to term `M'` and `M'` reduces to some type `A`, then so will `M`. The proof proceeds by structural induction on the object `[ ⊢ A]`. Extensions made to Beluga allow for totality checking and structural recursion over contextual LF objects [Pientka and Abel, 2015].

```
rec bwd_closed : {A:[ ⊢ tp]} {M: [ ⊢ term A]} {M': [ ⊢ term A]} [ ⊢ step M M']
                  → Reduce [ ⊢ A] [ ⊢ M'] → Reduce [ ⊢ A] [ ⊢ M] =
/ total a (bwd_closed a m m') /
mlam A, M, M' => fn s, r =>
  case [ ⊢ A] of
  | [ ⊢ b] => let I h = r in I (halts_step s h)
  | [ ⊢ arr T T1] =>
    let [ ⊢ S] = s in
    let Arr h f = r in
    Arr (halts_step s h)
          (mlam N => fn rn => bwd_closed _ _ _ [ ⊢ stepapp S] (f [ ⊢ N] rn))
;
```

Totality checking is employed by specifying the induction variable (**`total`** `a`) along with its index in the theorem (`bwd_closed a m m'`). We begin again by peeling off the universally quantified variables (**`mlam`** `A, M, M' =>`) and implication antecedents. We indicate which variable we are to perform case analysis on (**`case`** `[ ⊢ A]` **`of`**). If the specified variable is inductive, the respective induction hypotheses are automatically generated and made available for use in each respective inductive case. There are two constructors for the type `tp` therefore there are two subgoals to prove.

In the case `[ ⊢ A]` is the base type, type inference tells us that `r` has type `Reduce [ ⊢ b] [ ⊢ M']`. Since this type has only one constructor, we invert `r` revealing its constructor. We then construct our desired term using the previous lemma.

In the inductive case, `[ ⊢ A]` is the term `[ ⊢ arr T T1]` where `T` and `T1` are both of type `tp`. We first unbox assumption `s` to have it available for constructing `lf` proof terms. Then again, `r` has one constructor so we invert it. In order to construct a term of type

`Reduce [ ⊢ arr T T1] [ ⊢ M]` we must use the constructor `Arr` as it is the only constructor for this type. To solve the first subgoal of `Arr` we simply use the previous lemma again. The second subgoal is of function type, therefore we must construct a Beluga function. Since the type of `arr` is inductive, induction hypotheses are automatically generated for each subterm in the pattern `[ ⊢ arr T T1]`. In particular, we would like to find a term of type `Reduce [ ⊢ T1] [ ⊢ app M N]`, thus we employ the induction hypothesis generated for subterm `T1` by making a recursive call to `bwd_closed`. Due to type inference, we do not need to make explicit every parameter to the call. Instead, we only need to supply enough type information so that Beluga can infer the rest.

In the next sub-chapter, we introduce and provide motivation for Harpoon by demonstrating how one would use it to prove `bwd_closed`.

2.2.3 Harpoon: Constructing Proofs Interactively

Interactive theorem proving (ITP) comes in many forms, whether its simply a program that checks a human constructed proof or a user-guided computer generated proof. Whenever one uses a proof assistant to construct proofs they are likely using some sort of ITP. ITP differs from automated theorem proving, whereby a computer develops proofs with no user input. Programming an automated prover is a more difficult task due to the large size of proof search spaces and more importantly, because of the non-deterministic nature of proof search. That is why more often than not, proof assistants opt for an interactive proof development setting.

In order to have ITP, there needs to be a language that the user can use to communicate with the system. Often times this will be a high-level language that is more natural for users at the cost of involving some sort of automation. Normally, users interact with the system through the use of tactics. Tactics are like commands, which instruct the system on how to build a proof.

We provide a short overview of Harpoon, Beluga's interactive theorem proving environment [Errington et al., 2021]. Harpoon users develop proofs using a small set of tactics that manipulate the proof goal. Types of subgoals are closed thereby allowing users to solve subgoals in any order, independent of each other. This, along with the ability to undo incorrect actions and replace them by correct ones, makes proof development in Harpoon closely resemble proof development on paper. While users develop proofs, Harpoon builds a proof script that, upon completion of the proof, gets translated into a Beluga program which is then type-checked ensuring validity. Harpoon has been applied to a range of examples that cover all features supported by Beluga.

Tactics

The tactics within Harpoon mimic the proof steps often taken in informally written proofs. Harpoon supports building proofs via forwards and backwards reasoning, induction, and case-analysis. A few of the main tactics are presented below.

$$\text{Tactics } T ::= \texttt{msplit}\ X \mid \texttt{split}\ I \mid \texttt{by}\ I\ \texttt{as}\ y \mid \texttt{suffices by}\ I\ \texttt{to show}\ \tau \mid \texttt{solve}\ E$$

Figure 2.5: Harpoon tactic language.

There are two splitting tactics, `msplit` and `split` which split the specified (meta-) variable into a covering set of cases, initiating case analysis. If the variable was specified as an inductive variable, then induction hypothesis are automatically generated for each subcase. To introduce a lemma or appeal to an induction hypothesis, `by` is used and binds the result to some computational variable. Optionally, users can instead add the assumption to the meta-context by adding `unboxed` to the end of their `by` call. To conduct backwards-reasoning, users invoke `suffices` on a constructor or lemma. Users finish the proof a subgoal by providing the respective proof term as argument to the tactic `solve`.

As an option, there are tactics that get employed automatically depending on the goal. For example, if the goal is ever a function type (dependent or ordinary) then the tactic `auto-intros` is automatically deployed which strips the assumptions off the goal and adds them to their respective contexts. Further, if the goal can be trivially solved simply by using an assumption in the contexts, `auto-solve-trivial` will finish the proof automatically.

Interactive Proof Development

We walk-through how the theorem `bwd_closed` from Section 2.2.2 would be constructed within Harpoon. The proof-session begins by loading the file containing our formalized theory of the simply typed lambda-calculus and its meta-theory which consists of the lemma `halts_step`. We are then asked by Harpoon to provide a theorem name and statement, along with the index of the argument we will perform induction on (if any).

```
Name of theorem: bwd_closed
Statement of theorem: {A:[ ⊢ tp]} {M: [ ⊢ term A]} {M': [ ⊢ term A]} [ ⊢ step M M']
                            → Reduce [ ⊢ A] [ ⊢ M'] → Reduce [ ⊢ A] [ ⊢ M]
Induction order (empty for none): 1
```

Harpoon then asks us if there is another theorem name we would like to provide. This is used when we would like to prove multiple statements via mutual induction. Since we do not, we leave this blank. Harpoon presents us with our initial goal state, whereby assumptions have been automatically collected and placed in the respective contexts. Our goal is therefore `Reduce [ ⊢ A] [ ⊢ M]`.

```
Meta-context:
  A : (⊢  tp)
  M : ( ⊢ term A)
  M' : ( ⊢ term A)


Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ A] [ ⊢ M']
-----------------------------------
Reduce [ ⊢ A] [ ⊢ M]
> msplit A
```

```
(case arr)
Meta-context:
  T : ( ⊢ tp)
  T1 : ( ⊢ tp)
  M : ( ⊢ term (arr T T1))
  M' : ( ⊢ term (arr T T1))

Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ arr T T1] [ ⊢ M']
-----------------------------------
Reduce [ ⊢ arr T T1] [ ⊢ M]
> unbox s as [ ⊢ S]
```

```
(case arr)
Meta-context:
  T : ( ⊢ tp)
  T1 : ( ⊢ tp)
  M : ( ⊢ term (arr T T1))
  M' : ( ⊢ term (arr T T1))
  S : ( ⊢ step M M')

Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ arr T T1] [ ⊢ M']
-----------------------------------
Reduce [ ⊢ arr T T1] [ ⊢ M]
> invert r
```

The first few steps follow exactly as in our written proof. We begin by splitting on the meta-variable `A`. We are presented with the induction case first, namely `arr` and our new goal, `Reduce [ ⊢ arr T T1] [ ⊢ M]`. We then unbox our boxed assumption `s`, introducing the new meta-variable `S` into the meta-context.

```
(case arr)
Meta-context:
  T : ( ⊢ tp)
  T1 : ( ⊢ tp)
  M : ( ⊢ term (arr T T1))
  M' : ( ⊢ term (arr T T1))
  S : ( ⊢ step M M')


Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ arr T T1] [ ⊢ M']
  f : {N : ( ⊢ term T)}
      Reduce [ ⊢ T] [ ⊢ N] →
      Reduce [ ⊢ T1] [ ⊢ app M' N]
  h : [ ⊢ halts M']

-----------------------------------
Reduce [ ⊢ arr T T1] [ ⊢ M]
> suffices by Arr toshow _, _
```

```
(case arr)
Meta-context:
  T : ( ⊢ tp)
  T1 : ( ⊢ tp)
  M : ( ⊢ term (arr T T1))
  M' : ( ⊢ term (arr T T1))
  S : ( ⊢ step M M')
  N : ( ⊢ term T)

Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ arr T T1] [ ⊢ M']
  f : {N1 : ( ⊢ term T)}
      Reduce [ ⊢ T] [ ⊢ N1] →
      Reduce [ ⊢ T1] [ ⊢ app M' N1]
  h : [ ⊢ halts M']
  r' : Reduce [ ⊢ T] [ ⊢ N]
-----------------------------------
Reduce [ ⊢ T1] [ ⊢ app M N]
> solve (bwd_closed _ _ _
           [⊢  stepapp S]
            (f [⊢  N] r'))
```

```
(case arr)
Meta-context:
  T : ( ⊢ tp)
  T1 : ( ⊢ tp)
  M : ( ⊢ term (arr T T1))
  M' : ( ⊢ term (arr T T1))
  S : ( ⊢ step M M')


Computational context:
  s : [ ⊢ step M M']
  r : Reduce [ ⊢ arr T T1] [ ⊢ M']
  f : {N : ( ⊢ term T)}
      Reduce [ ⊢ T] [ ⊢ N] →
      Reduce [ ⊢ T1] [ ⊢ app M' N]
  h : [ ⊢ halts M']

-----------------------------------
[ ⊢ halts M]
> solve (halts_step s h)
```

After inverting `r` we are presented with two new assumptions, `f` and `h`. We perform backwards reasoning by invoking suffices on the term `Arr` which can only be used once we solve the two subgoals, the first being: `Reduce [ ⊢ T1] [ ⊢ app M N]`. We appeal to one of the induction hypotheses by making a recursive call to `bwd_closed`, omitting arguments that Beluga can infer during type reconstruction. Our second subgoal, `[ ⊢ halts M]` can easily be discharged by applying the `halts_step` lemma.

```
(case b)
Meta-context:
  M : ( ⊢ term b)
  M’ : ( ⊢ term b)

Computational context:
  s : [ ⊢ step M M’]
  r : Reduce [ ⊢ b] [ ⊢ M’]


____________________________________
Reduce [ ⊢ b] [ ⊢ M]
> invert r
```

```
(case b)
Meta-context:
  M : ( ⊢ term b)
  M’ : ( ⊢ term b)

Computational context:
  s : [ ⊢ step M M’]
  r : Reduce [ ⊢ b] [ ⊢ M’]
  h : [ ⊢ halts M’]

____________________________________
Reduce [ ⊢ b] [ ⊢ M]
> solve (I (halts_step s h))
```

Our base case (`[ ⊢ A]` = `[ ⊢ b]`) is solved exactly as in the user-supplied proof. At the completion of the proof we are presented with the proof script, and translated Beluga program.

Harpoon makes it simple for users to interact with Beluga’s proof engine and allows them to develop proofs similarly to how they would develop them on paper. Unfortunately, much human interaction is still required to construct a proof even for simple lemmas like `bwd_closed`.

2.3 Other Meta-Theoretic Proof Assistants

We separate our discussion of meta-theoretic proof assistants based on those that use HOAS encodings and those that do not. Those most similar to Beluga being Twelf [Pfenning and Schürmann, 1999] and Abella [Gacek, 2008] utilize HOAS. We also discuss the more established proof assistants Coq [Bertot and Castéran, 2004] and Isabelle/HOL [Paulson, 1994].

There is a significant difference in the automation efforts between established general purpose proof systems and those supporting HOAS encodings. As we will see, automation in Twelf and Abella is less sophisticated than in Coq or Isabelle. As such, it is therefore of interest to investigate automated reasoning over HOAS encodings and their meta theories.

2.3.1 HOAS-Based Languages

Twelf Proof Assistant

Twelf currently provides the most automation out of all HOAS-based proof assistants designed to formalize programming languages, although for a fragment that is more restrictive than all others. Its fully automatic and simple proof search loop has been used to prove many interesting examples, including the Church-Rosser theorem for the untyped lambda-calculus and cut-elimination for full first-order intutionistic logic [Schürmann, 2000]. Its search loop alternates between three stages that Pfenning and Schürmann call filling, recursion, and splitting [Pfenning and Schürmann, 2002]. It begins with filling; a bounded direct search for witnesses to existential quantifiers. If the goal is not solved it proceeds to recursion, in which induction hypotheses are applied (with smaller arguments). Finally the loop moves to bounded splitting; where a heuristic is used to determine which variable to conduct a case split on. The loop continues on each subgoal and finishes with either completion or failure, which occurs when none of the stages are possible.

A drawback to having a completely automated theorem prover is that proving capabilities of the system are limited by its heuristic choices. Further, the system does not support backtracking and commits to the decisions it makes, even if they are the wrong ones. Therefore there is no way to prove a theorem for which the search loop fails on. Adding support for backtracking and interactive theorem proving would aid users when constructing more difficult proofs.

As previously mentioned, Twelf's prover does not produce proof terms and therefore provides no way to verify its proofs. Such a choice requires more trust from users.

Another important design choice to consider is contexts. Although directly supported, contexts are kept implicit throughout a derivation in Twelf. Therefore derivations maintain a single implicit context across their developments. Consequently, theorems which reference terms dependent on different contexts are inexpressible.

The logic of Twelf is more restrictive compared to Beluga. Firstly, it is only capable of expressing Π_2 statements, which are simple implications from universally quantified arguments to existentially quantified outputs. Secondly, there is no way to define recursive data-type definitions about LF objects, as Twelf's logic does not support inductive or co-inductive types. Therefore it cannot present proofs that proceed via logical relations, as statements require recursive definitions and nesting of implications. Logical relations are a common proof technique in PL theory, and can sometimes provide the most direct proofs, as was the case in the extensional equivalence proof involving singleton types for the mechanization of Standard ML [Stone and Harper, 2006; Lee et al., 2007; Abel et al., 2019]. Lastly, there is no support to reason about open (contextual) LF objects or simultaneous substitutions.

Abella Proof Assistant

Abella is another proof assistant with the capabilities to mechanize meta-theoretic proofs about programming languages [Gacek, 2008]. It is based on $\mathcal{G}$, an intuitionistic higher-order logic with (co)induction using fixed-points [Gacek et al., 2008]. Formal systems are specified within an implementation of the logic of hereditary Harrop formulas in $\mathcal{G}$, which allows for specifications using HOAS. Reasoning is performed over these encodings using the nabla (∇) quantifier of $\mathcal{G}$, a generalization of the $\forall$ operator to model variable binding. Semi-automation exists in Abella provided in the form of tactics that are similar to the ones implemented in Harpoon. Expanding automation using a focusing proof search strategy has been proposed in [Chaudhuri et al., 2018] but has yet to be implemented.

Contexts and (simultaneous) substitutions in Abella are not treated as first-class like they are in Beluga. This means that in order to use such constructs, users must manually define them and any properties they wish to utilize about them (e.g. context weakening). These can further add to the complexity of formal theorem proving. Abella also does not construct proof objects during proof development so there is no way to independently type-check proofs. Thus users have no way to determine if their proofs are valid.

2.3.2 Other Languages

The proof assistants surveyed below do not contain built-in methods to handle encoding languages with variable binders as Beluga, Twelf, or Abella do. Instead, these tools depend on extensions (like packages and libraries) which allow them to handle such encodings.

Coq Proof Assistant

Perhaps the most well-known proof assistant, the logic of Coq is based on the Calculus of Inductive Constructions [Bertot and Castéran, 2004]. Although this rich higher-order logic supports dependent types, inductive definitions, pattern matching, and structural recursion, it does not have much built-in support for meta-theoretical reasoning; contexts must be handled explicitly and HOAS encodings are not supported. However there are many extensions and libraries to Coq which allow for specialized mechanizations and in different logics, like Hybrid [Ambler et al., 2002; Momigliano et al., 2008] and Autosubst [Schäfer et al., 2015]. Hybrid aims to provide support for encoding formal systems using full HOAS and Autosubst is a library for working with de Bruijn indices and parallel substitutions. Although Coq's library approach offers more flexibility, there is much more to learn. Therefore it may benefit some users, especially novice ones, to construct their mechanizations in smaller systems like Beluga or Twelf. On the other-hand Coq provides polymorphism unlike Beluga which allows for reuse of code.

On the theorem-proving side, Coq has extensive tactic languages and proof term construction capabilities. As users develop proofs using tactics, Coq builds a proof term from a set of terms that represent each primitive inference. Once completed, the proof term is checked independently by Coq's type checker, which reduces to verifying the correctness of each individual primitive inference.

The tactics of HARPOON are largely inspired by those in Coq. Both allow for forward and backward reasoning, introduction of assumptions, case analysis, and inductive reasoning. A big difference between the two systems is that Coq also supports the use of *tacticals*. These are tactics which act on other tactics. That is, we can actually build our own tactics from primitive ones. As an example, the `repeat` tactical repeatedly applies the supplied tactic until it fails. The automation status of Coq is impressive given its application to a wide range of mechanizations. Currently automated proof search tactics (auto, eauto, iauto and jauto) do not conduct any case analysis (including inversions), inductions, or rewritings, and are intended to finish a proof instead of complete it entirely [Pierce et al., 2022].

Isabelle/HOL Proof Assistant

Similar to Coq, Isabelle is a generic proof assistant that provides a meta-logic which can be used to encode various logics including higher-order logic (HOL) making it suitable to prove meta-theoretic properties about formal systems [Paulson, 1994]. Isabelle/HOL is the generalization of Isabelle for HOL and is one of the many implementations of HOL [Gordon and Melham, 1993; Harrison, 2009]. Like Coq, HOAS encodings are not directly supported but instead added through the Hybrid system [Ambler et al., 2002].

Users have the option, and are encouraged, to develop proofs within Isar (Intelligent semi-automated reasoning), which allows for the construction of more human-readable proofs [Wenzel, 1999]. Otherwise, users can use the standard technique of developing proofs using a set of low-level single-step tactics, similar to those in Coq. Users also have the ability to build their own tactics through the use of tacticals.

For much of its automation, Isabelle elicits the help of several external solvers. Sledgehammer [Paulson and Susanto, 2007], for example, takes a goal and heuristically chooses from Isabelle's libraries containing various lemmas, definitions, and axioms, a few hundred applicable ones to perform search over. Then, translates the goal and each of these assumptions to SMT (first-order logic) and sends the query off to an external SMT or resolution-based solver. It can also generate counterexamples using Quickcheck [Claessen and Hughes, 2000] and Nitpick's [Blanchette and Nipkow, 2010] countermodel generation and testing capabilities. In its own system, Isabelle performs various general-purpose proof search methods which help discharge simple parts of a proof allowing users to focus on the main ones [Blanchette et al., 2011]. They also have several strengthened endgame tactics which are meant to finish a proof but perform slower due to the increased automation and provide no hints upon failing.

Chapter 3

Theoretical Foundation of Proof Search in Beluga

Full automation for Beluga requires automatic case-splitting, generation of induction hypotheses if necessary, and, since the underlying logic of Beluga is a two-level logic, proof search on each of these two levels. This is a significant task. As such, in this thesis I only be focus on the fragment of Beluga that has been semi-automated. This fragment excludes substitution and parameter variables, block context schemas, inductive types, recursion, and pattern-matching. For a more in depth discussion of these features in Beluga, readers are invited to checkout these papers [Cave and Pientka, 2012; Pientka, 2008].

3.1 The Beluga Language

We begin by presenting the core of Beluga's specification logic: a dependently-typed version of contextual LF [Pientka, 2008]. We say an LF term is *canonical* if no beta- or eta-redexes are possible. Following [Harper and Pfenning, 2005], we only characterize canonical objects due to the fact that LF encodings admit a compositional bijection between object-level terms and the canonical LF objects, ensuring adequate encodings. We separate the presentation of contextual LF into two grammars, one for an extended LF and one for the meta-level.

Kinds	K	$::=$	$\mathbf{type} \mid \Pi x : A.\, K$
Atomic Types	P, Q	$::=$	$\mathbf{a}\overrightarrow{M}$
Types	A, B	$::=$	$P \mid \Pi x : A.\, B$
Neutral Terms	R	$::=$	$x \mid \mathbf{c} \mid RN \mid u[\sigma]$
Normal Terms	M, N	$::=$	$R \mid \lambda x.\, M$
Substitutions	σ	$::=$	$\cdot \mid id_\psi \mid \sigma, M$
Signatures	Σ	$::=$	$\cdot \mid \Sigma,\ \mathbf{c} : A \mid\ \mathbf{a} : K$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \Psi, x : A \mid \psi$
Contextual Variables	X	$::=$	$u[\sigma] \mid \psi$

Figure 3.1: Grammar of LF with contextual variables.

We extend LF terms to include meta-variables $u[\sigma]$. Meta-variables denote possibly open objects that come paired with a post-poned simultaneous substitution σ (by convention written to the right of a term) that gets applied as soon as we know what the variable u stands for. We separate terms into two categories, neutral and normal. We characterize neutral terms to be those that do not cause beta-redexes when they are applied in function application. Terms are classified by types, and are either type constants $\mathbf{a}$ that may be indexed by terms $M_1, .., M_n$ or dependent types. We write $A \to B$ in place of $\Pi x : A.B$ when x does not occur in B.

Simultaneous substitutions σ provide a mapping from one context of variables Φ to another Ψ. We do not always make the domain of the substitution explicit, but one can think of the i-th element of σ corresponding to the i-th declaration in Φ. We write $[\sigma]_\Psi$ for the substitution with domain Ψ. We assume all substitutions are hereditary substitutions [Watkins et al., 2004]. Hereditary substitutions continue to reduce terms in the presence of beta-reductions, which guarantees that the resulting type be in beta-normal form. They are also by definition capture-avoiding.

Variables in a contextual LF expression may be bound by one of two contexts. There is the LF context Ψ that holds typings for ordinary variables, and there are is a meta-context Δ (introduced in Figure 3.3) which holds typings for contextual variables, uniformly denoted by X. Contextual variables include meta-variables $u[\sigma]$ and context variables ψ. Context variables are an interesting aspect of this logic, providing a way to abstract over contexts which is required for recursion over HOAS specifications.

Following Pientka and Dunfield [2008] and Beluga's implementation, we present a bi-directional type system where types are synthesized for neutral terms and checked against normal terms. judgments have access to two explicit contexts, Δ and Ψ, as well as the implicit signature Σ. Variable and constant declarations in contexts and signatures are unique, therefore when we write $\Psi, x : A$ we assume x is not already declared in Ψ. We present the principal judgments below. We omit judgments on types, kinds, contexts, and definitional equality.

$\Delta; \Psi \vdash M \Leftarrow A$	Normal term M checks against type A
$\Delta; \Psi \vdash R \Rightarrow A$	Neutral term R synthesizes type A
$\Delta; \Psi \vdash \sigma \Leftarrow \Phi$	Substitution σ has domain Φ and range Ψ

The typing rules for LF extended with contextual variables are given in Figure 3.2 and are straightforward. All contexts are ordered due to type dependencies and we assume they are well-formed. We make use of hereditary substitutions [Watkins et al., 2004], written $[N/x]_A B$.

We pair LF objects with contexts, creating contextual objects. From Ψ we construct $\hat{\Psi}$, which contains only variable names. The contextual object $(\hat{\Psi} \vdash R)$ describes a neutral LF term R whose free LF ordinary variables are bound by $\hat{\Psi}$, thus all contextual objects are closed. Contextual objects are useful because they allow us to write compact and elegant proofs since we can store an entire hypothetical derivation within one (closed) object.

In order to uniformly abstract over meta-objects, we lift contextual LF objects to meta-

Neutral Terms $\boxed{\Delta; \Psi \vdash R \Rightarrow A}$

$$\frac{\Sigma(c) = A}{\Delta; \Psi \vdash c \Rightarrow A} \qquad \frac{\Psi(x) = A}{\Delta; \Psi \vdash x \Rightarrow A} \qquad \frac{\Delta; \Psi \vdash R \Rightarrow \Pi x : A.B \quad \Delta; \Psi \vdash N \Leftarrow A}{\Delta; \Psi \vdash RN \Rightarrow [N/x]_A B}$$

$$\frac{\Delta(u) = (\Phi \vdash P) \quad \Delta; \Psi \vdash \sigma \Leftarrow \Phi}{\Delta; \Psi \vdash u[\sigma] \Rightarrow [\sigma]_\Phi P}$$

Normal Terms $\boxed{\Delta; \Psi \vdash M \Leftarrow A}$

$$\frac{\Delta; \Psi \vdash R \Rightarrow P \quad P = Q}{\Delta; \Psi \vdash R \Leftarrow Q} \qquad \frac{\Delta; \Psi, x : A \vdash M \Leftarrow B}{\Delta; \Psi \vdash \lambda x.M \Leftarrow \Pi x : A.B}$$

Substitutions $\boxed{\Delta; \Psi \vdash \sigma \Leftarrow \Phi}$

$$\frac{}{\Delta; \Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta; \Psi \vdash \sigma \Leftarrow \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]_\Phi A}{\Delta; \Psi \vdash \sigma, M \Leftarrow \Phi, x : A} \qquad \frac{}{\Delta; \psi, \Psi \vdash id_\psi \Leftarrow \psi}$$

Figure 3.2: Typing rules for LF with contextual variables.

Context Schemas	G	$::=$	$\exists\overrightarrow{(x : A_o)}.\ A \mid G + \exists\overrightarrow{(x : A_o)}.\ A$
Meta Terms	C	$::=$	$(\hat{\Psi} \vdash R) \mid \Psi$
Meta Types	U	$::=$	$(\Psi \vdash P) \mid G$
Meta Substitutions	θ	$::=$	$\cdot \mid \theta, C/X$
Meta Contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$

Figure 3.3: Grammar of meta-level.

types U and meta-terms C. Our meta language's terms include contextual terms as well as LF contexts. The meta-type $(\Psi \vdash P)$ denotes the type of a meta-variable u and stands for a contextual term. Context schemas G are constructed from schema elements $\exists\overrightarrow{(x : A_o)}.A$ using $+$. We say a context Ψ checks against schema G if each of the declarations in Ψ are an instance of an element in G. As an example, the context `x: exp nat, y: exp bool` checks against the schema $\exists T :$ `tp. exp` T.

$\Delta \Vdash C \Leftarrow U$ Check meta-term C against meta-type U in meta-context Δ

$\Delta \Vdash \theta \Leftarrow \Delta'$ Check that meta-substitution θ has domain Δ' and range Δ

Meta Terms $\boxed{\Delta \Vdash C \Leftarrow U}$

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \Vdash (\hat{\Psi} \vdash R) \Leftarrow (\Psi \vdash P)} \qquad \frac{}{\Delta \Vdash \cdot \Leftarrow G} \qquad \frac{\Delta(\psi) = G}{\Delta \Vdash \psi \Leftarrow G}$$

$$\frac{\Delta \Vdash \Psi \Leftarrow G \quad \exists\overrightarrow{(x : B')}.B \in G \quad A = [\sigma] \xrightarrow[(x:B')]{} B \quad \Delta; \Psi \vdash \sigma \Leftarrow \overrightarrow{(x : B')}}{\Delta \Vdash \Psi, x : A \Leftarrow G}$$

Meta-Substitutions $\boxed{\Delta \Vdash \theta \Leftarrow \Delta'}$

$$\frac{}{\Delta \Vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \Vdash \theta \Leftarrow \Delta' \quad \Delta \Vdash C \Leftarrow [\![\theta]\!]_{\Delta'} U}{\Delta \Vdash \theta, C/X \Leftarrow \Delta', X : U}$$

Figure 3.4: Typing rules for meta-level.

In order to check that a meta-term has a valid meta-type, we revert to LF type checking. LF contexts must be well-formed and check against a context schema.

The single meta-substitution $[\![C/X]\!]_U(*)$, where $* = A, M, \sigma,$ or Ψ is defined inductively on the structure of X. The most common case is when X stands for a meta-variable u and so C stands for a contextual-object $(\hat{\Psi} \vdash R)$. This substitution gets pushed through λ-expressions and gets applied only when we reach a meta-variable $u[\sigma]$. In this case, we apply the meta-substitution $[\![(\hat{\Psi} \vdash R)/u]\!]$ to σ, obtaining $R[\sigma']$, and then finally apply σ' to R. We also have the notion of simultaneous substitutions for meta-substitutions, denoted $[\![\theta]\!]_\Delta U$. More discussion on simultaneous substitutions and the full definitions of meta-substitutions can be found previously described in Cave and Pientka [2012]; Pientka [2008].

We present some properties about meta-substitutions with respect to the typing systems above.

Lemma 1 (Meta-Substitution Properties for Contextual LF Typings)**.**
a) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X : U; \Psi \vdash R \Rightarrow A$ *then* $\Delta; [\![C/X]\!]\Psi \vdash [\![C/X]\!]R \Rightarrow [\![C/X]\!]A$
b) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X : U; \Psi \vdash M \Leftarrow A$ *then* $\Delta; [\![C/X]\!]\Psi \vdash [\![C/X]\!]M \Leftarrow [\![C/X]\!]A$
c) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X : U; \Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Delta; [\![C/X]\!]\Psi \vdash [\![C/X]\!]\sigma \Leftarrow [\![C/X]\!]\Phi$

Proof. By mutual induction on the structure of the second typing derivation. □

On top of the specification layer is the reasoning layer, whose logic is comparable to a dependently typed first-order logic. It is here that we omit inductive types, recursion, and pattern-matching.

Types	τ	$::=$	$[\Psi \vdash P] \mid \tau_1 \to \tau_2 \mid \Pi^{\Box} X : U.\, \tau$
Synthesized Expressions	I	$::=$	$y \mid I\ E \mid I\ \lceil C \rceil \mid (E : \tau)$
Checked Expressions	E	$::=$	$I \mid [\hat{\Psi} \vdash R] \mid \text{fn}\ y.E \mid \lambda^{\Box} X.E \mid \text{let box}\ X = I\ \text{in}\ E$
Computation-level Contexts	Γ	$::=$	$\cdot \mid \Gamma, y : \tau$

Figure 3.5: Grammar of the computational logic.

This layer, also called the computational layer, is used to describe the programs that operate on data. The computation types include atomic boxed-types $[\Psi \vdash P]$, computation level function abstraction, as well as abstraction over various contextual objects.

We separate computations based on whether we synthesize their types or check them against types. Ordinary functions are created using fn $y.E$ and applied using $I\ E$. Dependent functions are created by abstracting over meta-objects $\lambda^{\Box} X.E$ and applied to meta-objects using $I\ \lceil C \rceil$.

$\Delta; \Gamma \Vdash I \Rightarrow \tau$	Expression I synthesizes type τ
$\Delta; \Gamma \Vdash E \Leftarrow \tau$	Expression E checks against type τ

Synthesized Expressions $\boxed{\Delta; \Gamma \Vdash I \Rightarrow \tau}$

$$\frac{\Gamma(y) = \tau}{\Delta; \Gamma \Vdash y \Rightarrow \tau} \qquad \frac{\Delta; \Gamma \Vdash I \Rightarrow \tau' \to \tau \quad \Delta; \Gamma \Vdash E \Leftarrow \tau'}{\Delta; \Gamma \Vdash I\ E \Rightarrow \tau}$$

$$\frac{\Delta; \Gamma \Vdash I \Rightarrow \Pi^{\square} X : U.\tau \quad \Delta \Vdash C \Leftarrow U}{\Delta; \Gamma \Vdash I\lceil C \rceil \Rightarrow [\![C/X]\!]_U \tau} \qquad \frac{\Delta; \Gamma \Vdash E \Leftarrow \tau}{\Delta; \Gamma \Vdash (E : \tau) \Rightarrow \tau}$$

Checked Expressions $\boxed{\Delta; \Gamma \Vdash E \Leftarrow \tau}$

$$\frac{\Delta; \Gamma \Vdash I \Rightarrow \tau \quad \tau = \tau'}{\Delta; \Gamma \Vdash I \Leftarrow \tau'} \qquad \frac{\Delta \Vdash (\hat{\Psi} \vdash R]) \Leftarrow (\Psi \vdash P)}{\Delta; \Gamma \Vdash [\hat{\Psi} \vdash R] \Leftarrow [\Psi \vdash P]} \qquad \frac{\Delta; \Gamma, y : \tau_1 \Vdash E \Leftarrow \tau_2}{\Delta; \Gamma \Vdash \mathrm{fn}\ y.E \Leftarrow \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Gamma \Vdash I \Rightarrow [\Psi \vdash P] \quad \Delta, X : (\Psi \vdash P); \Gamma \Vdash E \Leftarrow \tau}{\Delta; \Gamma \Vdash \text{let box } X = I \text{ in } E \Leftarrow \tau} \qquad \frac{\Delta, X : U; \Gamma \Vdash E \Leftarrow \tau}{\Delta; \Gamma \Vdash \lambda^{\square} X.E \Leftarrow \Pi^{\square} X : U.\tau}$$

$$\frac{\Delta; \Gamma \Vdash I \Rightarrow \tau' \quad \Delta; \Gamma, y : \tau' \Vdash E \Leftarrow \tau}{\Delta; \Gamma \Vdash [I/y]E \Leftarrow \tau}$$

Figure 3.6: Typing rules for computations.

The typings for computations are mostly all straightforward. An interesting case is checking a boxed meta-term $[\hat{\Psi} \vdash R]$ against a boxed meta-type $[\Psi \vdash P]$ in two contexts Δ and Γ. Since meta-objects can only depend on other meta-objects, this simply reverts to checking $(\hat{\Psi} \vdash R)$ in the presence of the meta-context.

3.2 Sequent Calculus

Given the grammar for Beluga's core in the previous chapter we now wish to assign meaning to the connectives with respect to the rules of deduction, and ultimately describe Beluga's proof system.

The sequent calculus is a style for presenting the rules of a proof system, first introduced by Gerhard Gentzen in 1935 [Gentzen, 1935; Szabo, 1969]. It is ideal for presenting inference rules which are to be implemented in proof assistants because of its characterization of *normal* proofs, which limits the non-determinism in proof search, and is preferred to natural deduction due to its handling of assumptions. As such, we present our proof system using two sequent calculi.

On the specification level, we encode propositions into hereditary Harrop formulas [Miller et al., 1991]. Viewing types as propositions allows us to assign a logic programming interpretation to LF types. Atomic types $\mathbf{a}\overrightarrow{M}$ correspond to atomic propositions, non-dependent function types $A \to B$ correspond to implications, and dependent function-types $\Pi x : A.\ B$ correspond to universal statements. As such, moving forward we distinguish between dependent and non-dependent function types in LF. The fragment corresponding to hereditary Harrop formulas used for logic programming in contextual LF is thus:

$$\begin{array}{llcl} \text{Types} & A, B & ::= & P \mid A \to B \mid \Pi x : A.\ B \\ \text{Environment} & \Psi & ::= & \cdot \mid \Psi, x : A \\ \text{Contextual Environments} & \Delta & ::= & \cdot \mid \Delta, X : (\Psi \vdash P) \end{array}$$

Recall, P ranges over atomic propositions $\mathbf{a}\overrightarrow{M}$. We extend typical logic programming over LF with the addition of *contextual environments* which hold contextual LF assumptions. In order to use such an assumption $(\Psi \vdash P)$, we must find a substitution from Ψ to the local context in which it will be used. We can interpret this substitution as a *verification* of Ψ. We build proof terms, representing proof *witnesses*, simultaneously when developing proofs.

We present the sequent calculus for contextual LF, which is based off of the sequent calculus for intuitionistic contextual modal logic presented in Nanevski et al. [2008]. Contexts take the traditional form as ordered lists. As the order in our contexts matter (due to type dependencies), they only allow for weakening and contraction. We say that everything to the left of $\Longrightarrow$ can be used to construct a proof of the proposition(s) to the right of $\Longrightarrow$. We assume all sequents are well-formed.

The right rules introduce variable declarations into the local context Ψ. However, those introduced via the ΠR rule are simply parameters that are not used during proof search,

$\Delta; \Psi \Longrightarrow M : A$	M is a proof of the proposition A using assumptions from Δ and Ψ
$\Delta; \Psi \Longrightarrow \sigma : \Phi$	σ is a proof of the propositions in Φ using assumptions from Δ and Ψ

$$\frac{\mathbf{c} : A \in \Sigma}{\Delta; \Psi \Longrightarrow \mathbf{c} : A}\ \text{init}^{\Sigma} \qquad \frac{}{\Delta; \Psi, x : A \Longrightarrow x : A}\ \text{init}^{\Psi} \qquad \frac{\Delta; \Psi, x : A \Longrightarrow M : B}{\Delta; \Psi \Longrightarrow \lambda x.M : \Pi x : A.\ B}\ \Pi R$$

$$\frac{\Delta; \Psi, x_1 : \Pi x : A.\ B \vdash M \Leftarrow A \quad \Delta; \Psi, x_1 : \Pi x : A.\ B, x_2 : [M/x]B \Longrightarrow N : A'}{\Delta; \Psi, x_1 : \Pi x : A.\ B \Longrightarrow [x_1 M/x_2]N : A'}\ \Pi L$$

$$\frac{\Delta; \Psi, x_1 : A \to B \Longrightarrow M : A \quad \Delta; \Psi, x_1 : A \to B, x_2 : B \Longrightarrow N : A'}{\Delta; \Psi, x_1 : A \to B \Longrightarrow [x_1 M/x_2]N : A'} \to L$$

$$\frac{\Delta; \Psi, x : A \Longrightarrow M : B}{\Delta; \Psi \Longrightarrow \lambda x.M : A \to B} \to R \qquad \frac{}{\Delta; \Psi \Longrightarrow \cdot : \cdot}\ \text{sub}_1$$

$$\frac{}{\Delta; \psi, \Psi \Longrightarrow id_{\psi} : \psi}\ \text{sub}_2 \qquad \frac{\Delta; \Psi \Longrightarrow \sigma : \Phi \quad \Delta; \Psi \Longrightarrow N : [\sigma]B}{\Delta; \Psi \Longrightarrow (\sigma, N) : (\Phi, x : B)}\ \text{sub}_3$$

$$\frac{\Delta, u : (\Phi \vdash P); \Psi \Longrightarrow \sigma : \Phi \quad \Delta, u : (\Phi \vdash P); \Psi, x : [\sigma]P \Longrightarrow M : A}{\Delta, u : (\Phi \vdash P); \Psi \Longrightarrow [u[\sigma]/x]M : A}\ \text{reflect}$$

Figure 3.7: Sequent calculus for contextual LF.

unlike those introduced via $\to R$. Similarly, to use a universally quantified assumption (i.e. a dependent function type) as in ΠL, we require that M checks against type A. In practice, we do not search for the term M but introduce meta-variables for such universally quantified variables which are later instantiated via unification. In contrast, using an assumption of ordinary function type, as in $\to L$, involves searching for a proof term of type A.

In the *reflect* rule we may use the contextually valid assumption $(\Phi \vdash P)$ to deduce P in the context Ψ if we can verify Φ. In order to verify Φ we need to find a substitution which maps all the variables in Φ to terms that make sense in Ψ. There are several ways to construct such a substitution, depending on the shape of Φ. If it is empty, we simply use an empty substitution (as P is closed). If it is a context variable ψ and we simply want to use $(\psi \vdash P)$ in a weaker context, we apply the identity substitution. Otherwise, Φ contains a variable declaration $x : B$ which requires proof search in order to find a term in Ψ of type $[\sigma]B$ which will replace x in P.

We show some results which will be used in later proofs. First, to make derivations simpler, we show that the general init rule is admissible if we restrict the rule to apply only to atomic formulas.

Lemma 2 (Admissibility of init$^\Psi$)**.**
If $\Delta; \Psi, x : P \Longrightarrow P$ then for all $\Delta, \Psi,$ and A such that $\Delta \vdash \Psi$ and $\Delta; \Psi \vdash A$, we have $\Delta; \Psi, x : A \Longrightarrow A$

Proof. By induction on the structure of A. □

We show that our meta-substitution properties for contextual LF extend to the sequent calculus formulation.

Lemma 3 (Meta-Substitution Properties in the Sequent Calculus for Contextual LF)**.**
a) If $\Delta \Vdash C \Leftarrow U$ and $\Delta, X : U, \Delta'; \Psi \Longrightarrow M : A$ then
$\Delta, [\![C/X]\!]\Delta'; [\![C/X]\!]\Psi \Longrightarrow [\![C/X]\!]M : [\![C/X]\!]A$
b) If $\Delta \Vdash C \Leftarrow U$ and $\Delta, X : U, \Delta'; \Psi \Longrightarrow \Phi : \sigma$ then
$\Delta, [\![C/X]\!]\Delta'; [\![C/X]\!]\Psi \Longrightarrow [\![C/X]\!]\Phi : [\![C/X]\!]\sigma$

Proof. By mutual induction on the structure of the second sequent. Uses Lemma 1. □

To prove contextual cut, we also require a lemma for contextual LF typings, related to the sequent calculus.

Lemma 4 (More Meta-Substitution Properties for Contextual LF Typings)**.**
a) If $\Delta; \Psi \Longrightarrow R : P$ and $\Delta, X : (\Psi \vdash P) \Vdash C \Leftarrow U$ then
$\Delta \Vdash [\![(\hat{\Psi} \vdash R)/X]\!]C \Leftarrow [\![(\hat{\Psi} \vdash R)/X]\!]U$
b) If $\Delta; \Psi \Longrightarrow R : P$ and $\Delta, X : (\Psi \vdash P); \Phi \vdash R \Rightarrow A$ then

$\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Phi \vdash [\![(\hat{\Psi} \vdash R)/X]\!]R \Rightarrow [\![(\hat{\Psi} \vdash R)/X]\!]A$

c) If $\Delta; \Psi \Longrightarrow R : P$ *and* $\Delta, X : (\Psi \vdash P); \Phi \vdash M \Leftarrow A$ *then*

$\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Phi \vdash [\![(\hat{\Psi} \vdash R)/X]\!]M \Leftarrow [\![(\hat{\Psi} \vdash R)/X]\!]A$

d) If $\Delta; \Psi \Longrightarrow R : P$ *and* $\Delta, X : (\Psi \vdash P); \Phi \vdash \sigma \Leftarrow \Phi'$ *then*

$\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Phi \vdash [\![(\hat{\Psi} \vdash R)/X]\!]\sigma \Leftarrow [\![(\hat{\Psi} \vdash R)/X]\!]\Phi'$

Proof. By mutual induction on the structure of the second sequent. □

We turn our attention to proof search over computations. Computation-level contexts possess the properties of contraction and weakening, along with exchange. We assume all sequents are well-formed.

$$\Delta; \Gamma \Longrightarrow E : \tau \quad E \text{ is a proof of } \tau \text{ using assumptions from } \Delta \text{ and } \Gamma$$

$$\frac{}{\Delta; \Gamma, y : \tau \Longrightarrow y : \tau}\ \text{init}^{\Gamma} \qquad \frac{\Delta, X : U; \Gamma \Longrightarrow E : \tau}{\Delta; \Gamma \Longrightarrow \lambda^{\Box}X.E : \Pi^{\Box}X : U.\, \tau}\ \Pi^{\Box}R$$

$$\frac{\Delta \Vdash C \Leftarrow U \quad \Delta; \Gamma, y_1 : \Pi X : U.\, \tau', y_2 : [\![C/X]\!]\tau' \Longrightarrow E : \tau}{\Delta; \Gamma, y_1 : \Pi X : U.\, \tau' \Longrightarrow [y_1 \lceil C \rceil / y_2]E : \tau}\ \Pi^{\Box}L$$

$$\frac{\Delta; \Gamma, y : \tau_1 \Longrightarrow E : \tau_2}{\Delta; \Gamma \Longrightarrow \text{fn } y.\ E : \tau_1 \to \tau_2} \to R$$

$$\frac{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow E' : \tau_1 \quad \Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow E : \tau}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow [y_1 E' / y_2]E : \tau} \to L$$

$$\frac{\Delta \Longrightarrow (\hat{\Psi} \vdash R) : (\Psi \vdash P)}{\Delta; \Gamma \Longrightarrow [\hat{\Psi} \vdash R] : [\Psi \vdash P]}\ \Box R \qquad \frac{\Delta, X : (\Psi \vdash P); \Gamma, y : [\Psi \vdash P] \Longrightarrow E : \tau}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow \text{let box } X = y \text{ in } E : \tau}\ \Box L$$

Figure 3.8: Sequent calculus for the computation logic.

Our inference rules are mostly standard for a first-order logic. The $\Box R$ rule is the transition rule between contextual LF and computation-level proofs. In $\Box L$, we *unbox* a boxed assumption, thus adding it to Δ. We note that no assumptions in Δ are dependent on unboxed assumptions. Therefore we may exchange such assumptions within Δ. Using computation assumptions as in $\Pi^{\Box}L$ and $\rightarrow L$ is similar to how contextual LF assumptions are used. To use a universally quantified assumption (as in $\Pi^{\Box}L$), we require that C checks against U. Again, this term C is not explicitly constructed but found instead through unification.

We show more results which will be used later on. Similar to LF, we show that the general init rule is admissible if we restrict the rule to apply only to atomic formulas.

Lemma 5 (Admissibility of init$^{\Gamma}$)**.**
If $\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow [\Psi \vdash P]$ *then for all* Δ, Γ*, and* τ *such that* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma \vdash \tau$*, we have* $\Delta; \Gamma, y : \tau \Longrightarrow \tau$

Proof. By induction on the structure of τ. □

We present a property about meta-substitutions in the computation logic which will be used to prove cut elimination.

Lemma 6 (Meta-Substitution Property in the Sequent Calculus for the Computation Logic)**.**
If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X : U, \Delta'; \Gamma \Longrightarrow E : \tau$ *then*

$$\Delta, [\![C/X]\!]\Delta'; [\![C/X]\!]\Gamma \Longrightarrow [\![C/X]\!]E : [\![C/X]\!]\tau$$

Proof. By induction on the structure of the second sequent. Uses Lemma 3. □

We now show that the cut rule is admissible at both the meta-level and computation-level for the computation logic. These theorems were previously shown for contextual LF in [Nanevski et al., 2008], Theorem 3.1. Admissibility of cut has several corollaries including consistency of the logic. We prove the following theorem through mutual induction. For simplicity, when it is permitted we omit proof terms from our sequents.

Theorem 1 (Admissibility of Cut in the Computation Logic)**.**
*a) (**Cut**) If* $\Delta; \Gamma \overset{\mathcal{D}}{\Longrightarrow} \tau$ *and* $\Delta; \Gamma, y : \tau \overset{\mathcal{E}}{\Longrightarrow} \tau'$ *then* $\Delta; \Gamma \Longrightarrow \tau'$

*b) (**Contextual Cut**) If* $\Delta; \Psi \overset{\mathcal{D}}{\Longrightarrow} R : P$ *and* $\Delta, X : (\Psi \vdash \overset{\mathcal{E}}{P}), \Delta'; \Gamma \Longrightarrow \tau$ *then*
$\Delta, [\![(\hat{\Psi} \vdash R)/X]\!]\Delta'; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$

Proof of Theorem 1.a. The proof proceeds by structural induction on the cut formula τ and the derivations of the premises $\mathcal{D}$ and $\mathcal{E}$. In particular, we perform an outer induction over the structure of τ and an inner induction over the structures of $\mathcal{D}$ and $\mathcal{E}$. We appeal to the induction hypothesis using either a strictly smaller cut formula or an identical cut formula with only one strictly smaller derivation.

Case: $\mathcal{D}$ is an initial sequent.

$$\mathcal{D} = \frac{}{\Delta; \Gamma, y : \tau \Longrightarrow \tau}\ \text{init}^{\Gamma}$$

1. $\Delta; \Gamma, y : \tau, y' : \tau \Longrightarrow \tau'$ — Derivation $\mathcal{E}$ (assumption)
2. $\Delta; \Gamma, y : \tau \Longrightarrow \tau'$ — By contraction 1

Case: $\mathcal{E}$ is an initial sequent that uses the cut formula.

$$\mathcal{E} = \frac{}{\Delta; \Gamma, y : \tau \Longrightarrow \tau}\ \text{init}^{\Gamma}$$

$\Delta; \Gamma \Longrightarrow \tau$ — Derivation $\mathcal{D}$ (assumption)

Case: $\mathcal{E}$ is an initial sequent that doesn't use the cut formula.

$$\mathcal{E} = \frac{}{\Delta; \Gamma, y : \tau, y' : \tau' \Longrightarrow \tau'}\ \text{init}^{\Gamma}$$

$\Delta; \Gamma, y : \tau' \Longrightarrow \tau'$ — By init^{Γ}

Case: the cut formula τ is the principal formula of the final inference in both $\mathcal{D}$ and $\mathcal{E}$.

Subcase: $\tau = \Pi^{\Box} X : U.\, \tau$

$$\mathcal{D} = \dfrac{\Delta, X : \overset{\mathcal{D}'}{U}; \Gamma \Longrightarrow \tau}{\Delta; \Gamma \Longrightarrow \Pi^{\square} X : U.\, \tau}\ \Pi^{\square} R$$

and

$$\mathcal{E} = \dfrac{\Delta \Vdash \overset{\mathcal{E}_1}{C} \Leftarrow U \quad \Delta; \Gamma, y : \Pi^{\square} X : U.\, \overset{\mathcal{E}_2}{\tau}, y' : [\![C/X]\!]\tau \Longrightarrow \tau'}{\Delta; \Gamma, y : \Pi^{\square} X : U.\, \tau \Longrightarrow \tau'}\ \Pi^{\square} L$$

1. $\Delta; \Gamma, y' : [\![C/X]\!]\tau \Longrightarrow \Pi^{\square} X : U.\, \tau$	By weakening $\mathcal{D}$
2. $\Delta; \Gamma, y' : [\![C/X]\!]\tau \Longrightarrow \tau'$	By I.H. on $\Pi^{\square} X : U.\, \tau$, 1, and $\mathcal{E}_2$
3. $\Delta; [\![C/X]\!]\Gamma \Longrightarrow [\![C/X]\!]\tau$	By Lemma 6 on U, $\mathcal{E}_1$, and $\mathcal{D}'$
4. $\Delta; \Gamma \Longrightarrow [\![C/X]\!]\tau$	By fact X does not appear in Γ
5. $\Delta; \Gamma \Longrightarrow \tau'$	By I.H. on $[\![C/X]\!]\tau$, 4, and 2

Subcase: $\tau = \tau_1 \to \tau_2$

$$\mathcal{D} = \dfrac{\Delta; \Gamma, y_1 \overset{\mathcal{D}'}{:} \tau_1 \Longrightarrow \tau_2}{\Delta; \Gamma \Longrightarrow \tau_1 \to \tau_2}\ {\to} R$$

and

$$\mathcal{E} = \dfrac{\Delta; \Gamma, y : \tau_1 \overset{\mathcal{E}_1}{\to} \tau_2 \Longrightarrow \tau_1 \quad \Delta; \Gamma, y : \tau_1 \overset{\mathcal{E}_2}{\to} \tau_2, y_2 : \tau_2 \Longrightarrow \tau}{\Delta; \Gamma, y : \tau_1 \to \tau_2 \Longrightarrow \tau}\ {\to} L$$

1. $\Delta; \Gamma \Longrightarrow \tau_1$ By I.H. on $\tau_1 \to \tau_2$, $\mathcal{D}$, and $\mathcal{E}_1$
2. $\Delta; \Gamma \Longrightarrow \tau_2$ By I.H. on τ_1, 1, and $\mathcal{D}'$
3. $\Delta; \Gamma, y_2 : \tau_2 \Longrightarrow \tau_1 \to \tau_2$ By weakening $\mathcal{D}$
4. $\Delta; \Gamma, y_2 : \tau_2 \Longrightarrow \tau$ By I.H. on $\tau_1 \to \tau_2$, 3, and $\mathcal{E}_2$
5. $\Delta; \Gamma \Longrightarrow \tau$ By I.H. on τ_2, 2, and 4

Subcase: $\tau = [\Psi \vdash P]$

$$\mathcal{D} = \cfrac{\Delta; \Psi \overset{\mathcal{D}'}{\Longrightarrow} P}{\Delta; \Gamma \Longrightarrow [\Psi \vdash P]} \square R \qquad \mathcal{E} = \cfrac{\Delta, X : (\Psi \vdash P); \Gamma, y : [\Psi \vdash P] \overset{\mathcal{E}'}{\Longrightarrow} \tau'}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau'} \square L$$

1. $\Delta, X : (\Psi \vdash P); \Gamma \Longrightarrow [\Psi \vdash P]$ By weakening $\mathcal{D}$
2. $\Delta, X : (\Psi \vdash P); \Gamma \Longrightarrow \tau'$ By I.H. on $[\Psi \vdash P]$, 1, and $\mathcal{E}'$
3. $\Delta; \Gamma \Longrightarrow \tau'$ By Contextual Cut on $(\Psi \vdash P)$, $\mathcal{D}'$, and 2, and fact X does not appear in Γ and τ'

Case: the cut formula τ is not the principal formula of the final inference in $\mathcal{D}$.

Subcase: $$\mathcal{D} = \cfrac{\Delta \Vdash \overset{\mathcal{D}_1}{C} \Leftarrow U \quad \Delta; \Gamma, y_1 : \Pi^{\square} X : U.\, \overset{\mathcal{D}_2}{\tau''}, y_2 : [\![C/X]\!]\tau'' \Longrightarrow \tau}{\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\, \tau'' \Longrightarrow \tau} \Pi^{\square} L$$

1. $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\, \tau'', y_3 : \tau \Longrightarrow \tau'$ Derivation $\mathcal{E}$ (assumption)
2. $\Delta; \Gamma, y_1 \Pi^{\square} X : U.\, \tau'', y_2 : [\![C/X]\!]\tau'', y_3 : \tau \Longrightarrow \tau'$ By weakening and exchange $\mathcal{E}$
3. $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\, \tau'', y_2 : [\![C/X]\!]\tau'' \Longrightarrow \tau'$ By I.H. on τ, $\mathcal{D}_2$, and 2
4. $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\, \tau'' \Longrightarrow \tau'$ By $\Pi^{\square} L$ on $\mathcal{D}_1$ and 3

Subcase: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau_1} \quad \overset{\mathcal{D}_2}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow \tau}}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau} \to L$

1. $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_3 : \tau \Longrightarrow \tau'$	Derivation $\mathcal{E}$ (assumption)
2. $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2, y_3 : \tau \Longrightarrow \tau'$	By weakening and exchange $\mathcal{E}$
3. $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow \tau'$	By I.H. on τ, $\mathcal{D}_2$, and 2
4. $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau'$	By $\to L$ on $\mathcal{D}_1$ and 3

Subcase: $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Delta, X : (\Psi \vdash P); \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau}}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau} \Box L$

1. $\Delta; \Gamma, y : [\Psi \vdash P], y' : \tau \Longrightarrow \tau'$	Derivation $\mathcal{E}$ (assumption)
2. $\Delta, X : (\Psi \vdash P); \Gamma, y : [\Psi \vdash P], y' : \tau \Longrightarrow \tau'$	By weakening $\mathcal{E}$
3. $\Delta, X : (\Psi \vdash P); \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau'$	By I.H. on τ, $\mathcal{D}'$, and 2
4. $\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau'$	By $\Box L$ on 3

Case: the cut formula τ is not the principal formula of the final inference in $\mathcal{E}$.

Subcase: $\mathcal{E} = \dfrac{\overset{\mathcal{E}'}{\Delta, X : U; \Gamma, y : \tau \Longrightarrow \tau'}}{\Delta; \Gamma, y : \tau \Longrightarrow \Pi^{\Box} X : U.\, \tau'} \Pi^{\Box} R$

1. $\Delta; \Gamma \Longrightarrow \tau$	Derivation $\mathcal{D}$ (assumption)
2. $\Delta, X : U; \Gamma \Longrightarrow \tau$	By weakening $\mathcal{D}$
3. $\Delta, X : U; \Gamma \Longrightarrow \tau'$	By I.H. on τ, 2, and $\mathcal{E}'$
4. $\Delta; \Gamma \Longrightarrow \Pi^{\Box} X : U.\, \tau'$	By $\Pi^{\Box} R$ on 3

Subcase: $\mathcal{E} = \dfrac{\overset{\mathcal{E}'}{\Delta; \Gamma, y : \tau, y' : \tau_1 \Longrightarrow \tau_2}}{\Delta; \Gamma, y : \tau \Longrightarrow \tau_1 \to \tau_2} \to R$

1. $\Delta; \Gamma \Longrightarrow \tau$	Derivation $\mathcal{D}$ (assumption)
2. $\Delta; \Gamma, y' : \tau_1 \Longrightarrow \tau$	By weakening $\mathcal{D}$
3. $\Delta; \Gamma, y' : \tau_1 \Longrightarrow \tau_2$	By I.H. on τ, 2, and $\mathcal{E}'$ (with exchange)
4. $\Delta; \Gamma \Longrightarrow \tau_1 \to \tau_2$	By $\to R$ on 3

Subcase: $\mathcal{E} = \dfrac{\overset{\mathcal{E}'}{\Delta; \Psi \Longrightarrow P}}{\Delta; \Gamma, y : \tau \Longrightarrow [\Psi \vdash P]} \Box R$

$\Delta; \Gamma \Longrightarrow [\Psi \vdash P]$ By $\Box R$ on $\mathcal{E}'$

□

Proof of Theorem 1.b. The proof proceeds by structural induction on derivation $\mathcal{E}$. The proof is very similar to the above, therefore we only present the interesting cases.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}'}{\Delta, X : (\Psi \vdash P), X' : U'; \Gamma \Longrightarrow \tau}}{\Delta, X : (\Psi \vdash P); \Gamma \Longrightarrow \Pi^{\Box} X' : U'.\, \tau} \Pi^{\Box} R$

1. $\Delta; \Psi \Longrightarrow R : P$	Derivation $\mathcal{D}$ (assumpt.)
2. $\Delta, X' : [\![(\hat{\Psi} \vdash R)/X]\!]U'; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$	By I.H. on $(\Psi \vdash P)$, $\mathcal{D}$, and $\mathcal{E}'$
3. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow \Pi^{\Box} X' : [\![(\hat{\Psi} \vdash R)/X]\!]U'.\, [\![(\hat{\Psi} \vdash R)/X]\!]\tau$	By $\Pi^{\Box} R$ on 2
4. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\ \Pi^{\Box} X' : U'.\, \tau$	By def. of meta-sub.

Case: $\mathcal{E} = \Delta, X : (\Psi \vdash P); \Gamma, y_1 : \Pi X' : U'.\, \tau' \Longrightarrow \tau$ in

$$\frac{\Delta, X:(\Psi \vdash \overset{\mathcal{E}_1}{P}) \Vdash C \Leftarrow U' \quad \Delta, X:(\Psi \vdash P); \Gamma, y_1 : \Pi \overset{\mathcal{E}_2}{X'} : U'.\, \tau', y_2 : [\![C/X']\!]\tau' \Longrightarrow \tau}{\Delta, X:(\Psi \vdash P); \Gamma, y_1 : \Pi X' : U'.\, \tau' \Longrightarrow \tau}\ \Pi^{\square} L$$

1. $\Delta; \Psi \Longrightarrow R : P$ — Derivation $\mathcal{D}$ (assumpt.)
2. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!](\Gamma, y_1 : \Pi X' : U'.\, \tau', y_2 : [\![C/X']\!]\tau')$ $\Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ — By I.H. on $(\Psi \vdash P)$, $\mathcal{D}$, and $\mathcal{E}_2$
3. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma, y_1 : \Pi X' : [\![(\hat{\Psi} \vdash R)/X]\!]U'.\, [\![(\hat{\Psi} \vdash R)/X]\!]\tau'$, $y_2 : [\![[\![(\hat{\Psi} \vdash R)/X]\!]C/X']\!][\![(\hat{\Psi} \vdash R)/X]\!]\tau') \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ — By def. of meta-sub.
4. $\Delta \Vdash [\![(\hat{\Psi} \vdash R)/X]\!]C \Leftarrow [\![(\hat{\Psi} \vdash R)/X]\!]U'$ — By Lemma 4a.
5. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma, y_1 : \Pi X' : [\![(\hat{\Psi} \vdash R)/X]\!]U'.\, [\![(\hat{\Psi} \vdash R)/X]\!]\tau'$ $\Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ — By $\Pi^{\square}L$ on 4 and 3
6. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!](\Gamma, y_1 : \Pi X' : U'.\, \tau') \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ — By def. of meta-sub.

Case: $\mathcal{E} = \dfrac{\Delta, X:(\Psi \vdash \overset{\mathcal{E}'}{P}); \Phi \Longrightarrow Q}{\Delta, X:(\Psi \vdash P); \Gamma \Longrightarrow [\Phi \vdash Q]}\ \square R$

1. $\Delta; \Psi \Longrightarrow R : P$ — Derivation $\mathcal{D}$ (assumption)
2. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Phi \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]Q$ — By I.H. on $(\Psi \vdash P)$, $\mathcal{E}'$, $\mathcal{D}$
3. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow [[\![(\hat{\Psi} \vdash R)/X]\!]\Phi \vdash [\![(\hat{\Psi} \vdash R)/X]\!]Q]$ — By $\square R$ on 2
4. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!][\Phi \vdash Q]$ — By definition of meta-subst.

$$\text{Case: } \mathcal{E} = \dfrac{\overset{\mathcal{E}'}{\Delta, X : (\Psi \vdash P), X' : (\Phi \vdash Q); \Gamma, y : [\Phi \vdash Q] \Longrightarrow \tau}}{\Delta, X : (\Psi \vdash P); \Gamma, y : [\Phi \vdash Q] \Longrightarrow \tau} \; \Box L$$

1. $\Delta; \Psi \Longrightarrow R : P$ Derivation $\mathcal{D}$ (assumption)
2. $\Delta, X' : [\![(\hat{\Psi} \vdash R)/X]\!](\Phi \vdash Q);$

   $[\![(\hat{\Psi} \vdash R)/X]\!](\Gamma, y : [\Phi \vdash Q]) \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ By I.H. on $(\Psi \vdash P)$, $\mathcal{D}$, and $\mathcal{E}'$
3. $\Delta, X' : [\![(\hat{\Psi} \vdash R)/X]\!](\Phi \vdash Q);$

   $[\![(\hat{\Psi} \vdash R)/X]\!]\Gamma, y : [\![(\hat{\Psi} \vdash R)/X]\!][\Phi \vdash Q]$

   $\Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ By definition of meta-subst.
4. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!]\Gamma, y : [\![(\hat{\Psi} \vdash R)/X]\!][\Phi \vdash Q]$

   $\Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ By $\Box L$ on 3
5. $\Delta; [\![(\hat{\Psi} \vdash R)/X]\!](\Gamma, y : [\Phi \vdash Q]) \Longrightarrow [\![(\hat{\Psi} \vdash R)/X]\!]\tau$ By definition of meta-subst.

$\Box$

Using the cut theorems, we can prove invertibility of some of the inference rules of our calculi. The proofs are simple, and in addition to cut make use of weakening, init, and the respective left-rules.

Lemma 7 (Invertibility in the sequent calculi)**.**
a) (ΠR) If $\Delta; \Psi \Longrightarrow \Pi x : A.B$ then $\Delta; \Psi, x : A \Longrightarrow B$
b) ($\rightarrow R$) If $\Delta; \Psi \Longrightarrow A \rightarrow B$ then $\Delta; \Psi, x : A \Longrightarrow B$
c) ($\Pi^{\Box} R$) If $\Delta; \Gamma \Longrightarrow \Pi^{\Box} X : U.\tau$ then $\Delta, X : U; \Gamma \Longrightarrow \tau$
d) ($\rightarrow R$) If $\Delta; \Gamma \Longrightarrow \tau_1 \rightarrow \tau_2$ then $\Delta; \Gamma, y : \tau_1 \Longrightarrow \tau_2$
e) ($\Box L$) If $\Delta; \Gamma, y : [U] \Longrightarrow \tau$ then $\Delta, X : U; \Gamma, y : [U] \Longrightarrow \tau$

Notice that the modal box operator is invertible on the *left*. This causes implications in the focusing calculus.

3.3 Focused-Based Search

Using a sequent calculus is an excellent way of presenting readable inference rules for a logical system used to construct proofs (on paper) in a way that closely resembles how we normally perform deduction due to its characterization of normal proofs, thus limiting non-determinism in proof construction. However for an automated prover, the amount of remaining non-determinism in the logic still prevents it from being a practical calculus to implement. Therefore we instead turn our attention to building *uniform proofs* [Miller et al., 1991] over a *focusing calculus* [Andreoli, 1992].

Uniform proofs were developed by a group of researchers in the early 90's as a foundation for logic programming. It is a well-known proof building technique in which the logical connectives are perceived as *search instructions* for proof search. Uniform proofs abide by a set of rules. In particular, if the goal formula of a sequent in a proof is not atomic then it must be the lower sequent in the introduction (or right) rule of the goals's top-most connective. As a consequence, we can not work on assumptions or access the program until we have reached an atomic goal, and the procedure to reach such an atomic goal is deterministic.

The technique of focusing was first introduced by Jean-Marc Andreoli in 1992 for classical linear logic [Andreoli, 1992]. He intended to limit the amount of non-determinism within proof search by describing a procedure that produced proofs in some normal form. It is known that there is redundancy in proofs, in the sense that the order that some inference rules are applied does not matter. We call these "don't care" rules. The resulting calculus by Andreoli is one in which these "don't care" rules are grouped together into a macro-rule and performed in one step, in any order. Once all these "don't care" rules are applied, a choice must be made. A single formula is chosen successively from the list of assumptions to *focus* on. This chosen formula is then decomposed into the atoms it defines without utilizing any other assumption. Thus, focusing provides a systematic procedure to deal with the non-invertible rules of a logic, making it better suited for proof-finding implementations.

3.3.1 Focusing on Two Levels

Focusing systems have been designed for various logics as focusing provides a systematic and logical way to reduce non-deterministic choices during proof development and turn proof search into a more directed procedure. We have created a focusing calculus for the underlying logic of Beluga which we have used to design larger inference rules for Harpoon. This work extends the work done on uniform proofs, as constructing uniform proofs only (deterministically) instructs us on how to compute an atomic goal. We then use focusing to systematically handle the non-deterministic choices that follow.

Focusing can be seen as alternating applications of collections of inference rules. *Negative* phases consist of applying invertible rules, while *positive* phases consist of applying non-invertible rules. We also use this terminology, positive (also known as synchronous) and negative (also known as asynchronous), to classify logical connectives based on their behaviour during proof search. We consider those connectives with invertible right rules to be *negative* on-the-right (OTR) and those with non-invertible right rules to be *positive* OTR. Typically, if a connective is negative OTR it is positive on-the-left (OTL), and vice-versa. This leads to the classification of formulas within a sequent: a formula in a sequent is considered negative (respectively positive) if its top-most connective is negative (respectively positive). We can then give a classification of formulas in our reasoning logic, based on Lemma 7.

$$\begin{array}{lcl}
\text{Positive Right Formulas} & ::= & [\Psi \vdash P] \\
\text{Negative Right Formulas} & ::= & \tau_1 \to \tau_2 \mid \Pi^{\square} X : U.\tau \\
\text{Positive Left Formulas} & ::= & \tau_1 \to \tau_2 \mid \Pi^{\square} X : U.\tau \\
\text{Negative Left Formulas} & ::= & [\Psi \vdash P]
\end{array}$$

Since the box connective is invertible OTL, atomic box formulas are negative left formulas (and therefore positive right formulas). Although not mentioned in our grammar, we consider recursive types (like `Reduce` in Chapter 2.2.2) to be atomic computation-level propositions which are positive right and negative left formulas.

The two connectives, $\rightarrow$ and Π, in contextual LF both have invertible right rules and therefore construct negative right formulas. Since the assignment to atoms has no affect on provability [Miller and Saurin, 2007], for simplicity we choose to classify atomic LF formulas ($\mathbf{a}\overrightarrow{M}$) as positive left *and* positive right formulas.

$$\begin{array}{lcl} \text{Positive Right Formulas} & ::= & \mathbf{a}\overrightarrow{M} \\ \text{Negative Right Formulas} & ::= & A \rightarrow B \mid \Pi x : A.\ B \\ \text{Positive Left Formulas} & ::= & \mathbf{a}\overrightarrow{M} \mid A \rightarrow B \mid \Pi x : A.\ B \end{array}$$

This provides us a way to describe a homogeneous proof procedure for both layers of our logic. It proceeds as follows: we begin with a negative phase, applying all invertible rules. This phase ends when we have both a positive formula OTR and only positive formulas OTL. We then switch to the positive phase, where we elicit focusing by sequentially analyzing assumptions.

In our calculi, the uniform and focusing stages take the place of the negative and positive phases respectively. The focusing calculus we designed is actually two separate focusing calculi with a transition step in between. This again is due to the fact the logic we are dealing with is a two-level logic.

The goal of this logic is to formalize the proof search procedure that is implemented within HARPOON. This loop is fully automatic and therefore requires that non-determinism be handled with ease. The sequent calculus presented in Chapter 3.2 does not suffice as the rules do not provide any inherent direction for proof search. The rules of the following calculi guide better proof development. They are intended to be read bottom-up, which consequently forces proof search to develop bottom-up. We build uniform proofs by applying all invertible rules first and not working on assumptions until our goal is atomic. We then handle non-invertible rules systematically through focusing.

Focusing in LF

The focusing calculus for contextual LF consists of two main phases- a uniform and focusing phase. The uniform proof stage consists of applying the invertible right-rules to our goal until we reach on positive right formula. During focusing, we iterate through assumptions in the meta (Δ) and LF (Ψ) contexts. We omit proof terms for readability when permitted. We assume well-formedness of all sequents.

$\Delta; \Psi \overset{u}{\Rightarrow} A$	There exists a uniform proof of A in Δ and Ψ
$\Delta; \Psi \overset{u}{\Rightarrow} \sigma : \Phi$	σ is a uniform proof of Φ in Δ and Ψ
$\Delta; \Psi > x : A \rightrightarrows P$	There exists a focused proof of P in Δ and Ψ with focus on A

$$\frac{\Delta; \Psi, \hat{x} : A \overset{u}{\Rightarrow} B}{\Delta; \Psi \overset{u}{\Rightarrow} \Pi\hat{x} : A.\ B}\ \Pi R \qquad \frac{\Delta; \Psi, x : A \overset{u}{\Rightarrow} B}{\Delta; \Psi \overset{u}{\Rightarrow} A \to B} \to R$$

$$\frac{\Delta(X) = (\Phi \vdash Q) \quad \Delta; \Psi \overset{u}{\Rightarrow} \sigma : \Phi \quad [\sigma]Q = P}{\Delta; \Psi \overset{u}{\Rightarrow} P}\ \text{transition}^{\Delta}$$

$$\frac{\Psi(x) = A \quad \Delta; \Psi > x : A \rightrightarrows P}{\Delta; \Psi \overset{u}{\Rightarrow} P}\ \text{transition}^{\Psi}$$

$$\frac{}{\Delta; \Psi \overset{u}{\Rightarrow} \cdot : \cdot}\ \text{empty} \qquad \frac{}{\Delta; \psi, \Psi \overset{u}{\Rightarrow} id_{\psi} : \psi}\ \text{id} \qquad \frac{\Delta; \Psi \overset{u}{\Rightarrow} \sigma : \Phi \quad \Delta; \Psi \overset{u}{\Rightarrow} N : [\sigma]B}{\Delta; \Psi \overset{u}{\Rightarrow} (\sigma, N) : (\Phi, x : B)}\ \text{sub}$$

$$\frac{}{\Delta; \Psi > x : P \rightrightarrows P}\ \text{init}^{\Psi} \qquad \frac{\Delta; \Psi \overset{u}{\Rightarrow} A \quad \Delta; \Psi > x' : B \rightrightarrows P}{\Delta; \Psi > x : A \to B \rightrightarrows P} \to L$$

$$\frac{\Delta; \Psi \vdash M \Leftarrow A \quad \Delta; \Psi > x'' : [M/\hat{x}]B \rightrightarrows P}{\Delta; \Psi > x' : \Pi\hat{x} : A.\ B \rightrightarrows P}\ \Pi L$$

Figure 3.9: Focusing calculus for contextual LF.

This focusing calculus is mostly straightforward. We now distinguish between parameters that are introduced from the ΠR rule, labelled $\hat{x}$, and assumptions introduced from the $\to R$, labelled x, which are used in proof search. Proof development begins with the uniform phase in which parameters and assumptions are collected and placed in the LF context, concluding with an atomic positive goal P. We then try to find a solution by focusing on assumptions from the different contexts. It does not matter which context we attempt to focus on first. In the transition$^\Delta$ rule, using an assumption from Δ to complete a proof requires a simultaneous substitution (σ) to be constructed so that the assumption Q makes sense in the current LF context Ψ. We find such a substitution through uniform proof search. When focusing on assumptions from Ψ of function-type, we search for a proof of A if our assumption is of non-dependent function type. Otherwise the assumption is of dependent function type, in which case M is found via unification.

Focusing on the Computation Level

Similarly to focusing in LF, we perform all invertible rules first until we must make a choice on what to focus on. Unlike in LF proof search, proof search over computations requires two separate phases of inversions since the box connective has an invertible left rule. Further, the choices we have during the focusing phase increase as we may now also choose to conduct LF proof search to solve our goal, which corresponds to focusing on the right.

$\Delta; \Gamma \overset{R}{\Longrightarrow} \tau$	There is a uniform right proof of τ in Δ and Γ
$\Delta; \Gamma \gg \Gamma' \overset{L}{\Longrightarrow} [\Psi \vdash P]$	There is a uniform left proof of $[\Psi \vdash P]$ in Δ and Γ, Γ'
$\Delta; \Gamma > y : \tau \Rightarrow [\Psi \vdash P]$	There is a focused proof of $[\Psi \vdash P]$ in Δ and Γ with focus τ

There are four transition inference rules- left to right, focus to uniform, level, and blur. From these, it should be simple to determine which way proofs are constructed. We begin with a uniform right phase which ends with a positive goal formula, $[\Psi \vdash P]$. From there we transition to a uniform left phase, ending with only positive assumptions in Γ. The sequent depicting the uniform left phase is novel. We use the symbol $\gg$ as a way to distinguish positive assumptions (to the left of $\gg$) from (possibly) negative ones (to the right of $\gg$)

that have yet to be unboxed. Recall that the order of assumptions in Γ does not matter, therefore it is acceptable that the order reverses each time we complete a uniform left phase.

$$\dfrac{\Delta;\Gamma, y:\tau_1 \overset{\mathrm{R}}{\Rightarrow} \tau_2}{\Delta;\Gamma \overset{\mathrm{R}}{\Rightarrow} \tau_1 \to \tau_2} \to R \qquad \dfrac{\Delta, X:U;\Gamma \overset{\mathrm{R}}{\Rightarrow} \tau}{\Delta;\Gamma \overset{\mathrm{R}}{\Rightarrow} \Pi^{\Box} X:U.\, \tau} \Pi^{\Box} R \qquad \dfrac{\Delta;\cdot \gg \Gamma \overset{\mathrm{L}}{\Rightarrow} [\Psi \vdash P]}{\Delta;\Gamma \overset{\mathrm{R}}{\Rightarrow} [\Psi \vdash P]} \text{left to right}$$

$$\dfrac{\Delta, X:(\Phi \vdash Q);\Gamma \gg \Gamma' \overset{\mathrm{L}}{\Rightarrow} [\Psi \vdash P]}{\Delta;\Gamma \gg \Gamma', y:[\Phi \vdash Q] \overset{\mathrm{L}}{\Rightarrow} [\Psi \vdash P]} \Box L \qquad \dfrac{\tau \neq [\Phi \vdash Q] \quad \Delta;\Gamma, y:\tau \gg \Gamma' \overset{\mathrm{L}}{\Rightarrow} [\Psi \vdash P]}{\Delta;\Gamma \gg \Gamma', y:\tau \overset{\mathrm{L}}{\Rightarrow} [\Psi \vdash P]} \text{shift}$$

$$\dfrac{\Gamma(y) = \tau \quad \Delta;\Gamma > y:\tau \Rightarrow [\Psi \vdash P]}{\Delta;\Gamma \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]} \text{focus to uniform} \qquad \dfrac{\Delta;\Psi \overset{u}{\Rightarrow} P}{\Delta;\Gamma \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]} \text{level}$$

$$\dfrac{\Delta \Vdash C \Leftarrow U \quad \Delta;\Gamma > y' : [\![C/X]\!]\tau \Rightarrow [\Psi \vdash P]}{\Delta;\Gamma > y : \Pi^{\Box} X:U.\, \tau \Rightarrow [\Psi \vdash P]} \Pi^{\Box} L$$

$$\dfrac{\Delta;\Gamma \overset{R}{\Rightarrow} \tau_1 \quad \Delta;\Gamma > y' : \tau_2 \Rightarrow [\Psi \vdash P]}{\Delta;\Gamma > y : \tau_1 \to \tau_2 \Rightarrow [\Psi \vdash P]} \to L \qquad \dfrac{\Delta;\cdot \gg \Gamma, y' : [\Phi \vdash Q] \overset{L}{\Rightarrow} [\Psi \vdash P]}{\Delta;\Gamma > y' : [\Phi \vdash Q] \Rightarrow [\Psi \vdash P]} \text{blur}$$

Figure 3.10: Focusing calculus for the computation logic.

The uniform phases should be straight-forward- we collect assumptions and unbox boxed assumptions so they may be used during LF proof search. This is done because when we shift levels we only bring with us assumptions that are true across all levels (those in Δ), as computation assumptions do not make sense on the LF level. After the uniform proof stages, we are left with a positive goal and assumptions, so we enter into focusing. Focusing on the right, i.e. LF proof search, can only be applied if the goal is of box-type (which is currently the only atomic proposition in our logic). Focusing on the left is standard. Focusing commences once we have decomposed the focused formula to its atom ($[\Phi \vdash Q]$) as in the blur rule. At this point, we add the atomic formula to our computation assumptions and restart the process from the uniform left stage. In practice, we implement backtracking when focusing. If, for example, we cannot find a proof while focusing on the right, we backtrack and try focusing on the left. In the blur rule, we only add the new assumption to Γ if it has not been previously added. Otherwise, we backtrack and attempt to find a new, unique assumption.

In [Heilala and Pientka, 2007], Heilala and Pientka present proof search procedures over the propositional fragment of the intuitionistic modal logic IS4. They investigate modal logics that use the traditional meaning of validity, that is, no contextual validity. They too develop focused sequent calculi with the intention of presenting a fully automatic search procedure, but instead of loop detection, utilize *bidirectional proof search*. The idea behind this technique is to construct a set of inference rules that may be needed to solve a goal *prior* to conducting traditional proof search. These inference rules are intended to take the place of the left rules and are found via forward proof search techniques. This technique has yet to be explored for first- and higher-order modal logics such as the logics behind Beluga.

Soundness and Completeness

We show that, with respect to the sequent calculi presented in Figures 3.7 and 3.8, the focusing calculi presented above are sound and complete. The soundness proofs are straightforward and therefore their proofs are omitted.

Theorem 2 (Soundness)**.**
a) If $\Delta; \Psi \overset{u}{\Rightarrow} A$ *then* $\Delta; \Psi \Longrightarrow A$
b) If $\Delta; \Psi \overset{u}{\Rightarrow} \sigma : \Phi$ *then* $\Delta; \Psi \Longrightarrow \sigma : \Phi$
c) If $\Delta; \Psi > x : A \rightrightarrows P$ *then* $\Delta; \Psi, x : A \Longrightarrow P$
d) If $\Delta > X : U; \Psi \rightrightarrows P$ *then* $\Delta, X : U; \Psi \Longrightarrow P$
e) If $\Delta; \Gamma \overset{R}{\Rightarrow} \tau$ *then* $\Delta; \Gamma \Longrightarrow \tau$
f) If $\Delta; \Gamma \gg \Gamma' \overset{L}{\Rightarrow} [\Psi \vdash P]$ *then* $\Delta; \Gamma, \Gamma' \Longrightarrow [\Psi \vdash P]$
g) If $\Delta; \Gamma > y : \tau \Rightarrow [\Psi \vdash P]$ *then* $\Delta; \Gamma, y : \tau \Longrightarrow [\Psi \vdash P]$

Proof. a) - g) By structural induction on the given derivation. $\square$

Before we prove completeness, there are a number of lemmas that are required. First, the completeness proofs for contextual LF rely on some postponement results. Their proofs are straightforward.

Lemma 8 (Contextual LF Postponement 1).

a) If $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B, \Psi' \overset{u}{\Rightarrow} A'$ *then* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, \Psi' \overset{u}{\Rightarrow} A'$

b) If $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B > x' : A' \rightrightarrows P$ *then* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B > x' : A' \rightrightarrows P$

c) If $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A$ *and* $\Delta > X : U; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \rightrightarrows P$ *then* $\Delta > X : U; \Psi, x_1 : \Pi\hat{x} : A.\ B \rightrightarrows P$

d) If $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A$ *and* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \overset{u}{\Rightarrow} \Phi$ *then* $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \overset{u}{\Rightarrow} \Phi$

Proof. a) - d) By mutual structural induction on the second sequent. □

Lemma 9 (Contextual LF Postponement 2).

a) If $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ *and* $\Delta; \Psi, x_1 : A \to B, x_2 : B, \Psi' \overset{u}{\Rightarrow} A'$ *then* $\Delta; \Psi, x_1 : A \to B, \Psi' \overset{u}{\Rightarrow} A'$

b) If $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ *and* $\Delta; \Psi, x_1 : A \to B, x_2 : B > x' : A' \rightrightarrows P$ *then* $\Delta; \Psi, x_1 : A \to B > x' : A' \rightrightarrows P$

c) If $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ *and* $\Delta > X : U; \Psi, x_1 : A \to B, x_2 : B \rightrightarrows P$ *then* $\Delta > X : U; \Psi, x_1 : A \to B \rightrightarrows P$

d) If $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ *and* $\Delta; \Psi, x_1 : A \to B, x_2 : B \overset{u}{\Rightarrow} \Phi$ *then* $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} \Phi$

Proof. a) - d) By mutual structural induction on the second sequent. □

Lemma 10 (Contextual LF Postponement 3).

a) If $\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P, \Psi' \overset{u}{\Rightarrow} A$ *and* $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Rightarrow} \sigma : \Phi$ *then* $\Delta, X : (\Phi \vdash P); \Psi, \Psi' \overset{u}{\Rightarrow} A$

b) If $\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P > x' : A \rightrightarrows P'$ *and* $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Longrightarrow} \sigma : \Phi$ *then*
$\Delta, X : (\Phi \vdash P); \Psi > x' : A \rightrightarrows P'$

c) If $\Delta, X : (\Phi \vdash P) > X' : U'; \Psi, x : [\sigma]P \rightrightarrows P'$ *and* $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Longrightarrow} \sigma : \Phi$ *then*
$\Delta, X : (\Phi \vdash P) > X' : U'; \Psi \rightrightarrows P'$

Proof. a) - c) By mutual structural induction on the second sequent. □

The proofs for completeness of the computation logic require similar postponement lemmas.

Lemma 11 (Computation logic Postponement 1)**.**

a) If $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau_1$ $(\tau_2 \neq [\Psi \vdash P])$ *and* $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2, \Gamma' \overset{R}{\Longrightarrow} \tau$ *then*
$\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, \Gamma' \overset{R}{\Longrightarrow} \tau$

b) If $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau_1$ $(\tau_2 \neq [\Psi \vdash P])$ *and* $\Delta; \Gamma_1, y_2 : \tau_2, \Gamma_2 \gg \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$
(resp. $\Delta; \Gamma_1 \gg \Gamma_2, y_2 : \tau_2, \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$*) where* $y_1 \in \Gamma_1, \Gamma_2, \Gamma_3$*, then*
$\Delta; \Gamma_1, \Gamma_2 \gg \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$ *(resp.* $\Delta; \Gamma_1 \gg \Gamma_2, \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$*)*

c) If $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau_1$ $(\tau_2 \neq [\Psi \vdash P])$ *and* $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$
then $\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$

Proof. a) - c) By mutual structural induction on the second sequent. □

Lemma 12 (Computation logic Postponement 2)**.**

a) If $\Delta; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q] \overset{R}{\Longrightarrow} \tau_1$ *and* $\Delta, X : (\Phi \vdash Q), \Delta'; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q], \Gamma' \overset{R}{\Longrightarrow} \tau$ *then*
$\Delta, \Delta'; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q], \Gamma' \overset{R}{\Longrightarrow} \tau$

b) If $\Delta; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q] \overset{R}{\Longrightarrow} \tau_1$ *and* $\Delta, X : (\Phi \vdash Q); \Gamma_1 \gg \Gamma_2 \overset{L}{\Longrightarrow} [\Psi \vdash P]$ *where*
$y_1 \in \Gamma_1, \Gamma_2$ *then* $\Delta; \Gamma_1 \gg \Gamma_2 \overset{L}{\Longrightarrow} [\Psi \vdash P]$

c) If $\Delta; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q] \overset{R}{\Longrightarrow} \tau_1$ *and*
$\Delta, X : (\Phi \vdash Q); \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q] > y : \tau \Rightarrow [\Psi \vdash P]$ *then*
$\Delta; \Gamma, y_1 : \tau_1 \to [\Phi \vdash Q] > y : \tau \Rightarrow [\Psi \vdash P]$

Proof. a) - c) By mutual structural induction on the second sequent. □

Lemma 13 (Computation logic Postponement 3)**.**

a) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X' : [\![C/X]\!]\tau', \Delta'; \Gamma, y' : \Pi^{\square} X : U.\tau', \Gamma' \overset{R}{\Longrightarrow} \tau$ *(*$[\![C/X]\!]\tau' = [\Phi \vdash Q]$*)*
then $\Delta, \Delta'; \Gamma, y' : \Pi^{\square} X : U.\tau', \Gamma' \overset{R}{\Longrightarrow} \tau$

b) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X' : [\![C/X]\!]\tau'; \Gamma_1 \gg \Gamma_2 \overset{L}{\Longrightarrow} [\Psi \vdash P]$ *(*$[\![C/X]\!]\tau' = [\Phi \vdash Q]$*) where* $\Pi^{\square} X : U.\tau' \in \Gamma_1, \Gamma_2$ *then* $\Delta; \Gamma_1 \gg \Gamma_2 \overset{L}{\Longrightarrow} [\Psi \vdash P]$

c) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta, X' : [\![C/X]\!]\tau'; \Gamma, y' : \Pi^{\square} X : U.\tau' > y : \tau \Rightarrow [\Psi \vdash P]$ *(*$[\![C/X]\!]\tau' = [\Phi \vdash Q]$*) then* $\Delta; \Gamma, y' : \Pi^{\square} X : U.\tau' > y : \tau \Rightarrow [\Psi \vdash P]$

Proof. a) - c) By mutual structural induction on the second sequent. □

Lemma 14 (Computation logic Postponement 4)**.**

a) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\tau', y_2 : [\![C/X]\!]\tau', \Gamma' \overset{R}{\Longrightarrow} \tau$ *(*$[\![C/X]\!]\tau' \neq [\Phi \vdash Q]$*)*
then $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\tau', \Gamma' \overset{R}{\Longrightarrow} \tau$

b) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta; \Gamma_1, y_2 : [\![C/X]\!]\tau', \Gamma_2 \gg \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$
(resp. $\Delta; \Gamma_1 \gg \Gamma_2, y_2 : [\![C/X]\!]\tau', \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$*) (*$[\![C/X]\!]\tau' \neq [\Phi \vdash Q]$*) where*
$\Pi^{\square} X : U.\tau' \in \Gamma_1, \Gamma_2, \Gamma_3$ *then* $\Delta; \Gamma_1, \Gamma_2 \gg \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$
(resp. $\Delta; \Gamma_1 \gg \Gamma_2, \Gamma_3 \overset{L}{\Longrightarrow} [\Psi \vdash P]$*)*

c) If $\Delta \Vdash C \Leftarrow U$ *and* $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\tau', y_2 : [\![C/X]\!]\tau' > y : \tau \Rightarrow [\Psi \vdash P]$
($[\![C/X]\!]\tau' \neq [\Phi \vdash Q]$*) then* $\Delta; \Gamma, y_1 : \Pi^{\square} X : U.\tau' > y : \tau \Rightarrow [\Psi \vdash P]$

Proof. a) - c) By mutual structural induction on the second sequent. □

We have that the reflect rule holds for the uniform right phase of our focusing calculus for the computation logic. This result is also required for the completeness proof.

Lemma 15 (Reflect for Focusing)**.**
If $\Delta, X : (\Psi \vdash P), \Delta'; \Gamma, y : [\Psi \vdash P], \Gamma' \overset{R}{\Longrightarrow} \tau$ *then* $\Delta, \Delta'; \Gamma, y : [\Psi \vdash P], \Gamma' \overset{R}{\Longrightarrow} \tau$

Proof. By structural induction on the given derivation. □

We also require an intermediate lemma for our completeness theorem. To see why, consider the completeness of focused proofs for the computation logic. At first thought, we may take the theorem to be: "if $\Delta; \Gamma \Longrightarrow [\Psi \vdash P]$ then $\Delta; \Gamma > y : \tau \Rightarrow [\Psi \vdash P]$ for some $\tau \in \Gamma$", but recall during focusing, Γ must only contain positive assumptions, which this statement does not guarantee. Therefore we must prove something different. We first show that in fact it does not matter in which context atomic assumptions appear. That is, if there is a proof in our sequent calculus of $[\Psi \vdash P]$ (possibly) using some assumption $y : [\Phi \vdash Q]$ in Γ then there is also a proof of $[\Psi \vdash P]$ where y is omitted but under the added assumption $X : (\Phi \vdash Q)$ in Δ. This lemma helps allows us to properly rephrase our completeness theorem.

Before we begin, we introduce new notation.

Definition 3.3.1 ($\Delta^-_{\Delta,\Gamma}$).
Given Δ and Γ such that $\Delta \vdash \Gamma$, define $\Delta^-_{\Delta,\Gamma}$ to be

- *if $\Gamma = \cdot$, then Δ*
- *if $\Gamma = (\Gamma', y : [\Psi \vdash P])$, then $(\Delta^-_{\Delta,\Gamma'}, X : (\Psi \vdash P))$*
- *if $\Gamma = (\Gamma', y : \tau)$, $\tau \neq [\Psi \vdash P]$, then $\Delta^-_{\Delta,\Gamma'}$*

Definition 3.3.2 ($\Gamma^+_{\Delta,\Gamma}$).
Given Δ and Γ such that $\Delta \vdash \Gamma$, define $\Gamma^+_{\Delta,\Gamma}$ to be

- *if $\Gamma = \cdot$, then $\cdot$*
- *if $\Gamma = (\Gamma', y : [\Psi \vdash P])$, then $\Gamma^+_{\Delta,\Gamma'}$*
- *if $\Gamma = (\Gamma', y : \tau)$, $\tau \neq [\Psi \vdash P]$, then $(\Gamma^+_{\Delta,\Gamma'}, y : \tau)$*

Essentially, $\Delta^-_{\Delta,\Gamma}$ is Δ extended with the negative assumptions from Γ unboxed, and $\Gamma^+_{\Delta,\Gamma}$ contains only the positive assumptions from Γ. We may then present our lemma:

Lemma 16.
If $\Delta;\Gamma \Longrightarrow \tau$ *then* $\Delta^{-}_{\Delta,\Gamma};\Gamma^{+}_{\Delta,\Gamma} \Longrightarrow \tau$

Proof. By straightforward structural induction on the given derivation. □

Now, part of our completeness theorem is reformulated as "if $\Delta;\Gamma \Longrightarrow [\Psi \vdash P]$ then $\Delta^{-}_{\Delta,\Gamma};\Psi \overset{u}{\Rightarrow} P$". This however is not true. There may not necessarily be a right-focused proof using $\Delta^{-}_{\Delta,\Gamma}$. It could be the case that the right-focused proof requires a meta-context with more assumptions which may be derived from Γ. This may be best explained using an example.

```
LF nat : type =                     LF less_than : nat → nat → type =
| z : nat                           | lt : less_than N1 N2
| s : nat → nat                            → less_than (s N1) (s N2)
;                                   ;
```

Given a specification of natural numbers along with an (incomplete) theory of the less than relation, consider the following sequent, derivable in our sequent calculi:

$$\cdot; y : (\Pi^{\Box} N : (\vdash \text{nat}).\ [\vdash \text{less\_than}\ z\ N]) \Longrightarrow [\vdash \text{less\_than}\ (sz)\ (s(sz))]$$

There is however no right-focused from $\Delta^{-}_{\Delta,\Gamma}$. To derive a right-focused proof requires the assumption $(\vdash \text{less\_than}\ z\ (sz))$ be in the meta-context during LF proof search. This assumption may be introduced into Γ through the $\Pi^{\Box}L$ and blur rules, and subsequently added to Δ via $\Box L$. Only then may we transition of focusing on the right, and solve the goal.

For simplicity, we use C in place of the formula $\Pi^{\Box}N : (\vdash \mathit{nat}).[\vdash \text{less\_than}\ z\ N]$ and $\mathit{le\_z\_sz}$ in place of the LF type less_than $z\ (sz)$, and $\mathit{le\_sz\_ssz}$ in place of the goal formula less_than $(sz)\ (s(sz))$. We also highlight the principal formula in each sequent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{X : (\,\vdash \mathit{le\_z\_sz}); \cdot \overset{u}{\Rightarrow} \mathit{le\_sz\_ssz}}}{X : (\,\vdash \mathit{le\_z\_sz}); y : C \gg \cdot \overset{L}{\Rightarrow} [\,\vdash \mathit{le\_sz\_ssz}]}\text{level}}{X : (\,\vdash \mathit{le\_z\_sz}); \cdot \gg y : C \overset{L}{\Rightarrow} [\,\vdash \mathit{le\_sz\_ssz}]}\text{shift}}{\cdot; \cdot \gg y : C, y' : [\,\vdash \mathit{le\_z\_sz}] \overset{L}{\Rightarrow} [\,\vdash \mathit{le\_sz\_ssz}]}\Box L}{\cdot; y : C > y' : [\,\vdash \mathit{le\_z\_sz}] \Rightarrow [\,\vdash \mathit{le\_sz\_ssz}]}\text{blur}}{\cdot; y : C > y : C \Rightarrow [\,\vdash \mathit{le\_sz\_ssz}]}\Pi^{\Box}L}{\cdot; y : C \Longrightarrow [\,\vdash \mathit{le\_sz\_ssz}]}\text{focus to unif.}$$

Due to the nature of this two-level proof search, we must reformulate the theorem so that it accounts for these derivable assumptions when it comes time to focus on the right. We are then ready to state the completeness theorem:

Theorem 3 (Completeness)**.**
a) If $\Delta; \Psi \overset{\mathcal{D}}{\Longrightarrow} A$ *then* $\Delta; \Psi \overset{u}{\Rightarrow} A$

b) If $\Delta; \Psi \overset{\mathcal{D}}{\Longrightarrow} \sigma : \Phi$ *then* $\Delta; \Psi \overset{u}{\Rightarrow} \sigma : \Phi$

c) If $\Delta; \Psi \overset{\mathcal{D}}{\Longrightarrow} P$ *then either* $\Delta; \Psi > x : A \rightrightarrows P$ *for some* $A \in \Psi$ *or* $\Delta > X : U; \Psi \rightrightarrows P$ *for some* $U \in \Delta$

d) If $\Delta; \Gamma \overset{\mathcal{D}}{\Longrightarrow} \tau$ *then* $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \overset{R}{\Rightarrow} \tau$

e) If $\Delta; \Gamma \overset{\mathcal{D}}{\Longrightarrow} [\Psi \vdash P]$ *then* $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$

f) If $\Delta; \Gamma \overset{\mathcal{D}}{\Longrightarrow} [\Psi \vdash P]$ *then either* $\Delta^-_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$ *or* $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} > y : \tau \Rightarrow [\Psi \vdash P]$ *for some* $y : \tau \in \Gamma^+_{\Delta,\Gamma}$

Proof. Parts *a-e* are proven by mutual structural induction on the given derivations. Part *f* is proven by structural induction on derivation $\mathcal{D}$.

Part a):

Case: $\mathcal{D} = \dfrac{}{\Delta; \Psi, x : A \Longrightarrow A}\text{init}^{\Psi}$

By lemma 2.

Case: $\mathcal{D} = \dfrac{\Delta; \Psi, \hat{x} : A \Longrightarrow B}{\Delta; \Psi \Longrightarrow \Pi\hat{x} : A.\ B}\ \Pi R$

By induction hypothesis and ΠR.

Case: $\mathcal{D} = \Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \Longrightarrow A'$ in

$$\dfrac{\overset{\mathcal{D}_1}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A} \quad \overset{\mathcal{D}_2}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \Longrightarrow A'}}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \Longrightarrow A'}\ \Pi L$$

1. $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \overset{u}{\Rightarrow} A'$ By I.H. $\mathcal{D}_2$
2. $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \overset{u}{\Rightarrow} A'$ By lemma 8a.

Case: $\mathcal{D} = \dfrac{\Delta; \Psi, x : A \Longrightarrow B}{\Delta; \Psi \Longrightarrow A \to B} \to R$

By induction hypothesis and $\to R$.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Psi, x_1 : A \to B \Longrightarrow A} \quad \overset{\mathcal{D}_2}{\Delta; \Psi, x_1 : A \to B, x_2 : B \Longrightarrow A'}}{\Delta; \Psi, x_1 : A \to B \Longrightarrow A'} \to L$

1. $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ By I.H. $\mathcal{D}_1$
2. $\Delta; \Psi, x_1 : A \to B, x_2 : B \overset{u}{\Rightarrow} A'$ By I.H. $\mathcal{D}_2$
3. $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A'$ By lemma 9a.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta, X : (\Phi \vdash P); \Psi \Longrightarrow \sigma : \Phi} \quad \overset{\mathcal{D}_2}{\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P \Longrightarrow A}}{\Delta, X : (\Phi \vdash P); \Psi \Longrightarrow A}$ reflect

1. $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Rightarrow} \sigma : \Phi$	By I.H. $\mathcal{D}_1$
2. $\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P \overset{u}{\Rightarrow} A$	By I.H. $\mathcal{D}_2$
3.$\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Rightarrow} A$	By lemma 10a.

Part b):

Trivial.

Part c):

Case: $\mathcal{D} = \dfrac{}{\Delta; \Psi, x : P \Longrightarrow P}$ init$^\Psi$

1. $\Delta; \Psi, x : P > x : P \rightrightarrows P$	By init$^\Psi$

Case: $\mathcal{D} = \Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \Longrightarrow P$ in

$$\dfrac{\overset{\mathcal{D}_1}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \vdash M \Leftarrow A} \quad \overset{\mathcal{D}_2}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \Longrightarrow P}}{\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B \Longrightarrow P}\ \Pi L$$

Subcase: $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B > x' : A' \rightrightarrows P$, $x' \in \Psi, x_1, x_2$

1. $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B > x' : A' \rightrightarrows P$	By I.H. $\mathcal{D}_2$
2. $\Delta; \Psi, x_1 : \Pi\hat{x} : A.\ B > x' : A' \rightrightarrows P$	By lemma 8b.

Subcase: $\Delta > X : U; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \rightrightarrows P, X \in \Delta$

1. $\Delta > X : U; \Psi, x_1 : \Pi\hat{x} : A.\ B, x_2 : [M/\hat{x}]B \rightrightarrows P$ By I.H. $\mathcal{D}_2$
2. $\Delta > X : U; \Psi, x_1 : \Pi\hat{x} : A.\ B \rightrightarrows P$ By lemma 8c.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Psi, x_1 : A \to B \Longrightarrow A} \quad \overset{\mathcal{D}_2}{\Delta; \Psi, x_1 : A \to B, x_2 : B \Longrightarrow P}}{\Delta; \Psi, x_1 : A \to B \Longrightarrow P} \to L$

1. $\Delta; \Psi, x_1 : A \to B \overset{u}{\Rightarrow} A$ By I.H. $\mathcal{D}_1$

Subcase: $\Delta; \Psi, x_1 : A \to B, x_2 : B > x' : A' \rightrightarrows P, x' \in \Psi, x_1, x_2$

2. $\Delta; \Psi, x_1 : A \to B, x_2 : B > x' : A' \rightrightarrows P$ By I.H. $\mathcal{D}_2$
3. $\Delta; \Psi, x_1 : A \to B > x' : A' \rightrightarrows P$ By lemma 9b.

Subcase: $\Delta > X : U; \Psi, x_1 : A \to B, x_2 : B \rightrightarrows P, X \in \Delta$

2. $\Delta > X : U; \Psi, x_1 : A \to B, x_2 : B \rightrightarrows P$ By I.H. $\mathcal{D}_2$
3. $\Delta > X : U; \Psi, x_1 : A \to B \rightrightarrows P$ By lemma 9c.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta, X : (\Phi \vdash P); \Psi \Longrightarrow \sigma : \Phi} \quad \overset{\mathcal{D}_2}{\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P \Longrightarrow P'}}{\Delta, X : (\Phi \vdash P); \Psi \Longrightarrow P'}$ reflect

Subcase: $\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P > x' : A' \rightrightarrows P', x' \in \Psi, x$

1. $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Rightarrow} \sigma : \Phi$ By I.H. $\mathcal{D}_1$
2. $\Delta, X : (\Phi \vdash P); \Psi, x : [\sigma]P > x' : A' \rightrightarrows P'$ By I.H. $\mathcal{D}_2$
3. $\Delta, X : (\Phi \vdash P); \Psi > x' : A' \rightrightarrows P'$ By lemma 10b.

Subcase: $\Delta, X : (\Phi \vdash P) > X' : U'; \Psi, x : [\sigma]P \rightrightarrows P', X' \in \Delta, X$

1. $\Delta, X : (\Phi \vdash P); \Psi \overset{u}{\Rightarrow} \sigma : \Phi$ By I.H. $\mathcal{D}_1$
2. $\Delta, X : (\Phi \vdash P) > X' : U'; \Psi, x : [\sigma]P \rightrightarrows P'$ By I.H. $\mathcal{D}_2$
3. $\Delta, X : (\Phi \vdash P) > X' : U'; \Psi \rightrightarrows P'$ By lemma 10c.

Part d):

Case: $\mathcal{D} = \dfrac{}{\Delta; \Gamma, y : \tau \Longrightarrow \tau}\ \text{init}^{\Gamma}$

By lemma 5.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Delta; \Gamma, y : \tau_1 \Longrightarrow \tau_2}}{\Delta; \Gamma \Longrightarrow \tau_1 \to \tau_2} \to R$

Subcase: $\tau_1 \neq [\Psi \vdash P]$

Trivial.

Subcase: $\tau_1 = [\Psi \vdash P]$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma} \overset{R}{\Longrightarrow} \tau_2$ By I.H. $\mathcal{D}'$
2. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma}, y : [\Psi \vdash P] \overset{R}{\Longrightarrow} \tau_2$ By weakening
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : [\Psi \vdash P] \overset{R}{\Longrightarrow} \tau_2$ By lemma 15.
4. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \overset{R}{\Longrightarrow} \tau_1 \to \tau_2$ By $\to R$

$$\text{Case: } \mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau_1} \quad \overset{\mathcal{D}_2}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow \tau}}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau} \to L$$

Subcase: $\tau_2 = [\Psi \vdash P]$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau$ By lemma 12a.

Subcase: $\tau_2 \neq [\Psi \vdash P]$

1. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \overset{R}{\Longrightarrow} \tau$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Longrightarrow} \tau$ By lemma 11a.

$$\text{Case: } \mathcal{D} = \frac{\overset{\mathcal{D}'}{\Delta, X : U; \Gamma \Longrightarrow \tau}}{\Delta; \Gamma \Longrightarrow \Pi^{\Box} X : U.\, \tau} \, \Pi^{\Box} R$$

Trivial.

Case: $\mathcal{D} = \dfrac{\Delta \Vdash \overset{\mathcal{D}_1}{C} \Leftarrow U \quad \Delta; \Gamma, y : \Pi^{\Box} X : U.\, \overset{\mathcal{D}_2}{\tau'}, y' : [\![C/X]\!]\tau' \Longrightarrow \tau}{\Delta; \Gamma, y : \Pi^{\Box} X : U.\, \tau' \Longrightarrow \tau}\ \Pi^{\Box} L$

Subcase: $[\![C/X]\!]\tau' = [\Psi \vdash P]$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma}, y : \Pi^{\Box} X : U.\, \tau' \overset{R}{\Longrightarrow} \tau$	By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$	By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^{\Box} X : U.\, \tau' \overset{R}{\Longrightarrow} \tau$	By lemma 13a.

Subcase: $[\![C/X]\!]\tau' \neq [\Psi \vdash P]$

1. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^{\Box} X : U.\, \tau', y' : [\![C/X]\!]\tau' \overset{R}{\Longrightarrow} \tau$	By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$	By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^{\Box} X : U.\, \tau' \overset{R}{\Longrightarrow} \tau$	By lemma 14a.

Case: $\mathcal{D} = \dfrac{\Delta; \Psi \overset{\mathcal{D}'}{\Longrightarrow} P}{\Delta; \Gamma \Longrightarrow [\Psi \vdash P]}\ \Box R$

1. $\Delta; \Psi \overset{u}{\Longrightarrow} P$	By I.H. $\mathcal{D}'$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Longrightarrow} [\Psi \vdash P]$	By level and weakening
3. $\Delta^-_{\Delta,\Gamma}; \cdot \gg \Gamma^+_{\Delta,\Gamma} \overset{L}{\Longrightarrow} [\Psi \vdash P]$	By shift
4. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \overset{R}{\Longrightarrow} [\Psi \vdash P]$	By left to right

$$\text{Case: } \mathcal{D} = \cfrac{\Delta, X : (\Psi \vdash P); \overset{\mathcal{D}'}{\Gamma}, y : [\Psi \vdash P] \Longrightarrow \tau}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow \tau} \; \Box L$$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P), X' : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma} \overset{R}{\Rightarrow} \tau$ — By I.H. $\mathcal{D}'$
2. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma} \overset{R}{\Rightarrow} \tau$ — By contraction

Part e):

$$\text{Case: } \mathcal{D} = \cfrac{}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow [\Psi \vdash P]} \; \text{init}^\Gamma$$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P) > X : (\Psi \vdash P); \Psi \rightrightarrows P$ — By init$^\Delta$
2. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Psi \overset{u}{\Rightarrow} P$ — By transition$^\Delta$
3. $\Delta^-_{\Delta,\Gamma}, X : (\Psi \vdash P); \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ — By level

$$\text{Case: } \mathcal{D} = \cfrac{\Delta \Vdash \overset{\mathcal{D}_1}{C} \Leftarrow U \quad \Delta; \Gamma, y : \Pi^\Box X : U.\, \tau', \overset{\mathcal{D}_2}{y'} : [\![C/X]\!]\tau' \Longrightarrow [\Psi \vdash P]}{\Delta; \Gamma, y : \Pi^\Box X : U.\, \tau' \Longrightarrow [\Psi \vdash P]} \; \Pi^\Box L$$

Subcase: $[\![C/X]\!]\tau' = [\Phi \vdash Q]$

1. $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ — By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$ — By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ — By lemma 13b.

Subcase: $[\![C/X]\!]\tau' \neq [\Phi \vdash Q]$

1. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau', y' : [\![C/X]\!]\tau' \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$ By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By lemma 14b.

$$\text{Case: } \mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau_1} \quad \overset{\mathcal{D}_2}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow [\Psi \vdash P]}}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow [\Psi \vdash P]} \to L$$

Subcase: $\tau_2 = [\Phi \vdash Q]$

1. $\Delta^-_{\Delta,\Gamma}, X : \tau_2; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Rightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By lemma 12b.

Subcase: $\tau_2 \neq [\Phi \vdash Q]$

1. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Rightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By lemma 11b.

$$\text{Case: } \mathcal{D} = \frac{\overset{\mathcal{D}'}{\Delta; \Psi \Longrightarrow P}}{\Delta; \Gamma \Longrightarrow [\Psi \vdash P]} \Box R$$

1. $\Delta; \Psi \overset{u}{\Rightarrow} P$ By I.H. $\mathcal{D}'$
2. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By level and weakening

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Delta, X : (\Phi \vdash Q); \Gamma, y : [\Phi \vdash Q] \Longrightarrow [\Psi \vdash P]}}{\Delta; \Gamma, y : [\Phi \vdash Q] \Longrightarrow [\Psi \vdash P]}\ \Box L$

1. $\Delta^-_{\Delta,\Gamma}, X : (\Phi \vdash Q), X' : (\Phi \vdash Q); \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By I.H. $\mathcal{D}'$
2. $\Delta^-_{\Delta,\Gamma}, X : (\Phi \vdash Q); \Gamma^+_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By contraction

Part f):

Case: $\mathcal{D} = \dfrac{}{\Delta; \Gamma, y : [\Psi \vdash P] \Longrightarrow [\Psi \vdash P]}\ \text{init}^\Gamma$

1. $\Delta, X : (\Psi \vdash P) > X : (\Psi \vdash P); \Psi \rightrightarrows P$ By init^Δ
2. $\Delta, X : (\Psi \vdash P); \Psi \overset{u}{\Rightarrow} P$ By transition^Δ

Case: $\mathcal{D} = \dfrac{\Delta \Vdash \overset{\mathcal{D}_1}{C} \Leftarrow U \quad \Delta; \Gamma, y : \Pi^\Box X : U.\, \tau', \overset{\mathcal{D}_2}{y' : [\![C/X]\!]\tau' \Longrightarrow [\Psi \vdash P]}}{\Delta; \Gamma, y : \Pi^\Box X : U.\, \tau' \Longrightarrow [\Psi \vdash P]}\ \Pi^\Box L$

Subcase: $[\![C/X]\!]\tau' = [\Phi \vdash Q]$

Subcase: $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Psi \overset{u}{\Rightarrow} P$

1. $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Psi \overset{u}{\Rightarrow} P$	By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\, \tau' \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$	By level on 1
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\, \tau' \gg y' : [\![C/X]\!]\tau' \overset{L}{\Rightarrow} [\Psi \vdash P]$	By $\Box L$ on 2
4. $\Delta^-_{\Delta,\Gamma}; \cdot \gg \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\, \tau', y' : [\![C/X]\!]\tau' \overset{L}{\Rightarrow} [\Psi \vdash P]$	By shift and exchange on 3
5. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\, \tau' > y' : [\![C/X]\!]\tau' \Rightarrow [\Psi \vdash P]$	By blur on 4
6. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$	By weakening $\mathcal{D}_1$
7. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\, \tau' > y : \Pi^\Box X : U.\, \tau' \Rightarrow [\Psi \vdash P]$	By $\Pi^\Box L$ on 5 and 6

Subcase: $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$,
$y'' \in \Gamma^+_{\Delta,\Gamma}, y$

1. $\Delta^-_{\Delta,\Gamma}, X' : [\![C/X]\!]\tau'; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$	By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$	By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$	By lemma 13c.

Subcase: $[\![C/X]\!]\tau' \neq [\Phi \vdash Q]$

Subcase: $\Delta^-_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$

1. $\Delta^-_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$	By I.H. $\mathcal{D}_2$

Subcase: $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau', y' : [\![C/X]\!]\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$,

$$y'' \in \Gamma^+_{\Delta,\Gamma}, y, y'$$

1. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau', y' : [\![C/X]\!]\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma} \Vdash C \Leftarrow U$ By weakening $\mathcal{D}_1$
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y : \Pi^\Box X : U.\tau' > y'' : \tau \Rightarrow [\Psi \vdash P]$ By lemma 14c.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow \tau_1} \quad \overset{\mathcal{D}_2}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow [\Psi \vdash P]}}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow [\Psi \vdash P]} \to L$

Subcase: $\tau_2 = [\Phi \vdash Q]$

Subcase: $\Delta^-_{\Delta,\Gamma}, X : \tau_2; \Psi \overset{u}{\Rightarrow} P$

1. $\Delta^-_{\Delta,\Gamma}, X : \tau_2; \Psi \overset{u}{\Rightarrow} P$ By I.H. $\mathcal{D}_2$
2. $\Delta^-_{\Delta,\Gamma}, X : \tau_2; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \gg \cdot \overset{L}{\Rightarrow} [\Psi \vdash P]$ By level on 1
3. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \gg y_2 : \tau_2 \overset{L}{\Rightarrow} [\Psi \vdash P]$ By $\Box L$ on 2
4. $\Delta^-_{\Delta,\Gamma}; \cdot \gg \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \overset{L}{\Rightarrow} [\Psi \vdash P]$ By shift and exchange on 3
5. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y_2 : \tau_2 \Rightarrow [\Psi \vdash P]$ By blur on 4
6. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Rightarrow} \tau_1$ By Theorem 3.d. on $\mathcal{D}_1$
7. $\Delta^-_{\Delta,\Gamma}; \Gamma^+_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y_1 : \tau_1 \to \tau_2 \Rightarrow [\Psi \vdash P]$ By $\to L$ on 5 and 6

Subcase: $\Delta^{-}_{\Delta,\Gamma}, X : \tau_2; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y : \tau \Rightarrow [\Psi \vdash P], y \in \Gamma^{+}_{\Delta,\Gamma}, y_1$

1. $\Delta^{-}_{\Delta,\Gamma}, X : \tau_2; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Rightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$ By lemma 12c.

Subcase: $\tau_2 \neq [\Phi \vdash Q]$

Subcase: $\Delta^{-}_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$

1. $\Delta^{-}_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$ By I.H. $\mathcal{D}_2$

Subcase: $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 > y : \tau \Rightarrow [\Psi \vdash P], y \in \Gamma^{+}_{\Delta,\Gamma}, y_1, y_2$

1. $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$ By I.H. $\mathcal{D}_2$
2. $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 \overset{R}{\Rightarrow} \tau_1$ By I.H. $\mathcal{D}_1$
3. $\Delta^{-}_{\Delta,\Gamma}; \Gamma^{+}_{\Delta,\Gamma}, y_1 : \tau_1 \to \tau_2 > y : \tau \Rightarrow [\Psi \vdash P]$ By lemma 11c.

$$\text{Case: } \mathcal{D} = \frac{\Delta; \Psi \overset{\mathcal{D}'}{\Longrightarrow} P}{\Delta; \Gamma \Longrightarrow [\Psi \vdash P]} \ \Box R$$

1. $\Delta; \Psi \overset{u}{\Rightarrow} P$ By Theorem 3.a. on $\mathcal{D}'$
2. $\Delta^{-}_{\Delta,\Gamma}; \Psi \overset{u}{\Rightarrow} P$ By weakening 1

$$\text{Case: } \mathcal{D} = \frac{\Delta, X : (\Phi \vdash Q); \Gamma, y : [\Phi \vdash Q] \overset{\mathcal{D}'}{\Longrightarrow} [\Psi \vdash P]}{\Delta; \Gamma, y : [\Phi \vdash Q] \Longrightarrow [\Psi \vdash P]} \ \Box L$$

Subcase: $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q), X' : (\Phi \vdash Q); \Psi \overset{u}{\Rightarrow} P$

1. $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q), X' : (\Phi \vdash Q); \Psi \overset{u}{\Rightarrow} P$ By I.H. $\mathcal{D}'$
2. $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q); \Psi \overset{u}{\Rightarrow} P$ By contraction 1

Subcase: $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q), X' : (\Phi \vdash Q); \Gamma^{+}_{\Delta,\Gamma} > y : \tau \Rightarrow [\Psi \vdash P], y \in \Gamma^{+}_{\Delta,\Gamma}$

1. $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q), X' : (\Phi \vdash Q); \Gamma^{+}_{\Delta,\Gamma} > y : \tau \Rightarrow [\Psi \vdash P]$ By I.H. $\mathcal{D}'$
2. $\Delta^{-}_{\Delta,\Gamma}, X : (\Phi \vdash Q); \Gamma^{+}_{\Delta,\Gamma} > y : \tau \Rightarrow [\Psi \vdash P]$ By contraction

$\square$

Chapter 4

The Tactic Implementation

In the previous chapter we presented the theoretical foundation behind our tactics. We now provide a summary of the implementation work that has been done. In Chapters 4.1 and 4.2 we outline the automated proof search procedures for contextual LF and its meta-logic implemented in Beluga. These search procedures are encompassed within the tactics `auto-invert-solve` and `inductive-auto-solve`. In Chapter 4.3 we present an overview of the applications of our tactics, and discuss the limitations of their use in Chapter 4.4.

4.1 Theorem Prover for Contextual LF

It is well-known that meta-theoretic reasoning often also involves reasoning within theories. Therefore in order to achieve automated meta-theorem proving, we must also automate theorem proving within theories. As such, Beluga includes an automated theorem prover for (a subset of) contextual LF. As theories are encoded in LF using the propositions-as-types perspective, proof search proceeds by searching for a proof term with the appropriate type. We examine the contextual LF theorem prover by walking through two examples which will highlight its solving capabilities. Although proof terms are constructed during proof search, we will omit them here for readability. We begin by analyzing the `halts_step` lemma, previously presented in Chapter 2.2.2. This proof will highlight proof search over LF types.

```
rec halts_step : [ ⊢ step M M’] → [ ⊢ halts M’] → [ ⊢ halts M] =
fn s, h =>
  let [ ⊢ halts/m MS V] = h in let [ ⊢ S] = s in [ ⊢ halts/m (sstep S MS) V]
;
```

We concentrate on the construction of the final proof term:

```
[ ⊢ halts/m (sstep S MS) V]
```

Just before solving this subgoal, the state of the proof search in Beluga looks like:

```
MS: ( ⊢ steps M’ V’), V: ( ⊢ val V’), S: ( ⊢ step M M’)
;
⊢ [ ⊢ halts M]
```

where unbound meta-variables are left implicit. We present the state of proof search in Beluga in this way. With the two contexts appearing to the left of the sequent, meta-context denoted first, and separated by a semicolon. The goal will appear to the right of the sequent.

Beluga first attempts to look for a proof in LF, that is, by focusing on the right. By executing the level rule we transition to proof search in LF and the new goal then becomes:

```
MS: ( ⊢ steps M’ V’), V: ( ⊢ val V’), S: ( ⊢ step M M’)
;
⊢ halts M
```

Since the goal above is a positive formula, the uniform stage is trivially complete. The algorithm then proceeds to the LF focusing stage. The algorithm first attempts focusing on assumptions in Δ, executing the transition$^{\Delta}$ rule. In Beluga, all assumptions (both LF and computation-level) are compiled into hereditary Harrop formulas which consists of a head, subgoals, and bound variables. For example, the type of the term `halts/m`:

$$\Pi A : \text{tp}.\ \Pi M : \text{term } A.\ \Pi M' : \text{term } A.\ \text{steps } M\ M' \to \text{val } M' \to \text{halts } M$$

where A, M, and M' are implicit, is viewed as the horn clause:

$$(\text{steps } M\ M' \wedge \text{val } M') \to \text{halts } M$$

which gets compiled to:

```
{ head = halts M
; subgoals = steps M M', val M'
; boundVars = A: tp, M: term A, M': term A }
```

This form allows easy access to the head of assumptions. As subgoals are independent of one another, they are solved from right-to-left, but the constructed proof term maintains the order in which they occur in the goal formula.

In the implementation, we only focus in LF on assumptions whose head unifies with the goal. Therefore, the algorithm does not proceed with focusing on any assumption in Δ. Next we look for a solution in the dynamic assumptions in Ψ. Since Ψ is empty here, we then move on to focusing on assumptions from our signature Σ. These focusing rules are exactly like the focusing rules presented for Ψ. There is one constructor for `halts` in Σ, namely `halts/m`. Since the head of `halts/m` unifies with the goal we proceed with focusing on this assumption.

The list of bound variables in the assumption is non-empty, which implies that the assumption is of Π type. In the implementation of the ΠL rule, a substitution is created which holds instantiations for the bound variables in the assumption. This substitution is composed of meta-variables which represent existential variables that are to be instantiated during unification later.

Beluga then checks if there are any subgoals, implying that the focus is on an assumption of $\to$ type. Starting with the right-most assumption, Beluga searches for a uniform proof for each subgoal, following the $\to L$ rule. This process restarts the search loop for each subgoal.

```
MS: ( ⊢ steps M' V'), V: ( ⊢ val V'), S: ( ⊢ step M M')
;
⊢ val ?V
```

where `?V` stands for a meta-variable yet to be instantiated. This assumption is discharged by focusing on `V` in Δ which unifies `?V` with `V'`. The algorithm then proceeds to the next subgoal:

```
MS: ( ⊢ steps M' V'), V: ( ⊢ val V'), S: ( ⊢ step M M')
;
⊢ steps M V'
```

If no solution to this subgoal can be found, the algorithm backtracks to the previous state in the proof and resumes finding a proof of `val ?V`.

Returning to the search, no solution is found when focusing on Δ or Ψ, so it proceeds to look into Σ. Here, the only constructor with such whose head unifies with the goal is `sstep`. Repeating the steps above, the algorithm builds a substitution which holds the instantiations of the bound variables of `sstep` and proceeds to solving its first subgoal:

```
MS: ( ⊢ steps M' V'), V: ( ⊢ val V'), S: ( ⊢ step M M')
;
⊢ steps ?M V
```

This subgoal is discharged by focusing on `MS` which unifies `?M` with `M'`. The final subgoal to `sstep` is:

```
MS: ( ⊢ steps M' V'), V: ( ⊢ val V'), S: ( ⊢ step M M')
;
⊢ step M M'
```

which can be discharged with assumption `S`. This ends the proof.

In the implementation, proof search in LF is *bounded* in order to ensure the procedure terminates. We bound search by the depth of the search tree. That is, we increment depth when we attempt to solve the first subgoal for an assumption. Therefore if an assumption has more than one subgoal, the depth only increases once. By default, the depth is set to 3 for LF proof search, which in most cases is high enough.

The next example we present highlights the search in LF for substitutions. Recall, we need to build substitutions whenever we want to use a meta-assumption. In the previous example, we made this search implicit as all of the goals and meta-assumptions had empty local contexts. Therefore the substitution needed to use the meta-assumptions in all cases was the empty substitution, which can be solved trivially and is left implicit in the proof term.

To showcase this aspect of the solver, we introduce a new example. We show type preservation for the STLC. This property states that the typing of a term is preserved after the term has been evaluated. To show this, we require an extrinsically typed representation:

```
LF tp : type =
| b : tp
| arr : tp → tp → tp
;
```

```
LF exp : type =
| c : exp
| abs : tp → (exp → exp) → exp
| app : exp → exp → exp
;
```

We encode typing and evaluation rules:

```
LF oft : exp → tp → type =
| tp_c : oft c b
| tp_abs : ({x:exp} oft x T1 → oft (E x) T2)
           → oft (abs T1 (λx . E x)) (arr T1 T2)
| tp_app : oft E2 T2 → oft E1 (arr T2 T1)
           → oft (app E1 E2) T1
;

LF eval : exp → exp → type =
| ev_c : eval c c
| ev_abs : eval (abs T (λx. E x)) (abs T (λx. E x))
| ev_app : eval E1 (abs T (λx. E x)) → eval E2 V2
          → eval (E V2) V → eval (app E1 E2) V
;
```

Our type preservation theorem and proof then takes the form:

```
rec pres: {E : ( ⊢ exp)} {V : ( ⊢ exp)} {T : ( ⊢ tp)} [ ⊢ eval E V]
            → [ ⊢ oft E T] → [ ⊢ oft V T] =
/ total e (pres x v t e) /
mlam E, V, T ⇒ fn e, o ⇒ case e of
  | [⊢ ev_c] ⇒ o
  | [⊢ ev_abs] ⇒ o
  | [⊢ ev_app E1 E2 E3] ⇒
       let [ ⊢ tp_app O1 O2] = o in
       let [⊢ X] = pres [ ⊢ _] [ ⊢ _] [ ⊢ _] [ ⊢ E2] [ ⊢ O1] in
       let [⊢ tp_abs (λx. λz. O3)] =
         pres [ ⊢ _] [ ⊢ _] [ ⊢ arr _ T] [ ⊢ E1] [ ⊢ O2] in
       pres [⊢ _] [⊢ _] [⊢ _] [⊢ E3] [⊢ O3[_, X]]
;
```

In the proof above, notice that the meta-assumption in the final proof term (`[⊢ O3[_, X]]`) is paired with an explicit substitution. We will focus our attention to the construction of this term. Our proof state just before this point, and once transitioning to proof search in LF, is:

```
V: (⊢ exp), T: ( ⊢ tp), E1: (⊢ eval X1 (abs T1 (λx. X3 x))), E2: ( ⊢ eval X2 V2),
E3: ( ⊢ eval (X3 V2) V), O1: (⊢ oft X2 T1), O2: (⊢ oft X1 (arr T1 T)),
X: (⊢ oft V2 T1), O3: (x: exp, z: oft x T1 ⊢ oft (X3 x) T)
;
⊢ oft (X3 V2) T
```

Focusing in Δ first, there is one assumption whose head unifies with the goal, namely `O3`. Unlike the previous examples above, `O3` has a non-empty local context. Since we would like to use this assumption in our current goal state in which Ψ is empty, we must find the required substitution σ where $\Delta; \cdot \vdash \sigma : (x : \text{exp}, z : \text{oft } x\ T1)$.

Solving for LF substitutions is one area of incompleteness in the implementation. Currently, we may only search for the appropriate substitution terms in Δ. Fortunately, this is enough for us to complete this goal.

Once Beluga focuses on `O3`, it executes the sub rule. Again, a substitution is created in which the bound variables `x` and `z` are replaced by meta-variables, but this time we must search for the instantiations. The algorithm searches Δ to solve the subgoals, starting with the right-most one:

```
V: (⊢  exp), T: ( ⊢ tp), E1: (⊢  eval X1 (abs T1 (λx. X3 x))), E2: ( ⊢ eval X2 V2),
E3: ( ⊢ eval (X3 V2) V), O1: (⊢  oft X2 T1), O2: (⊢  oft X1 (arr T1 T)),
X: (⊢  oft V2 T1), O3: (x: exp, z: oft x T1 ⊢ oft (X3 x) T)
;
⊢ oft V2 T1
```

The assumption `X` discharges this goal. The final goal, `(⊢  exp)` is then instantiated with `V2`. Since this assumption is implicit in Δ, it appears as an underscore in the final proof term.

These two examples demonstrate the automated prover in Beluga for contextual LF. As it searches for a proof, this solver also constructs proof terms for LF types and substitutions. It executes a bounded search which backtracks upon encountering failure.

4.2 Beluga's Meta-Theorem Prover

The automated meta-theorem search procedure for Beluga is composed of two operations: search and split. The search phase proceeds similarly to proof search in LF, described in the previous chapter. The split phase may take several directions, depending on what the user or Beluga requests. They may request to do any type of (meta-)variable split, only inversions, or no splits at all. The search procedure is again bounded by the depth of the search tree, and implements backtracking if it encounters failure.

To demonstrate the capabilities of the meta-theorem prover, we examine how Beluga automatically solves the `bwd_closed` lemma previously presented in Chapter 2.2.2. We show the procedure which allows for any split to be made. The solver takes as input the goal state, consisting of the relevant contexts, the goal type, split index, and depth bound. A proof term is simultaneously constructed during the search, but we omit it in our example. We begin with empty contexts, an induction argument index of 1, and a depth bound of 3. The index is given with respect to the (meta-)variable's position in the overall goal, counted from left-to-right.

```
;
⊢ {A:[ ⊢ tp]} {M: [ ⊢ term A]} {M': [ ⊢ term A]}
    [ ⊢ step M M'] → Reduce [ ⊢ A] [ ⊢ M'] → Reduce [ ⊢ A] [ ⊢ M]
```

The search procedure begins with the uniform phase, beginning with uniform right. This concludes with the following proof state:

```
A: (⊢  tp), M: (⊢  term A), M': (⊢  term A)
; s: [⊢  step M M'], r: Reduce [⊢  A] [⊢  M']
⊢ Reduce [ ⊢ A] [ ⊢ M]
```

Once a positive goal is reached, Beluga enters into the uniform left phase. This concludes when Γ contains only positive assumptions.

```
A: (⊢  tp), M: (⊢  term A), M': (⊢  term A), S: (⊢  step M M')
; r: Reduce [⊢  A] [⊢  M']
⊢ Reduce [ ⊢ A] [ ⊢ M]
```

Since we explicitly stated a split index, the implemented search procedure detours from the theoretical procedure and splits on the respective variable immediately. As `tp` has two constructors, there are two cases to solve. Solving for the case of $A =$ `b` the goal becomes:

```
M: (⊢  term b), M': (⊢  term b), S: (⊢  step M M')
; r: Reduce [⊢  b] [⊢  M']
⊢ Reduce [ ⊢ b] [ ⊢ M]
```

After a split, the procedure immediately applies any inversions (splits that result in one subgoal). It first inverts `r`, introducing the assumption `h: [⊢ halts M']` then immediately inverts `h`, introducing assumptions `SS: (⊢ steps M' N)` and `V: (⊢ val N)` for implicit term `N: (⊢ term c)`. Finally, it inverts `V`, which introduces no new assumptions. The resulting proof state is then:

```
M: (⊢  term b), M': (⊢  term b), S: (⊢  step M M'), SS: (⊢  steps M' N)
;
⊢ Reduce [ ⊢ b] [ ⊢ M]
```

Once all inversions have been made, we return to the search process and proceed with focusing. We start with a blur phase that is skipped here as it is inapplicable, but will be discussed in the next example. Continuing with focusing, since the goal is not of box type, we cannot focus on the right. The algorithm then proceeds with focusing on the left, beginning with the computational signature, then local context Γ, ending with the available induction hypotheses. When focusing, assumptions whose head unifies with the goal are prioritized. It then focuses on the constructor `I` in the computational signature. This time, a meta-substitution is built, following the $\Pi^{\square}$ rule, which holds the instantiations of the bound meta-variables in the assumption. Beluga then checks if there are any subgoals to the assumption. There is one subgoal to solve, namely `[⊢ halts M]`.

```
M: (⊢ term b), M’: (⊢ term b), S: (⊢ step M M’), SS: (⊢ steps M’ c)
;
⊢ [⊢ halts M]
```

The procedure then restarts, this time with no explicit split index stated. Since the goal and assumptions are already all positive and no split index is specified, we advance straight to the focusing phase. We do not blur when solving subgoals coming from focusing. As the goal is of box-type, it tries first to focus on the right. There, the matching proof term is constructed: `[ ⊢ halts/m (sstep S SS) val/c]`.

In the second case, $A =$ `arr T1 T2`, the proof state begins at:

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M’: (⊢ term (arr T1 T2))
, S: (⊢ step M M’)
; r: Reduce [⊢ arr T1 T2] [⊢ M’]
⊢ Reduce [⊢ arr T1 T2] [⊢ M]
```

Similarly to above, immediately after splitting comes inversions. After the inversions we are left with:

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M’: (⊢ term (arr T1 T2))
, S: (⊢ step M M’), SS: (⊢ steps M’ (abs T1 N’))
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M’ N1]
⊢ Reduce [⊢ arr T1 T2] [⊢ M]
```

Focusing on the left, on the term `Arr`, presents us with two new subgoals. Subgoals are again solved from right-to-left.

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M’: (⊢ term (arr T1 T2))
, S: (⊢ step M M’), SS: (⊢ steps M’ (abs T1 N’))
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M’ N1]
⊢ {N: (⊢ term T1)} Reduce [⊢ T1] [⊢ N] → Reduce [⊢ T2] [⊢ app M N]
```

After the uniform phase, the proof state is:

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M’: (⊢ term (arr T1 T2))
, S: (⊢ step M M’), SS: (⊢ steps M’ (abs T1 N’)), N: (⊢ term T1)
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M’ N1]
, y: Reduce [⊢ T1] [⊢ N]
⊢ Reduce [⊢ T2] [⊢ app M N]
```

When focusing on the left, the only assumption with a head that unifies with the goal is an induction hypothesis/schema. In Beluga this schema looks like:

```
bwd_closed [ ⊢ T2] : {M1 : ( ⊢ term T2)}{M2 : ( ⊢ term T2)} [ ⊢ step M1 M2]
                     → Reduce [ ⊢ T2] [ ⊢ M2] → Reduce [ ⊢ T2] [ ⊢ M1]
```

These assumptions are treated the same as others, and get compiled into hereditary Harrop formulas. A meta-substitution is then created from the bound meta-variables (`M1` and `M2`). Since the assumption is of → type, there are subgoals to solve.

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M’: (⊢ term (arr T1 T2))
, S: (⊢ step M M’), SS: (⊢ steps M’ (abs T1 N’)), N: (⊢ term T1)
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M’ N1]
, y: Reduce [⊢ T1] [⊢ N]
⊢ Reduce [⊢ T2] [⊢ ?M]
```

The uniform stage is completed, trivially, so we move to focusing. The head of assumption `f` (i.e. `Reduce [⊢ T2] [⊢ app M' N1]`) unifies with our goal so we focus on `f`. There is then one subgoal to solve.

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M': (⊢ term (arr T1 T2))
, S: (⊢ step M M'), SS: (⊢ steps M' (abs T1 N')), N: (⊢ term T1)
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M' N1]
, y: Reduce [⊢ T1] [⊢ N]
⊢ Reduce [⊢ T1] [⊢ ?N]
```

Focusing on assumption `y` finishes this branch in the search tree. We move on to the second subgoal of the induction hypothesis.

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M': (⊢ term (arr T1 T2))
, S: (⊢ step M M'), SS: (⊢ steps M' (abs T1 N')), N: (⊢ term T1)
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M' N1]
, y: Reduce [⊢ T1] [⊢ N]
⊢ [⊢ step (app M N) (app M' N)]
```

Again, the goal is positive, so we proceed to focusing. Since the goal is of box-type, we first attempt to focus on the right. LF proof search finds a matching proof term, namely `[⊢ stepapp S]`, which completes the proof of the first subgoal of `Arr`.

```
T1: (⊢ tp), T2: (⊢ tp), M: (⊢ term (arr T1 T2)), M': (⊢ term (arr T1 T2))
, S: (⊢ step M M'), SS: (⊢ steps M' (abs T1 N')), N: (⊢ term T1)
; f: {N1: (⊢ term T1)} Reduce [⊢ T1] [⊢ N1] → Reduce [⊢ T2] [⊢ app M' N1]
, y: Reduce [⊢ T1] [⊢ N]
⊢ [⊢ halts M]
```

The next subgoal is also solved from focusing on the right, with the proof term `[⊢ halts/m (sstep S SS) (val/abs)]`. The final proof term Beluga constructs looks like:

```
mlam A, M, M' ⇒ fn s, r ⇒
  let [ ⊢ S] = s in
  case [ ⊢ A] of
  | [ ⊢ b] =>
    let I h = r in
    let [ ⊢ halts/m SS V] = h in
    let [ ⊢ val/c] = [ ⊢ V] in I [ ⊢ halts/m (sstep S SS) val/c]
  | [ ⊢ arr T1 T2] =>
    let (Arr h f : Reduce [ ⊢ arr T1 T2] [ ⊢ M']) = r in
    let ([ ⊢ halts/m SS V] : [ ⊢ halts M']) = h in
    let [ ⊢ val/abs ] = [ ⊢ V] in
    Arr [ ⊢ halts/m (sstep S SS) (val/abs )]
      (mlam N => fn y =>
      bwd_closed [ ⊢ T2] [ ⊢ app M N] [ ⊢ app M' N] [ ⊢ stepapp S]
        (f [ ⊢ N] y))
```

Our last example showcases the applications of the blur rule. This rule is applied when the assumption we focus on is atomic but does not unify with the goal. In this case, the atomic assumption is added to Γ and we resume focusing. This purpose of this rule is to introduce lemmas that are needed for LF proof construction.

For simplicity, we choose a basic example: value soundness for the the theory of natural numbers.

```
LF nat : type =
| z : nat
| suc : nat → nat
;
```

```
LF val  : nat → type =
  | v_z  : val z
  | v_s  : val N → val (suc N)
;
```

```
LF eval   : nat → nat → type =
  | ev_z  : eval z z
  | ev_s  : eval N V
              → eval (suc N) (suc V)
;
```

The soundness lemma and proof then take the form:

```
rec sound : {N : ( ⊢ nat)} {V : ( ⊢ nat)} [ ⊢ eval N V] → [ ⊢ val V] =
/ total e (sound n v e) /
mlam N, V => fn e => case e of
  | [ ⊢ ev_z] =>
    [ ⊢ v_z]
  | [ ⊢ ev_s E'] =>
    let y = sound [ ⊢ _] [ ⊢ _] [ ⊢ E'] in
    let [ ⊢ Z] = y in [ ⊢ v_s Z]
;
```

This property may be proved fully automatically in Beluga by induction on the third argument, `e`. The proof state at the start of the proof looks like:

```
N: (⊢ nat), V: (⊢ nat)
; e: [⊢ eval N V]
⊢ [⊢ val V]
```

We then call upon Beluga's theorem prover to finish the proof. The uniform right stage is trivially complete, so it moves on to the uniform left stage. Since we indicated that we are to perform induction on `e`, this argument is not unboxed for simplicity. We then immediately split. We are presented with the first case, when `e` is `[ ⊢ ev_z]`.

```
;
⊢ [⊢ val z]
```

This goal is easily discharged by performing LF proof search, or focusing on the right. We then move to the inductive case, that is, when `e` is `[ ⊢ ev_s E']` for `E': [⊢ eval N' V']`.

```
E': (⊢ eval N' V')
;
⊢ [⊢ val (suc V')]
```

There is one induction hypothesis generated, namely:
`sound [ ⊢ _] [ ⊢ _] [ ⊢ E'] : [ ⊢ val V']`. There are no inversions to be made, so the search loop restarts. The uniform phase is completed trivially; moving on to focusing. Before heading to proof search in LF, there is a blurring/lemma application phase that occurs. During this phase, we focus on the assumptions whose head does not necessarily unify with our goal and attempt to introduce the atoms they define. We begin by blurring on the assumptions in Γ. There are none, so we move on to the induction hypotheses; focusing on the only hypothesis. This hypothesis contains no universally quantified variables that require instantiation nor does it have any subgoals that require solving, thus the assumption `[ ⊢ val V']` is easily introduced.

```
E': (⊢ eval N' V')
; y: [⊢ val V']
⊢ [⊢ val (suc V')]
```

In the case that there are universally quantified variables requiring instantiation, the loop attempts to instantiate the variables with all possible combinations of meta-variables from Δ. In the case that there are subgoals to be solved, the prover will attempt to solve each subgoal by calling upon the search loop enforcing a search depth of 1. Therefore only trivial lemmas may be introduced.

Once the blurring phase is complete, we enter back into the uniform left phase, unboxing any newly introduced box-type assumptions.

```
E': (⊢ eval N' V'), Z: (⊢ val V')
;
⊢ [⊢ val (suc V')]
```

Finally, the loop enters the standard focusing phase, beginning with focusing on the right, in which it finds the desired proof term: `v_s Z`.

The lemma application rule is implemented naively, meaning we introduce every possible lemma that can be found. Unfortunately this sometimes means that *too many* assumptions are introduced that remain unused during proof development, which clutters proofs.

This completes our description of Beluga's meta-theorem prover. These examples showcased the main capabilities of the solver. In both examples, we used the tactic **`inductive-auto-solve`** which performs automatic induction on the specified argument, along with proof search on each case. We did not showcase any uses of the tactic **`auto-invert-solve`**, which is meant to be applied to the easily solvable subgoals of a complex proof. We discuss some of the uses of this tactic in the next subchapter.

4.3 Evaluation

The proof search procedures behind `inductive-auto-solve` and `auto-invert-solve` are the beginning of the implementation of the focusing calculi presented in Chapter 3.3. There are several areas of incompleteness that may be improved in the future. Nevertheless, the tactics are able to prove a number of interesting theorems both semi- and fully-automatically. We provide a summary of these case studies here.

Case study	Automation	Difficulty	Interesting proof features
MiniML/fix type preservation	Full	Advanced	Solving substitutions, I.H. appeal, inversions
MiniML/fix value soundness	Full	Basic	I.H. appeal
STLC weak-head normalization lemmas	Full	Intermediate	Inversions, depth bound, higher-order solving, I.H. appeal
STLC type uniqueness	Partial	Basic	I.H. appeal, inversions
Untyped lambda-calculus ordinary and parallel reduction lemmas	Full/Partial	Basic	I.H. appeal

Table 4.1: Overview of case studies

We highlight our case studies by the amount of automation that may be successfully applied. Theorems are proven with full automation using `inductive-auto-solve`, or with partial automation using `auto-invert-solve`. Partial automation is used on induction proofs when not all the cases fall within the prover's applicable subset. All proofs proceed by induction along with various features that we have outlined.

We use our tactics to prove key lemmas required to prove weak-head normalization for the STLC. These include in particular the termination property (`halts_step`) and backwards closed (`bwd_closed`) lemmas. The backwards closed lemma is particularly interesting as it requires bounded depth search, and higher-order function type solving. Type uniqueness for the STLC can be proven semi-automatically as two of its cases involve parameter variables and context block schemas.

We prove all lemmas regarding ordinary reduction for the untyped lambda-calculus automatically, and all but one lemma regarding parallel reduction automatically. These lemmas are used to prove equivalence of the ordinary and parallel reductions, and ultimately the Church-Rosser theorem for each reduction procedure.

For MiniML without fixpoints, we are able to prove type preservation and value soundness fully-automatically. Preservation in particular showcases the solvers ability to solve for substitutions.

Beluga's level of automation does not yet surpass that of Twelf's. However, Beluga is able to reason directly using logical relations, unlike Twelf. Certain properties, like normalization theorems, are most commonly proven using logical relations. In Twelf, users must find alternative proof methods [Schürmann and Sarnat, 2008; Abel, 2008], which may be conceptually different from on-paper formulations and require more work from the user to construct additional machinery. In Beluga, such logical relations may be directly translated from on-paper formulations and their proofs become simplified with the use of our automation tactics.

4.4 Limitations

As is the case with automated theorem provers, there are limits to what may be solved with our tactics. Some of these limitations are due to incompleteness and overall design of Beluga. We discuss a few of those here. It is important for users to understand the scope of what is provable with our tactics, so that they make the best use of them.

Incompleteness We previously mentioned that one area of incompleteness comes from fact that we do not perform full proof search when searching for a verifying substitution in the transition$^{\Delta}$ rule from the focusing calculus for LF. When we require a substitution of type $x_1 : B_1, ..., x_n : B_n$ we only search in Δ for the matching terms. This choice prevents us from automatically proving properties that require for example assumptions from Ψ to build substitutions, as in the proof of the admissibility of cut for intuitionistic sequent calculus. Another area of incompleteness arises from the blur rule in the focusing calculus for the computation logic. Currently, we only apply this rule on assumptions from Γ or induction hypotheses. We only focus on assumptions from the signature if the head of their compiled clauses unifies with our goal, as in LF proof search. Showing the equivalence of ordinary and parallel reductions for the untyped lambda calculus requires lemma application (which results from blurring) of previously proven theorems in the signature. Currently, we cannot prove such a theorem automatically.

Beluga design In order to use the tactics to assist in induction proofs, the goal formula should uphold certain conditions. For one, users must ensure they only quantify over variables that are bound within the body of the theorem. As an example, it is fine to write theorems as $\Pi^{\Box}\ X_1 : U.\ \Pi^{\Box}\ X_2 : (\Psi \vdash P).\ \tau$ where X_2 doesn't appear in τ. If we were to perform induction on X_1 however, this will cause issues for the prover if we choose to focus on an induction hypothesis. Applying the $\Pi^{\Box}L$ rule, there will never be an instantiation for the meta-variable replacing X_2 found as it does not occur in τ. This results in uninstantiated meta-variables being left in the final proof term. Therefore users must be conscientious when formulating their meta-theorems. Users should also formulate their theorems in a way that if their induction variable is an ordinary variable, it should be the left-most antecedent. That is, if we want to induct on the assumption `[⊢ eval E V]` in the theorem `[ ⊢ oft E T] → [ ⊢ eval E V] → [ ⊢ oft V T]` then we should instead formulate the theorem as: `[ ⊢ eval E V] → [ ⊢ oft E T] → [ ⊢ oft V T]`. This is because issues can arise in the generation of the induction hypotheses. Induction schema generation is still an area of active development in Beluga.

Chapter 5

Conclusion

5.1 Future Work

We discuss here possible implementation improvements that may be made to the Beluga theorem prover. We do not include extensions that would make the prover complete with respect to the logic presented in this thesis.

Support other meta-types

In Beluga's implementation of contextual LF, meta-types also include parameter and substitution variables [Cave and Pientka, 2013]. Parameter variables allow users to reference assumptions within LF contexts. Substitution variables allow for the first-class treatment of substitutions, providing a richer, more expressive logic. Substitution variables have shown to be particularly useful for implementing normalization proofs [Cave and Pientka, 2013]. Currently, Beluga's theorem prover lacks support for such meta-types. This also limits the theorem prover's expressive power. Extending both the prover but also the logic presented in this thesis with these meta-types would be interesting future research, and due to the uniform treatment of meta-types in Beluga's logic, this extension should be straight forward. Supporting these meta-types would allow for more automatic reasoning about open LF objects, which is required for many interesting theorems.

Context blocks

The implementation is also lacking support for another interesting aspect of BELUGA's logic, context block schemas. Such a schema is a generalization of regular context schemas whereby each schema element is constructed from a Σ-type, which allows grouping of multiple declarations. These too are used to reason about open LF objects. This extension would however be less straightforward due to BELUGA's induction hypothesis generation. During this process, blocks are flattened and thus lose their schema's shape. Therefore more research will need to be done to correctly incorporate these schemas into the automated prover.

Automatic splitting

A more experimental aspect of the prover that we have not discussed yet is its ability to conduct case analysis and inductions automatically. This feature is far from being robust but would be an interesting addition to make. Currently, there is an option to allow automatic splitting to occur whenever a loop of the search procedure has conducted and no proof has been found. In this case, a variable is chosen using a basic heuristic and the solver attempts to solve each subgoal. The problem of choosing the right variable to split on is infamously difficult, and currently there seems to be no simple way to solve this. Therefore it is crucial the solver handles failure well.

Improved failure handling

Planning for failure is something all theorem proving designers must keep in mind when creating their systems. If not, a search may easily result in a non-terminating loop, which deters users. Ideally, if a proof does not exist for a proposition, the solver will fail quickly.

This is of course easier said then done. Currently, the practices in place to plan for failure in Beluga include bounding the search tree depth, and bounding the number of times a case analysis or induction may occur. There are more sophisticated techniques researchers have studied which have the potential to improve failure handling in Beluga that would be interesting to explore.

Memoization is one such technique used to speed up computation by reusing subcomputations [Michie, 1968; Pientka, 2003, 2005]. Normally in proof search if a branch of the search tree is taken and found to not be the correct one, the system backtracks and forgets everything learned while computing that branch. Using memoization, the system stores learned information within a table and can refer to it later, like caching. This technique can also be used to detect loops, also a way to speed up search (and failure). Implementing memoization in Beluga would probably take extensive work, but it has the potential to make the system much more robust.

Machine learning has also been applied to theorem provers in various settings [Denzinger et al., 1997; Meng and Paulson, 2009; Goller, 1997]. One such way is to determine the best heuristic to use during search. Algorithms can find similarities and differences between the structures of propositions and from these, determine the possible proof structure of an unproven statement. As research in this field progresses, it may be worth exploring the application of machine learning to meta-theorem provers, such as Beluga.

5.2 Summary

In closing, we have presented the the theorem and meta-theorem provers behind Beluga. These provers perform proof search over a core subset of Beluga's logic which allows for the automatic completion of many simple lemmas and cases of PL theory proofs. Users of Beluga may now bypass these simple proof and focus their energy on the interesting cases. This is the first step in adding full automation to Beluga. Along with our implementation, we provide a theoretical foundation for our solvers in the form of a cut-free sequent calculus, which is easy to understand, and a sound and complete focusing calculus, which closely reflects our implementation. These provide us with a way to study our implementation and ensure its correctness.

Our next steps are to expand the solver so that its proving capabilities are equivalent to that of the logic presented in this thesis. After that, we plan to add support for context block schemas, and substitution and parameter variables, which should bring its proving power up to that of Twelf's.

Bibliography

Andreas Abel. Normalization for the Simply-Typed Lambda-Calculus in Twelf. volume 199, pages 3–16, 2008. doi: https://doi.org/10.1016/j.entcs.2007.11.009. URL `https://www.sciencedirect.com/science/article/pii/S1571066108000753`. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing Proofs by Logical Relations. *Journal of Functional Programming*, 29, 2019. doi: 10.1017/S0956796819000170.

Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lecture Notes in Computer Science (LNCS)*, page 13–30, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540440399.

Jean-Marc Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. doi: 10.1093/logcom/2.3.297. URL `https://doi.org/10.1093/logcom/2.3.297`.

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 Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05*, volume 3606 of *Lecture Notes in Computer Science (LNCS)*, page 50–65, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540283722. doi: 10.1007/11541868_4. URL `https://doi.org/10.1007/11541868_4`.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, Nate Foster, Benjamin Pierce, Peter Sewell, Jeff Vaughan, Dimitris Vytiniotis, Geoff Washburn, Stephanie Weirich, and Steve Zdancewic. POPLmark. `https://www.seas.upenn.edu/~plclub/poplmark/`, April 2012.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series (TTCS). Springer Berlin, Heidelberg, 2004.

Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science (LNCS)*, pages 131–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14052-5.

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems (FroCoS)*, volume 6989 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 12–27, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24364-6.

Andrew Cave and Brigitte Pientka. Programming with Binders and Indexed Data-Types. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles* *of Programming Languages (POPL'12)*, page 413–424, New York, NY, USA, January 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103705. URL `https://doi.org/10.1145/2103656.2103705`.

Andrew Cave and Brigitte Pientka. First-Class Substitutions in Contextual Type Theory. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, page 15–24, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323826. doi: 10.1145/2503887.2503889. URL `https://doi.org/10.1145/2503887.2503889`.

Kaustuv Chaudhuri, Ulysse Gérard, and Dale Miller. Computation-as-deduction in Abella: work in progress. In *13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'18)*, Oxford, United Kingdom, July 2018. URL `https://hal.inria.fr/hal-01806154`.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Notices*, 35(9):268–279, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351266. URL `https://doi.org/10.1145/357766.351266`.

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. ISSN 1385-7258. doi: https://doi.org/10.1016/1385-7258(72)90034-0.

Jorg Denzinger, Marc Fuchs, and Matthias Fuchs. High Performance ATP Systems by Combining Several AI Methods. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence*, IJCAI'97, page 102–107. Morgan Kaufmann Publishers Inc., 1997. ISBN 15558604804.

Maarten H. Van Emden and Robert A. Kowalski. The Semantics of Predicate Logic as a

Programming Language. *Journal of the Association for Computing Machinery (ACM)*, 23 (4):733–742, Oct 1976.

Jacob Errington, Junyoung Jang, and Brigitte Pientka. Harpoon: Mechanizing Metatheory Interactively: (System Description). In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event*, volume 12699 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 636–648. Springer, Cham, July 2021. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_38.

Andrew Gacek. The Abella Interactive Theorem Prover (System Description). In *Automated Reasoning - 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, volume 5195 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 154–161, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining Generic Judgments with Recursive Definitions. In *23rd Annual IEEE Symposium on Logic in Computer Science*, pages 33–44, Los Alamitos, CA, USA, June 2008. IEEE Computer Society. doi: 10.1109/LICS.2008.33.

Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935. URL `http://eudml.org/doc/168546`. English translation in Szabo [1969].

Christoph Goller. *A connectionist approach for learning search-control heuristics for automated deduction systems*. PhD thesis, Technical University of Munich, 1997.

Michael J.C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

Robert Harper and Frank Pfenning. On Equivalence and Canonical Forms in the LF Type Theory. *ACM Transactions on Computational Logic*, 6(1):61–101, jan 2005. doi: 10.1145/1042038.1042041.

Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, January 1993.

John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science (LNCS)*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Samuli Heilala and Brigitte Pientka. Bidirectional Decision Procedures for the Intuitionistic Propositional Modal Logic IS4. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, page 116–131, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540735946. doi: 10.1007/978-3-540-73595-3_9. URL `https://doi.org/10.1007/978-3-540-73595-3_9`.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, page 173–184, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595935754. URL `https://doi.org/10.1145/1190216.1190245`.

Xavier Leroy. A locally nameless solution to the POPLmark challenge. Technical Report 6098, INRIA, January 2007. URL `https://hal.inria.fr/inria-00123945v2`.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. doi: 10.1145/1538788.1538814. URL `https://hal.inria.fr/inria-00415861`.

Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2007.07.004. URL `https://www.sciencedirect.com/science/article/pii/S1570868307000626`.

Donald Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, 1968.

Dale Miller and Alexis Saurin. From Proofs to Focused Proofs: A Modular Proof of Focalization in Linear Logic. In Jacques Duparc and Thomas A. Henzinger, editors, *Annual Conference for Computer Science Logic*, pages 405–419, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74915-8.

Dale A. Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic,*, 51:125–157, 1991.

Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax. In *Proceedings of the Second International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07)*, volume 196 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 85–93, January 2008. doi: https://doi.org/10.1016/j.entcs.2007.09.019.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic*, 9(3), June 2008. ISSN 1529-3785. doi: 10.1145/1352582.1352591. URL `https://doi.org/10.1145/1352582.1352591`.

Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

Lawrence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving* *in Higher Order Logics (TPHOLs'07)*, volume 4732 of *Lecture Notes in Computer Science (LNCS)*, pages 232–245, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 202–206. Springer, 1999.

Frank Pfenning and Carsten Schürmann. Twelf User's Guide- Version 1.4, Dec 2002. `http://www.cs.cmu.edu/~twelf/guide-1-4/`.

Brigitte Pientka. *Tabled Higher-Order Logic Programming*. PhD thesis, Carnegie Mellon University, 2003.

Brigitte Pientka. Tabling for Higher-Order Logic Programming. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE-20)*, Lecture Notes in Artificial Intelligence (LNAI), pages 54–68, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31864-4. doi: https://doi.org/10.1007/11532231_5.

Brigitte Pientka. A Type-Theoretic Foundation for Programming with Higher-Order Abstract Syntax and First-Class Substitutions. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, volume 43, page 371–382, New York, NY, USA, January 2008. Association for Computing Machinery. doi: 10.1145/1328897.1328483. URL `https://doi.org/10.1145/1328897.1328483`.

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'15)*, volume 38 of *Leibniz International Proceedings in Informatics*

(LIPIcs), pages 273–287, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming Proofs. In *25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science (LNCS)*, pages 272–281. Springer, 2015. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6_18.

Brigitte Pientka and Jana Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'08)*, PPDP '08, pages 163–173, New York, NY, USA, July 2008. Association for Computing Machinery Press.

Brigitte Pientka and Jana Dunfield. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In Jürgen Giesl and Reiner Hähnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, volume 6173 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 15–21, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14203-1.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2022. Version 6.2, `http://softwarefoundations.cis.upenn.edu`.

Andrew M. Pitts. Nominal Logic: A First Order Theory of Names and Binding. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software*, pages 219–242, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45500-4.

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*, pages 359–374, Cham, 2015. Springer International Publishing.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Carnegie Mellon University, 2000.

Carsten Schürmann and Jeffrey Sarnat. Structural Logical Relations. In *23rd Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 69–80, Pittsburgh, PA, USA, 2008. IEEE Computer Society. doi: 10.1109/LICS.2008.44.

Johanna Schwartzentruber and Brigitte Pientka. Semi-Automation of Meta-Theoretic Proofs in Beluga. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'23)*, Electronic Notes in Theoretical Computer Science (ENTCS), 2023. to appear.

Christopher A. Stone and Robert Harper. Extensional Equivalence and Singleton Types. *ACM Transactions on Computational Logic (TOCL)*, 7(4):676–722, October 2006. ISSN 1529-3785. doi: 10.1145/1183278.

M.E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969. English translation of Gentzen [1935].

William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967. doi: 10.2307/2271658.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework: The Propositional Fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 355–377, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24849-1.

Markus Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine

Paulin, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, volume 1690 of *Lecture Notes in Computer Science (LNCS)*, page 167–183, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48256-7.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *ACM SIGPLAN Notices*, 46(6):283–294, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL `https://doi.org/10.1145/1993316.1993532`.