Realizing Dependently Typed Logic Programming

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zachary Snow

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Gopalan Nadathur, Advisor
May, 2010

Acknowledgments

I owe many people a debt of gratitude for supporting me during this thesis. First, my thanks to Gopalan Nadathur, my advisor, for his encouragement and criticism, and for his insistent motivation throughout. I would like to thank David Baelde, for his insights, generous help, and friendship. And thanks to Andrew Gacek, for lending me his experience. Thanks to Eric Van Wyk and Wayne Richter for serving on my committee. And finally, thank you Molly, for everything else.

This work has been supported by the NSF grants CCR-0429572 and CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Abstract

Dependently typed λ-calculi such as the Edinburgh Logical Framework (LF) can encode relationships between terms in types and can naturally capture correspondences between formulas and their proofs. Such calculi can also be given a logic programming interpretation: the Twelf system is based on such an interpretation of LF. We consider here whether a conventional logic programming language can also provide the benefits of a Twelf-like system for encoding type and term dependencies through dependent typing, and whether it can do so in an efficient manner. In particular, we present a simple mapping from LF specifications to a set of formulas in the higher-order hereditary Harrop (*hohh*) language, that relates derivations and proof-search between the two frameworks. We then show that this encoding can be improved by exploiting knowledge of the well-formedness of the original LF specifications to elide much redundant type-checking information. The resulting logic program has a structure that closely follows the original specification, thereby allowing LF specifications to be viewed as meta-programs that generate *hohh* programs. We prove that this mapping is correct, and, using the Teyjus implementation of λProlog, we show that our translation provides an efficient means for executing LF specifications, complementing the ability the Twelf system provides for reasoning about them. In addition, the translation offers new avenues for reasoning about such specifications, via reasoning over the generated *hohh* programs.

Contents

Chapter 1

Introduction

There is a significant and growing interest in mechanisms for specifying, prototyping and reasoning about formal systems that are described by syntax-directed rules. Attention has recently been focused on these aspects by the desire to mechanically validate properties of programming languages [1], but there are also more practical and longer-term motivations arising from concerns about software correctness and security (*e.g.*, see [2, 3, 4]).

The operational semantics of formal systems, particularly when defining them or reasoning over them "by hand", are often given by syntax-directed descriptions. There are many ways in which such *specifications* can be formalized, differentiated in particular by the choice of logic in which to work. For instance, specifications can be written in a style in which predicates are used to define all of the relationships necessary to fully describe the system. Another technique employs types to encode these relationships.

The goal of this thesis to describe these techniques, to identify a correspondence between them, and then to exploit this correspondence in two ways. First, to provide an effective implementation of a types-based encoding by employing existing research on implementing predicate-based ones. And second, to open avenues for reasoning over specifications written in the first style by employing existing tools for reasoning over those written in the latter. Each of these goals are accomplished by *translating* formal specifications from a types-based encoding to a predicate based encoding.

1.1 Formal systems and syntax-directed descriptions

In order to develop a feeling for the kinds of formal systems we are concerned with, how such systems can be described, and how these descriptions can be used, let us consider an example: that of the simply typed λ-calculus (*stlc*), along with a call-by-value evaluation strategy for it. This problem will provide a running example throughout; it is intended to reflect the real-world application of LF in specifying formal systems, and it has been chosen to eventually illustrate various properties of both LF and of our implementation approach.

We first treat the syntax of terms in the language, restricting ourselves to a definition of

$$\frac{}{\Gamma \Vdash \mathit{void} : \mathit{unit}}\ \textit{type-void}$$

$$\frac{}{\Gamma_1, x : T, \Gamma_2 \Vdash x : T}\ \textit{type-var}$$

$$\frac{\Gamma, x : T_1 \Vdash M : T_2}{\Gamma \Vdash \lambda x{:}T_1.M : T_1 \to T_2}\ \textit{type-abs}$$

$$\frac{\Gamma \Vdash N : T_1 \qquad \Gamma \Vdash M : T_1 \to T_2}{\Gamma \Vdash (M\ N) : T_2}\ \textit{type-app}$$

Figure 1.1: Typing for the *stlc*

$$\frac{}{\mathit{void} \Rightarrow \mathit{void}}\ \textit{eval-void}$$

$$\frac{}{\lambda x{:}T.M \Rightarrow \lambda x{:}T.M}\ \textit{eval-abs}$$

$$\frac{M_1 \Rightarrow \lambda x{:}T.M_1' \quad M_2 \Rightarrow M_2' \quad M_1'[M_2'/x] \Rightarrow N}{M_1\ M_2 \Rightarrow N}\ \textit{eval-app}$$

Figure 1.2: Evaluation for the *stlc*

the object language that consists of terms that are applications, abstractions, and variables, along with types that are arrows. To these we add the atomic type *unit*, and the term *void* of type *unit*. Letting T range over types, and M and N range over terms, the syntax is as follows:

$$\begin{array}{lcl} T & := & \mathit{unit} \mid T \to T \\ M, N & := & x \mid \mathit{void} \mid M\ N \mid \lambda x{:}T.M \end{array}$$

Once we have fixed the syntax of the language, we can think of specifying various semantical aspects of it through rules based on this syntax. One example of this is that of relating terms with types; we define the judgment $\Gamma \Vdash M : T$, denoting that M has type T under a context Γ. The rules in Figure 1.1 formalize this relationship. Another kind of relation that one might want to define on two *stlc* terms is that of evaluation. For instance, we define a call-by-value evaluation strategy for the *stlc* in the judgment $M \Rightarrow N$, indicating that M evaluates to N using this strategy. Derivations for this judgment are constructed using the inference rules of Figure 1.2.

These rules are *syntax directed* in that a particular rule applies to *stlc* terms or types whenever their syntactic structure matches that prescribed by the rule. Each of these inference rules applies to a particular syntactic structure; for instance, in the case of evaluation, *eval-abs* is responsible for evaluating an abstraction, and states that any abstraction evaluates to itself. Using these rules we can show, for instance, that $(\lambda x{:}unit.x)\ void$ evaluates to *void* by constructing a derivation for $(\lambda x{:}unit.x)\ void \Rightarrow void$. Guided by the syntax of $(\lambda x{:}unit.x)\ void$, we see first that we must apply the *eval-app* rule, and therefore must find derivations for $(\lambda x{:}unit.x) \Rightarrow (\lambda x{:}unit.x)$ and $void \Rightarrow void$; the first is by application of *eval-abs*, the second by application of *eval-void*.

Given such a specification, we can think of employing it in several ways. First, we might use it to reason about the formal system. In our example, we might think of various properties we wish to prove about the *stlc*. We may want to show that the types of terms are preserved under the evaluation relation; this property is often referred to as *subject reduction* and is a key to using types to guarantee the absence of certain kinds of errors during evaluation. Subject reduction for the *stlc* can be expressed as the following property of our specifications: if $\Gamma \Vdash M : T$ and $M \Rightarrow N$, then it must be the case that $\Gamma \Vdash N : T$. We might prove this property by reasoning over the structure of our formal specification.

Second, an important reason to formally specify a particular language or logic is so that it is possible to develop a conforming implementation. For instance, we might define an operational semantics for a programming language, and then use that to ensure that a particular implementation is correct. However, if we have specified a language in an appropriate specification language, we might also think of using the specification itself as an implementation, or prototype thereof. In our example of the *stlc*, we could think of using our specification of call-by-value evaluation to actually evaluate terms under this strategy. For instance, whereas above we were able to *check* that a particular term M evaluates to another term N, here we could attempt to evaluate the term $M = (\lambda x{:}unit.x)\ void$ by *searching* for a term N such that $M \Rightarrow N$ has a derivation.

1.2 The dependently typed λ-calculus as a specification language

Dependently typed λ-calculi such as the Edinburgh Logical Framework (LF) [5] provide many conveniences from a specification perspective. In LF we may define types, and define objects of a given type. However, unlike in many (perhaps more familiar) systems, types in LF can depend on, or be *indexed by*, terms. As an example of a type that is dependent in this sense, consider the traditional encoding of lists of natural numbers. Calling this type *list*, we can think of constructing lists from *nil*, the empty list, and *cons*, a constructor taking a natural number and a list. The type *list* yields very little information about the structure or properties of a particular list; for instance, we cannot say whether a particular list has any elements based solely on its type. This can lead to problems when defining partial functions on lists, like *hd* that returns the first element of a non-empty list: as the type *list* cannot indicate that a given list must be non-empty, the type of *hd* cannot require that its argument be a non-empty list; it can only require that its argument be a list. However, we can think of indexing the type of lists by another term, this one a natural number indicating the length of the list. Then by careful definition of the constructors for this new type we can guarantee that the number by which the type of a list is indexed corresponds exactly to the length of the list. For instance, we can say that *nil* has type *list* 0, indicating a list of length 0, and that *cons* takes a natural number and an object of type *list* n, returning an object of type *list* $(n + 1)$. Now the type of a given list *does* contain useful information, and we can use this information to write more precise types of functions like *hd*: in this case, *hd* should only be applied to an object of type *list* n, where $n > 0$.

To understand how the dependently typed λ-calculus can be used in the task of specifying formal systems, we return to our running example. We might think of simply encoding the types of the *stlc* as objects of type *ty* in the dependently typed λ-calculus, and encoding terms of the *stlc* as objects of type *tm*. For instance, we could introduce a new object *unit* of type *ty*, an object *void* of type *tm*, and a constructor *app* taking two arguments of type *tm* and building an object of type *tm*; we are simply specifying the abstract syntax of terms and types in the *stlc*.

Given this encoding of the syntax of the *stlc*, we could proceed to defining various aspects of its semantics. Here again we turn to types, and in particular dependent types. We can think of evaluation $M \Rightarrow N$ as a relationship between two terms M and N, and in fact we can encode a relation between terms in LF using a type that depends on these terms; here we have one representing M, and the other representing N. Calling this new type *eval* m n (where m and n are encodings of M and N in the dependently typed λ-calculus, respectively), we define constructors for this type such that some LF term has this type *only when* $M \Rightarrow N$ has a derivation. With this in mind, we can think of any such term as representing a *proof* that M indeed evaluates to N. We can thus understand type checking a term of this type as *verifying* that M evaluates to N.

The notion that type checking can be viewed as proof checking is a powerful one, and leads to a particularly useful observation: given that an term of this type corresponds to a proof of the relevant judgment, we can think of *searching* for a term of this type as being equivalent to searching for a proof of that judgment. This presents an obvious route toward employing such specifications in the prototyping of formal systems.

1.3 Predicate logic as a specification language

Another approach is to employ predicate logics when defining specifications of formal systems. In this setting we can think of defining, for each judgment we require, a predicate whose clauses correspond to inference rules for constructing proofs of the judgment. In the context of our running example of the *stlc*, we might define a binary predicate *eval* corresponding to call-by-value evaluation, that holds whenever the first term evaluates to the second. We then can define clauses for this predicate, one for each different inference rule. For instance, as we have seen, the evaluation of an abstraction yields the abstraction itself; we would therefore define a clause for *eval* that holds whenever both arguments represent the same abstraction.

To make this discussion concrete we first consider encoding the abstract syntax of the terms and types of the *stlc* as λ-terms in a predicate logic. We follow the same approach as in Section 1.2, introducing a new constant *void*, and constructors *app* and *abs*. In the case of *abs* we employ an abstraction at the level of the predicate logic in which we are working to encode *stlc*-level binding. That is, the constructor *abs* takes an argument representing an *stlc* type, and a argument that represents an *stlc* term, *abstracted* at the predicate logic level to represent binding within the term.

Next we define the evaluation relation as a predicate relating two λ-terms encoding *stlc* terms. As in the dependently typed λ-calculus our definition of this relation is syntax directed. Notice that for this specification to be meaningful as written, the predicate logic in which it is written must allow for predicates over λ-terms, and for analyzing abstractions in such terms.

$$\begin{array}{l} eval\ void\ void, \\ \forall t.\forall m.eval\ (abs\ t\ m)\ (abs\ t\ m), \\ \forall t.\forall m.\forall m'.\forall n.\forall n'.\forall v.eval\ m\ (abs\ t\ m') \supset eval\ n\ n' \supset eval\ (m'\ n')\ v \supset \\ \qquad eval\ (app\ m\ n)\ v \end{array}$$

Then, as in the dependently typed λ-calculus, we can verify that a particular term M evaluates to N under this semantics by providing a derivation in the predicate logic of *eval* M N. And given tools for reasoning over such specifications we might also prove various properties, for instance subject reduction, about the semantics of the *stlc* through theorems about this specification.

There are several benefits to this approach to specifying formal systems. First, the granularity of this approach is very fine: we can think of introducing additional conditions at any point in a specification very easily, by simply adding an additional premise to a clause. Second, there has already been a significant amount of research into both the efficient implementation of logics, and tools and techniques for reasoning over specifications written in such logics. However, along with this granularity comes a problem: specifications defined in this way can become very *ad hoc*, making it easier to under-specify, or even incorrectly specify, a particular formal system.

1.4 Relating the specification approaches

Given that we have in mind two distinct approaches to specifying formal systems, we turn to investigating the relationship between them. For example, as we have just seen that we can encode the evaluation relationship between two *stlc* terms either by using a predicate definition or by using a collection of dependent types. This implies that it might be possible to represent many, or even all, dependent types based specifications using predicate definitions. Moving in the other direction, we can think of the property of an object being well-typed in LF as being defined by a predicate between a context, an object, and a type. We could therefore define *this* relation in a predicate logic. From this we can infer that there is perhaps some level of equivalence between dependently typed logics and predicate logics, and therefore between specifications written in a dependently typed logic and predicate-based specifications.

We therefore might ask whether such kinds of transformations can be realized automatically, in such a way as to provably preserve the properties of the transformed relations. Given that there exist efficient implementations of predicate logics, we might also wonder whether animation of specifications written in a dependently typed system might be accomplished by transformation to a predicate logic, employing existing implementations of such logics. And finally, we might wonder whether such a translation can be made transparent, so that the programmer can treat specifications written in a dependently typed style as *meta-programs* that generate predicate specifications. Our answers to each of these questions are affirmative, and constitute the body of this thesis.

1.5 Outline of the thesis

This thesis is organized as follows. In the first chapter we introduce dependently typed logic programming, and describe existing languages and systems that implement it. We then describe the higher-order predicate logic that underlies our implementation of dependently typed logic programming, and similarly describe a language and system that implements it. The primary contributions of this thesis then follow: we first describe in Chapter 4 a translation from the programs written in the former to programs in the latter as a means to implementing dependently typed logic programming, and prove its correctness. We identify some problems with this initial translation, which we address in Chapter 5 with an improved translation, which we also prove correct. In Chapter 6 we provide experimental evidence that this technique for implementing dependently typed logic programming is more than competitive with existing methods in many cases. We conclude with a description of related work and an indication of some future directions for this research.

Chapter 2

Dependently Typed Logic Programming

In this chapter we introduce the notion of logic programming based on a dependently typed λ-calculus. Within such a calculus, types can have a sophisticated structure: in particular, they can be parameterized by terms. In this setting, we can interpret types as formulas using the well-known Curry-Howard isomorphism [6] between these two sorts of objects. In logic programming, we are typically interested in proving particular formulas. Under the mentioned isomorphism, this becomes identical to the task of showing that a type is inhabited; that is, that there exists some object having the type. Notice that in solving the inhabitation problem, we will actually be finding terms that encode proofs for the formulas to which they correspond.

There are several advantages of this approach to logic programming. First, under this interpretation it is straightforward to not only execute a particular logic program, thereby proving that a formula has a derivation, but also extract a proof of the given formula. This has applications in domains such as proof carrying code and certified compilers. Second, given that terms correspond to proofs and types to judgments, we can apply type-checking to proof terms to mechanically verify that they are indeed proofs of a particular judgment. Finally, reasoning over dependently typed logic programs is simplified by the fact that programs are already formulas, whose validity may be ascertained through type-checking, so that it is possible to specify properties of such programs same calculus.

We develop this model of logic programming using a specific dependently typed λ-calculus: the Edinburgh Logical Framework [5]. In the next section, we present this framework and then describe how it can be used for logic programming. We describe Twelf, an implementation of the Edinburgh Logical Framework with extensions that add convenience and power to the logic programming experience. And we give an example of logic programming in Twelf that will serve to illustrate both the power of dependent types, and several key concepts throughout this thesis. Finally we describe reasoning over Twelf specifications.

2.1 The Edinburgh Logical Framework

The Edinburgh Logical Framework is a language for specifying formal systems like logics and programming languages. Expressions in LF fall into three categories. There are *kinds*, *type families*, and *objects* (which are also called *terms*). In LF objects are classified by type families, and type families are classified by kinds. We refer to a nullary type family as simply a *type*. We assume that there are two denumerable sets of variables: object variables, denoted x and y, and type variables, denoted u and v. Then, letting K range over kinds, A and B over type families, and M and N over objects, the following defines the syntax of LF expressions.

$$\begin{array}{rcl} K & := & \mathit{Type} \mid \Pi x{:}A.\ K \\ A & := & u \mid \Pi x{:}A.\ B \mid \lambda x{:}A.B \mid A\ M \\ M & := & x \mid \lambda x{:}A.M \mid M\ N \end{array}$$

We use P and Q to refer to expressions of any of these three categories, and let w range over both object and type variables. Here λ corresponds to traditional λ-abstraction, and Π corresponds to typed universal quantification. The scope of a λ-abstraction or Π-quantifier extends as far to the right as possible, with parentheses used only for grouping. We write $P \to Q$ for $\Pi x{:}P.\ Q$ when x does not occur free in Q. Finally, expressions differing only in the names of bound variables are identified.

Having defined the syntax of LF expressions, there are several important properties of such expressions that we should like to determine — Is a given object well-typed? Is a given type valid? All of these questions suggest the fact that the definition of LF syntax above admits some invalid expressions; we are eventually only interested in those that satisfy these kinds of properties, however the dependent nature of LF leads to a kind of circularity in defining them. The well-typedness of LF objects, and the well-kindedness of LF type families, are expressed relative to *contexts*. Contexts are finite sets of assignments of types to object variables, and kinds to type variables. Letting Γ and Δ range over contexts, their syntax is defined as follows:

$$\Gamma \quad := \quad \cdot \mid \Gamma, u : K \mid \Gamma, x : A$$

Here $\cdot$ indicates the empty context. We write $\mathit{dom}(\Gamma)$ to refer to the object and type variables bound by Γ. We sometimes refer to the elements of a context as *context items*.

One can think of an LF context as consisting of two parts: an initial segment, that contains bindings for the various constants in a specification, and a dynamic segment, that may grow and shrink when deriving particular judgments. The first part is often referred to, especially in a programming setting, as a *signature*.

Given the above definition of contexts, we can now write formally those important properties of LF expressions. The first states that a given context is *valid*, the second that a kind is well-formed, the third that a type family has a particular kind, and finally that an object has a particular type. We present these simultaneously because they are, as mentioned, intertwined; for instance, when checking that a particular type is valid under some context, we will eventually be obligated to show that the context is valid — but checking a context for validity will in turn necessitate checking that a separate type is itself valid. Note that some presentations of LF derivations separate the notion of context presented here into the aforementioned two parts: a signature for initial variable, or constant, bindings, and a context for dynamic variable bindings. Such presentations are equivalent to the one given here.

$$\begin{gathered} \vdash \Gamma\ \mathit{ctx} \\ \Gamma \vdash K\ \mathit{kind} \\ \Gamma \vdash A : K \\ \Gamma \vdash M : A \end{gathered}$$

The rules for deriving judgments of these forms are given in Figure 2.1. This presentation of the rules of LF is due to [5], where it is called *simplified* LF. Notice that for a context to be well-formed it must not contain multiple assignments to the same variable. To adhere to this requirement, bound variable renaming may be entailed in the use of the *pi-kind*, *pi-fam*, *abs-fam* and *abs-obj* rules.

Given this definition of LF we can think of formally specifying a system; for simplicity we pick lists. However, to better demonstrate the use of dependent types, we take lists of natural numbers, and *index* them by their length. First we define natural numbers in the style of Peano arithmetic:

$$\begin{gathered} \mathit{nat} : \mathit{Type} \\ z : \mathit{nat} \\ s : \mathit{nat} \to \mathit{nat} \end{gathered}$$

Next we define the type family of lists, which depend on a natural number. We define constructors for lists in such a way as to ensure that well-formed lists have a type that corresponds to their length:

$$\frac{}{\vdash \cdot\ \mathit{ctx}}\ \mathit{null\text{-}ctx}$$

$$\frac{\Gamma \vdash K\ \mathit{kind} \quad \vdash \Gamma\ \mathit{ctx} \quad u \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, u : K\ \mathit{ctx}}\ \mathit{kind\text{-}ctx}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \vdash \Gamma\ \mathit{ctx} \quad x \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, x : A\ \mathit{ctx}}\ \mathit{type\text{-}ctx}$$

$$\frac{\vdash \Gamma\ \mathit{ctx}}{\Gamma \vdash \mathit{Type}\ \mathit{kind}}\ \mathit{type\text{-}kind} \qquad \frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash K\ \mathit{kind}}{\Gamma \vdash (\Pi x{:}A.\ K)\ \mathit{kind}}\ \mathit{pi\text{-}kind}$$

$$\frac{\vdash \Gamma\ \mathit{ctx} \quad u : K \in \Gamma}{\Gamma \vdash u : K}\ \mathit{var\text{-}fam}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash (\Pi x{:}A.\ B) : \mathit{Type}}\ \mathit{pi\text{-}fam}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : K}{\Gamma \vdash (\lambda x{:}A.B) : (\Pi x{:}A.\ K)}\ \mathit{abs\text{-}fam} \qquad \frac{\Gamma \vdash A : \Pi x{:}B.\ K \quad \Gamma \vdash M : B}{\Gamma \vdash (A\ M) : K[M/x]}\ \mathit{app\text{-}fam}$$

$$\frac{\Gamma \vdash A : K \quad \Gamma \vdash K'\ \mathit{kind} \quad K \equiv K'}{\Gamma \vdash A : K'}\ \mathit{conv\text{-}fam}$$

$$\frac{\vdash \Gamma\ \mathit{ctx} \quad x : A \in \Gamma}{\Gamma \vdash x : A}\ \mathit{var\text{-}obj}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M) : (\Pi x{:}A.\ B)}\ \mathit{abs\text{-}obj} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.\ B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B[N/x]}\ \mathit{app\text{-}obj}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : \mathit{Type} \quad A \equiv A'}{\Gamma \vdash M : A'}\ \mathit{conv\text{-}obj}$$

Figure 2.1: Rules for inferring assertions in Simplified LF

$$\begin{gathered} list : nat \to Type \\ nil : list\ z \\ cons : \Pi m{:}nat.\ \Pi n{:}nat.\ \Pi l{:}list\ n.\ list\ (s\ n) \end{gathered}$$

Under this specification we can rest assured that, if we have a natural number n of type *nat*, and a list l of type *list* n, then l has n elements. For example, a list with a single element $s\ z$ would be represented with the term *cons* $(s\ z)\ z\ nil$; the first argument of type *nat* corresponds to the list element, and the second such argument corresponds to the length of the tail of the list — the tail itself is the third argument.

The rules for derivations above make use of β-reduction, an operation with which LF expressions are equipped. β-reduction is defined by the rule $(\lambda x{:}A.P)\ N \to_\beta P[N/x]$. All LF expressions that are well-formed in the sense formalized above are strongly normalizing under this reduction relation [5]. Moreover any well-typed expression P has a unique normal form (up to changes in bound variable names); we denote this normal form by P^β. Terms P and Q are equivalent, written $P \equiv Q$, when they share the same unique normal form (that is, when $P^\beta = Q^\beta$).

Proposition 2.1.1 (Strong Normalization)**.** *Given a context* Γ *and LF expressions* P *and* Q*,* P *is strongly normalizing whenever* $\Gamma \vdash P : Q$ *has a derivation.*

An object variable x that appears in an LF expression P that is well-formed with respect to a context Γ has type of kind *Type* associated with it through either an assignment in Γ or a binding operator. And a type variable u appearing in the same has a kind associated with it in the same way. Moreover, the normal form of this kind or type must have a prefix of Πs. If the length of this prefix is n, then an occurrence of x is *fully applied* if it appears in a sub-term of the form $x\ M_1\ \ldots\ M_n$; similarly, an occurrence of u is fully applied if it appears in a sub-term of the form $u\ M_1\ \ldots\ M_n$. Further, P is *canonical* with respect to Γ if it is in normal form and if every variable occurrence in it is fully applied. A well-formed context Γ is canonical if the type or kind it assigns to each variable is canonical relative to Γ. For any type variable v, a well-formed type of the form $v\ M_1\ \ldots\ M_n$ that is fully applied is called a *base type*. The LF system admits a notion of η-expansion, using which any well-formed expression can be converted into a canonical form.

2.2 Algorithmic LF

The description of simplified LF given above, while straightforward, is not particularly amenable to reasoning over or use in an implementation: the conversion rules *conv-fam*

$$\frac{}{\vdash \cdot\ \mathit{ctx}}\ \mathit{null\text{-}ctx}$$

$$\frac{\Gamma \vdash K\ \mathit{kind} \quad \vdash \Gamma\ \mathit{ctx} \quad u \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, u : K\ \mathit{ctx}}\ \mathit{kind\text{-}ctx}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \vdash \Gamma\ \mathit{ctx} \quad x \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, x : A\ \mathit{ctx}}\ \mathit{type\text{-}ctx}$$

$$\frac{\vdash \Gamma\ \mathit{ctx}}{\Gamma \vdash \mathit{Type}\ \mathit{kind}}\ \mathit{type\text{-}kind} \qquad \frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash K\ \mathit{kind}}{\Gamma \vdash (\Pi x{:}A.\ K)\ \mathit{kind}}\ \mathit{pi\text{-}kind}$$

$$\frac{\vdash \Gamma\ \mathit{ctx} \quad u : K \in \Gamma}{\Gamma \vdash u : K^{\beta}}\ \mathit{var\text{-}fam}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash (\Pi x{:}A.\ B) : \mathit{Type}}\ \mathit{pi\text{-}fam}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : K}{\Gamma \vdash (\lambda x{:}A.B) : (\Pi x{:}A^{\beta}.\ K)}\ \mathit{abs\text{-}fam} \qquad \frac{\Gamma \vdash A : \Pi x{:}B.\ K \quad \Gamma \vdash M : B}{\Gamma \vdash (A\ M) : (K[M/x])^{\beta}}\ \mathit{app\text{-}fam}$$

$$\frac{\vdash \Gamma\ \mathit{ctx} \quad x : A \in \Gamma}{\Gamma \vdash x : A^{\beta}}\ \mathit{var\text{-}obj}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M) : (\Pi x{:}A^{\beta}.\ B)}\ \mathit{abs\text{-}obj} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.\ B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : (B[N/x])^{\beta}}\ \mathit{app\text{-}obj}$$

Figure 2.2: Rules for inferring assertions in Algorithmic LF

and *conv-obj*, along with β-conversion in the conclusions of some inference rules, make it very difficult to analyze the structure of derivations in proofs, and of terms within such derivations. As an example of the first issue, consider using the specification of lists indexed by their lengths, given above, for *checking* whether a given list l of length n is valid. To do so we must attempt to find a derivation for the judgment $\Gamma \vdash l : \mathit{list}\ n$; if a derivation is found the l does indeed represent a list of length n. In setting out to prove this judgment we might think of applying the conversion rule *conv-obj* immediately. We could repeat this process over and over, never making progress; indeed, there is nothing in the system to suggest that we *should not* proceed in this fashion. To address this issue we present *algorithmic* LF in Figure 2.2, which will form the basis of the definition of LF that we will use throughout the thesis. Algorithmic LF is a formulation of LF that does not include the conversion rules for type families and objects.

These inference rules allow for the derivation of an assertion of the form $\Gamma \vdash M : A$

only when A is in normal form. To verify such an assertion when A is not in normal form, we first derive $\Gamma \vdash A : \mathit{Type}$ and then verify $\Gamma \vdash M : A^\beta$. A similar observation applies to $\Gamma \vdash A : K$.

Note that in the rules above we are often required to find the normal form of a type A, however we have said that such normal forms are only known to exist when A is valid. Here we make use of the fact that at the point of this normalization (for instance, in the *abs-obj* rule) we already have a derivation of $\Gamma \vdash A : \mathit{Type}$ and so are assured that A does in fact have a normal form.

This presentation renders the conversions rules unnecessary, but still requires us to consider β-conversions that can impede analysis of derivations. In later sections we shall consider LF derivations in which *all* expressions in the the end assertion are in normal form. This entails that every expression in the entire derivation must also be in such a form, as in judgments of the forms $(\lambda x{:}A.B) : (\Pi x{:}A'.\ K)$ and $(\lambda x{:}A.M) : (\Pi x{:}A'.\ B)$ it must be the case that A and A' are identical. Finally, normalization need not be considered in the use of the *var-fam* and *var-obj* rules. With these restrictions we can state the following proposition, proved in [5]:

Proposition 2.2.1 (Soundness and Completeness). *Given LF expressions P and Q, the judgment $\Gamma \vdash P : Q$ has derivation in simplified LF if and only if $\Gamma^\beta \vdash P : Q^\beta$ has a derivation in algorithmic LF.*

We therefore may henceforth restrict attention to algorithmic LF judgments of the form $\Gamma \vdash P : Q$, along with their derivations, where Γ is a canonical context, and Q are in normal form. The following property of substitutions for such derivations that follows easily from the results in [5] will be useful later; here α stands for any judgment, and substitution and normalization over α and Γ correspond to distributing these operations to the expressions appearing in them.

Proposition 2.2.2 (Substitution). *Let Γ_1, Γ_2 be canonical contexts, and A be a type in normal form. If $\Gamma_1 \vdash M : A$ has a derivation, and $\Gamma_1, x : A, \Gamma_2 \vdash \alpha$ has a derivation, then $\Gamma_1, (\Gamma_2[M/x])^\beta \vdash (\alpha[M/x])^\beta$ has a derivation as well.*

Additionally we will use a second property of algorithmic LF derivations, which follows from Proposition 2.2.2.

Proposition 2.2.3 (Renaming). *Let $\Gamma = \Gamma_1, x : P, \Gamma_2$ be a canonical context, let P be a type or kind in normal form, and let α be a judgment in normal form. Let y be a variable* *not bound in* Γ*, and not occurring in* α*. Then* $\Gamma_1, x : P, \Gamma_2 \vdash \alpha$ *has a derivation if and only if* $\Gamma_1, y : P, \Gamma_2[y/x] \vdash \alpha[y/x]$ *has one.*

2.3 A logic programming interpretation of LF

LF as described above is a specification language; it is useful for describing formal systems, but it has no operational semantics of its own. However, LF admits a logic programming interpretation that is rooted in the "formulas as types" principle identified in the Curry-Howard isomorphism. In this setting a formula is represented by a type family, and inference rules defining the formula are represented by constructors for this type family.

Logic programming therefore proceeds by the search for an inhabitant of a type that corresponds to a formula for which a derivation must be found. Any inhabitant that is discovered can be interpreted as a derivation for the formula; such inhabitants are called *proof terms* because they represent a proof of the formula. In this setting, we begin with an initial LF context that is called a *specification*, which corresponds to the Prolog notion of a *logic program*. Then a type for which an inhabitant is sought can be viewed in the Prolog sense as a *goal* to be proved using such a program.

To see how this plays out more concretely, consider the simple logic programming example of appending lists of natural numbers (this time we consider traditional lists, not those indexed by their length, for simplicity of presentation). We first write down LF context items that describe the type of such lists.

$$\begin{gathered} nat : Type,\ \ z : nat,\ \ s : nat \to nat, \\ list : Type,\ \ nil : list,\ \ cons : nat \to list \to list \end{gathered}$$

Next we are interested in constructing proofs of formulas of the form $append\ l\ k\ m$, for lists l, k, and m. Given that formulas can be represented using types, we must first define a type representing this particular formula. As *append* is a relation over three lists, we define the type family $append : list \to list \to list \to Type$ the is indexed by three lists. We can apply this family to three lists l, k, and m to create a type corresponding to the formula $append\ l\ k\ m$.

Next, given that proofs of a particular formula correspond to terms of the corresponding type, we require a means for actually constructing such terms. In the case of appending lists, we obtain these means by defining object level constants that have a target type of the form $append\ l\ k\ m$ and that correspond to the usual means for constructing proofs of such formulas; we refer to these as constructors. This leads in particular to the following

LF context items; here each constructor corresponds to the one of the traditional clauses for appending lists.

$$\begin{aligned}
&\mathit{append} : \mathit{list} \to \mathit{list} \to \mathit{list} \to \mathit{Type},\\
&\mathit{append}_{\mathit{nil}} : \Pi l{:}\mathit{list}.\ \mathit{append}\ \mathit{nil}\ l\ l,\\
&\mathit{append}_{\mathit{cons}} : \Pi x{:}\mathit{nat}.\ \Pi l{:}\mathit{list}.\ \Pi k{:}\mathit{list}.\ \Pi m{:}\mathit{list}.\ \mathit{append}\ l\ k\ m \to\\
&\qquad \mathit{append}\ (\mathit{cons}\ x\ l)\ k\ (\mathit{cons}\ x\ m)
\end{aligned}$$

Given this specification, when searching for an inhabitant of the type

$$\mathit{append}\ (\mathit{cons}\ z\ \mathit{nil})\ \mathit{nil}\ (\mathit{cons}\ z\ \mathit{nil})$$

(that is, searching for a proof that *cons z nil* appended to *nil* yields *cons z nil*) we proceed by backchaining on constructors for *append*. The first is $\mathit{append}_{\mathit{nil}}$; backchaining fails as *cons z nil* does not match *nil*. The second is $\mathit{append}_{\mathit{cons}}$; here backchaining succeeds and we must search for an inhabitant of *append nil nil nil*. Here we once again backchain on $\mathit{append}_{\mathit{nil}}$, which succeeds. Therefore search succeeds, and the inhabitant that is found is:

$$\mathit{append}_{\mathit{cons}}\ z\ \mathit{nil}\ \mathit{nil}\ \mathit{nil}\ (\mathit{append}_{\mathit{nil}}\ \mathit{nil})$$

Thus the general process of searching for an inhabitant of a given type A, given an LF specification, is as follows. First, start with a context Γ that is just the original LF specification. Then, if A is a base type, try to match it to target type of the type of a bound variable x of the context. If a match is found, recursively find inhabitants for each argument type of the type of x. If A is instead of the form $\Pi y{:}A_1.\ A_2$, construct a new context Γ' consisting of Γ extended with $y : A_1$, and attempt to find an inhabitant of A_2 under Γ'.

2.4 Twelf

Twelf is a robust, widely used, and many-featured implementation of LF. It includes a language for writing LF contexts and judgments, and an implementation of logic programming search in the style described above. In addition, it supports various extensions to LF, both for the purposes of logic programming and reasoning over specifications.

The concrete syntax of Twelf is almost a direct translation of the abstract syntax of LF. We write `{x : P} Q` in Twelf for the LF quantification $\Pi x{:}P.\ Q$, and write `[x : P] Q` for the LF abstraction $\lambda x{:}P.Q$. The LF shorthand $P \to Q$ has a corresponding Twelf shorthand, `P -> Q`, and a reverse shorthand: `Q <- P` is equivalent to `P -> Q`.

```
nat : type.
z : nat.
s : nat -> nat.

list : type.
nil : list.
cons : nat -> list -> list.

append : list -> list -> list -> type.
append-nil : {L : list}append nil L L.
append-cons : {X : nat}{L : list}{K : list}{M : list}
              append L K M -> append (cons X L) K (cons K M).
```

Figure 2.3: *append* in Twelf

Twelf logic programs simply consist of a specification, elements of which ($x : P, y : Q$, and so on) are written in order, each with a terminating period. Thus the LF specification defining *append* over lists of natural numbers already given can be written as the Twelf specification shown in Figure 2.3.

Logic programming in Twelf is achieved through an interactive prompt. Given a type, Twelf searches for an inhabitant in the manner described, returning success if one is found, and failure if not. For example, searching for an inhabitant of *append nil nil nil* yields a single solution:

```
?- append (cons z nil) nil (cons z nil).
Solving...
Empty substitution.
```

Similarly, searching for a type which has no inhabitants yields no solutions:

```
?- append nil nil (cons z nil).
Solving...
No solutions.
```

In addition, Twelf includes a number of extensions to LF to ease programming. For instance, objects and type families may be given different associativities and precedences, and may be made infix, prefix, or postfix. These features are orthogonal to the main emphasis of this thesis, and so details on their implementation are omitted; in practice these issues can be treated separately, so that the result of parsing a given expression is more or less pure LF.

However, an important extension, when it comes to logic programming search, merits further description. In Twelf the types for which an inhabitant is to be sought can contain *meta-variables*, which are to be filled in during search. This allows for using Twelf to not only determine whether a relation holds, but to discover on exactly *which* terms the relation holds, as well. For instance, the primary use of a judgment like *append* is not to *verify* that two lists append to form a third, but rather to determine *which* list the two construct.

```
?- append (cons z nil) nil L.
Solving...
L = cons z nil.
```

This indicates that an inhabitant was found for the LF type

$$\mathit{append}\ (\mathit{cons}\ z\ \mathit{nil})\ \mathit{nil}\ (\mathit{cons}\ z\ \mathit{nil}).$$

In some cases meta-variables in the type for which an inhabitant is being sought are not instantiated, as they are under-constrained by the specification. When this occurs the interpretation is that, for any term t of the meta-variable's type, the goal type has an inhabitant when the meta-variable is instantiated with t.

2.5 An extended example

Consider the problem of encoding in LF both the syntax of the simply typed λ-calculus (*stlc*), and the semantics of evaluation; this is the sort of problem at which LF excels. In this context, in which we use LF to encode a different language, we call LF the *meta-language*, and the encoded language (here the *stlc*) the *object language*. We have given the object language already, in Section 1.1.

First we must represent *stlc* types in LF; we do so using LF objects of type *ty*. Next we make use of dependency by representing terms of the *stlc* using LF objects of type family *tm*. This type family depends on an object of type *ty*; the constructors for *tm* maintain the invariant that an *stlc* term of type t is represented by an LF object of type *tm* t', where t' is the LF representation of the type t. For instance, *void* has type *tm unit*, indicating that it is a term of type *unit*.

```
ty : type.
arrow : ty -> ty -> ty.
unit : ty.

tm : ty -> type.
abs : {T1 : ty}{T2 : ty} (tm T1 -> tm T2) -> tm (arrow T1 T2).
app : {T1 : ty}{T2 : ty} tm (arrow T1 T2) -> tm T1 -> tm T2.
void : tm unit.
```

Of particular interest is the definition for *abs*; here we make use of the abstraction construct of LF to represent abstraction in the *stlc*. This technique is known as *higher order abstract syntax* [7] (HOAS), and is a powerful mechanism for keeping the intricacies of binding, names, and substitutions within the meta-language instead of re-implementing it anew for each object language. Specifically, if we were to manage object level bound variables "by hand" within the representation of the *stlc* (say by introducing a new term constructor *var*, taking perhaps an integer representing its identity), we would be obligated to also implement capture-avoiding substitution, which could involve bound-variable renaming, and which in turn might require us to develop a notion of a fresh variable. Instead we represent variables in the object language using variables of the meta-language, and use meta-level abstraction to represent abstraction in the object language; thus we can directly use meta-level applications to perform object-level substitutions, and we inherit from the meta-language properties such as α-equivalence.

There are many relations over such terms that are traditionally of interest; we consider that of evaluation. A typical description of a call-by-value evaluation strategy for the *stlc* has been given by syntax-directed rules in Figure 1.2, where we used the notation $E \Rightarrow V$ to indicate that E evaluates to V under this strategy. The rules shown provide the means for deriving such judgments.

We then can formalize this specification of evaluation in LF using the type family *eval*, which takes three objects, first one encoding an *stlc* type, and then two encoding *stlc* terms of that type. We define constructors for *eval* such that an LF object has type $eval\ t\ e\ v$ only when the *stlc* term e' that e represents evaluates to the *stlc* term v' that v represents under the call-by-value evaluation strategy of Figure 1.2, where both e' and v' have the *stlc* type represented by t. This encoding employs the Curry-Howard Isomorphism as described: here evaluation is the judgment we want to encode using the *eval* type, the constructors for correspond to inference rules for deriving this judgment, and terms of this type correspond to proofs of the judgment.

```
eval : {A : ty} tm A -> tm A -> type.
eval-void : eval unit void void.
eval-abs :
  {T1 : ty}{T2 : ty}{E : tm  T1 -> tm T2}
  eval (arrow T1 T2) (abs T1 T2 E) (abs T1 T2 E).
eval-app :
  {T1 : ty}{T2 : ty}{A : tm (arrow T1 T2)}{B : tm T1}
  {F : tm T1 -> tm T2}{X : tm T1}{V : tm T2}
    eval T2 (app T1 T2 A B) V <-
      eval (arrow T1 T2) A (abs T1 T2 ([x : tm T1] F x)) <-
      eval T1 B X <-
      eval T2 (F X) V.
```

Notice that the type of *eval* requires that terms of a particular type evaluate to terms of the same type. This is quite a powerful invariant, as it means that, so long as the LF specification is valid, we can be sure that our definitions for the constructors of *eval* are type-preserving. Furthermore, we need not bake this requirement into each of the definitions of the constructors explicitly; a constructor that does not maintain this invariant cannot be well-typed!

Given the above definitions of types, terms, and evaluation, we can write the following *adequacy* proposition, which states that the evaluation relation defined in LF is equivalent to the *stlc* evaluation judgment $\bullet \Rightarrow \bullet$:

Proposition 2.5.1 (Adequacy)**.** *Let T be an* stlc *type, and M and N* stlc *terms of type T. Let Γ be the well-formed LF context encoding the* stlc *as defined above, and let T' be the LF object of type ty that is the encoding of T. Let M' and N' be LF objects of type tm T' that are encodings of M and N, respectively. Then $M \Rightarrow N$ has a derivation if and only if $\Gamma \vdash P : eval\ T'\ M'\ N'$ has a derivation, for some LF object P.*

An encoding of the *stlc* (to which our encoding above is closely related) is given in [8]; proof of Proposition 2.5.1 follows from that work.

We can now make use of this specification in logic programming. In particular, given a well-typed *stlc* term α of type τ, we can determine what it evaluates to as follows. First, let t be the LF object representing τ, and a be the LF object representing α; then a has type *tm t*. Next, by our definition of *eval* (assuming that we have correctly defined it, and it is an adequate encoding), we know that if the type *eval t a b* has an inhabitant for some LF object b of type *tm t*, then α evaluates to some term β whose LF representation is b.

For instance, if $\tau = unit$ and $\alpha = (\lambda x{:}unit.x)\ void$ then

$$t = tm\ unit \quad \text{and} \quad a = app\ unit\ (abs\ unit\ unit\ (\lambda x{:}tm\ unit.x))\ void$$

As we have seen, we can search for such an inhabitant in Twelf by invoking the following goal:

```
?- eval unit
        (app unit unit (abs unit unit ([x : tm unit] x)) void)
        B.
Solving...
B = void.
```

Here `B` is a meta-variable. On success Twelf returns with a binding for `B` that is `void`, which means that there exists an inhabitant of type:

$$eval\ unit\ (app\ unit\ (abs\ unit\ unit\ (\lambda x{:}tm\ unit.x))\ void)\ void$$

Twelf provides the inhabitant, which is quite large:

```
  eval-app unit unit (abs unit unit ([x:tm unit] x))
    void ([x:tm unit] x) void void eval-void eval-void
    (eval-abs unit unit ([x:tm unit] x))
```

This term, by Proposition 2.5.1, corresponds to a proof that α evaluates to β under the call-by-value evaluation strategy defined in Figure 1.2 that the *eval* type family and its constructors embody.

2.6 Reasoning over LF specifications

Twelf includes, in addition to an implementation of logic programming search for LF, features for reasoning over specifications; in this context we refer to theorems over LF judgments and specifications as *meta-theorems*. These features take two forms. First, there are several analyses that can be used to prove certain general properties of specifications and judgments. These include properties like termination of a judgment under a particular ordering, and totality and coverage checking. In order to understand what a property like termination means regarding a judgment encoded as a type family, the notion of a *mode* is introduced. Modes [9] encode the meanings of arguments to a type family, in terms of whether each argument should be interpreted as *input*, *output*, or *unrestricted* (that is, neither). For instance, in our previous example of *eval*, we might wonder whether we can always use it to see to what a particular *stlc* term evaluates. In this usage, the first two arguments to *eval* would be input, and the final argument would be output. We can than think of asking whether the judgment terminates *when used in this mode*; of course our conclusions will be different for a different mode. Twelf is also able to *check* the validity of the modes ascribed to a type family by a programmer.

When reasoning in this fashion, the essence of meta-theorem proving in Twelf is to encode the particular theorem that is to be proved *as an LF specification*, and then prove the aforementioned properties about the judgments encoding meta-theorems. It is possible to use careful encodings of theorems as judgments so that when, for example, a judgment corresponding to a meta-theorem is *total*, the meta-theorem itself is true. In this way it is possible to prove complex theorems about judgments. For example, returning to our simple encoding of *append* in LF, we can think of proving that appending *nil* to any list results in the same list: $\forall l : list.append\ l\ nil\ l$. That is, we ask whether, for any list l, the relation *append* l *nil* l holds. By our definition of *append* we know that this is equivalent to asking whether, for any list l, the LF type *append* l *nil* l has an inhabitant. We therefore define a judgment (that is, a type family *encoding* a judgment) corresponding to a relation between a list and a proof that appending *nil* to said list results in the same list. We assign a mode to indicate that the new judgment is to be used in the following way: given a list, it should *produce* the relevant proof.

```
append-unit : {L1 : list} append L1 nil L1 -> type.
%mode append-unit +L -A.
```

Next we actually "fill in", for each kind of list by which this type family might be indexed, how to construct a proof of the relevant judgment. There are two cases: first, when the list is *nil*, we provide a proof that appending *nil* to *nil* yields *nil* again — the term representing this proof is just `append-nil`. Second, we must provide a proof that the property holds for a non-empty list; this is accomplished by first recursively producing a proof for the tail of the list and then using the `append-cons` constructor to produce a proof for the entire list.

```
append-unit-nil : append-unit nil append/nil.
append-unit-cons :
  {N : nat}{L : list}{A : append L nil L}
  append-unit L A ->
  append-unit (cons N L) (append-cons N L nil L A)
```

Finally we use the totality checker to show that, given *any* list, the definition of `append-unit` is able to produce a proof that appending *nil* to the list results in the same list.

```
%total D (append-unit D _).
```

So long as `append-unit` is total, we can rest assured that our original theorem, namely that $\forall l : \mathit{list.append}\ l\ \mathit{nil}\ l$, is correct.

The second form of reasoning over Twelf specifications is by way of an (experimental) automatic meta-theorem prover. This prover is able to automatically prove various properties of specifications nearly automatically, which is appealing in that it does not require significant user intervention. However, this tool suffers from several limitations: it can only prove theorems that have a particular form (namely, those specified by universal and then existential quantification), does not have fundamental support for inductive proofs; instead it relies first on the user specifying an induction order, and then on a form of cycle detection over proofs to deal with theorems requiring such kinds of reasoning. In its current incarnation this precludes proofs involving nested inductions, and automatic generation and use of arbitrary invariants.

It is also useful to consider the possibility of embedding LF specifications into some richer logic, in order to more easily and directly reason about them. For instance, following the *two-level* approach [10], we take a powerful logic for reasoning (called the *meta-logic*), and develop an encoding of the logic over which we seek to reason (called the *object logic*) in the meta-logic. We can then directly encode formulas from the object logic in the meta-logic, and thereby reason over these formulas using the tools provided by the meta-logic. The meta-logic $\mathcal{G}$ [11] has been used in this context; it includes, amongst other properties, straightforward facilities for reasoning about binding and variables, and has been used to encode the a higher-order predicate logic. Recall that one of the goals of this thesis is to enable reasoning over LF specifications by translation to such a logic — these tools and techniques can enable programmer to reason more directly and openly over LF specifications by first translating a specification to predicate logic and then reasoning over the generated specification using, for instance, $\mathcal{G}$. This could allow the programmer to prove arbitrary theorems without the need to appeal to encodings as in Twelf.

Chapter 3

Higher-order hereditary Harrop formulas

Recall that our eventual goal is to develop a translation from LF to a predicate based logic. We have two primary concerns when selecting to which predicate logic we should translate, under the consideration that it is to be used when implementing logic programming search for LF. First, the logic should admit an efficient implementation, so that our LF implementation can be efficient as well. Second, the logic should be sufficiently powerful to allow for a clear and simple translation from LF. This is important first because it allows for a translation that is readily understandable, and obviously correct; and second, it means that the efficiency of the implementation will not be lost due to the possibly complex machinery of the translation.

We select the logic of higher-order hereditary Harrop formulas (*hohh*); both LF and *hohh* have some of the same properties, including a logic programming interpretation and, importantly, a built-in treatment of binding via HOAS. Indeed, as we shall see, the essential difference between the logics of LF and *hohh* is to be found in the typing disciplines of the two logics. Whereas in LF we are presented with the richness of dependent types, *hohh* lacks this facility, but includes support for *parametric polymorphism* as is present in, for example, programming languages like ML [12]. Thus neither logic has a type system that is the superset of the other.

In this chapter we first present the syntax of *hohh*, and then give a logic programming interpretation for it. We then describe a programming language based on *hohh*, and describe one efficient implementation of it. Finally we return to our running example, to give a feel for how programming in an *hohh* setting differs from programming in LF.

3.1 Syntax

The logic of *hohh* formulas is based on an intuitionistic version of Church's simple theory of types [13], both of which are built over a typed form of the λ-calculus. While we have stated that *hohh* has a type system that includes parametric polymorphism, as such types are not required for our specific work, we do not discuss those aspects here. Instead, we will restrict ourselves to those types that are constructed from a finite collection of atomic types that includes o, the type of propositions, and at least one other type, using $\rightarrow$, the infix, right associative function type constructor. Given this syntax any type τ can be written in the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \sigma$, where σ is an atomic type. Understanding a type this way, we refer to the (possibly empty) sequence $\tau_1, \ldots, \tau_n$ as its *argument* types and to σ as its *target* type.

We assume that we are given sets of typed variables and typed constants. The full collection of (typed) terms is generated from these by the usual abstraction and (left associative) application operators. Terms that differ only in the names of their bound variables are not distinguished. We further assume a notion of equality $P \equiv Q$ that encompasses α-, β- and η-conversion. It is well-known that every term has a unique normal form under these reduction operations in this simply-typed setting. We shall implicitly assume all terms are converted into such a form prior to their consideration in any context. Moreover, we shall use the $\beta\eta$-long normal form for terms, *i.e.*, the β-normal form in which every occurrence of a variable x whose type has n argument types is ultimately in the context of a term of the form $(x\ t_1\ \ldots\ t_n)$. We write $t[s_1/x_1, \ldots, s_n/x_n]$ to denote the result of simultaneously replacing the variables $x_1, \ldots, x_n$ with the terms $s_1, \ldots, s_n$ in the term t, renaming bound variables as needed to avoid accidental capture. This substitution operation is defined only when s_i and x_i are of the same type for $1 \leq i \leq n$.

We will use only a fragment of the full *hohh* logic here; this fragment still possesses the proof-theoretic properties that are fundamental to the logic programming interpretation of the *hohh* logic. The constants from which terms are constructed are differentiated into *non-logical* ones, that constitute a *signature*, and logical ones. Non-logical constants whose target type is o are called *predicates*; other non-logical constants are *constructors*. We do not permit o to appear in the type of the arguments of non-logical constants and variables. The logical constants are restricted to $\top$ of type o, $\supset$ of type $o \rightarrow o \rightarrow o$ that is written in the customary infix form and, for each type α, Π of type $(\alpha \rightarrow o) \rightarrow o$. The constant $\top$ represents the true proposition, and $\supset$ the usual binary implication connective (which will be written in the customary infix, right associative form). Π represents the universal quantifier as a function over sets. We abbreviate $\Pi\ (\lambda x.F)$ by $\forall x.F$. An *atomic formula*, denoted by A, is a term of type o of the form $p\ t_1\ \ldots\ t_n$ where p is a non-logical constant. The logic of interest is characterized by two collections of terms called G- and D-formulas that are defined mutually recursively by the following syntax rules:

$$\frac{}{\Sigma;\Gamma \longrightarrow \top}\,\top R \qquad \frac{\Sigma;\Gamma\cup\{D\} \longrightarrow G}{\Sigma;\Gamma \longrightarrow D \supset G}\,\supset R \qquad \frac{c \notin \Sigma \quad \Sigma\cup\{c\};\Gamma \longrightarrow G[c/x]}{\Sigma;\Gamma \longrightarrow \forall x.G}\,\forall R$$

$$\frac{D \in \Gamma \quad \Sigma;\Gamma \xrightarrow{D} A}{\Sigma;\Gamma \longrightarrow A}\,\mathit{decide} \qquad \frac{}{\Sigma;\Gamma \xrightarrow{A} A}\,\mathit{init}$$

$$\frac{t \text{ is a } \Sigma\text{-term} \quad \Sigma;\Gamma \xrightarrow{D[t/x]} A}{\Sigma;\Gamma \xrightarrow{\forall x.D} A}\,\forall L \qquad \frac{\Sigma;\Gamma \longrightarrow G \quad \Sigma;\Gamma \xrightarrow{D} A}{\Sigma;\Gamma \xrightarrow{G\supset D} A}\,\supset L$$

Figure 3.1: Derivation rules for the *hohh* logic

$$\begin{array}{lcl} G & := & \top \mid A \mid D \supset G \mid \forall x.G \\ D & := & A \mid G \supset D \mid \forall x.D \end{array}$$

A specification or logic program is a finite collection of closed D-formulas that are also called *program clauses* and a goal is a closed G-formula. Notice that the presented program clauses extend those in a Prolog-like setting by allowing for embedded implications and universal quantifiers in the body of a clause and by extending the arguments of atomic formulas from first-order terms to simply typed λ-terms.

3.2 Logic programming with higher-order hereditary Harrop formulas

Logic programming in *hohh* corresponds to searching for a proof of a formula given a collection of formulas called a logic program. This process is formalized as the search for a derivation of a sequent of a particular form. Specifically, we seek a derivation of $\Sigma;\Gamma \longrightarrow G$. Here Σ is an initial signature, the collection of constants used to construct terms. Next, Γ is a collection of D-formulas representing the logic program that are taken to be true. Finally, G is a G-formula corresponding to the formula to be proved. A derivation of such a sequent then shows that the goal follows from the context. Figure 3.1 presents the rules for constructing such a derivation; note that the expression "t is a Σ-term" in the $\forall L$ rule means that t is a closed term all of whose non-logical constants are contained in Σ.

The derivation rules manifest a goal-directed character: to find a derivation for $\Sigma;\Gamma \longrightarrow G$, we simplify G based on its logical structure and then use *decide* to select a formula from the logic program for solving an atomic goal. Notice also that the *decide* rule initiates the consideration of a focused sequence of rules that is similar to backchaining. In particular, if the formula selected from Γ has the structure $(\forall x_1.F_1 \supset \ldots \supset \forall x_n.(F_n \supset A')\ldots)$ then this sequence is equivalent to the following rule:

$$\frac{\Sigma;\Gamma \longrightarrow F'_1 \qquad \ldots \qquad \Sigma;\Gamma \longrightarrow F'_n}{\Sigma;\Gamma \longrightarrow A}\ \textit{backchain}$$

This rule has the proviso that for some Σ-terms $t_1, \ldots, t_n$ that have the same types as $x_1, \ldots, x_n$, respectively, it is the case that A is equal to $A'[t_1/x_1, \ldots t_n/x_n]$ and, for $1 \leq i \leq n$, F'_i is equal to $F_i[t_1/x_1, \ldots, t_i/x_i]$. Read in a proof search direction, $\forall R$ leads to an expansion of the signature in the sequent whose derivation is sought and $\supset R$ similarly causes an addition to the logic program.

The proof system defined above, using just $\supset R$, $\forall R$, *backchain*, and $\top R$ is in fact equivalent to intuitionist sequent calculus, with a few provisos: namely, each D-formula must be converted into the form $\forall x_1.F_1 \supset \ldots \supset \forall x_n.(F_n \supset A)$ by the insertion of vacuous quantifications and implications (that is, $\forall x.D$ where x does not appear in D, and $\top \supset D$). The system generates *uniform* proofs [14], which are essentially those that correspond to a complete, backwards-chaining, goal directed search.

An example should make the model of computation described above clear. Consider again the problem of encoding lists of natural numbers, along with the *append* operation, this time in *hohh*. Again we are first tasked with encoding such lists; here we choose to introduce two atomic types *nat* and *list*. We next define constructors for these types: z of type *nat*, s of type $nat \rightarrow nat$, *nil* of type *list*, and *cons* of type $nat \rightarrow list \rightarrow list$. Finally we must define the *append* predicate, giving it type $list \rightarrow list \rightarrow list \rightarrow o$; that is, it is a relation over three lists. And we write down two D-formulas that specify when particular lists are in the relation:

$$\forall l.append\ nil\ l\ l$$
$$\forall x.\forall l.\forall k.\forall m.append\ l\ k\ m \supset append\ (cons\ x\ l)\ k\ (cons\ x\ m)$$

The first corresponds to the base case of the canonical definition of *append*, the second to the recursive case.

Now given we can determine whether particular lists, for instance (*cons z nil*), *nil*, and (*cons z nil*) are in the relation. That is, we can ask whether *nil* appended to (*cons z nil*) is equal to *cons z nil*; we do so as follows. First, we construct Σ by taking all of the types and non-logical constants already described. Second, we construct the logic program Γ by taking the two formulas above. And finally, we search for a derivation of $\Sigma;\Gamma \longrightarrow append\ (cons\ z\ nil)\ nil\ (cons\ z\ nil)$. The derivation can proceed first by *backchain* on the second clause for *append*; here x is instantiated with z, and all three of l, k, and m with *nil*, leaving us to develop a derivation of $\Sigma;\Gamma \longrightarrow append\ nil\ nil\ nil$. We therefore may apply *backchain* on the first clause for *append*, instantiating l with *nil* and yielding no more subgoals to prove and completing the derivation. In the future we will write this as just $\Gamma \longrightarrow \mathit{append}\ (\mathit{cons}\ z\ \mathit{nil})\ \mathit{nil}\ (\mathit{cons}\ z\ \mathit{nil})$ when Σ is clear from context.

3.3 A higher-order logic programming language

The higher-order logic programming language λProlog is founded on *hohh*, with several extensions that add power and convenience to the programming experience. We restrict our attention to those aspects of λProlog that are necessary to the rest of this thesis, and that correspond to the presentation of *hohh* already given.

In λProlog the *hohh* type o is represented by `o`. One can introduce new atomic types using the `kind` construction; for instance, one indicates that tm is a type using the syntax `kind tm type`. Additionally, it is possible to define new non-interpreted constructors for these types using the `type` construction; for instance we can define *app* as having type $tm \to tm \to tm$ using the syntax `type app tm -> tm -> tm`. These constructions are used to populate the signature Σ.

Abstractions of the form $\lambda x.M$ are written `x\ M`, and applications are written by juxtaposition; parentheses are used only for grouping. Next, each logical constant of *hohh* has a corresponding syntax in λProlog; $\top$ is `true`, $\forall.t$ is written `pi t`, and $G \supset D$ is written `G => D`. In addition, one can write `D :- G` for $G \supset D$.

Finally, the context is built up by writing down formulas and terminating them with periods. As a shorthand, capitalized names in such context formulas are taken as being universally quantified at the outside. Thus, coming again back to the *append* example:

```
kind nat type.
type z nat.
type s nat -> nat.

kind list type.
type nil list.
type cons nat -> list -> list.

append nil L L.
append (cons X L) K (cons X M) :- append L K M.
```

Interaction with a λProlog program takes place in a similar way to that of Twelf. For instance, if we want to check if *append nil nil nil* holds, we write:

```
?- append nil nil nil.
```

Finally, λProlog extends *hohh* in several ways; of primary importance for our purposes is that it allows for meta-variables in queries, which are instantiated by unification during search; this is again quite similar to the approach taken in Twelf. To determine what list is produced when appending *nil* to *cons z nil*, for instance, the interaction would proceed as follows:

```
?- append (cons z nil) nil L.
L = (cons z nil).
```

Here search succeeds, instantiating the meta-variable `L` to the result of the *append*.

The logic that we have described has been given an efficient implementation in the *Teyjus* system [15]. As this system will eventually provide a basis for our implementation of logic programming search for LF, we detail a few important characteristics of the Teyjus implementation. Teyjus is a system composed of several parts. First, a compiler which generates relocatable bytecode for individual modules of a logic program. Second, a linker for composing the bytecode for several modules into a single program. And finally, a simulator, for executing and interacting with the final bytecode program. The bytecode and simulator are based on Warren's abstract machine [16]. While the specifics of this implementation technique are beyond the scope of this thesis, an understanding of certain aspects of how backchaining is performed will eventually become necessary.

Usually at compile time the particular predicate on which backchaining is to be performed is known (indeed, this will be true in all examples in this thesis). Therefore backchaining is invoked within the simulator by jumping to the first clause head for the given predicate, and then each clause head is tried in turn. In addition, the optimization known as *indexing*, also due to Warren, is used to improve performance during backchaining by allowing the structure of the first argument to a predicate to guide dispatch and skip clauses that cannot match. Notice that this is only effective when the first argument is *not* an uninstantiated meta-variable.

3.4 An extended example

Here we return to the example of encoding the *stlc*, this time in λProlog. Again, our first task is to provide a representation of *stlc* terms as λProlog terms. Whereas in our LF encoding of the *stlc* we could make use of dependent types to encode an invariant stating that any term should be well-typed, in λProlog we have no dependent types and so cannot define our types in such a way. We can, however, still make use of the HOAS style to represent the binding of `abs` in terms of λProlog binding, just as in LF. Therefore we

```
kind ty type.
type unit ty.
type arrow ty -> ty -> ty.

kind tm type.
type void tm.
type abs ty -> (tm -> tm) -> tm.
type app tm -> tm -> tm.

type eval tm -> tm -> o.
eval void void.
eval (abs T F) (abs T F).
eval (app F X) V :-
  eval F (abs T R),
  eval X Y,
  eval (R Y) V.
```

Figure 3.2: The *stlc* in λProlog

introduce new atomic types for terms and types, `tm` and `ty`, respectively, along with the obvious constructors.

Next we define evaluation, which is a logical relation between two terms. Thus its type has two arguments of type `tm`, and which has as a target the type of propositions, `o`. Note that we make use of the λProlog operator "`,`", which corresponds to the *hohh* conjunction $\wedge$, for readability and to more closely reflect standard λProlog programming practice. To recover a syntax without this operator we can make use of the equivalence between $(\alpha \wedge \beta) \supset \gamma$ and $\alpha \supset \beta \supset \gamma$, which is sufficient for all of the uses below. Figure 3.2 shows the λProlog program implementing the *stlc* and its evaluation.

Given this program we can run some of the same queries in λProlog as we could in LF; we can, for instance, determine what the *stlc* term $(\lambda x.x)$ *void* evaluates to using by proving the following goal:

```
?- eval (app (abs unit (x\ x)) void) V.
V = void
```

Here the binding for `V` indicates that the term evaluates to the *stlc* term *void*.

There are a few important points to make, all related to the fact that we have lost an important property of the LF encoding: we cannot be sure that an *hohh* term always represents a well-typed *stlc* term. First, it is not obvious from the code that a given term must evaluate to a term of the same type, whereas in LF our encoding (in Section 2.5) makes this clear just from the type of *eval*. Second, it is not clear what will happen if we attempt to evaluate a term that does not correspond to a well-typed *stlc* term; again, in LF need not ask this question, as all LF terms of type *tm* correspond to well-typed *stlc* terms. For instance, the λProlog term `app (abs unit (x\ void x)) void` does not correspond to a well-typed *stlc* term, but is a well-typed λProlog term. If we try to evaluate this term we will get the *hohh* term `app void void`, which again does not correspond to a well-typed *stlc* term, but without an easy way to determine this fact. Finally the argument for the adequacy of this encoding with respect to the *stlc* is actually significantly complicated by the laxness of the encoding. In order to recover all of these properties, and enable a proof of adequacy, we will need to introduce an explicit typing predicate defining the typing relation given in Figure 1.1, and insert in various places assertions that terms have the correct type.

This all suggests that encoding dependencies between terms in this *ad hoc* fashion can be difficult, and that it can be non-obvious what such changes will entail for the rest of the program. However, this is not to say that programming in *hohh* is not without its own strengths, which include the strength and efficiency of the compiled implementation that it admits, and its facilities for parametric and *ad hoc* polymorphism.

Chapter 4

A Straightforward Translation into Predicate Logic

We are interested in developing an implementation of logic programming search for LF via translation to *hohh*. As previously described, the language λProlog admits an efficient implementation, manifest in the Teyjus system; therefore, if we can develop a translation that is sound and complete with respect to derivability in LF, we can exploit this efficient implementation to perform logic programming in LF.

An implementation based on such a technique could proceed as follows. First, a given LF specification is translated to a corresponding *hohh* logic program. This logic program contains some pieces that are concerned only with the mechanics of the translation, and some pieces that are based on the specifics of the given LF specification. Next, an LF judgment is translated into a corresponding *hohh* goal. Finally, we search for a derivation of the goal under the logic program. So long as there exists an equivalence between derivations in LF and derivations in *hohh* for the relevant programs, we can understand the proof of the *hohh* goal as being a proof to the original LF judgment.

A straightforward mapping of LF specifications and judgments into *hohh* is presented in [17]. The motivation for this translation is simply to demonstrate the possibility of encoding LF derivations in *hohh* derivations. It is an appealing translation in that, given an LF context and judgment, and their translations, the LF derivation of the judgment under the given context has a direct structural correspondence to the *hohh* derivation of the translated judgment under the translated context. This leads to a proof of correctness based on simply building an *hohh* derivation from the LF derivation, and vice versa, without many complications.

However, this translation is not directly suitable for our purposes for two reasons. First, it is not based exclusively on LF types, but assumes also the availability of objects; that is, it provides that given an LF context Γ, object M, and type A, $\Gamma \vdash M : A$ has a derivation if and only if the translation of this judgment has a derivation in *hohh*. However, for the purpose of logic programming search, the object M is not known, as it is exactly said objects that are being sought. Second, the correctness result for the translation only states an equivalence between LF derivability and *hohh* derivability for *known* LF assertions, and does not consider whether it is possible for non-canonical or ill-formed objects to be produced by the *hohh* specification.

We describe a translation from LF to *hohh* that is suitable for logic programming, and also exhibits structural correspondence to the original LF specification. We give an example of how the translation plays out in practice. We then prove its correctness, and return to the example to discuss the implications of the translation's correctness. Finally, we identify some issues that must be addressed before this translation can form the basis of a practical implementation.

4.1 A translation for logic programming

We adapt the earlier translation here in a way that allows us to use it in logic programming discussions; we call this new translation the *simplified* translation in that it is conceptually simpler than the earlier translation. These adaptations take two forms. In order to accommodate a mode in which we do not have access to the object, and only have a type for which an inhabitant should be sought, as described above, we must ensure that the translation is guided strictly by the type.

Second, to simplify the definition of the translation, and ease reasoning over the programs it generates, we seek to restrict the translation to operating on those judgments that are of use in logic programming: those that assign types to objects. But derivations of such judgments, those of the form $\Gamma \vdash M : A$, can themselves contain sub-derivations of judgments of the form $\Gamma \vdash A : K$; indeed, when A is of the form $\Pi x{:}A_1.\ A_2$ a derivation of $\Gamma \vdash M : A$ concludes by the *abs-obj* rule:

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.M) : (\Pi x{:}A.\ B)}\ \mathit{abs\text{-}obj}$$

Clearly such a derivation contains a sub-derivation corresponding to well-kinding, namely $\Gamma \vdash A : \mathit{Type}$; this implies that our translation must treat judgments assigning kinds to types. However, we can design the translation so that it only accounts for judgments of the form $\Gamma \vdash M : A$ through the following observation: if $\Gamma \vdash A : \mathit{Type}$ is known to have a derivation, and the last rule of the derivation of $\Gamma \vdash M : A$ is *abs-obj*, then $A = (\Pi x{:}B.\ A')$. The sub-derivation of $\Gamma \vdash (\Pi x{:}B.\ A') : \mathit{Type}$ clearly contains one of $\Gamma \vdash B : \mathit{Type}$ (by *pi-fam*), and so we need not consider such a derivation in the derivation of $\Gamma \vdash M : A$. Therefore, by imposing the restriction that a type A on which the translation is to be run is valid (that is, that $\Gamma \vdash A : \mathit{Type}$ has a derivation), we can use the

$$\begin{gathered}\phi(A) := \mathit{lf\text{-}obj} \text{ when } A \text{ is a base type}\\ \phi(\Pi x{:}A.\ P) := \phi(A) \to \phi(P)\\ \phi(\mathit{Type}) := \mathit{lf\text{-}type}\end{gathered}$$

Figure 4.1: Encoding of LF types in *hohh*

$$\begin{gathered}\langle u\ M_1 \dots M_n\rangle := u\ \langle M_1\rangle \dots \langle M_n\rangle\\ \langle x\ M_1 \dots M_n\rangle := x\ \langle M_1\rangle \dots \langle M_n\rangle\\ \langle \lambda x{:}A.M\rangle := \lambda^{\phi(A)} x.\langle M\rangle\end{gathered}$$

Figure 4.2: Encoding of LF terms in *hohh*

observation above and only treat judgments of the relevant form. Finally, note that we do not define the translation on types of the form $P \to Q$; this really is treated as a shorthand, as previously described, and so a variable w not appearing in Q is introduced so that the type takes the form $\Pi w{:}P.\ Q$.

The translation first encodes LF types and objects as *hohh* terms, as presented in Figures 4.1 and 4.2; here an object (type) of type (kind) P is represented by an *hohh* term of simple type $\phi(P)$, built from the atomic types *lf-type* and *lf-obj*. The encoding of an object of base type Q is then given by $\langle Q\rangle$ (note that the definition of the translation ensures that no non-base type is encoding using the $\langle \bullet \rangle$); variable names are reused with an appropriate type as part of the *hohh* signature in the process. For example, the LF type *append nil nil nil* gets translated to the same term in *hohh*, where it has type *lf-type*. This translation behaves well with respect to substitution and β-conversion, and is injective for objects of the same type, and for types of the same kind.

During this process a significant amount of typing information is lost; it is then recovered in the translation of LF judgements, shown in Figure 4.2. Here a clause for the predicate *hastype* of type *lf-obj* $\to$ *lf-type* $\to o$, which associates the encoding of an LF term M with the encoding of the LF type A of that term, are generated from an LF judgment $\Gamma \vdash M : A$. To emphasize the reliance only on the structure of the LF type, we first translate A into an *hohh* predicate $[A]$ describing what it means to be an object of type A, and then write

$$\begin{gathered}[\Pi x{:}A.\ B] := \lambda M.\forall x.[A](x) \supset [B](Mx)\\ [A] := \lambda M.\mathit{hastype}\ M\ \langle A\rangle \text{ where } A \text{ is a base type}\end{gathered}$$

Figure 4.3: Simplified translation of LF judgments to *hohh*

$$nat : \textit{lf-type},\ \ z : \textit{lf-obj},\ \ s : \textit{lf-obj} \to \textit{lf-obj},$$

$$list : \textit{lf-type},\ \ nil : \textit{lf-obj},\ \ cons : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj},$$

$$\begin{gathered} append : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-type}, \\ append_{nil} : \textit{lf-obj} \to \textit{lf-obj}, \\ append_{cons} : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \end{gathered}$$

Figure 4.4: *hohh* signature for the simplified translation of the LF specification for *append*

$[M : A]$ for $[A](\langle M\rangle)$; this allows for the translation of a type A for which an inhabitant is to be *sought* as an *hohh* formula $[A](\langle M\rangle)$, where M is left uninstantiated. This translation is defined on all canonical LF types.

Finally, in order to use the *hohh* logic program and goal generated from an LF context Γ and type A, we must give an *hohh* signature Σ. To generate Σ we take each item $x : A$ in Γ and add $x : A'$ to Σ, where A' is A translated to *hohh* via a *type-level* translation informed by the one in Figure 4.1: if A is a base type, $[\Pi x_1{:}B_1.\ \ldots \Pi x_n{:}B_n.\ A]$ has type $\tau \to o$ where $\tau = \phi(A)$. Therefore every constructor introduced in an LF specification will yield *two* things: first, a clause definition for *hastype* in the logic program, and second an element of the signature. For instance, given the LF assertion for successor, $s : nat \to nat$, we generate the *hohh* formula $\forall x. hastype\ x\ nat \supset hastype\ (s\ x)\ nat$, and an element of the *hohh* signature $s : nat \to nat$. Note that in this signature *nat* is an *hohh* atomic type, introduced by the LF assertion $nat : Type$.

Thus valid Twelf specifications are encoded by dropping all kind assignments and translating each type assignment they contain. As an example, the specification of *append* (Section 2.3) is translated into the logic program given in Figure 4.5, which are written assuming a signature containing the non-logical constants given, along with their types, in Figure 4.4. From these clauses we can, for example, find a derivation of $hastype\ (cons\ (s\ z)\ nil)\ list$, and we could search for terms X such that

$$hastype\ X\ (append\ (cons\ z\ nil)\ (cons\ (s\ z)\ nil)\ (cons\ z\ (cons\ (s\ z)\ nil)))$$

has a derivation under the relevant logic program and signature.

$$\begin{array}{l}
\textit{hastype } z \textit{ nat}, \\
\forall n.\ \textit{hastype } n \textit{ nat} \supset \textit{hastype } (s\ n) \textit{ nat}, \\
\\
\textit{hastype nil list}, \\
\forall n.\ \textit{hastype } n \textit{ nat} \supset \forall l.\ \textit{hastype } l \textit{ list} \supset \textit{hastype } (\textit{cons } n\ l) \textit{ list}, \\
\\
\forall l.\ \textit{hastype } l \textit{ list} \supset \textit{hastype } (\textit{append}_{\textit{nil}}\ l)\ (\textit{append nil } l\ l), \\
\forall x.\ \textit{hastype } x \textit{ nat} \supset \forall l.\ \textit{hastype } l \textit{ list} \supset \forall k.\ \textit{hastype } k \textit{ list} \supset \\
\qquad \forall m.\ \textit{hastype } m \textit{ list} \supset \forall a.\ \textit{hastype } a\ (\textit{append } l\ k\ m) \supset \\
\qquad \textit{hastype } (\textit{apppend}_{\textit{cons}}\ x\ l\ k\ m\ a)\ (\textit{append } (\textit{cons } x\ l)\ k\ (\textit{cons } x\ m))
\end{array}$$

Figure 4.5: Simplified translation of the LF specification for *append*

4.2 An extended example

Returning to the example of the *stlc* encoded in LF, let us consider the result of converting the LF specification given in Section 2.5 to *hohh* using the above translation. Although the translation is given in terms of generating *hohh* terms and formulas, we show the result of the translation as a λProlog program for readability. First we present the generated λProlog signature in Figure 4.6; this contains both the new atomic types *lf-type* and *lf-obj*, as well as all constants found in the initial specification. Each constant is given along with its type, as is required by λProlog. In addition, the constant *hastype* is defined with the relevant type. Notice that the type-level translation, which yields the type of each constant, eliminates a significant amount of information, leaving only the general structure of the original LF type, which can be quite illegible.

Next we present the logic program associated with the translation of the LF specification in Figure 4.7. As previously described λProlog allows implicit universal quantification of capitalized identifiers, which we make use of to simplify the presentation. Given our understanding of the mechanics of the translation, along with the original LF specification, we can identify various aspects of the specification within the logic program: each clause definition for *hastype* corresponds to an object that acts as a constructor for a type family, and each constant corresponds to either an object, a type, or type family.

Now let us consider the translation of an LF judgment. Again from our running example, we evaluate the *stlc* term $(\lambda x{:}\textit{unit}.x)\ \textit{void}$, which is encoded as the following term:

$$\alpha = \textit{app unit } (\textit{abs unit unit } (\lambda x{:}\textit{tm unit}.x))\ \textit{void}$$

Thus we must search for an inhabitant of the type *eval unit* α *void*, to show that α evaluates

```
kind lf-type     type.
kind lf-obj      type.
type hastype     lf-obj -> lf-type -> o.

type ty     lf-type.
type arrow  lf-obj -> lf-obj -> lf-obj.
type unit   lf-obj.

type tm     lf-obj -> lf-type.
type abs    lf-obj -> lf-obj -> (lf-obj -> lf-obj) -> lf-obj.
type app    lf-obj -> lf-obj -> lf-obj -> lf-obj -> lf-obj.
type void   lf-obj.

type eval       lf-obj -> lf-obj -> lf-obj -> lf-type.
type eval-void  lf-obj.
type eval-abs   lf-obj -> lf-obj -> (lf-obj -> lf-obj) -> lf-obj.
type eval-app   lf-obj -> lf-obj -> lf-obj -> lf-obj ->
                  (lf-obj -> lf-obj) -> lf-obj -> lf-obj ->
                  lf-obj -> lf-obj -> lf-obj -> lf-obj.
```

Figure 4.6: Simplified translation of the *stlc*; signature

to *void*. From the Teyjus top-level we attempt to prove the goal corresponding to $[\alpha](M)$ for a meta-variable M:

```
?- hastype M (eval unit
                  (app unit unit
                    (abs unit unit (x\ x)) void)
                  void).
M = eval-app unit unit (abs unit unit (W1\ W1)) void (W1\ W1)
      void void eval-void eval-void (eval-abs unit unit (W1\ W1))
```

The substitutions given indicate that the type does have an inhabitant, and in fact the inhabitant is exactly the binding for `M`. Thus, we can intuitively understand this solution to mean that the *stlc* term $(\lambda x{:}unit.x)$ *void* does in fact evaluate to *void*; we shall develop a formal footing for this intuition in the next section.

Making further use of meta-variables, we can use the translation to determine the *stlc* term to which $((\lambda x{:}unit.x)$ *void*$)$ evaluates, instead of assuming it before hand and then checking; here we replace the final argument to `eval` with the meta-variable `B`, which the system instantiates during search:

```
hastype T1 ty => hastype T2 ty => hastype (arrow T1 T2) ty.
hastype unit ty.

hastype T1 ty => hastype T2 ty =>
  (pi x\ hastype x (tm T1) => hastype (F x) (tm T2)) =>
  hastype (abs T1 T2 F) (arrow T1 T2).
hastype T1 ty => hastype T2 ty =>
  hastype L (tm (arrow T1 T2)) => hastype R (tm T1) =>
  hastype (app T1 T2 L R) (tm T2).
hastype void (tm unit).

hastype eval-void (eval unit void void).
hastype T1 ty => hastype T2 ty =>
  (pi V10\ hastype V10 (tm T1) => hastype (E V10) (tm T2)) =>
  hastype (eval-abs T1 T2 E)
          (eval (arrow T1 T2) (abs T1 T2 E) (abs T1 T2 E)).

hastype T1 ty => hastype T2 ty =>
  hastype A (tm (arrow T1 T2)) => hastype B (tm T1) =>
  (pi V14\ hastype V14 (tm T1) =>
    hastype (F V14) (tm T2))
  hastype X (tm T1) => hastype V (tm T2) =>
  hastype V11 (eval T2 (F X) V) =>
  hastype V12 (eval T1 B X) =>
  hastype V13 (eval (arrow T1 T2) A (abs T1 T2 (x\ F x))) =>
  hastype (eval-app T1 T2 A B F X V V11 V12 V13)
          (eval T2 (app T1 T2 A B) V).
```

Figure 4.7: Simplified translation of the *stlc*; logic program

```
?- hastype M (eval unit
                  (app unit unit
                    (abs unit unit (x\ x)) void)
                  B).
B = void
M = eval-app unit unit (abs unit unit (W1\ W1)) void (W1\ W1)
      void void eval-void eval-void (eval-abs unit unit (W1\ W1))
```

The meta-variable is instantiated to `void`, demonstrating that we can use the translation not only for checking whether a type is inhabited, but also for determining what conditions allow for the type to be inhabited, as we could in Twelf.

4.3 Correctness of the translation

We now show that the translation presented above is correct. The statement of correctness has two parts; the first, roughly that any LF derivation has a corresponding derivation in *hohh*, is relatively straightforward to state. The second, however, merits a closer look. The idea is that we would like to know that, given an LF type A, if there is an *hohh* derivation that yields a proof term for this type, then it corresponds to an LF object M, and M has type A. That is to say, we are not only interested in whether, given M and A, there is a derivation of $\Gamma \vdash M : A$ whenever there is one of $[\Gamma] \longrightarrow [M : A]$. We are also concerned with whether the translation will allow us to *produce* such an M given only A, and only do so when $\Gamma \vdash M : A$. This formulation makes it clear that when search succeeds it results always in a valid proof term.

Theorem 4.3.1 (Correctness of the simplified translation)**.** *Let Γ be a well-formed canonical LF context and let A be a canonical LF type such that $\Gamma \vdash A : Type$ has a derivation. If $\Gamma \vdash M : A$ has a derivation for a canonical object M, then there is a derivation of $[\Gamma] \longrightarrow [M : A]$. Conversely, if $[\Gamma] \longrightarrow [A](M)$ for an arbitrary* hohh *term M, then there is a canonical LF object M' such that $M = \langle M' \rangle$ and $\Gamma \vdash M' : A$ has a derivation.*

Proof. To prove completeness we proceed by induction on the derivation of $\Gamma \vdash M : A$ to build a derivation of $[\Gamma] \longrightarrow [M : A]$. We proceed by case analysis on the canonical type A.

- If A is of the form $\Pi x{:}B.\ A'$ then the LF derivation ends with the *abs-obj* rule and M must be of the form $\lambda x{:}B.M'$; note that despite the definition of the *abs-obj* rule in this case we need not consider B^β as A is already canonical, as described in Section 2.2.

$$\frac{\Gamma \vdash B : \mathit{Type} \quad \Gamma, x : B \vdash M' : A'}{\Gamma \vdash (\lambda x{:}B.M') : (\Pi x{:}B.\ A')}\ \mathit{abs\text{-}obj}$$

The derivation of $\Gamma \vdash (\Pi x{:}B.\ A') : \mathit{Type}$ contains a derivation of $\Gamma \vdash B : \mathit{Type}$ by *pi-fam*. The induction hypothesis gives us a derivation of $[\Gamma, x : B] \longrightarrow [M' : A']$ and by applying the rules $\forall R$ and $\supset R$ we derive $[\Gamma] \longrightarrow \forall x.\ [x : B] \supset [M' : A']$. This is a derivation of the expected goal $[(\lambda x{:}B.M') : (\Pi x{:}B.\ A')] = \forall x.\ [x : B] \supset [A'](\langle \lambda x{:}B.M' \rangle\ x)$, and $\langle M' \rangle = \langle \lambda x{:}B.M' \rangle\ x$ by β-conversion.

- If A is a base type then the canonical LF derivation ends with a chain of *app-obj* rules following a *var-obj* rule on $(x : \Pi \overrightarrow{y{:}B}.A') \in \Gamma$ with $A = A'[\overrightarrow{N/y}]$:

$$\frac{\{\Gamma \vdash N_i : B_i[N_1/x_1, \ldots, N_{i-1}/x_{i-1}]\}_i}{\Gamma \vdash x\overrightarrow{N} : A}$$

By the inductive hypothesis we obtain derivations $\mathcal{D}_i$ of

$$[\Gamma] \longrightarrow [N_i : B_i[N_1/x_1, \ldots, N_{i-1}/x_{i-1}]]$$

Moreover, $[\Gamma]$ contains the encoding of the context item used above, that is:

$$\forall y_1.\ ([B_1](y_1) \supset \ldots \supset \forall y_n.\ ([B_n](y_n) \supset \mathit{hastype}\ (x\ \overrightarrow{y})\ \langle A' \rangle))$$

By applying *backchain* to that clause, choosing $\langle N_i \rangle$ for y_i and using the derivations $\mathcal{D}_i$, we obtain a derivation of $[\Gamma] \longrightarrow \mathit{hastype}\ (x\ \overrightarrow{\langle N \rangle})\ (\langle A' \rangle[\overrightarrow{\langle N \rangle / y}])$, which is precisely $[x\overrightarrow{N} : A'[\overrightarrow{N/y}]]$.

We prove the soundness direction by induction on the derivation of $[\Gamma] \longrightarrow [A](M)$: assuming that $\Gamma \vdash A : \mathit{Type}$ has a derivation, we establish that $M = \langle M' \rangle$ and construct a derivation of $\Gamma \vdash M' : A$. We proceed by case analysis on the canonical type A.

- If A is of the form $\Pi x{:}B.\ A'$ then the structure of $[A]$ forces the *hohh* derivation to conclude as follows:

$$\frac{[\Gamma, x : B] \longrightarrow [A'](Mx)}{[\Gamma] \longrightarrow \forall x.\ [B](x) \supset [A'](Mx)}\ \forall R, \supset R$$

Since A is a valid *Type* under Γ, B must also be, and A' must be valid under $(\Gamma, x : B)$. We can thus apply the inductive hypothesis, and we obtain that $Mx = \langle M' \rangle$ and that

$\Gamma, x : B \vdash M' : A'$ is derivable. Since x does not occur free in M, we conclude that $M = (\lambda x.\langle M'\rangle) = \langle \lambda x{:}B.M'\rangle$, and we derive $\Gamma \vdash (\lambda x{:}B.M') : (\Pi x{:}B.\ A')$ using the *abs-obj* rule and our derivation of $\Gamma \vdash B : \mathit{Type}$.

- Otherwise, A is a base type, and our derivation of $\mathit{hastype}\ M\ \langle A\rangle$ must proceed by a *backchain* on some clause $\forall y_1.\ ([B_1](y_1) \supset \ldots \supset \forall y_n.\ ([B_n](y_n) \supset \mathit{hastype}\ (x\ \overrightarrow{y})\ \langle A'\rangle))$ where $A = \langle A'\rangle[\overrightarrow{N/y}]$ for some *hohh* terms $\overrightarrow{N}$:

$$\frac{\{[\Gamma] \longrightarrow ([B_i](y_i))[N_1/y_1 \ldots N_i/y_i]\}_i}{[\Gamma] \longrightarrow \mathit{hastype}\ (x\overrightarrow{N})\ (\langle A'\rangle[\overrightarrow{N/y}])}\ \mathit{backchain}$$

We know that $(x : \Pi\overrightarrow{y{:}B}.A') \in \Gamma$, and from $\vdash \Gamma\ \mathit{ctx}$ we deduce that $\Pi\overrightarrow{y{:}B}.A'$ is valid under Γ. We first establish that each $N_i = \langle N_i'\rangle$, and $B_i[N_1'/y_1 \ldots N_{i-1}'/y_{i-1}]$ is a valid type under Γ, by an inner induction on i: having derivations of $\Gamma \vdash B_j[N_1/y_1 \ldots N_{j-1}'/y_{j-1}] : \mathit{Type}$ for all $j < i$, we substitute them in the derivation of $\Gamma, x_1 : B_1, \ldots, x_{i-1} : B_{i-1} \vdash B_i : \mathit{Type}$ that we obtain from $\Gamma \vdash \Pi\overrightarrow{y{:}B}.A' : \mathit{Type}$ using Proposition 2.2.2. We can now apply our outer inductive hypothesis to obtain that $N_i = \langle N_i'\rangle$ and $\Gamma \vdash N_i' : B_i[N_1'/y_1 \ldots N_{i-1}'/y_{i-1}]$ has a derivation. Therefore we have that $\vdash \Gamma\ \mathit{ctx}$ has a derivation and that $\Gamma \vdash x : \Pi y_1{:}B_1.\ \ldots \Pi y_n{:}B_n.\ A'$ has a derivation (by *var-obj* on $x \in \Gamma$). From these we construct a derivation of $\Gamma \vdash x\overrightarrow{N'} : A'[\overrightarrow{N'/y}]$ by n applications of *app-obj*, employing the derivations of $\Gamma \vdash N_i' : B_i[N_1'/y_1 \ldots N_{i-1}'/y_{i-1}]$ to discharge the assumptions introduced by each application.

$\square$

4.4 An extended example, revisited

Let us revisit the example of Section 4.2, armed now with the knowledge that the translation is, in fact, correct. Recall that we are attempting to show that the *stlc* term $(\lambda x{:}\mathit{unit}.x)\ \mathit{void}$ evaluates to *void*:

```
?- hastype M (eval unit
                  (app unit unit
                    (abs unit unit (x\ x)) void)
                  void).
M = eval-app unit unit (abs unit unit (W1\ W1)) void (W1\ W1)
      void void eval-void eval-void (eval-abs unit unit (W1\ W1))
```

By Theorem 4.3.1 we know that the corresponding LF judgment, namely

$$\Gamma \vdash M' : \mathit{eval}\ \mathit{unit}\ (\mathit{app}\ \mathit{unit}\ \mathit{unit}\ (\mathit{abs}\ \mathit{unit}\ \mathit{unit}\ (\lambda x{:}\mathit{tm}\ \mathit{unit}.\mathit{void}))\ \mathit{void})\ \mathit{void}$$

has a derivation, where Γ is our LF specification from Section 2.5 and M' is an LF object such that $M = \langle M' \rangle$. Then by Proposition 2.5.1 we can conclude that $(\lambda x{:}\mathit{unit}.x)\ \mathit{void} \Rightarrow \mathit{void}$ has a derivation, and that indeed $(\lambda x{:}\mathit{unit}.x)\ \mathit{void}$ evaluates to *void* under a call-by-value evaluation strategy.

Note that, while the above proof of Theorem 4.3.1 assumes closed LF types, we have not yet considered the meaning of meta-variables in such types, what it means when a meta-variable is not bound during search, nor whether bindings for them are correct. Here there are two approaches. First, recall our previous discussion about meta-variables in Twelf: when meta-variables remain *unbound* after search, the interpretation is that for *any* term t of the meta-variable's type, there is an inhabitant of the sought type when the meta-variable is bound to t. Drawing from this understanding, we can treat unbound meta-variables in λProlog in the same way: consider searching for an inhabitant of the LF type A', where some sub-term of the encoding of A' has been replaced with a meta-variable M of type τ; call this A. If M remains unbound after successful search (returning proof-term P) we can conclude that, given a λProlog term N corresponding to the encoding of an LF term N' of type τ, there is an inhabitant of the same type as the meta-variable, then there is an inhabitant of A with the M bound to N, and it is exactly P. What is more, we know that these are encodings of the LF terms A', M', and N'. We can then extend this treatment even to meta-variables that *are* bound during search, by simply checking after search succeeds that the meta-variables have been properly instantiated. In practice this process is less intensive than proof search proper, and tends not to be overly expensive.

Second, it in fact should be the case that, under the translation described above, no meta-variable could possibly be bound to a term that corresponds to an LF term of incorrect type. In the example above, `B` was bound to the term `void`, which clearly has the correct LF type *tm unit*. The intuition here is that the only time a meta-variable is bound in the logic programs generated by the translation is when it is matched with the head of a clause. Since the original specification is valid, any such matching clause should impose only the correct type on the meta-variable. However, the statement and proof of this theorem is not at all obvious, and is further stymied by the fact that it is not clear how exactly this extension to Twelf, which we are seeking to emulate, should behave.

4.5 Issues associated with the translation

Unfortunately, the simplified translation presented in this section cannot be the basis of a practical implementation of logic programming in LF. There are a number of observations to be made in the context of the goals of the translation, namely transparency and efficiency. First, it is not at all clear, without a thorough understanding of how the translation applies to different LF constructs, that the λProlog program corresponds or is equivalent to the original LF specification. This makes it very difficult to determine whether the λProlog program is correct, and it makes the runtime characteristics of the program opaque. However, not all is for ill: recall that both our original LF specification and our λProlog program encoding the *stlc* used HOAS in the definition of *abs*; the translation is able to maintain this usage, as is visible in the *hastype* clause corresponding to $eval_{app}$.

Second, from the perspective of the type system of λProlog, the translation is a very shallow one. All LF objects have type *lf-obj*, all LF types have type *lf-type*, and no effort is made to make use of the rich typing information existing in the original LF specification when assigning types to constants. This has effects not only on the transparency of the translation, as described, but also on the efficiency of the translation. In particular, when backchaining on a goal of the form `hastype M A` we must investigate every single clause for `hastype`, *even those that cannot ever apply.* For instance, consider the goal `hastype M (arrow T1 T2)`. Given that the term `arrow T1 T2` corresponds to the LF type *ty*, we are assured that the only clauses that could match this goal are those that correspond to constructors for *ty*. However, the logic program above does not distinguish between clauses for different LF types, and so might attempt to backchain on every other clause before reaching the correct one, even those that construct unrelated types like $tm\ \alpha$.

However there is a far more important criticism to be levied in the context of efficiency. Proof search using a program the simplified translation produces may involve repeatedly proving goals of the form $hastype\ M\ A$ for (encodings of) the same object M and type A. This can be seen clearly from the example in Figure 4.5. The formulation of *append* described has an algorithmic complexity that is intuitively linear in the length of the first list: we peel off the head of the list repeatedly until we reach the constant time base case. However, consider how search proceeds on the goal `hastype M (append l k m)` for lists l, k, and m of arbitrary lengths: each time we peel off the head of l, we are forced to verify that the tail of the list is actually a list; that is, we must prove the goal `hastype l' list` where `l'` is the tail of `l`. Proving such a goal is linear in the length of the list, and so the complexity of *append* under the translation is *quadratic* in the length of the first list; we check subparts of the list for type correctness over and over again, even though a single check is logically sufficient. What's more, if we know at the outset that each argument to *append* (that is, each of `l`, `k`, and `m`) is already of the correct type, we should *never* develop these redundant derivations. This increase in complexity is an artifact of the translation: we know, having already checked that the first argument is in fact a list, that it will remain so throughout the course of proof search. However, this redundancy in "type-checking" is not easily detectable from the *hohh* program that is generated; rather, it must be determined, and shown to be safely eliminable, based on deeper properties of LF objects and derivations. It is this issue that we take up in the next section.

Chapter 5

An Optimized Translation into Predicate Logic

Having identified various issues with the translation of Chapter 4, we now set out to rectify them. In order to do so we introduce two modifications to the translation of Chapter 4. First, we identify and eliminate the redundancies in search that the translation causes, as described in Section 4.5. And second, we improve the transparency of the translation by making better use of the type system afforded us by *hohh*. We then present an example that serves to highlight the changes caused by these modifications in programs resulting from the optimized translation, and demonstrate some of the improvements associated with them. We prove that the optimized translation yielded by these modifications is correct, and finally revisit the example to understand the practical relevance of correctness in this context.

5.1 Eliminating redundancies in search

In order to eliminate some of the redundancies in proof search that we have seen, we must first identify them. We exploit the fact that we are considering derivations $\Gamma \vdash M : A$ where Γ and A have been type-checked; that is, we have derivations of both $\vdash \Gamma\ ctx$ and $\Gamma \vdash A : Type$. The essential observation has actually already been made in reference to the manner in which search proceeds in the example of *append*: certain type checking goals are evaluated over and over, even when it is clear that they need not be checked at all.

To make these ideas more concrete, suppose we are trying to determine whether the type

$$A = append\ (cons\ z\ nil)\ nil\ (cons\ z\ nil)$$

is inhabited given a context Γ defining lists of natural numbers and list concatenation, as in Figure 2.3. Given that Γ is valid we already know that A is a valid type, which notably means that $(cons\ z\ nil)$ is a valid object of type *list*: *append* is defined in Γ as taking three lists, and so the derivation of $\Gamma \vdash append\ (cons\ z\ nil)\ nil\ (cons\ z\ nil) : Type$ necessarily contains one of $\Gamma \vdash (cons\ z\ nil) : list$. Therefore, when searching for an inhabitant of this type, we should avoid proving such a typing judgment. We could factor that kind of

$$\frac{\Gamma; \cdot; x \sqsubset_o A_i \text{ for some } A_i}{\Gamma; x \sqsubset_t c\overrightarrow{A}}\ \textsc{app}_\mathrm{t} \qquad \frac{y_i \in \delta \text{ for each } y_i \quad y_i \text{ distinct}}{\Gamma; \delta; x \sqsubset_o x\ \overrightarrow{y}}\ \textsc{init}_\mathrm{o}$$

$$\frac{\Gamma, y; x \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.\ B}\ \textsc{pi}_\mathrm{t} \qquad \frac{y \notin \Gamma \text{ and } \Gamma; \delta; x \sqsubset_o M_i \text{ for some } i}{\Gamma; \delta; x \sqsubset_o y\ \overrightarrow{M}}\ \textsc{app}_\mathrm{o}$$

$$\frac{\Gamma; \delta, y; x \sqsubset_o M}{\Gamma; \delta; x \sqsubset_o \lambda y{:}A.M}\ \textsc{abs}_\mathrm{o}$$

Figure 5.1: Rigidly occurring variables in types and objects

observation into our optimized translation by removing some well-typedness constraints in the translation of type assignments of the form $\Gamma \vdash M : \Pi x{:}B.\ A$, thereby eliminating some redundancies.

We are therefore interested in methods that will allow us to statically identify situations in which type checking the target type of the instantiation of a universally quantified type will automatically ensure that the instantiating terms are themselves of the correct type. While the example of *append* might lead us to imagine that simply *appearing* in the target type of a constructor is sufficient, in fact this is not the case when higher order variables come into play. Instead the variables that our technique identifies as not needing to be checked are those x_i that occur in A in such a way that, no matter how the other variables $\overrightarrow{x}$ are instantiated, the term t_i instantiating it will still appear in $B[\overrightarrow{t/x}]$, and hence in the constructed type. This notion of *rigid occurrence* is expressed by the judgment $\overrightarrow{x}; \cdot; x_i \sqsubset_o B$, defined in Figure 5.1. The rules $\textsc{app}_\mathrm{t}$ and $\textsc{pi}_\mathrm{t}$ act on LF types, and the rules $\textsc{init}_\mathrm{o}$, $\textsc{app}_\mathrm{o}$, and $\textsc{abs}_\mathrm{o}$ act on LF objects; as usual, both types and objects are assumed to be in canonical form.

The following fundamental property of rigidly occurring variables will prove key to proving the correctness of the optimized translation. We first state the property in terms of LF, as it gives a feel for the essential content of the lemma.

Lemma 5.1.1. *Let $\overrightarrow{t}$ be a vector of LF objects, let Γ, $\Gamma_0 = x_1 : B_1, \ldots, x_n : B_n$, and Δ be LF contexts, let M be an LF object, and let A be an LF type, all being assumed canonical. Suppose that there are derivations of $\overrightarrow{x}; x_i \sqsubset_t A$, $\Gamma, \Gamma_0, \Delta \vdash M : A$ and $\Gamma, \Delta[\overrightarrow{t/x}] \vdash M : A[\overrightarrow{t/x}]$. Then there is a derivation of $\Gamma \vdash t_i : B_i[t_1/x_1, \ldots, t_{i-1}/x_{i-1}]$.*

While this statement, as we shall see, is true for LF, in order to accommodate our general result we must reformulate Lemma 5.1.1 in a rather technical way. Namely, we must allow for encoded types that are the result of instantiations of (a priori) arbitrary *hohh* terms. This generalization is necessitated by the fact that, at the outset, we are concerned with constructing *hohh* terms that correspond to objects inhabiting a given type, and so must take care to ensure that all *hohh* terms do in fact have an associated LF object of the correct type. First we define a relation to capture to fact that two *hohh* terms are in fact encodings of the same LF term; note that one of these *hohh* terms is under a substitution of other *hohh* terms.

Definition 5.1.2. *Let $\overrightarrow{t}$ be a vector of* hohh *terms, and $\overrightarrow{x}$ a vector of variables of the same length. If M and N are LF objects, then we write $(M \sim N)[\overrightarrow{t/x}]$ when $\langle M \rangle = (\langle N \rangle[\overrightarrow{t/x}])$. For LF types A and B, we write $(A \sim B)[\overrightarrow{t/x}]$ when the two types are equal up to $(\bullet \sim \bullet)[\overrightarrow{t/x}]$ on objects within. Finally we extend this notion to contexts of the same length by pushing it down to the types bound by the context. We shall omit $\overrightarrow{t}$ and $\overrightarrow{x}$ when they are obvious from the context, simply writing $P \sim Q$.*

We employ this definition in Lemma 5.1.3, below, to capture and maintain the validity of *hohh* terms. This statement is therefore essentially the same as that of Lemma 5.1.1, except that it is over encodings of LF objects and types.

Lemma 5.1.3. *Let $\overrightarrow{t}$ be a vector of* hohh *terms, $\overrightarrow{x}$ a vector of variables, and $\overrightarrow{B}$ of canonical LF types, all of same length, such that $t_j = \langle t'_j \rangle$ for $j < i$. Let Γ be an LF context, and let $\Gamma_0 = x_1 : B_1, \ldots, x_n : B_n$.*

1. *Let Δ be an LF context, M an LF object and A a type, all being assumed canonical. Suppose that there are derivations of $\overrightarrow{x}; \delta; x_i \sqsubset_o M$, of $\Gamma, \Gamma_0, \Delta \vdash M : A$, and of $\Gamma, \Delta' \vdash M' : A'$, with $A' \sim A$, $M' \sim M$ and $\Delta' \sim \Delta$. Then t_i is of the form $\langle t'_i \rangle$ and there is a derivation of $\Gamma \vdash t'_i : B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]$.*

2. *Let $\Pi\overrightarrow{x{:}B}.A$ be a canonical base type. Suppose that there are derivation of $\Gamma \vdash \Pi\overrightarrow{x{:}B}.A : Type$, of $\overrightarrow{x}; x_i \sqsubset_t A$, and of $\Gamma \vdash A' : Type$ for $A' \sim A$. Then $t_i = \langle t'_i \rangle$ and there is a derivation of $\Gamma \vdash t'_i : B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]$.*

Proof. We prove part (1) by induction on the structure of the derivation of $\overrightarrow{x}; \delta; x_i \sqsubset_o M$. We let $\mathcal{D}$ be the derivation of $\Gamma, \Gamma_0, \Delta \vdash M : A$, and $\mathcal{D}'$ be the derivation of $\Gamma, \Delta' \vdash M' : A'$.

- In the base case of $\textsc{init}_o$ $M = x_i \overrightarrow{y}$ and $\mathcal{D}$ ends with an a series of applications of *app-obj* followed by an application of *var-obj*:

$$\frac{\vdash \Gamma, \Gamma_0, \Delta \ ctx \quad (x_i : \Pi \overrightarrow{z{:}C}.D) \in (\Gamma, \Gamma_0, \Delta)}{\Gamma, \Gamma_0, \Delta \vdash x_i : A_0} \ \textit{var-obj}$$

Therefore $A = D[\overrightarrow{y}/\overrightarrow{z}]$. Note that, because the variables y_i are distinct bound variables that are fresh with respect to D, this substitution can be inverted, and we thus have $A[\overrightarrow{z}/\overrightarrow{y}] = D$. The other sub-derivations of the chain of *app-obj* applications establish $\Gamma, \Gamma_0, \Delta \vdash y_i : C_i[y_1/z_1 \dots y_{i-1}/z_{i-1}]$; In fact, those sub-derivations must end with *var-obj* on $(y_i : C_i[\overrightarrow{y}/\overrightarrow{z}]) \in \Delta$ and hence $(y_i : C'_i[\overrightarrow{y}/\overrightarrow{z}]) \in \Delta'$ for $C'_i \sim C_i$.

We next determine t'_i. By η-equivalence we can assume that t_i to be of the form $\lambda z_1 \dots \lambda z_n.u$. We have $\langle M' \rangle = t_i \overrightarrow{y} = u[\overrightarrow{y/z}]$, and hence $u = \langle M' \rangle[\overrightarrow{z/y}] = \langle M'[\overrightarrow{z/y}]\rangle$, by first inverting the substitution, as it is injective, and then pushing it inside the encoding, as the encoding is the identity on variables. Let $t'_i = \lambda \overrightarrow{z{:}C'}.u'$ where $u' = M'[\overrightarrow{z/y}]$ Then we have:

$$\langle t'_i \rangle = \lambda z_1 \dots \lambda z_n.\ \langle M' \rangle[\overrightarrow{z/y}] = \lambda z_1 \dots \lambda z_n.\ u = t_i$$
$$M' = t'_i\ y_1 \dots y_n = u'[\overrightarrow{y/z}]$$

Given that $\mathcal{D}'$ derives $\Gamma, \Delta' \vdash M' : A'$, we obtain a derivation of $\Gamma, \Delta'[\overrightarrow{z}/\overrightarrow{y}] \vdash u' : A'[\overrightarrow{z/y}]$ by renaming variables $\overrightarrow{y}$ into $\overrightarrow{z}$, employing Proposition 2.2.3. The context $\Delta'[\overrightarrow{z/y}]$ contains assignments $(z_i : C'_i)$ and the other variables in its domain do not occur in u' nor $A'[\overrightarrow{z/y}]$ (since $A' \sim A$, $A = D[\overrightarrow{y/z}]$ and D is a sub-term of B_i which cannot contain any y_i). We then have $\Gamma \vdash (\lambda \overrightarrow{z{:}C'}.u') : (\Pi \overrightarrow{z{:}C'}.A'[\overrightarrow{z/y}])$ by weakening unused variables and using *abs-obj* to introduce the variables $\overrightarrow{z}$. This is a typing derivation for t'_i; we must now show that the associated type is actually $B_i[t'_1/x_1 \dots t'_{i-1}/x_{i-1}]$.

Since $B_i = \Pi \overrightarrow{z{:}C}.A[\overrightarrow{z}/\overrightarrow{y}]$, we have $\langle A \rangle[t_1/x_1 \dots t_n/x_n] = \langle A[t'_1/x_1 \dots t'_{i-1}/x_{i-1}]\rangle$. From $A' \sim A$ we thus obtain, by injectivity of $\langle \bullet \rangle$, that $A' = A[t'_1/x'_1 \dots t'_{i-1}/x_{i-1}]$. From this we obtain that $A'[\overrightarrow{z}/\overrightarrow{y}] = A[t'_1/x'_1 \dots t'_{i-1}/x_{i-1}][\overrightarrow{z}/\overrightarrow{y}]$. We permute these substitutions, and hence $\Pi \overrightarrow{z{:}C'}.A'[\overrightarrow{z}/\overrightarrow{y}]$ is indeed $B_i[t'_1/x'_1 \dots t'_{i-1}/x_{i-1}]$.

- Next consider the case of INIT$_\circ$ when $M = x_i\ y$ for some $y \in \Delta$. In this case $\mathcal{D}$ ends with an application of *app-obj*

$$\frac{\Gamma \vdash x_i : \Pi z{:}A_1.\ A_2 \quad \Gamma \vdash y : A_1}{\Gamma \vdash (x_i\ y) : (A_2[y/z])^\beta} \ \textit{app-obj}$$

The first premise is immediate satisfied using *var-obj* as in the base case, and as Γ_0 contains $x_i : B_i$ it must be the case that $B_i = \Pi z{:}A_1.\ A_2$. The second premise must be satisfied using *var-obj* as well, as Δ binds y. Therefore $A' \sim A_2[y/z]$.

- In the ABS$_\mathrm{o}$ case, we have $M = \lambda y{:}A_1.N$ and $\mathcal{D}$ ends with the *abs-obj* rule as follows:

$$\frac{\Gamma, \Gamma_0, \Delta \vdash A_1 : \mathit{Type} \quad \Gamma, \Gamma_0, \Delta, y : A_1 \vdash N : A_2}{\Gamma, \Gamma_0, \Delta \vdash (\lambda y{:}A_1.N) : (\Pi y{:}A_1.\ A_2)}\ \mathit{abs\text{-}obj}$$

  Then $A' \sim \Pi y{:}A_1.\ A_2$, and so A' must be of the form $\Pi y{:}A_1'.\ A_2'$ where $A_i' \sim A_i$. Similarly, we have $M' = \lambda y{:}A_1'.N'$ with $N' \sim N$. Then, $\mathcal{D}'$ must contain a derivation of $\Gamma, \Delta', y : A_1' \vdash N' : A_2'$, and we conclude by inductive hypothesis.

- In the APP$_\mathrm{o}$ case, we have $M = y\overrightarrow{N}$, $y \notin \overrightarrow{x}$ and $\overrightarrow{x}; \delta; x_i \sqsubset_o N_j$. The derivation $\mathcal{D}$ starts with a chain of *app-obj* applications, followed by *var-obj* on some $y : \Pi\overrightarrow{y{:}C}.A_1$, with $A = A_1[\overrightarrow{N/y}]$. The premise corresponding to N_j establishes that

$$\Gamma, \Gamma_0, \Delta \vdash N_j : C_j[N_1/y_1, \ldots, N_{j-1}/y_{j-1}]$$

  Since $A' \sim \Pi\overrightarrow{y{:}C}.A_1$, A' is of the form $\Pi\overrightarrow{y{:}C'}.A_1'$ with $C_j' \sim C_j$. Since $M' \sim y\overrightarrow{N}$ and since y is not affected by the instantiation of $\overrightarrow{x}$, it must be that M' is of the form $y\overrightarrow{N'}$ with all $N_j' \sim N_j$. The derivation $\mathcal{D}'$ must proceed in a similar fashion, namely a chain of *app-obj* applications followed by *var-obj* on y. Therefore we have a derivation of $\Gamma, \Delta' \vdash N_j' : C_j'[N_1'/y_1, \ldots, N_{j-1}'/y_{j-1}]$. We can conclude by the inductive hypothesis because $C_j'[N_1'/y_1, \ldots, N_{j-1}'/y_{j-1}] \sim C_j[N_1/y_1, \ldots, N_{j-1}/y_{j-1}]$ (which relies on the disjointness of $\overrightarrow{x}$ and $\overrightarrow{y}$).

The proof of (2) follows similar mechanisms. First, by a straightforward inspection of the first rules of the derivation of $\Gamma \vdash \Pi\overrightarrow{x{:}B}.A : \mathit{Type}$ we extract a derivation of $\Gamma, \Gamma_0 \vdash A : \mathit{Type}$. Then, since A is a base type, it must be (by APP$_\mathrm{t}$) that x_i rigidly occurs in one of its arguments M. Note that A and A' have the same structure on the path leading to M, since no objects are involved there. Hence, a simultaneous inspection of the first rules of the derivations of $\Gamma, \Gamma_0 \vdash A : \mathit{Type}$ and $\Gamma \vdash A' : \mathit{Type}$, yields derivations of $\Gamma, \Gamma_0 \vdash M : T$ and $\Gamma \vdash M' : T'$ for $M' \sim M$ and $T' \sim T$. We can conclude using part (1). □

Despite the technical nature of the statement and proof of Lemma 5.1.3, the more intuitive statement about LF actually follows directly from part (2); we simply instantiate $\overrightarrow{t}$ with *hohh* terms that already correspond to the encodings of the relevant LF objects.

The definition of rigidity described above might seem very restrictive. In particular, one might want to allow $\Gamma; \delta; x \sqsubset_o x\overrightarrow{N}$ in the $\textsc{init}_o$ rule; that is, when $\overrightarrow{N}$ can contain any terms at all, including those from Γ. However, with such a rule the rigidity lemma described above is no longer true. Consider a signature with $num : nat \to type$ and $num_n : \Pi n{:}nat.\ (num\ n)$. If we let $t = num_n$ then we must have a derivation of $\Gamma \vdash (t\ z) : (num\ z)$, and also one of $\Gamma, x : (nat \to num\ z) \vdash (x\ z) : (num\ z)$. However, we clearly do not have $\Gamma \vdash t : nat \to num\ z$: the context dictates that $t : \Pi n{:}nat.\ (num\ n)$. This is a counter-example to Lemma 5.1.3, part (1). Such counter-examples are a direct result of dependent types; in a simply-typed setting the essential property that given an application $M\ N$, its type, and the type of N, we can recover the type of M is easily shown to be true.

5.2 A deeper use of types

Next we address the issue of the transparency of generated logic programs by making a more essential use of the type system of *hohh*. The idea here is to reflect LF types as *hohh* types as accurately as possible, under the clear constraint that LF allows for dependencies between types and terms, whereas *hohh* does not.

Consider the fragment of LF in which types do not depend on terms; that is, the syntax of types is $A := u \mid \Pi x{:}A_1.\ A_2$. Certainly A_2 cannot depend on x, and so we might as well write the second case as $A_1 \to A_2$. This restricts the set of kinds we need to simply $K := Type$. Under these constraints it becomes clear how we might encode an LF type A as a *hohh* type: when A is atomic, introduce a new *hohh* atomic type of the same name and use this, and when A is of the form $A_1 \to A_2$ we simply construct the type using the $\to$ of *hohh* from the encodings of A_1 and A_2.

It thus remains to determine how to encode dependent types in *hohh*. In fact we cannot by making use of the type system alone. Instead, when A is of the form $\Pi x{:}A_1.\ A_2$ and A_2 does depend on x, we will use the same encoding above. This leaves open the possibility that some *hohh* term will not correspond to a well-typed LF term. We therefore will still be required to include in relevant places proof obligations corresponding to type checking, as we have under the simplified translation of Chapter 4. This ensures that, while the encoding admits some "invalid" terms as before, nevertheless they will not cause derivations to go awry. To see how this plays out, we will consider the example of *append* once more. Whereas under the simplified translation lists, natural numbers, and derivations of *append* all had type *lf-obj*, here we make richer use of types when defining constants.

```
kind nat type.
type z nat.
type s nat -> nat.

kind list type.
type nil list.
type cons nat -> list -> list.

kind append type.
type append-nil list -> append.
type append-cons nat -> list -> list -> list -> append -> append.
```

Next, we address the fact that the only predicate to be found under the simplified translation is *hastype*. Here we recognize that, when translating an LF context item or judgment, we always know what type family a Π-bound variable should have; the type system of LF does not allow for parametric polymorphism. We can thus think of defining several $hastype_c$ predicates, one for each type constructor c in the LF context. When translating a context item or judgment of the form $M : A$ when A is a base type (that is, $A = c\overrightarrow{N}$), instead of generating the *hohh* term *hastype* $\langle M\rangle$ $\langle A\rangle$ we could generate $hastype_c$ $\langle M\rangle$ $\overrightarrow{\langle N\rangle}$. This has the benefit, from the perspective of an implementation using Teyjus, of optimizing backchaining by limiting the number of clauses that must be investigated. In addition, it improves the transparency of the translation, by making it completely clear to which type a goal corresponds.

Returning to *append*, and employing the technique described above, we arrive at the following logic program:

```
hastype-nat z.
hastype-nat N => hastype-nat (s N).

hastype-list nil.
hastype-nat N => hastype-list L => hastype-list (cons N L).

hastype-append (append-nil L) nil L L.
hastype-nat X =>
  hastype-list L => hastype-list K => hastype-list M =>
  hastype-append V L K M =>
  hastype-append (append-cons X L K M V) (cons X L) K (cons X M).
```

Finally there is one optimization based on the nature of the Teyjus implementation that we will mention here: as previously described, the *indexing* optimization is only useful when the first argument to a predicate is not an uninstantiated meta-variable. As during proof search the argument corresponding to the proof term is often uninstantiated, we should avoid allowing the proof term to take first position. Instead of placing it first, as described above and in a fashion similar to the translation of Chapter 4, we can place it last in the list of arguments, so that instead of generating $\mathit{hastype}_c\ \langle M\rangle\ \overrightarrow{\langle N\rangle}$ we generate $\mathit{hastype}_c\ \overrightarrow{\langle N\rangle}\ \langle M\rangle$; we call this slight modification the *indexing extension.* Note that it is not always the case that the proof-term is uninhabited; during search we may be obligated to *check* that a particular term has a given type, and in this case the term is already known.

5.3 An optimized translation for logic programming

Given the above techniques for improving the simplified translation, we can think of the optimized translation as a two step process. First, the specification (call it Γ) is statically checked to ensure that it has certain properties (namely, that $\vdash \Gamma\ \mathit{ctx}$). Then, the specification is translated in a manner similar to the original translation, but with various checks removed; exactly *which* checks are elided will be dictated by rigidity described in Section 5.1, and these checks are guaranteed to succeed so long as the static checking succeeded. That this enhancement of our translation will not only yield an optimization in terms of runtime performance, it will also makes the specification more readable, and effectively restore its intended operational behavior. In addition, the optimized translation embodies the ideas expressed in Section 5.2. Note however that we do not include the indexing extension here; in fact the definition of the optimization translation, and the proof of its correctness, are almost entirely unaffected by the use of this extension.

The $[\![\bullet]\!]^+$ translation is used on type assignments appearing negatively (notably context items) and $[\![\bullet]\!]^-$ on positive typing judgments (notably the conclusion of LF assertions). As before, that translation is entirely guided by the type, and defined for all canonical types. We shall use the notation $[\![M : A]\!]^+$ for $[\![A]\!]^+(\langle M\rangle)$ and $[\![M : A]\!]^-$ for $[\![A]\!]^-(\langle M\rangle)$, and define $[\![\Gamma]\!]^+$ as the result of applying the translation to each context item, dropping kind assignments. Note that instead of replacing unnecessary typing judgments with $\top$ we could simply elide them all together; we use $\top$ as a placeholder because it simplifies later proofs.

Whereas in the simplified translation only two new atomic types we introduced (namely *lf-type* and *lf-obj*), under the optimized translation we add several. For each context item $u : K$ in Γ we introduce a new atomic type u_{type} (the subscript is added only for easy disambiguation, as eventually we shall introduce a constant named u). Given these new atomic types we can define a new type-level translation of an LF type A into an *hohh* type:

- When A is a base type, $A = u\overrightarrow{M}$ and the *hohh* type corresponding to A is just u_{type}.

$$
\begin{aligned}
[\![\Pi x{:}A.\ B]\!]^+_\Gamma &:= \begin{cases} \lambda M.\ \forall x.\ \top \supset [\![B]\!]^+_{\Gamma,x}(M\ x) & \text{if } \Gamma; x \sqsubset_t B \\ \lambda M.\ \forall x.\ [\![A]\!]^-(x) \supset [\![B]\!]^+_{\Gamma,x}(M\ x) & \text{otherwise} \end{cases} \\
[\![u\overrightarrow{N}]\!]^+_\Gamma &:= \lambda M.\ u\ M\ \overrightarrow{\langle N \rangle} \text{ where } A \text{ is a base type} \\
[\![\Pi x{:}A.\ B]\!]^- &:= \lambda M.\ \forall x.\ [\![A]\!]^+_\cdot(x) \supset [\![B]\!]^-(M\ x) \\
[\![u\overrightarrow{N}]\!]^- &:= \lambda M.\ u\ M\ \overrightarrow{\langle N \rangle} \text{ where } A \text{ is a base type}
\end{aligned}
$$

Figure 5.2: Optimized translation of LF specifications and judgments to *hohh*

- When $A = \Pi x{:}A_1.\ A_2$ then the *hohh* type corresponding to A is $\tau_1 \to \tau_2$, where τ_i is the type-level translation of A_i.

Having the type-level translation, we can proceed to the generation of the signature, which is more involved than it was for the simplified translation. Given an LF context Γ we generate Σ as follows:

- For each context item $x : A$ we introduce a constant x whose type is given by the type-level translation of A.

- For each context item $u : K$ we introduce a constant u whose type is $u_{type} \to \tau$, where τ is determined as follows: if $K = Type$, then $\tau = o$; otherwise when $K = A_1 \to \ldots \to A_n \to Type$ then $\tau = \tau_1 \to \ldots \to \tau_n \to o$ where τ_i is the type-level translation of A_i. Notice that this constant corresponds, with only a slight change in name for readability, to the $hastype_u$ constants, one for each type family u, described in Section 5.2.

5.4 An extended example

We once again return to our running example: encoding the *stlc*. The result of translating the LF specification of Section 2.5 using the optimized translation is given in Figure 5.3, which includes both the λProlog signature and logic program. Notice that we have renamed new atomic types of the form u_{type} to simply `u`; this is possible because in λProlog the sets of types and constants are separate. Also, we again make use of the implicit universal quantification afforded by λProlog.

First let us consider the generated signature. Each LF type and type family $u : K$ has given rise to a new λProlog constant of the same name. This constant takes several

```
kind ty          type.
type ty          ty -> o.

type arrow       ty -> ty -> ty.
type unit        ty.

kind tm          type.
type tm          tm -> ty -> o.
type abs         ty -> ty -> (tm -> tm) -> tm.
type app         ty -> ty -> tm -> tm -> tm.
type void        ty.

kind eval        type.
type eval        eval -> ty -> tm -> tm -> o.

type eval-void   eval.
type eval-abs    ty -> ty -> (tm -> tm) -> eval.
type eval-app    ty -> ty -> tm -> tm ->
                   (tm -> tm) -> tm -> tm ->
                   eval -> eval -> eval -> eval.

ty V1 => ty V2 => ty (arrow V1 V2).
ty unit.

(pi x\ tm x T1 => tm (V x) T2) => tm (abs T1 T2 V) (arrow T1 T2).
ty T1 => tm V1 (arrow T1 T2) => tm V2 T1 => tm (app T1 T2 V1 V2) T2.
tm void unit.

eval eval-void unit void void.
eval (eval-abs T1 T2 E) (arrow T1 T2) (abs T1 T2 E) (abs T1 T2 E).
(pi V13\ tm V13 T1 => tm (F V13) T2) => tm X T1 =>
  eval V11 T1 B X  => eval V10 T2 (F X) V =>
  eval V12 (arrow T1 T2) A (abs T1 T2 (x\ F x)) =>
  eval (eval-app T1 T2 A B F X V V10 V11 V12) T2 (app T1 T2 A B) V.
```

Figure 5.3: Optimized translation of the *stlc*

arguments, the first corresponding to a proof term, the rest corresponding to the arguments of the type family itself. Clauses for this constant should be understood as defining a judgment derivable only when the first argument (the proof term) has the type LF type constructed from the LF type family u and LF objects whose encodings under $\langle\bullet\rangle$ are the remaining arguments. In addition, each object constructor $x : A$ has a matching λProlog constant. Notice that the type is transparently equivalent to the original LF type, and, for non-dependent types like *ty*, is also transparently equivalent to the corresponding type in λProlog.

We can furthermore compare the generated logic program with our original λProlog implementation of evaluation for the *stlc*, show in Figure 3.2. First, let us rewrite the clauses for `eval`, again making use of the equivalence between $\alpha \wedge \beta \supset \gamma$ and $\alpha \supset \beta \supset \gamma$. This mechanical translation (which can be performed using the Teyjus system) generates a set of clauses that more closely match the usual style of λProlog logic programs:

```
eval eval-void unit void void.
eval (eval-abs T1 T2 E) (arrow T1 T2) (abs T1 T2 E) (abs T1 T2 E).
eval (eval-app T1 T2 A B F X V V10 V11 V12) T2 (app T1 T2 A B) V :-
  eval V12 (arrow T1 T2) A (abs T1 T2 (x\ F x)),
  eval V11 T1 B X,
  eval V10 T2 (F X) V,
  tm X T1,
  (pi V13\ tm V13 T1 =>
    tm (F V13) T2).
```

Notice that the number of clauses for `eval` is in each case the same, as expected. Furthermore, the clause for the base case of evaluating *void* is almost exactly the same, save for the addition of the argument `eval-void` in the case of the optimized translation. This argument corresponds to the proof term, which has no relative in the context of straight λProlog. This pattern is played out for all clauses in fact; all arguments but the first are exactly the same.

Next we consider the bodies of clauses; let us first study the second base case, that of evaluating an abstraction. The hand-written λProlog program appears slightly simpler here, but this is just due to the fact that λProlog cannot express the dependent type that *abs* has in LF; otherwise the essence of the clause is exactly the same, stating that abstractions evaluate to themselves. Of greater interest is the clause for evaluation of application. Here the λProlog program is straightforward, evaluating the head and argument of the application before using a meta-level application to compute the object level substitution. Finally the result of this substitution is itself evaluated. In the case of the translation, there is more to it. First the head and argument are evaluated, followed by evaluation of the result of the substitution, as in the hand handwritten program. However, after this the head and the argument are additionally checked to have the same (LF level) type after evaluation as before. Upon reflection the presence of these additional type checking goals is easily explained: the LF specification required this property, through the use of dependent types. The types provided by λProlog are not sufficient to maintain this invariant, and so we must discharge these extra proof obligations.

Indeed, if we wish to write an LF specification that corresponds exactly to the handwritten λProlog program we must consider that the handwritten program *does not* maintain as strong invariants about the structure of terms encoding *stlc* terms as our original LF specification. Instead we should define a weaker term representation, as follows:

```
tm : type.
app : tm -> tm -> tm.
abs : (tm -> tm) -> tm.
void : tm.
```

This representation clearly does not require that a term represent a well-typed *stlc* term, and on this weaker representation the optimized translation generates a logic program that is exactly the same as the handwritten one, again save for the fact that the generated program includes a proof term as the first argument to each predicate.

5.5 Correctness of the optimized translation

We shall now establish the correctness of the optimized translation, by showing it equivalence to the simplified translation.

Theorem 5.5.1. *Let Γ be an LF context and A an LF type, both canonical, such that $\vdash \Gamma$ ctx and $\Gamma \vdash A$: Type are derivable. Then when M is an arbitrary* hohh *term, $[\Gamma] \longrightarrow [A](M)$ has a derivation if and only if $[\![\Gamma]\!]^+ \longrightarrow [\![A]\!]^-(M)$ has a derivation.*

Proof. We establish the soundness direction by induction on the derivation of the optimized translation, maintaining the assumptions about Γ and A.

- If A is of the form $\Pi x{:}B.\ A'$ our derivation ends as follows:

$$\frac{[\![\Gamma, x : A']\!]^+ \longrightarrow [\![B]\!]^-(M\ x)}{[\![\Gamma]\!]^+ \longrightarrow [\![\Pi x{:}B.\ A']\!]^-(M)}\ \forall R, \supset R$$

First, since Γ and A are well-formed, $\Gamma \vdash B : \mathit{Type}$, $\vdash (\Gamma, x : B)\ \mathit{ctx}$ and $\Gamma, x : B \vdash A' : \mathit{Type}$ must have derivations. We can thus apply the inductive hypothesis, obtaining that $[\Gamma, x : B] \longrightarrow [A'](M\ x)$ has a derivation. By $\forall R$ and $\supset R$, $[\Gamma] \longrightarrow [\Pi x{:}B.\ A'](M)$ has one as well.

- If A is a base type, then our derivation starts with a backchaining on the encoding of some $(y : \Pi\overrightarrow{x{:}B}.A') \in \Gamma$, *i.e.* $\forall x_1.\ ([\![B_1]\!]^-(x_1) \supset \ldots \supset \forall x_n.\ ([\![B_n]\!]^-(x_n) \supset (u\ (y\ \overrightarrow{x})\ \overrightarrow{\langle N\rangle})))$:

$$\frac{[\![\Gamma]\!]^+ \longrightarrow F_1 \quad \ldots \quad [\![\Gamma]\!]^+ \longrightarrow F_n}{[\![\Gamma]\!]^+ \longrightarrow (u(y\,\overrightarrow{x})\overrightarrow{\langle N\rangle})[\overrightarrow{t/x}]}\ \mathit{backchain}$$

Here F_i is either $([\![B_i]\!]^-(x_i))[t_1/x_1, \ldots, t_i/x_i]$ or $\top$. We perform an inner induction on $i \leq n$, showing that for all $j \leq i$, $t_j = \langle t'_j\rangle$ for some LF object t'_j, and that we have derivations of $[\Gamma] \longrightarrow [B_j[t'_1/x_1, \ldots, t'_{j-1}/x_{j-1}]](t'_j)$ and of $\Gamma \vdash t'_j : B_j[t'_1/x_1, \ldots, t'_{j-1}/x_{j-1}]$.

  - We first treat the case where $F_i = \top$, *i.e.* there is a derivation of $\overrightarrow{x}; x_i \sqsubset_t A'$. We assumed that $\Gamma \vdash A : \mathit{Type}$, and since Γ is valid we also have a derivation of $\Gamma \vdash \Pi\overrightarrow{x{:}B}.A' : \mathit{Type}$. We can thus apply Lemma 5.1.3, to obtain t'_i and a derivation of $\Gamma \vdash t'_i : B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]$, and we conclude $[\Gamma] \longrightarrow [B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]](t_i)$ by Theorem 4.3.1.
  - When $F_i \neq \top$, we can see that within the derivation of $\Gamma \vdash \Pi\overrightarrow{x{:}B}.A' : \mathit{Type}$ there is a derivation of $\Gamma, x_1 : B_1, \ldots, x_{i-1} : B_{i-1} \vdash B_i : \mathit{Type}$. By substituting (Proposition 2.2.2) the derivations provided by the inner inductive hypothesis on this formula we construct a derivation of $\Gamma \vdash B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}] : \mathit{Type}$. We can now apply the outer inductive hypothesis on F_i, to conclude that $[\Gamma] \longrightarrow [B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]](t_i)$ has a derivation. By Theorem 4.3.1, we finally obtain that t_i is of the form $\langle t'_i\rangle$.

  We compose all derivations $[\Gamma] \longrightarrow [B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]](t_i)$ by *backchain* on the encoding of $(y : \Pi\overrightarrow{x{:}B}.A') \in \Gamma$, obtaining the expected derivation of $[\Gamma] \longrightarrow \mathit{hastype}\ (y\ \overrightarrow{t})\ (u\ \overrightarrow{\langle N\rangle})[\overrightarrow{t/x}]$, which concludes this direction of the proof.

Completeness is proved by an induction on the derivation of $[\Gamma] \vdash [M] : [A]$. Note that for this direction of the proof we are simply dropping information (sub-derivations) and so we do not rely on Γ being a valid specification or A being a valid type. We proceed by induction on the structure of the derivation of $[\Gamma] \longrightarrow [A](M)$, followed by case analysis on A.

- If A is of the form $\Pi x{:}B.\ A'$ our derivation ends as follows:

$$\frac{[\Gamma, x : B] \longrightarrow [A']\ (M\ x)}{[\Gamma] \longrightarrow [\Pi x{:}B.\ A']\ M}\ \forall R, \supset R$$

  By the inductive hypothesis $[\![\Gamma, x : B]\!]^+ \longrightarrow [\![A']\!]^-\ (M\ x)$ has a derivation, and by applying $\forall R$ and $\supset R$ to this derivation we can construct a derivation of $[\![\Gamma]\!]^+ \longrightarrow [\![\Pi x{:}B.\ A']\!]^-\ M$.

- Otherwise, A is a base type and our derivation proceeds by backchaining on some $(y : \Pi \overrightarrow{x{:}B}.A') \in \Gamma$, with $\langle A\rangle = \langle A'\rangle[t_1/x_1 \dots t_n/x_n]$:

$$\frac{[\Gamma] \longrightarrow F_1 \quad \dots \quad [\Gamma] \longrightarrow F_n}{[\Gamma] \longrightarrow [A]\ (y\ \overrightarrow{t}\,)}\ \textit{backchain}$$

  Here, $F_i = ([B_i]\ x_i)[t_1/x_1 \dots t_n/x_n]$. As in the completeness proof of the simplified encoding, we obtain by an inner induction that each t_i is of the form $\langle t'_i\rangle$ and thus that $F_i = [B_i[t'_1/x_1 \dots t'_n/x_n]](t_i)$.

  We shall build the derivation of $[\![\Gamma]\!]^+ \longrightarrow [\![A]\!]^-(y\ \overrightarrow{t}\,)$ by using *backchain* on the optimized translation of $(y : \Pi \overrightarrow{x{:}B}.A') \in \Gamma$, by choosing $\overrightarrow{t}$ for $\overrightarrow{x}$. The resulting premises are either

$$[\![\Gamma]\!]^+ \longrightarrow [\![B_i[t'_1/x_1 \dots t'_n/x_n]]\!]^-\ t_i$$

  when x_i does not occur rigidly in A', and this case is provided for by the inductive hypothesis, or $\top$ otherwise, which we derive using $\top R$.

□

Therefore, by Theorems 4.3.1 and 5.5.1, intuitionistic provability under the optimized translation is equivalent to provability in LF, and the following is a theorem.

Theorem 5.5.2 (Correctness of the optimized translation)**.** *Let* Γ *be an LF specification such that* $\vdash \Gamma$ *ctx has a derivation, and let* A *be an LF type such that* $\Gamma \vdash A :$ *Type has a derivation. Then, for any LF object* M *such that* $\Gamma \vdash M : A$ *has a derivation,* $[\![\Gamma]\!]^+ \longrightarrow [\![M : A]\!]^-$ *is derivable. Moreover, if* $[\![\Gamma]\!]^+ \longrightarrow [\![A]\!]^-(M)$ *for an arbitrary* hohh *term* M*, then it must be that* $M = \langle M'\rangle$ *for some canonical LF object such that* $\Gamma \vdash M' : A$ *has a derivation.*

$$\frac{\Gamma, y; x \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.\ B}\ \text{PI}_{\text{t-1}}$$

$$\frac{\Gamma; x \sqsubset_t A \quad \Gamma; y \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.\ B}\ \text{PI}_{\text{t-2}}$$

Figure 5.4: Extension to rigidity

Finally, there is a straightforward extension to the definition of rigidity that can enable the elimination of some additional redundancies in search: we can identify those non-rigid variables as rigid that appear rigidly in the *type* of another rigidly occurring variable (in fact, such a scenario is quite rare, but it is useful to understand that rigidity as defined above captures most "intuitive" redundancies, leaving out only those occurrences that yield sub-goals that correspond to actual computation). Consider the case when some variable x is rigid in $\Pi x{:}B.\ A$. In the arguments above, we conclude that, when $A[t/x]$ is a valid type, then t must have the correct type, that is that roughly speaking $t : B$. Now suppose that instead we have $\Pi x_1{:}B_1.\ \Pi x_2{:}B_2.\ A$, with x_2 rigid therein, and similarly that $A[\overrightarrow{t/x}]$ is a valid type. When x_1 appears rigidly in B_2, the type of x_2, we can say that x_1 also occurs rigidly in the type as a whole. This is formalized, as shown in Figure 5.4, by the redefinition of the PI$_{\text{t}}$ rule as two new rules: PI$_{\text{t-1}}$, which is just PI$_{\text{t}}$ as before, and PI$_{\text{t-2}}$, which embodies the notion described above.

Instead of modifying the proof of Lemma 5.1.3, we extend the correctness argument of Theorem 5.5.1 to accommodate this modification by considering the *backchain* case of the soundness direction; the following sketch should be sufficient to express the nature of this proof. We return to the inner induction on the number of proof obligations corresponding to LF judgments of the form $t_i : B_i[\overrightarrow{t/x}]$ used previously to treat this case. We must consider, along with the original possibilities that a term t_i corresponds to a variable x_i that either was or was not rigidly occurring in the type $A[\overrightarrow{t/x}]$ for which an inhabitant is being sought, a third possibility: that x_i occurs rigidly in the type of some rigidly occurring x_j, where $i < j$. Clearly if x_j occurs rigidly we have derivations of both $\Gamma \vdash t_j : B_j[\overrightarrow{t/x}]$ and $\Gamma \vdash B_j[\overrightarrow{t/x}] :$ by Lemma 5.1.3. Given that we also have a derivation of $\Gamma \vdash \Pi\overrightarrow{x{:}B}.B_j : Type$ where $\overrightarrow{B}$ has $B_1 \ldots\ B_{j-1}$, we can reapply the lemma to conclude that $\Gamma \vdash t_i : B_i[\overrightarrow{t/x}]$, with the relevant additional facts about the validity of t_i as an encoding of an LF object t'_i.

Therefore we can indeed extend our definition of rigidity as described above to provide a slightly improved translation, in the sense that more redundancies are eliminated, without sacrificing correctness. Note however that we do not include this slight extension in further discussion or analysis.

5.6 An extended example, revisited

As in Section 4.4 we can now confidently claim that our translated specification is correct. Recall that we are still attempting to show that the *stlc* term $(\lambda x{:}unit.x)$ *void* evaluates to *void*:

```
?- hastype M (eval unit
                  (app unit unit
                    (abs unit unit (x\ x)) void)
                  void).
M = eval-app unit unit (abs unit unit (W1\ W1)) void (W1\ W1)
      void void eval-void eval-void (eval-abs unit unit (W1\ W1))
```

In the same way as before we can conclude, by Theorem 5.5.2 and by Proposition 2.5.1, that $(\lambda x{:}unit.x)$ *void* does indeed evaluate to *void* under a call-by-value evaluation strategy. And we are once again faced with the same issues associated with proving correctness in a setting where there are meta-variables in the type for which an inhabitant is being sought. These issues should be understood in the same way as they were in Section 4.4, and addressed in the same fashion.

Chapter 6

An Implementation and its Experimental Evaluation

We have claimed two properties for our translation: that it produces an *hohh* program which corresponds closely to the original LF specification, and that this program provides an effective means for executing the specification. Evidence for the first claim is provided by the translation of the specification encoding the *stlc* presented in Figure 5.3, and analyzed in detail in Section 5.4. As described, the translation is both transparently related to the original LF specification (that is, the original may be "read off" of the translated program immediately), and corresponds closely to what a programmer might write in λProlog directly.

To test the second claim, we have carried out several performance comparisons between the Twelf implementation of logic programming search for LF, and an implementation obtained by using the translation into *hohh* programs with the Teyjus system. We first describe our implementation, and then detail benchmarks designed to demonstrate the second claim, and analyze their results.

6.1 The Parinati implementation

We have implemented the various translations described here in Parinati. Given an LF specification written in the concrete syntax of Twelf, Parinati generates a λProlog program that can be compiled and run using the Teyjus system. All translations described in this thesis, along with a few "intermediate" translations that encompass only parts of the optimized translation, can be applied to generate this code. In addition, certain optimizations (for instance, the indexing extension) can be applied during translation, in any combination. Parinati is written in Objective Caml [18], a multi-paradigm functional programming language. The implementation has been released under the GNU General Public License version 3, and its source, along with several examples and tools, is available for download [19].

As described in Section 2.4, Twelf allows for various extensions to LF proper. These include an *implicit* syntax mode, wherein capitalized identifiers are assumed to be Π-quantified at the outermost level, and where certain arguments to constructors may be omitted when doing so is not ambiguous. Parinati does not handle implicit syntax; Twelf itself can be used to generate a fully explicit specification from an implicit one. Furthermore Parinati assumes the correctness of the given specification, and does only cursory type-checking. Again it is possible to employ Twelf in the process of these static analyses.

A typical Parinati interaction proceeds as follows. First, a given Twelf specification is statically analyzed for correctness using the Twelf system; thereafter the specification and with any types for which inhabitants are to be sought are assumed to be correct. Next, Parinati is run on the correct specification, generating a λProlog module and signature based on the selected translation. For instance, in the interaction below the specification `lists.lf` is translated to λProlog using the optimized translation described in Section 5.3, along with the indexing extension.

```
$ parinati --input lists.lf --translation extended --opt-index
```

This generates the module `lists.mod`, and the corresponding signature `lists.sig`. These two pieces, which are always correct so long as the original specification was correct, are then given to the Teyjus compiler. The compiler then generates a bytecode file that may be linked and then run on the Teyjus simulator.

```
$ tjcc lists
$ tjlink lists
$ tjsim lists
```

At this point the user may interact with the LF specification by querying the loaded module from within the simulator; these queries should take the form of translations of LF typing judgments. These can also be automatically generated from LF judgments by Parinati. For example, assuming that `lists.lf` contains a suitable definition for `append`, we might run the following query:

```
?- append ProofTerm nil nil A.
A = nil
ProofTerm = append_nil nil.
```

6.2 Experiments and results

We have tested our implementation using several benchmarks; the specifics of the benchmarking process are as follows:

- Twelf was benchmarked using Twelf version 1.5R3 using the SML top-level `twelf-sml`. The top-level was run under SML of New Jersey version 110.71. Only default optimizations for Twelf were used, and tabling was not enabled. In addition, only time spent actually solving goals was recorded; time spent parsing, type checking, *etc.* was not included.

- The translation-based implementation proceeded by first translating an LF specification to a λProlog program using `parinati` and then compiling the program using the Teyjus compiler `tjcc`, again with default optimizations. Finally, the resulting bytecode was run using the Teyjus simulator. Version 2.0-b2 of the Teyjus system was used. As Teyjus does not provide as fine-grained access to timing information within the simulator, timing was performed by running the simulator with a bytecode program without proving any goals to determine the base cost of invoking the software. Then the simulator was run with same bytecode program along with the relevant goal; the final time was computed by subtracting the base cost from this time.

Each benchmark was run 5 times, with the best time used. A system with a 3.00GHz Intel processor and one gigabyte of physical memory was used to run these benchmarks.

We present results here over programs that have a few different characteristics, intended to provide a reasonably complete overview and test of the kinds of programs regularly written in LF (and in particular those written for and run on the Twelf implementation of LF):

- First, as we are interested in logic programming in LF, the traditional logic program for naively reversing a constant list n times (*rev*) is included.

- The encoding of evaluators for various languages is a common usage of LF. Therefore the *miniml* benchmark consists of an encoding of Mini-ML, along with an encoding of addition. The benchmark tests adding n to 10.

- The *miniml* specification does not make essential use of dependent types. The *typed miniml* benchmark, which consists of an evaluator for Mini-ML in which terms are indexed by their type, uses dependent types to ensure that terms are well-formed. The Mini-ML program that was run is a typed version of the encoding of addition.

- An implementation of a meta-interpreter for intuitionistic non-commutative linear logic, INCLL, has been proposed as a test program by [20]. This benchmark, *perm*, tests list permutation encoded in INCLL and run using the meta-interpreter for lists of varied lengths.

- The last benchmark, *num*, involves rewriting arithmetic expressions into an equivalent normal form. This example again makes essential use of dependent types by associating with each equivalence of two such terms a proof of their equivalence. The benchmark tests rewriting expressions of size n.

Figure 6.2 presents data comparing the simplified translation, the translation with redundant typing judgments removed, the fully optimized translation, and the optimized translation with indexing extension against the standard of Twelf with default optimizations on these benchmarks — that is, all data has been normalized, so that Twelf always receives a score of 1.0, and the others are scored relative to this. In the data presented, overflow indicates a heap overflow in the Teyjus simulator, and ∞ means that the program ran more than 1000 times longer than Twelf. The most optimized translation, with or without the indexing extension, leads to better performance in most cases, often significantly so. On the other hand, the simplified translation yields a program that is generally slower than Twelf. In particular, performance tends to deteriorate with larger problems sizes, in keeping with the difficulty that we noted with this translation in Section 4.5. However, the simplified translation is still comparable to Twelf on some benchmarks (such as *rev(n)*); this is likely due to the efficiency of the virtual machine underlying the Teyjus implementation rather than to properties of the translation.

We make several observations about specific benchmarks. First, in the case of *rev(n)* we find that in all cases the translation-based implementation performs far better than Twelf; as mentioned we ascribe this primarily to the efficiency of the underlying virtual machine. The same is true in the case of *append(n)*. Next, note that in the case of *perm(n)*, the most optimized translation's performance falls off more quickly than the performance of Twelf as the problem size increases. The exact cause of this behavior has still to be pinpointed: we suspect that it may be due at least in part to a difference in how and when Twelf and Teyjus perform the occurs check, as system employs different but partially overlapping optimizations in this respect.

In the cases of *miniml* and *typed miniml* we can see the former more efficient than the latter when compared with Twelf. This result is very likely due to the imperfect nature of our redundancy elimination technique, in so far as increase redundancy elimination leads to improved performance. In particular, in the case of *miniml* there are far fewer typing constraints for individual rules for evaluation, and therefore fewer possibilities for

Example	Twelf	Simplified	Optimized	Typed Optimized	Indexing
rev(10)	1.0	0.40	0.14	0.07	0.08
rev(20)	1.0	0.57	0.19	0.12	0.11
rev(30)	1.0	0.63	0.20	0.14	0.11
rev(40)	1.0	0.41	0.13	0.10	0.07
rev(50)	1.0	0.46	0.15	0.10	0.08
miniml(50)	1.0	0.74	0.25	0.18	0.08
miniml(100)	1.0	1.25	0.44	0.30	0.17
miniml(150)	1.0	1.75	0.56	0.41	0.25
miniml(200)	1.0	2.89	0.83	0.62	0.41
typed miniml(50)	1.0	2.27	1.07	0.57	0.48
typed miniml(100)	1.0	2.22	0.76	0.49	0.38
typed miniml(150)	1.0	3.49	1.44	0.67	0.55
typed miniml(200)	1.0	3.70	0.92	0.67	0.55
perm(10)	1.0	overflow	3.13	0.94	0.72
perm(20)	1.0	overflow	1.75	0.78	0.44
perm(30)	1.0	overflow	3.05	1.52	0.81
perm(40)	1.0	overflow	3.95	2.15	1.14
perm(50)	1.0	overflow	5.05	2.88	1.59
num(64)	1.0	158.19	0.25	0.23	0.21
num(128)	1.0	∞	0.10	0.10	0.07
num(256)	1.0	∞	0.15	0.14	0.13
num(512)	1.0	∞	0.003	0.003	0.003

Figure 6.1: Experimental data

redundancy, when compared to *typed miniml*. Thus we can expect that, while in the first case we might manage to eliminate most or all redundancies, in the second case we are likely to eliminate fewer. To take a specific example, the evaluation rule for let-bound variables in *miniml* is as follows:

```
letv : exp -> (exp -> exp) -> exp.
```

Under the optimized translation we have the following program, presented in idiomatic λProlog:

```
exp (letv E F) :-
  exp E,
  pi x\ exp x => exp (F x).
```

Notice that we seemingly have no redundancies: checking that a term has type `exp` must involve checking its sub-terms for correctness. Contrast this with the definition of the same rule in LF:

```
letv : {T1 : ty}{T2 : ty} exp T1 -> (exp T1 -> exp T2) -> exp T2.
```

And its translation in *typed miniml*:

```
exp (letv T1 T2 E F) :-
  ty T1,
  exp E T1,
  pi x\ exp x T1 => exp (F x) T2.
```

Here we are unable to elide the typing subgoal associated with `T1`, but we are able to elide the one associated with `T2`. As there are more such subgoals involved in the translations of constructors in the *typed miniml* example, so too is it likely that some of these subgoals will be "missed" by the redundancy elimination optimization, leading to possibly unnecessary re-derivations during proof search.

A case of particular interest is that of *num(n)*. On a problem of size 512, the most optimized translation takes .003 of the time of Twelf. The precipitous falling off in performance for Twelf here seems to be due to excessive memory consumption. In fact, for all the benchmarks, very large examples lead to this kind of behavior. The source of this problem is perhaps the fact that Twelf is implemented using the SML language: it has been argued that realizing a logic programming language in a functional programming setting can lead to poor memory reclamation and eventually to shortage of space [21].

Finally, in only one case is the translation slower than Twelf; this is the *perm(n)* case. We have yet to pinpoint the reason for this—the program is large and difficult to analyze in detail—but we suspect that the linear head optimization which this benchmark has been used to analyze, that delays expensive unification computation till after simpler checks have been made, may have something to do with this. The fact that ordering arguments to take advantage of term indexing causes significant improvement gives credence to this observation. Furthermore, this benchmark makes extensive use of dependent types not only for computation but also for representation, as in *typed miniml*; the same observations made about that benchmark apply here.

In conclusion, the evidence described here suggests that our implementation of logic programming search based on translation to λProlog can be practical and efficient. It is worth emphasizing that, in developing this implementation, we have refrained from introducing additional (possibly *ad hoc*) optimizations, though they might seem straightforward, apart from those already described. This leaves an implementation that is quite easy for a programmer to understand, both in the sense of how translation proceeds, and how the resulting program will behave in terms of performance and other runtime characteristics, which allows the programmer to view LF as a meta-programming language for λProlog. Even in its early form, this implementation can handily out-perform the Twelf implementation in many cases.

Chapter 7

Conclusion

We have considered in this paper a translation of LF specifications into logic programs in the *hohh* language. We have illustrated the efficiency and transparency of the translation through various examples that are representative of the kinds of specifications regularly written in LF. Our translation is significantly more efficient on most problems than the widespread existing implementation of LF, Twelf. What is more, our translation allows programmers to execute programs on problems far larger than those possible using the existing implementation, which has implications for the practical application of LF in such areas as proof-carrying code [3] and certified compiler implementations: these applications can involve intensive computations that often cannot be carried out in Twelf in a reasonable amount of time. An essential part of our work is the recognition of certain situations in which type checking during search is redundant, and hence can be avoided. We identify these situations as being rooted in fundamental properties of LF derivations and expressions.

Our optimized translation produces a program that has a close correspondence to the original LF specification; it is a highly transparent translation, which allows the programmer to treat LF as a meta-programming language for writing λProlog programs that make use of the richness of dependent types. For instance, we have demonstrated that fundamental idioms present in the original LF specification, such as the use of HOAS to represent binding in an object logic, are maintained by the translation. Not only this, the translation that we have developed generates *hohh* programs such that *derivations* in *hohh* exhibit a strong structural correspondence to the LF derivations to which they are equivalent. Finally, we have proved that this translation is correct not just in the sense of generating equivalent *hohh* programs, as previous work has demonstrated; we have also shown that, in the context of logic programming, it generates only *hohh* terms that do correspond to LF objects of the correct type.

We now discuss extant work related to both our technique for optimizing search, and methods for implementing logic programming search for LF. And we describe various ways of building on the work presented in this thesis going forward.

7.1 Related work

There does exist some work in the context of eliminating various sorts of redundancies in LF terms and derivations. Reed [22] approaches this problem from a different perspective, and with a different goal: that of reducing the *size* of proof-terms yielded during logic programming search, but not reducing the cost of actually discovering these terms, motivated by the fact that in some applications proof-terms must be transmitted or manipulated. He does so by developing a technique for identifying redundancies in terms, through a notion of *strictness* that is similar to rigidity, that he uses to identify sub-terms of LF objects that can be reconstructed, either from the types of nearby sub-terms, or from the type of an object itself. He describes two modes for omitting sub-terms, *synthesis* based omission and *inheritance* based omission, and uses strictness to determine which kind of omission, if any, is possible. In omission by inheritance knowledge of the type that a term has is used to elide (and later reconstruct) sub-terms of the term; in omission by synthesis the types of nearby sub-terms are used to elide and eventually reconstruct a given sub-term, when the sub-term being omitted appears (in a sufficient manner) in said type.

There are clearly differences in motivation between the approach described in this thesis and that of Reed: whereas we are concerned with optimizing search, and eliminating redundant type checking, his work focuses on optimizing an LF object (that is, a proof term) for size by eliminating redundant parts of the object itself, and without particular concern for how such a term is discovered. Nevertheless, the essential redundancies that are identified in this process are similar. First, those terms that can be omitted by inheritance are exactly those for which type checking can be eliminated under our system, as they are just those sub-terms that also appear in the type in a sufficient way (that is, the definitions of strictness and rigidity closely coincide in this case). Second, those terms that can be omitted by synthesis are similar to those that are identified as rigid by the extension to rigidity described in Section 5.5. In his system, however, Reed does not face the same issues described in Section 5.5 for further extending rigidity: whereas he is working with LF expressions that are already known to be well-formed, we have been concerned all along with ensuring that all *hohh* objects discovered or used during search actually correspond to the encodings of LF objects. Furthermore, as Reed does not provide the necessary proofs for directly making use of his system, it is not clear whether the technical problems we have faced are surmountable. Still, a better understanding of how we might apply his system to the problem of logic programming search might lead both to an improvement of our translation, and to the ability to shorten LF proof terms that are needed in applications such as that of proof-carrying code.

7.2 Future work

We concluded the previous section by indicating a possible application of our results directly to LF. In this section we discuss another kind of avenue for further work, in particular directions yet to be explored in animating LF specifications. The specific work undertaken here can be extended in a few different ways. First, we can consider modifying the definition of rigidity so as to improve performance, based on the understanding that identifying more variables as rigid implies type checking fewer terms at runtime. Possible directions include identifying a definition of rigidity that captures omission by synthesis, as described above. In addition, Theorems 4.3.1 and 5.5.1 could be extended to capture the correctness of the translations in the presence of "open" terms, *i.e.* those containing meta-variables.

From an implementation perspective, another possible optimization is to avoid constructing an LF object explicitly when the task has been identified as that of only determining whether a type has an inhabitant. This is actually related to the more common forms of logic programming embodied by languages like Prolog and λProlog. In general, one is interested in whether a judgment has a derivation, and usually more importantly, for what arguments. In this setting, one is not concerned with the specifics of the derivation of the judgment at all, and indeed such systems tend not to expose such information. The savings that might be attained, when one is willing to forgo a proof term, can be immense, as such terms can become exceedingly large on even moderate problem sizes. Our initial experiments in this direction indicate in some cases a ten-fold performance improvement over the optimized translation.

Furthermore, there are a number of optimizations to logic programming search in LF in the context of its Twelf implementation that could be applied to our approach as well. Some examples of these include *tabling* for memoizing the results of searches so they need not be run again, and more radical *indexing* based on the structure of an LF specification (instead of simply allowing Teyjus to perform indexing based on the structure of the translated logic program, as the current implementation does).

One such implementation-level optimization deserves further discussion. The minimization of occurs checking is of critical importance when implementing unification-based systems. In the case of Teyjus the system makes use of what is called the *first occurrence* optimization for variables. This technique takes advantage of the fact that when unifying the first occurrence of a variable with a term for the first time it is not necessary to check whether the term contains the variable, as clearly it cannot by definition. While this is a well-understood technique when it comes to WAM-based implementations of logic programming languages, it is not employed in the interpretation-based Twelf. Pientka and Pfenning [20] remark that "it is not clear how to generalize" this optimization in the dependently typed setting. They then present a technique for minimizing occurs checks using *head linearization*, which essentially results in unifications requiring occurs checks being performed after an "assignment" check that compares the rigid structure of terms. This can lead to faster failures, as mismatches in rigid structure can be detected before any occurs checking takes place. They find that these optimizations prove critical to the efficiency of the Twelf implementation, when compared to a system in which all occurs checks are performed. However, we have experimented with automatically applying head linearization to all clauses of the generated λProlog programs, and have not found general performance improvements; instead we have found that in certain cases, with carefully "hand-applied" linearizations, performance can be increased, but that in others performance suffers. We believe that this could be because the first occurrence optimization is also in effect, even though neither technique subsumes the other; a more rigorous analysis of *when* linearization can improve performance in our setting might lead to additional improvements in occurs check elimination.

While we have focused here on realizing LF through a translation to λProlog. A different approach is that of compiling LF specifications directly to bytecode for the virtual machine underlying the Teyjus system. Pientka notably describes a technique for optimizing unification [23] that could perhaps be applied if only the LF specification were compiled directly to the Teyjus virtual machine's bytecode. Furthermore, direct compilation could allow us to regain opportunities for those improvements that might be lost by translating first to λProlog and then relying on its implementation that is not specially optimized to treat LF-specific programs. However, this would clearly eliminate the possibility of treating LF as a meta-programming language for writing complex λProlog programs, as the requirement of transparency could not be fulfilled.

Finally, a more ambitious line of development concerns meta-reasoning over specifications. Existing tools like Abella [24] and Tac [25], developed for reasoning over a subset of *hohh*, can be used to reason about LF programs via the translation; here the transparency of the translation becoming essential. Anecdotal evidence suggests that this transparency is not only enabling, it is also elucidating: the generated *hohh* program is easier to reason about because it highlights those types that could have logical importance, and elides those that do not.

Bibliography

[1] 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 *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer-Verlag, 2005.

[2] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[3] George C. Necula. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97*, pages 106–119, Paris, France, 1997. ACM Press.

[4] Michael William Whalen. *Trustworthy translation for the requirements state machine language without events*. PhD thesis, Minneapolis, MN, USA, 2005. Adviser-Heimdahl, Mats Per.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[6] William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.

[7] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. Logic Programming*, pages 379–388, 1994.

[8] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, 2007.

[9] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conference on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[10] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In A. Abel and C. Urban, editors, *LFMTP 2008: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 75–89, 2008.

[12] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[13] Alonzo Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.

[14] Dale 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.

[15] Andrew Gacek, Steven Holte, Gopalan Nadathur, Xiaochu Qi, and Zachary Snow. The Teyjus system – version 2, March 2008. Available from `http://teyjus.cs.umn.edu/`.

[16] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, October 1983.

[17] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.

[18] The Objective Caml language. Available from `http://caml.inria.fr/ocaml/index.en.html`.

[19] Zachary Snow. Parinati. `http://www.cs.umn.edu/~snow/parinati`, 2010.

[20] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer-Verlag, 2003.

[21] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of lambdaprolog: Prolog/mali. In *ILPS Workshop: Implementation Techniques for Logic Programming Languages*, 1994.

[22] Jason Reed. Redundancy elimination for LF. *Electron. Notes Theor. Comput. Sci.*, 199:89–106, 2008.

[23] Brigitte Pientka. Eliminating redundancy in higher-order unification: A lightweight approach. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2006.

[24] Andrew Gacek. Abella, 2009. Available from `http://abella.cs.umn.edu/`.

[25] David Baelde, Zachary Snow, and Alexandre Viel. Tac: A generic and adaptable interactive theorem prover, 2010. Available from http://slimmer.gforge.inria.fr/tac/.