Translating Specifications in a Dependently Typed Lambda Calculus into a Predicate Logic Form

Mary Southern  
Gopalan Nadathur

Computer Science and Engineering  
University of Minnesota  
Minneapolis, MN 55455

Abstract

Dependently typed lambda calculi such as the Edinburgh Logical Framework (LF) are a popular means for encoding rule-based specifications concerning formal syntactic objects. In these frameworks, relations over terms representing formal objects are naturally captured by making use of the dependent structure of types. We consider here the meaning-preserving translation of specifications written in this style into a predicate logic over simply typed \( \lambda \)-terms. Such a translation can provide the basis for efficient implementation and sophisticated capabilities for reasoning about specifications. We start with a previously described translation of LF specifications to formulas in the logic of higher-order hereditary Harrop (hohh) formulas. We show how this translation can be improved by recognizing and eliminating redundant type checking information contained in it. This benefits both the clarity of translated formulas, and reduces the effort which must be spent on type checking during execution. To allow this translation to be used to execute LF specifications, we describe an inverse transformation from hohh-terms to LF expressions; thus computations can be carried out using the translated form and the results can then be exported back into LF. Execution based on LF specifications may also involve some forms of type reconstruction. We discuss the possibility of supporting such a capability using the translation under some reasonable restrictions on the structure of specifications.

1. Introduction

The Edinburgh Logical Framework (LF) has proven to be a useful device for specifying formal systems such as logics and programming languages. At its core, LF is a dependently typed lambda calculus. By exploiting the abstraction operator that is part of the syntax of LF, it is possible to succinctly encode formal objects whose structure embodies binding notions. Because types can be indexed by terms, we can use them to express relations between the formal objects encoded in terms. If we view such types as formulas, terms that have a given type can be interpreted as proofs of the formula that type represents. Thus, LF specifications can be given a logic programming interpretation using a notion of proof search that corresponds to determining inhabitation of given types. The Twelf system is an implementation of LF that is based on such an interpretation.

An alternative approach to specifying formal systems is to use a predicate logic. Objects treated by the formal systems can be represented by the terms of this logic and relations between them can be expressed through predicates over these terms. If the terms include a notion of abstraction (e.g., if they encompass simply typed lambda terms) they provide a convenient means for representing binding notions. While an unrestricted predicate logic would be capable of describing relations adequately, it is preferable to limit the permitted formulas so that the desired interpretation of rule based specifications can be modeled via a constrained proof search behavior. The logic of higher-order hereditary Harrop formulas (hohh) has been designed with these ideas in mind and many experiments have shown this logic to be a useful specification device (e.g., see [5]). This logic has also been given a computational interpretation in the language \( \lambda \)Prolog [6]. Moreover, an efficient implementation of \( \lambda \)Prolog has been developed in the Teyjus system [7].

There are obvious similarities between the two different approaches to specification, making it interesting to explore the connections between them more formally. In early work, Felty et. al. showed that LF derivations could be encoded in hohh derivations by describing a translation from the former to the latter [2]. This translation demonstrated the expressive power of hohh, but was not directly usable in relating proof search behavior. To rectify this situation, Snow et. al. showed how to translate LF specifications into hohh formulas in such a way that the process of constructing a derivation could be related [3]. This work provided the basis for an alternative implementation of Twelf. The translation also has the potential to be useful in bringing the power of the Abella prover [3] to bear on reasoning about Twelf specifications.

This paper continues the work described in [3]. There are four specific contributions it makes in this setting:

1. An important part of the translation is the recognition and elimination of redundant typing information in specifications. We describe an improvement to the criterion presented in [3] for this purpose.

2. In contrast to [3], we show how to modularize the proof of redundancy of typing information, establishing a result concerning LF first and then lifting this result to the translation. This enables us to present results that also apply directly to LF.

3. If we are to use the translation as a means for implementing proof search in Twelf, we need also a way to return to Twelf expressions after completing execution in \( \lambda \)Prolog. We describe such an inverse transformation.

4. Logic programming in Twelf includes a process of type reconstruction. We begin an analysis of the translation towards understanding whether type reconstruction on the translated expression will agree with Twelf’s behavior. This analysis is incomplete, but we believe the approach to be sound and the remaining work to be mainly that of elaborating the details.

The next two sections describe LF and the hohh logic respectively and discuss their computational interpretations. Section 4 then presents a simple translation of LF specifications into hohh ones. The following section takes up the task of improving
this translation. In particular, it characterizes certain bound variable occurrences in types using a notion called \textit{strictness} and uses this characterization to identify redundancy in typing. We are then able to eliminate such redundancy in translation. Section 2 describes the inverse translation from \texttt{holh} terms found via proof search to LF expressions in the originating context for the translation. Section 3 contains a discussion on the treatment of type reconstruction in Twelf proof search. We end the paper with a discussion of future directions to this work in Section 8.

2. Logical Framework

This section introduces dependently typed \(\lambda\)-calculi as a means for specifying formal systems. A unique aspect of these calculi is that they let us define types which are indexed by terms. This can be a more intuitive method of encoding relationships between terms and types within a specification than using predicates, such as in Prolog. To take a computational view, we interpret types as formulas, and proving such formulas then reduces to checking that a certain type is inhabited. The particularly dependently typed \(\lambda\)-calculus we shall use in this paper is called the Edinburgh Logical Framework or LF. We describe this calculus below, then exhibit its use in specifying relations and finally explain how it can be given an executable interpretation.

2.1 The Edinburgh Logical Framework

There are three categories of LF expressions: kinds, type families which are classified by kinds, and objects which are classified by types. Below, \(x\) denotes an object variable, \(c\) an object constant, and \(a\) a type constant. Letting \(K\) range over kinds, \(A\) and \(B\) over types, and \(M\) and \(N\) over objects, the syntax of these terms are as follows:

\[
\begin{align*}
K & ::= \text{Type} \mid \Pi x : A. K \\
A & ::= a \mid \Pi x : A. B \mid A M \\
M & ::= c \mid x \mid \lambda x : A. M \mid M N
\end{align*}
\]

Both \(\Pi\) and \(\lambda\) are binders which assign a type to a variable over the term. The shorthand \(A \rightarrow P\) is used for \(\Pi x : A. P\) when \(x\) does not appear free in \(P\). Terms differing only in bound variable names are identified. We use \(U\) and \(V\) below to stand ambiguously for types and object expressions. We write \(U[M_1/x_1, \ldots, M_n/x_n]\) to denote the capture avoiding substitution of \(M_1, \ldots, M_n\) for free occurrences of \(x_1, \ldots, x_n\) in \(U\).

LF type family and object expressions are formed starting from a signature \(\Sigma\) that identifies constants together with their kinds or types. In addition, in determining whether or not an expression is well-formed, we need to consider contexts, denoted by \(\Gamma\), that assign types to variables. The syntax for signatures and contexts is as follows:

\[
\begin{align*}
\Sigma & ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
\Gamma & ::= \cdot \mid \Gamma, x : A
\end{align*}
\]

In what follows, the signature (which is user-defined) will not change over the course of time. Given this, for simplicity, we will leave this signature implicit in our discussions.

Not all the LF expressions identified by the syntax rules above are considered to be well-formed. The following five forms of judgments are relevant to deciding the ones that are:

\[
\begin{align*}
\Sigma \text{ sig} & \vdash \Gamma \text{ ctx} \\
\Gamma & \vdash K \text{ kind} \\
\Gamma & \vdash A : K \\
\Gamma & \vdash M : A
\end{align*}
\]

The judgments on the first line assert, respectively, that \(\Sigma\) is a valid signature and that \(\Gamma\) is a valid context, implicitly in \(\Sigma\). The judgments on the second line assert that \(K\) is a valid kind in the context \(\Gamma\). \(A\) is a valid type of (valid) kind \(K\) in \(\Gamma\), and \(M\) is a valid object of (valid) type \(A\) in \(\Gamma\); all these judgments also verify that the context \(\Gamma\) and the implicit signature \(\Sigma\) are both valid. In stating the rules for deriving these judgments, we shall make use of an equality notion for expressions that is based on \(\beta\)-conversion, i.e., the reflexive and transitive closure of a relation that equates two expressions that differ only in that a subexpression of the form \((\lambda x : A. M) N\) in one is replaced by \(M[N/x]\) in the other. We shall write \(U^\beta\) for the \(\beta\)-normal form of an expression, i.e., for an expression that is equal to \(U\) and that does not contain any subexpressions of the form \((\lambda x : A. M) N\). The rules for deriving the five different LF judgments are presented in Figure 1.

Notice that we allow for the derivation of judgments of the form \(\Gamma \vdash A : K \land \Gamma \vdash M : B\) only when \(K\) and \(B\) are in \(\beta\)-normal form. We also observe that such forms are not guaranteed to exist for all LF expressions. However, they do exist for well-formed LF expressions, a property that is ensured to hold for each relevant LF expression by the premises of every rule whose conclusion requires the \(\beta\)-normal form of that expression.

The notion of equality that we use for LF terms also includes \(\eta\)-conversion, i.e., the congruence generated by the relation that equates \(\lambda x : A. (M x)\) and \(M\) if \(x\) does not appear free in \(M\). Observe that \(\beta\) and \(\eta\) forms for the different categories of expressions have the following structure:

\[
\begin{align*}
\text{Kind} & \quad \Pi x_1 : A_1, \ldots, \Pi x_n : A_n. \text{Type} \\
\text{Type} & \quad \Pi y_1 : B_1, \ldots, \Pi y_m : B_m. a_1 \ldots a_n. M_1 \ldots M_n \\
\text{Object} & \quad \lambda x_1 : A_1, \ldots, \lambda x_n : A_n. u M_1 \ldots M_n
\end{align*}
\]

where \(u\) is an object constant or variable and where the subterms and subtypes appearing in the expression recursively have the same form. We refer to the the part denoted by a \(M_1 \ldots M_n\) in a type expression in such a form as its \textit{target} type and to \(B_1, \ldots, B_m\) as its \textit{argument} types. Let \(w\) be a variable or constant which appears in the well-formed term \(U\) and let the number of \(\Pi\)s that appear in the prefix of its type or kind be \(n\). We say \(w\) is \textit{fully applied} if every occurrence of \(w\) in \(U\) has the form \(w M_1 \ldots M_n\). A type of the form \(a_1 \ldots a_n. M_1 \ldots M_n\) where \(a\) is fully applied is a \textit{base type}. We also say that \(U\) is \textit{canonical} if it is in normal form and every occurrence of a variable or constant in it is fully applied. It is a known fact that every well-formed LF expression is equal to one in canonical form by virtue of \(\beta\eta\)-conversion.

2.2 Specifying Relations in LF

LF can be used to formalize different kinds of rule based systems by describing a signature corresponding to the system, as we now illustrate. In presenting particular signatures, we will use a more machine-oriented syntax for LF expressions: we write \(\{x : U\} V\) for \(\Pi x : U. V\) and \(\{x : A\} M\) for \(\lambda x : A. M\).

The first example we consider is that of the natural number machine. In formulating this system we must, first of all, provide a representation for the numbers. This is easy to do: we pick a type corresponding to these numbers and then provide an encoding for zero and the successor constructor. The first three items in the signature shown in Figure 2 suffice for this purpose. The next thing to do is to specify operations on natural numbers. In LF we think of doing this through relations: thus, addition would be specified as a relation between three numbers. To describe relations we use \textit{dependent} types. For example, the addition relation might be encoded as a type constant that takes three natural number \textit{objects} as arguments. The real interest is in determining when such a relation holds. In rule based specifications this is typically done through inference rules. Thus, using the LF notation that we have just described, addition might be defined by the rules:

\[
\begin{align*}
\text{plus } z X X & \\
\text{plus } N M L & \\
\text{plus } (s N) M (s L)
\end{align*}
\]
in which tokens represented by uppercase letters constitute schema variables. In an LF specification, such rules correspond to object constants whose target type is the representation of the rule’s conclusion and whose argument types are the types of the schema variables and the representations of the premises. As a concrete example, the object constants \( \text{plusZ} \) and \( \text{plusS} \) defined in Figure 2 represent the two addition rules shown. The question of whether a relation denoted by a type holds now becomes that of whether a given object expression \( M \) has the type \( B \). If so, then \( M \) is derivable. We can match this type with the target type of \( \text{append} \) and we are then left with finding a term \( T \) such that \( \text{append} L M N \) and \( \text{append} (\text{cons} \ x \ l \ m \ n) \) is derivable. We can match this type with the target type of \( \text{appCons} \) and we are then left with finding a term \( N \) such that \( \vdash \ N : \text{append} \) is derivable. Notice that this step also results in \( M \) being instantiated to \( \text{appCons} \) in the new goal of \( \text{append} \). In the more general case, \( A \) may not be a base type, i.e. it may actually have the structure \( \{x_1 : A_1, \ldots, x_m : A_m\}B \) where \( B \) is a base type. In this case, we first transform the task to trying to show that the object expression \( M x_1 \ldots x_m \) has type \( B \) where we treat \( x_1, \ldots, x_m \) as new constants of type \( A_1, \ldots, A_m \), respectively, that are dynamically added to the signature.

For a concrete example of this behavior, let our signature be \( \{ \text{nat}, \text{list}, \text{cons} \} \) and \( \text{append} \) and \( \text{appCons} \) as new constants of type \( \text{nat} \to \text{list} \to \text{list} \). The rules for proving this relation are the following

\[
\frac{\vdash \text{append} \cdot \text{nil} \ L \ L}{\vdash \text{append} \ . \text{nil} \ L \ L}
\]

Following the structure described earlier, the object constants \( \text{appNil} \) and \( \text{appCons} \) shown in Figure 2 represent these rules.

### 2.3 Logic Programming

The Twelf system gives LF specifications a logic programming interpretation. Computation is initiated in Twelf by presenting it with a type. Such a type, as we have explained earlier, corresponds to a formula and the task is to find a proof for it or, more precisely, to find an inhabitant for the provided type.

The search problem is actually better viewed as that of checking if a given object expression \( M \) has a given type \( A \); this formulation subsumes the case where only the type is given because we allow \( M \) to contain variables that may become instantiated as the search progresses. In the simple case \( A \) is a base type. Here, computation proceeds by looking for an object declaration

\[
c : \{x_1 : B_1, \ldots, x_n : B_n\}A'
\]

in the signature at hand and checking if there are object expressions \( M_1, \ldots, M_n \) such that \( A'[M_1/x_1, \ldots, M_n/x_n] = A \). If this is the case and if it is also the case that \( M \) and \( c \) \( M_1 \ldots M_n \) can be unified, then the task reduces, recursively, to checking if \( M_i \) has the type \( B_i \) for \( 1 \leq i \leq n \). In this model of computation, the types associated with object constants in a signature are often referred to as \textit{clauses} and the process of picking an object declaration and trying to use it to solve the inhabitation question is referred to as \textit{backchaining} on a clause.
We have, at this point determined that this object expression inhabits the type \( \text{append} \ (\text{cons} \ z \ \text{nil}) \ (\text{cons} \ z \ \text{nil}) \).

3. Specifications in Predicate Logic

Another approach to specification uses a predicate logic, where relations are encoded as predicates rather than in types. The idea of executing the specifications then corresponds to constructing a proof for chosen formulas in the relevant logic. To yield a sensible notion of computation, the specifications must also be able to convey information about how a search for a proof should be conducted. Not all logics are suitable from this perspective. Here we describe the logic of higher-order hereditary Harrop formulas that does have an associated computational interpretation and that, in fact, is the basis for the programming language λProlog \[1\]. We present the syntax of the formulas in this logic in the first subsection below and then explain their computational interpretation. The \emph{hohh} logic will be the target for the translation of Twelf that is the focus of the rest of the paper.

3.1 Higher-order hereditary Harrop formulas

The \emph{hohh} logic is based on Church’s Simple Theory of Types \[1\]. The expressions of this logic are those of a simply typed λ-calculus. The types are constructed from the atomic type \( o \) of propositions and a finite set of other atomic types by using the function type constructor \( \to \). There are assumed to be two sets of atomic expressions, one corresponding to variables and the other to constants, in which each member is assumed to have been given a type. All typed terms can be constructed from these typed sets of constants and variables by application and \( \lambda \)-abstraction. As in LF, terms differing only in bound variable names are identified. The notion of equality between terms is further enriched by \( \beta \)- and \( \eta \)-conversion. When we orient these rules and think of them as reductions, we are assured in the simply typed setting of the existence of a unique normal form for every well-formed term under these reductions. Thus, equality between two terms becomes the same as the identity of their normal forms. For simplicity, in the remainder of this paper we will assume that all terms have been converted to normal form. We use \( t[s_1/x_1, \ldots, s_n/x_n] \) to denote the capture avoiding substitution of the terms \( s_1, \ldots, s_n \) for free occurrences of \( x_1, \ldots, x_n \) in \( t \).

Further qualifications are required to introduce logic into this setting. First, the constants mentioned above are divided into the categories of logical and non-logical constants. Next, we restrict the constants so that only the logical constants can have argument types containing the type \( o \). Finally, we limit the logical constants to the following:

- \( \top \) of type \( o \)
- \( \top \) of type \( o \to o \to o \)
- \( \bigodot \) of type \( (\tau \to o) \to o \) for each valid type \( \tau \)

\( \Pi \) denotes universal quantification, and the shorthand \( \forall x. F \) is used for \( \Pi(\lambda x. F) \).

The set of non-logical constants is typically called the signature, and as mentioned \( o \) cannot appear in the type of any argument of these constants. However, \( o \) is allowed as the target type for non-logical constants. Constants with target type \( o \) are called predicates; those with any other target type are called constructors.

For a non-logical constant \( c \) of type \( \tau_1 \to \ldots \to \tau_n \to o \) and terms \( t_1, \ldots, t_n \) of type \( \tau_1, \ldots, \tau_n \), we call the term \( c \, t_1 \ldots t_n \) of type \( o \) an \emph{atomic formula}. Using the set of logical constants, we construct sets of \( G \) and \( D \)-formulas from the set of atomic formulas. The syntax of these two sets is the following:

\[
G ::= \top \mid \{ A \mid D \supseteq G \mid \forall x. G \}
\]
\[
D ::= \{ A \mid G \supseteq D \mid \forall x. D \}
\]

where \( A \) denotes an atomic formula.

The \( D \) formulas described above are also called higher-order hereditary formulas. A specification in this setting consists of a set of such formulas. To illustrate how such specifications may be constructed in practice, let us consider the encoding of the append relation on lists of natural numbers. The first step in formalizing this relation is to describe a representation for the data objects in its domain. Towards this end, we introduce two atomic types, \( \text{nat} \) and \( \text{list} \). Our signature should then identify the obvious constructors with each of these types:

\[
\begin{align*}
&z \quad \text{of type} \quad \text{nat} \\
s \quad \text{of type} \quad \text{nat} \to \text{nat} \\
nil \quad \text{of type} \quad \text{list} \\
\text{cons} \quad \text{of type} \quad \text{nat} \to \text{list} \to \text{list}
\end{align*}
\]

As a concrete example, the list that has 0 and 1 as its elements would be represented by the term \( \text{cons} \, z \, (\text{cons} \, s \, \text{nil}) \).

The append relation will now be encoded via a \emph{predicate} constant, i.e., a non-logical constant that has \( o \) as its target type. In particular, we might use the constant \( \text{append} \) that has the type

\[
\text{list} \to \text{list} \to \text{list} \to o
\]

for this purpose. To define the relation itself, we might use the following two \( D \)-formulas:

\[
\begin{align*}
&\forall l. (\text{append} \, \text{nil} \, l) \\
&\forall x. \forall l_1. \forall l_2. (\text{append} \, l_1 \, l_2) \supset (\text{append} \, (\text{cons} \, x \, l_1) \, l_2 \, (\text{cons} \, x \, l_2))
\end{align*}
\]

These formulas, that are also often referred to as the \emph{clauses} of a specification or program, can be visualized as defining the append relation by recursion on the structure of the list that is its first argument. The first formulas treats the base case, when this list is empty. The second formula treats the recursive case; the conclusion of the implication is conditioned by the relation holding in the case where the first argument is a list of smaller size. This pattern, of using universal quantifications over atomic formulas to treat the base cases of a relation and such quantifications over formulas that have an implication structure to treat the recursive cases is characteristic of relational specifications in the \emph{hohh} logic.

3.2 Logic Programming

The computational interpretation of the \emph{hohh} logic consists of thinking of a collection of \( D \)-formulas as a program against which we can solve a \( G \)-formula. More formally, computation in this setting amounts to attempting to construct a derivation for a sequent of the form \( \Xi; P \to G \), where \( \Xi \) is a signature, \( P \) is a set of program clauses, and \( G \) a goal formula. The computation that results from such a sequent consists of first decomposing the goal \( G \) in a manner determined by the logical constants that appear in it and then, once \( G \) has been broken up into its atomic components, picking a formula from \( P \) and using this to solve the resulting goals.

The precise derivation rules for the \emph{hohh} logic are given in Figure 3. These rules can be understood as follows. In a sequent of the form \( \Xi; P \to G \), if \( G \) is not an atomic formula, then it must have one of the forms \( \top, D \supseteq G \) or \( \forall x. G' \). The first kind of goal has an immediate solution. In the second case, we extend the logic program \( P \) with \( D \) and continue search with \( G' \) as the new goal formula. In the last case, i.e., when \( G \) is of the form \( \forall x. G' \), we expand \( \Xi \) with a new constant \( c \) and the new goal becomes \( G'[c/x] \). Once we have arrived at an atomic formula \( A \), we pick a clause from \( P \) whose head eventually “matches” \( A \), spawning off new goals to solve in the process. The exact manner in which this kind of simplification of atomic goals takes place is determined by the last four rules in Figure 3.
kind nat type.
kind list type.


type z nat

type s (nat \rightarrow nat).

type nil list.

type cons (nat \rightarrow list \rightarrow list).

\pi (L \rightarrow (append \ nil \ L \ L)).
\pi x \rightarrow (pi \ L1 \setminus (pi \ L2 \setminus (pi \ L3 \setminus
(append \ L1 \ L2 \ L3 \Rightarrow
append \ (cons \ X \ L1) \ L2 \ (cons \ X \ L3))))).

Figure 4. An example of a \lambda Prolog program

A special case for treating atomic goals arises when the clause selected from the program has the structure

\forall x_1. (F_1 \supset \cdots \supset \forall x_n. (F_n \supset A') \ldots),

and where it is the case that for terms t_1, \ldots, t_n of correct type, A = A'[t_1/x_1, \ldots, t_n/x_n]. The effect of the sequence of rule applications that results in this case is reflected in the following derived rule

\Xi; \Gamma \rightarrow F_1 \supset \cdots \supset \Xi; \Gamma \rightarrow F_n \supset A \supset \text{backchain}

in which F_i = F_i[t_i/x_i, \ldots, t_i/x_i] for 0 < i \leq n. We shall find this rule, which we have labeled backchain for obvious reasons, useful in the analyses that appear in later sections.

The \lambda Prolog language can be viewed as a programming rendition of the hohh logic that we have discussed here. In \lambda Prolog, the user can introduce new atomic types through declarations that begin with the keyword kind and new constructors by using declarations that identify the signature, but also of clauses defining relations. In the concrete syntax of \lambda Prolog, abstraction is written as the infix symbol \setminus, i.e., the expression \lambda x. F is rendered as x\setminus F. Moreover, the logical constants \top, \bot and \supset are written as true, pi and \Rightarrow respectively. Another option for expressing G \Rightarrow D is the notation D \supset G. Several of these aspects of \lambda Prolog syntax are illustrated in Figure 4 through the presentation of clauses defining the append relation.

The \lambda Prolog language has been given an efficient compilation-based implementation in the Teyjus system. One of the goals of our work is to leverage this implementation in providing also an efficient treatment of LF programs.

4. A Naive Translation

We present in this section a simple translation of LF specifications into hohh specifications. This translation is taken from [2] that builds on earlier work due to Felty [2]. After presenting the translation, we will prove a correspondence between its source and target. This property will ensure that reasoning based on the translation will correctly follow reasoning based on the original specification. In this way, we know that constructing a hohh proof of some judgment is equivalent to finding a derivation in LF. Unfortunately, the simple translation produces hohh formulas that contain a lot of redundant information related to type checking that can result in quite inefficient proof search behavior. We highlight this issue towards developing a better translation in the next section.

4.1 The Translation

We have previously seen two methods for specifying append, in Section 2 a dependently-typed calculus was used and in Section 3 we utilized a relational style. Similarities between these two styles should have become apparent from this simple example. The signature we defined consisted of expressions which are essentially the same between LF and the simply typed \lambda calculus. Differences appear when defining dependencies between objects and types. In LF these relations are defined in the types and so we defined objects appendNil and appendCons. hohh is simply typed, and so relations are encoded using predicates and D-formulas are constructed to define exactly when the relation holds. There is then, a clear connection between the dependent types in LF, and the program clauses in hohh. The closeness of these two approaches is important in determining a translation from LF to hohh specifications.

As we have seen in Section 3b the goal of proof search in Twelf is to determine if an object of a particular type can be constructed. We will mimic this situation in \lambda Prolog by examining if we can construct a proof for an hohh formula that is obtained from the LF type. The translation presented by Felty relies on having in hand both the LF type and the LF object, but this is obviously too much to expect if proof search is intended to be the main focus. To overcome this difficulty, Snow et. al. adapted Felty’s translation so that it was based solely on the type [3]; the LF object is then uncovered incrementally by proof search in the hohh logic from the corresponding specification.
The following theorem makes precise our informal description of the property of our translation and also provides the basis for using \( hohh \) proof search in answering LF queries.

**Theorem 1.** Let \( \Gamma \) be a well-formed canonical LF context and let \( A \) be a canonical LF type such that \( \Gamma \vdash A : \text{Type} \) has a derivation. If \( \Gamma \vdash M : A \) has a derivation for a canonical object \( M \), then there is a derivation of \( \{ \Gamma \} \rightarrow \{ M : A \} \). Conversely, if \( \{ \Gamma \} \rightarrow \{ M' \} \) for an arbitrary \( hohh \) term \( M' \), then there is a canonical LF object \( M \) such that \( M' \equiv \{ M \} \) and \( \Gamma \vdash M : A \) has a derivation.

The proof of this theorem can be found in [9]. To summarize the proof, the completeness argument proceeds by induction on the derivation of \( \Gamma \vdash M : A \). To show how to construct a derivation for \( \{ \Gamma \} \rightarrow \{ M : A \} \), similarly, for soundness it uses induction on the derivation of \( \{ \Gamma \} \rightarrow \{ M' \} \) to extract from \( M' \) an LF object \( M \) of the required type.

### 4.2 Some Issues With The Translation

The translation described here has been shown correct. However, because LF expressions contain a lot of redundant information, and because of the context in which we want to use the translation, it is possible to produce a version that is more optimized for proof search. A key fact to bear in mind is that when we consider judgments of the form \( \Gamma \vdash M : A \) in the setting of logic programming, we would have already verified that \( A \) is a valid type. This knowledge gives us additional typing related information. For example, suppose that

\[
A = \text{append nil} (\text{cons z nil}) (\text{cons z nil})
\]

If we know that \( A \) is a valid type, then clearly \( \text{cons z nil} \) must be of type \( \text{list} \). In fact, looking at the \text{app-fam} rule tells us that a derivation of \( \Gamma \vdash A : \text{Type} \) must have a derivation of \( \Gamma \vdash \text{cons z nil} : \text{list} \). Thus, in deriving the \( hohh \) goal

\[
\text{hastype M (append nil (cons z nil) (cons z nil))}
\]

it is unnecessary to show that \( \text{hastype (cons z nil) list} \) holds as the translation of the type of \( \text{appCons} \) that is shown in Figure 6 requires us to do.

Removing tests like those above that arise from binders in LF types would certainly simplify the \( hohh \) specification and would thereby allow for more efficient proof search. However, not all such binders can be ignored in LF types: some of them also play a role in addressing inhabitation questions and are not just relevant to type checking. For example, consider the (well-formed) type

\[
\text{append (cons z nil) (cons z nil)}
\]

To form an object of this type based on the \( \text{appCons} \) constructor, we need to in hand have an object of type

\[
\text{append z (cons z nil) nil}
\]

Thus, the translation of the type of \( \text{appCons} \) in whose binder this type occurs must preserve the subgoal corresponding to finding such an object. Clearly then, we need some method of determining which tests are redundant and so can be correctly removed and which must be preserved.

### 5. Improving the Translation

The redundancy issue highlighted in the previous section can be rephrased as follows. We are interested in translating an LF type of the form \( \Pi x_1 : A_1, \ldots, \Pi x_n : A_n, B \) into an \( hohh \) clause that can be used to determine if a type \( B' \) can be viewed as an instance \( B[M_1/x_1, \ldots, M_n/x_n] \) of the target type \( B \). This task also requires us to show that \( M_1, \ldots, M_n \) are inhabitants of the types \( A_1, \ldots, A_n \); in the naive translation, this job is done by the

---

1 To translate LF signatures in their entirety, we also have to describe a translation of kinds. However, these translations will not be used in the derivations in \( hohh \) and so we make them explicit.
Suppose we know that this is a valid type. Then we would already know that $t_2$ has the type

$$\{ z : b t_1 \} c (\{ y : \text{nat} \} y) (\{ w : \text{nat} \rightarrow \text{nat} \} [y : \text{nat}] t_1 (w y)) z$$

and hence would not need to check this explicitly. The fact that $t_2$ has this type follows from looking at its occurrence in the type known to be valid and noting that the checking of the type of $f$ has already established that any instance of the third argument of $d$ in this setting must have as its type the corresponding instance of the type of $y$:

$$\{ z : b t_1 \} c (\{ y : \text{nat} \} y) (\{ w : \text{nat} \rightarrow \text{nat} \} [y : \text{nat}] x (w y)) z.$$  

Analyzing this more closely, we see that the critical contributing factors (to $t_2$ occurring in this way in the type) are that the path down to the occurrence of $y$ is rigid, i.e., it cannot be modified by substitution and $y$ is not applied to arguments in a way that could change the structure of the expression substituted for it. These properties were formalized in a notion of strictness in [9], there inappropriately referred to as rigidity.

The criterion described in [3] actually fails to recognize some further cases in which dynamic type checking can be avoided. To understand this, consider the occurrence of $x$ in the target type of $f$. This occurrence applies to an argument that could end up “hiding” the actual structure of any instantiation of $x$. We see this concretely in the instance

$$d (\{ y : \text{nat} \} y) (\{ w : \text{nat} \rightarrow \text{nat} \} [y : \text{nat}] t_1 (w y)) t_2.$$ 

considered earlier; we know something about the type of the term resulting from $t_1 (w y)$, but cannot conclude anything about the type of $t_1$ itself from this. Thus, this occurrence of $x$ is correctly excluded by the strictness condition presented in [9].

Observe, however, that $x$ has another occurrence in the type of $f$, in particular, in the type of the argument $y$. Further, because this argument $y$ occurs strictly in the instantiated target type, we would have statically checked its validity. Looking at that type, which is

$$\{ z : b t_1 \} c (\{ y : \text{nat} \} y) (\{ w : \text{nat} \rightarrow \text{nat} \} [y : \text{nat}] t_1 (w y)) z,$$

we would know that $b t_1$ is well formed and therefore $t_1$ is an inhabitant of the expected type.

In summary, it seems possible to extend the strictness condition recursively while preserving its utility in recognizing redundancy in type checking. We consider occurrences of bound variables to be strict in the overall type if they are strict in the types of other bound variables that occur strictly in the instantiated target type. The relation defined in Figure 7 formalizes this idea. Specifically, we say that the bound variable $x_i$ occurs strictly in the type $\Pi_{x_1 : A_1} \ldots \Pi_{x_n : A_n} B$ if it is the case that

$$\vdash x_i \in \text{ctx} \Pi_{x_1 : A_1} \ldots \Pi_{x_{i-1} : A_{i-1}} \Pi_{x_{i+1} : A_{i+1}} \Pi_{x_n : A_n} B$$

holds.

In the lemmas that follow, we formally prove the relationship between this notion and redundancy in type checking that we have discussed above.

**Lemma 2.** Let $N_1, \ldots, N_n$ be LF objects and $\Gamma, \Gamma_0$, and $\Delta$ be LF contexts where $\Gamma_0 = x_1 : B_1, \ldots, x_n : B_n$. Further, let $M$ be an LF object, $A$ an LF type and $\delta \subseteq \text{dom}(\Delta)$. Finally, suppose for some $i$ there are derivations of

1. $x_1, \ldots, x_n \vdash \delta; x_i \in \text{ctx} \Pi_{x_1 : A_1} \ldots \Pi_{x_{n-1} : A_{n-1}} \Pi_{x_{n+1} : A_{n+1}} \Pi_{x_n : A_n} B$.
2. $\Gamma, \Gamma_0 \vdash M : A$, and
3. $\Gamma, \Delta \Pi_{N_1/x_1} \ldots \Pi_{N_n/x_n} \vdash (M : A)[N_1/x_1 \ldots N_n/x_n]$.

Then there is a derivation of $\Gamma \vdash N_i : B_i[N_1/x_1 \ldots N_{i-1}/x_{i-1}]$.
Proof. By induction on the derivation of \(x_1, \ldots, x_n; \delta; x_i \sqsubseteq_o M\). The argument proceeds by considering the cases for the last rule used in the derivation.

The last rule is \texttt{int}. In this case, \(M = (x_1 y_1 \ldots y_k)\) for some distinct \(y_1, \ldots, y_k \in \delta\) and \(y_i\). Then \(M_1[x_1/x \ldots N_n/x_n]\) must in fact be \((N_1 y_1 \ldots y_k)\). From \((\texttt{2})\), it follows that \(B_i\), the type of \(x_i\), must be \(\Pi y_1:C_1 \ldots \Pi y_k:C_k. A\). Note that none of the variables in \(\text{dom}(\Delta)\) can appear in \(B_i\) (and hence \(A\)) or in \(N_j\) for \(1 \leq j \leq n\) and, further, that \(B_i\) cannot contain \(x_i\) if \(y_i \leq x_i\). Then we get from \((\texttt{2})\) that \(\Gamma \vdash (N_1 y_1 \ldots y_k) : B_i[N_1/x \ldots N_n/x_n]\) has a derivation. By using below this derivation a sequence of \texttt{abs-fam} rules and using the fact that the variables \(y_1, \ldots, y_k\) cannot appear in \(A[N_1/x_1 \ldots N_n/x_n]\), we see then that there must be a derivation of

\[
\Gamma \vdash \lambda y_1:C_1 \ldots \lambda y_k:C_k (N_1 y_1 \ldots y_k) : \Pi y_1:C_1 \ldots \Pi y_k:C_k. A[N_1/x \ldots N_n/x_n/x_i-1].
\]

But \(\lambda y_1:C_1 \ldots \lambda y_k:C_k (N_1 y_1 \ldots y_k)\) is equivalent (via the \(\eta\)-conversion rule) to \(N_1\) and

\[
\Pi y_1:C_1 \ldots \Pi y_k:C_k. A[N_1/x \ldots N_n/x_i-1/x_i-1]
\]
is identical to

\[
(\Pi y_1:C_1 \ldots \Pi y_k:C_k) [N_1/x \ldots N_n/x_i-1/x_i-1],
\]

i.e., to \(B_i[N_1/x \ldots N_n/x_i-1/x_i-1]\). Thus we actually have a derivation for

\[
\Gamma \vdash N_1 : B_i[N_1/x \ldots N_n/x_i-1/x_i-1],
\]
as desired.

The last rule is \texttt{app}. In this case, \(M = y M_1 \ldots M_k\) for some \(y \notin \Gamma\) of type \(\Pi z_1:C_1 \ldots \Pi z_k:C_k. A\) and \(x_1, \ldots, x_n; \delta; x_i \sqsubseteq_o M_j\) for some \(j\). Then by successive applications of \texttt{app-to} \((\texttt{2})\) there is a derivation of

\[
\Gamma, \Gamma_0, \Delta \vdash M_j : C_j.
\]

Similarly from \((\texttt{3})\) we find

\[
\Gamma, \Delta[N_1/x \ldots N_n/x_n] \vdash M_j[N_1/x \ldots N_n/x_n] : C_j[N_1/x \ldots N_n/x_n].
\]

We conclude by using the induction hypothesis.

The last rule is \texttt{abs}. In this case \(M = \lambda y C. M'\), \(A = \Pi y:C. A'\) and there is a derivation of \(x_1, \ldots, x_n; \delta; y; x_i \sqsubseteq_o M'\). Looking also at the other two derivations, \texttt{abs-obj} provides derivations of both

\[
\Gamma, \Gamma_0, \Delta, y : C \vdash M' : A'
\]
and

\[
\Gamma, \Delta[N_1/x \ldots N_n/x_n], y : C[N_1/x \ldots N_n/x_n] \vdash M'[N_1/x \ldots N_n/x_n] : A'[N_1/x \ldots N_n/x_n].
\]

Now we can conclude by the induction hypothesis. \(\square\)

**Lemma 3.** Let \(N_1, \ldots, N_n\) be LF objects and \(\Gamma, \Gamma_0, \text{ and } \Theta\) be LF contexts with \(\Gamma_0 \vdash x_1 : B_1, \ldots, x_n : B_n\). Further, let \(\Pi x_1:B_1, \ldots, \Pi x_n:B_n.A\) be an LF type with \(A\) a base type. Finally, suppose for some \(i\) there are derivations of

1. \(x_1 : B_1, \ldots, x_n : B_n, \Theta, x_i \sqsubseteq_t A\)
2. \(\Gamma, \Gamma_0, \Theta \vdash A : \text{Type}\)
3. \(\Gamma, \Theta[N_1/x_1 \ldots N_n/x_n] \vdash A[N_1/x_1 \ldots N_n/x_n] : \text{Type}\)

Then there is a derivation of \(\Gamma \vdash N_1 : B[N_1/x_1 \ldots N_n/x_n/x_i-1/x_i-1]\).

**Proof.** We prove the lemma by induction on the structure of the derivation of \(x_1 : B_1, \ldots, x_n : B_n; x_i \sqsubseteq_t A\). The argument proceeds by considering the case for the last rule in the derivation.

The derivation concludes by \texttt{app}. Then there is some \(c\) of type \(\Pi y_1:C_1 \ldots \Pi y_k:C_k. A\) and \(A\) is of the form \(c M_1 \ldots M_k\) where \(x_i\) occurs rigidly in some \(M_j\). Therefore we have a derivation of \(x_1, \ldots, x_n, \text{dom}(\Theta); \vdash x_i \sqsubseteq_o M_j\). Successive applications of \texttt{app-fam to} \((\texttt{2})\) and \((\texttt{3})\) simultaneously will provide derivations of \(\Gamma, \Gamma_0, \Theta \vdash M_j : C_j\) and

\[
\Gamma, \Theta[N_1/x_1 \ldots N_n/x_n] \vdash M_j[N_1/x_1 \ldots N_n/x_n] : C_j[N_1/x_1 \ldots N_n/x_n],
\]
respectively. We conclude by invoking \texttt{Lemma 2}\((\texttt{2})\). The derivation concludes by \texttt{ctx}. Then there is some \(x_j : B_j\) such that

\[
x_1 : B_1, \ldots, x_j-1 : B_{j-1}; x_i \sqsubseteq_t B_j
\]
and

\[
x_1 : B_1, \ldots, x_j-1 : B_{j-1}; x_i \sqsubseteq_t A.
\]

(i)

Now, by \((\texttt{4})\) and \((\texttt{5})\), there is a derivation of

\[
\Gamma \vdash N_j : B_j[N_1/x_1 \ldots N_{j-1}/x_{j-1}].
\]

And so there must be a derivation of

\[
\Gamma \vdash B_j[N_1/x_1 \ldots N_{j-1}/x_{j-1}] : \text{Type}.\]

(ii)

We also have a derivation of

\[
\Gamma, \Gamma_0, \Delta \vdash B_j : \text{Type}.\]

(iii)

If \(B_j\) is a base type we can conclude by the induction hypothesis. Otherwise \(B_j\) is of the form \(\Pi y_1:C_1 \ldots \Pi y_k:C_k.D\) with \(D\) a base type, and by \(\texttt{pi}\), there is a derivation of

\[
x_1 : B_1, \ldots, x_j-1 : B_{j-1}, \Theta', x_j \sqsubseteq_t D
\]
with \(\Theta' = y_1 : C_1, \ldots, y_k : C_k\). From \((\texttt{3})\) and \((\texttt{4})\) we obtain derivations of

\[
\Gamma, \Theta'[N_1/x_1 \ldots N_{j-1}/x_{j-1}] \vdash D[N_1/x_1 \ldots N_{j-1}/x_{j-1}] : \text{Type}
\]
and

\[
\Gamma, \Theta' \vdash D : \text{Type}
\]
by \texttt{pi-fam}. Since \(D\) must be a base type we can conclude by the induction hypothesis.

From these lemmas we can conclude that explicitly checking that the types of strict variables are inhabited is redundant as this is already guaranteed by the formation of the target type. We now turn leveraging this to improve the \texttt{hohh} clauses generated by translation of LF signatures.

### 5.2 Eliminating Redundancies in the Translation

In the simple translation presented in Section \((\texttt{4})\), every binder will generate a \texttt{hastype} formula which amounts to showing that a term is an inhabitant of a type. But we have shown that substitutions for strict variables must inhabit the correct type, and we would like to modify the translation so that this is taken into account. There are now two modes in which translation operates, the negative, \([\cdot]^\rightarrow\), which is essentially the same as before in that it does not check for strictness of bound variables, and the positive, \([\cdot]^\leftarrow\), which will only generate \texttt{hastype} formulas for variables which do not appear strictly. We do this to insur that the eliminations occur in situations in which it makes sense to think of the implication encoding an inhabitation check. The new rules for translating judgments can be found in Figure \((\texttt{9})\). The encoding of types and objects remains the same as in the simple translation.

It is useful to define a notion of equivalence between two \texttt{hohh} terms which encode the same LF term. The idea is that
Lemma 4. Let $\Gamma$, $\Delta_1$, and $\Delta_2$ be valid LF contexts, and let $\delta \subseteq \text{dom}(\Delta_1)$. Suppose there are derivations of

1. $\Gamma, x_1 : B_1, \ldots, x_n : B_n, \Delta_1 \vdash M : A$
2. $x_1, \ldots, x_n ; \delta ; x_i \sqsubseteq o M$, and
3. $\Gamma, \Delta_2 \vdash M' : A'$

where $(M' \sim M)[t_1/x_1, \ldots, t_n/x_n]$ and $(M' \sim M)[t_1/x_1, \ldots, t_n/x_n]$ for some hohh terms $t_1, \ldots, t_n$. Then there is an LF term $t_i'$ such that $t_i = \langle t_i' \rangle$.

Proof. The proof proceeds by induction on the structure of the derivation $x_1, \ldots, x_n; \delta; x_i \sqsubseteq o M$.

The derivation concludes by $\text{Init}$. Then $M$ has the form $x_i y_1 \ldots y_k$ for distinct $y$ from $\delta$. So $t_i$ must be a term of the form

$$\lambda z_1 \ldots \lambda z_k u.$$

We use $M' \sim M$ to determine that $\langle M' \rangle = u[y_1/z_1 \ldots y_k/z_k]$.

$$\langle M' \rangle = \langle M \rangle[t_1/x_1, \ldots, t_i/x_i] = (x_1 y_1 \ldots y_k)[t_1/x_1, \ldots, t_i/x_i] = (t_i y_1 \ldots y_k) = u[y_1/z_1 \ldots y_k/z_k]$$

Because the $y_i$'s are distinct the substitution on $u$ can be inverted and we find $u = \langle M' \rangle[y_1/z_1 \ldots y_k/z_k]$. In fact, because the encoding leaves variables unchanged we can determine that $\langle M' \rangle[y_1/z_1 \ldots y_k/z_k] = \langle M' \rangle[y_1/z_1 \ldots y_k/z_k]$. Each $y_i$ is a distinct variable from $\delta$, and so for some type $C_j$, there is a binding $y_j : C_j \in \Delta_1$. Let $t_i' = \lambda z_1 : C_1 \ldots \lambda z_k : C_k u'$ where $u' = M'[y_1/z_1 \ldots y_k/z_k]$. Then $\langle t_i' \rangle = t_i$.

The derivation concludes by abs. Then $M$ is of the form $\lambda y : C. N$ and from (3) we have a derivation of $x_1, \ldots, x_n; \delta; x_i \sqsubseteq o N$. Because $M' \sim M$ their structures must be similar, and so $M'$ has the form $\lambda y : C'. N'$ with $N' \sim N$. The type $A$ must be of the form $\Pi y : C.D$, and $A'$ will be of the form $\Pi y : C'. D'$. From (1) and (3) respectively, abs-obj provides derivations of

$$\Gamma, x_1 : B_1, \ldots, x_n : B_n, \Delta_1, y : C \vdash C' : C$$

and $\Gamma, \Delta_2, y : C' \vdash N' : C$. We can conclude by the inductive hypothesis.

The derivation concludes by app. Then $M$ has the form $y N_1 \ldots N_k$ with $y \neq x_j$ for $j \leq i$. So by the rules of strictness, there is a derivation of $x_1, \ldots, x_n; \delta; x_i \sqsubseteq o N_i$ for some $l < k$. Successive applications of app-obj to (1) followed by var-obj on some $y : \Pi z_1 : C_1, \ldots, \Pi z_k : C_k.D$ provide a derivation of

$$\Gamma, x_1 : B_1, \ldots, x_n : B_n, \Delta_1 \vdash N_i : C_i[N_1/z_1 \ldots N_{i-1}/z_{i-1}].$$

Since $M' \sim M = y N_1 \ldots N_k$, $M'$ has the form $y N_1' \ldots N_k'$ where $N_i' \sim N_i$. It should be clear that the derivation of (3) proceeded similarly to (1) but with some $y' : \Pi z'_1 : C'_1, \ldots, \Pi z'_k : C'_k.D'$. Thus we find a derivation of

$$\Gamma, \Delta_2 \vdash N'_i : C'_i[N'_1/z'_1 \ldots N'_{i-1}/z'_{i-1}].$$

We conclude by the induction hypothesis.

Lemma 5. Let $\Gamma$, $\Theta_1$, and $\Theta_2$ be valid LF contexts and

$$\Pi x_1 : B_1, \ldots, x_n : B_n, A$$

be a type with $A$ a base type. Suppose there are derivations of

1. $\Gamma, x_1 : B_1, \ldots, x_n : B_n, \Theta_1 \vdash A : \text{Type}$
2. $x_1 : B_1, \ldots, x_n : B_n, \Theta_1 \sqsubseteq o A$, and
3. $\Gamma, \Theta_2 \vdash A : \text{Type}$

where $(A' \sim A)[t_1/x_1, \ldots, t_n/x_n]$ for hohh terms $t_1, \ldots, t_n$. Then there is an LF term $t_i'$ such that $t_i = \langle t_i' \rangle$.

Proof. Proceed by induction on the derivation of $x_1 : B_1, \ldots, x_n : B_n, t_i \sqsubseteq o A$.

Note that the rule $\text{proj}$ will not apply because we know $A$ to be a base type.
The derivation concludes by app. Then A is a base type of the form $e \Pi_1 \ldots \Pi_k$ and there is a derivation of $x_1, \ldots, x_n \vdash t_1, \ldots, t_n$ for some $i < k$. Applying app-lam to $t_i$ provides the derivation of

$$\Gamma, x_1 : B_1, \ldots, x_n : B_n, \Theta \vdash N_i : T.$$  

Because $A' \sim A$, their structures must be similar and so $A'$ has the form $e \Pi_1', \ldots, N_i'$. Thus from (2) we can obtain a derivation of $\Gamma, \Theta_2 : N_i' : T$' where $N_i' \sim N_i$. We conclude by Lemma 3.

The derivation concludes by ctx. Then there must be derivations of

$$x_1 : B_1, \ldots, x_{j-1} : B_{j-1}; x_1 \vdash B_j,$$

and

$$x_1 : B_1, \ldots, x_n : B_n; x_j \vdash A.$$  

From the assumption (1), and the derivation rules of LF, there must be a derivation of $\Gamma, \Theta_1 \vdash N_i : T$ and so also one of

$$\Gamma, x_1 : B_1, \ldots, x_{j-1} : B_{j-1} \vdash B_j : Type.$$  

By the induction hypothesis on (3), there is an LF object $t'_j$ such that $t_j = \langle t'_j \rangle$. In addition, $t'_j$ must be of some type $B'_j$ where $B'_j \sim B_j[t_1/x_1, \ldots, t_n/x_n]$. Thus we conclude by the induction hypothesis.

We can now assume that any substitution $t$ for a strict variable $x$ which is performed in the translation, in fact is the encoding of a valid LF term $t'$. Because $t = \langle t' \rangle$ we can use Lemma 3 to guarantee $t'$ inhabits the correct type. Thus it is reasonable to remove the hohl formula which makes us to explicitly show that hohltype $t$ A holds for the appropriate type.

We show this new translation to be correct by relating the new translation with the simple one given in Section 4. From the correctness of this simple translation (Theorem (1)) then, we get that the improved translation is also correct. Showing completeness is straightforward because it simply erases information from derivations resulting from the simple translation. Soundness is more involved. We must reconstruct a derivation which has been eliminated by the strictness condition using properties of strict variables which have been shown through Lemmas 3 and 4.

**Theorem 6.** Let $\Gamma$ be a valid LF context and $A$ an LF type such that

1. $\Gamma \vdash A : Type$

is derivable. Then for arbitrary hohl term $M$, $\Gamma \vdash A(M)$ has a derivation if and only if $\Gamma^+ \vdash A^-(M)$ has a derivation.

**Proof. Completeness**

We proceed by induction on the derivation of $\Gamma \vdash A(M)$.

The derivation concludes by $\forall R$. Then $A$ must be of the form $\Pi x : B.A'$, and the derivation concludes by $\forall R$ and $\exists R$ as shown below.

$$\Gamma \vdash A'(\Pi x : B.A').(M) \forall R, \exists R$$

By assumption (1), and the derivation rules of LF, there must be a derivation of $\Gamma, x : B \vdash A' : Type$. By the induction hypothesis there is then a derivation of $\Gamma, x : B \vdash A^-(M)$ from this we can construct the following:

$$\Gamma, x : B \vdash A'(M) \forall R, \exists R$$

which is exactly the derivation desired.

**Theorem 6 concludes by $\exists R$. This case proceeds as for the previous case.**

The derivation concludes by backchain on the encoding of a term $(y : \Pi x_1 : B_1, \ldots, x_n : B_n, A') \in \Gamma$. Then $A$ is a base type of the form $u \Pi_1 \ldots \Pi_k$. It is clear then that $M$ is of the form $y t_1 \ldots t_n$ for some hohl terms $t_1, \ldots, t_n$ and $A \sim A'[t_1/x_1, \ldots, t_n/x_n]$. We are left with the collection of derivations shown below.

$$\Gamma \vdash A(M) \forall R, \exists R$$

To continue we show an inner induction on $i$ that asserts that for $j < i t_j = \langle t'_j \rangle$ for some LF object $t'_j$, then there is a $t'_j$ such that $t_j = \langle t'_j \rangle$.

Because $t_j = \langle t'_j \rangle$ for $j < i$, we can move the substitution inside of the encoding giving the equivalence

$$(\langle B_i \rangle x_1, \ldots, x_n) t_1 / x_1, \ldots, t_{i-1} / x_{i-1}) \sim (\langle B_i \rangle [t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}] \rangle t_i.$$

So there is a derivation $\Gamma \vdash B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}]$. Then we conclude by the correctness of the naive translation that there is an LF object $t'_j$ such that $t_i = \langle t'_i \rangle$.

By our assumption, $\Gamma$ is a valid context and so

$$\Gamma \vdash \Pi x_1 : B_1, \ldots, x_n : B_n, A' : Type$$

is derivable. We substitute the $t'_i$'s into this derivation to construct one of $\Gamma \vdash B_i[t'_1 / x_1, \ldots, t'_{i-1} / x_{i-1}] : Type$ for each $i$. By the outer induction we obtain the collection of derivations

$$\{ \Gamma \vdash B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}] \} \forall R, \exists R$$

To conclude we apply backchain with the improved translation of the same term $(y : \Pi x_1 : B_1, \ldots, x_n : B_n, A') \in \Gamma$ selecting the derivation $\Gamma \vdash \top$ if there is a strict occurrence of $x_i$ and

$$\Gamma \vdash B_i[t'_1/x_1, \ldots, t'_{i-1}/x_{i-1}] \forall R, \exists R$$

otherwise. Thus we have a derivation of

$$\Gamma \vdash A(M) \forall R, \exists R$$

as desired.

**Soundness**

We proceed by induction on the derivation of $\Gamma \vdash A(M)$.

The derivation concludes by $\forall R$. Then $A$ must be of the form $\Pi x : B.A'$, and the derivation concludes by $\forall R$ and $\forall R$ as shown below.

$$\Gamma, x : B \vdash A'(M) \forall R, \forall R$$

By assumption (1), there are derivations of $\Gamma \vdash B : Type$ and $\Gamma, x : B \vdash A' : Type$. From this and the assumption $\Gamma$ is a valid context, we can construct a derivation of $\Gamma \vdash B x : Bctx$. Now by the inductive hypothesis there is a derivation of

$$\Gamma, x : B \vdash A'(M x) \forall R, \exists R$$

To conclude we build the desired derivation as shown below.

$$\Gamma \vdash A'(M x) \forall R, \exists R$$

The derivation concludes by $\exists R$. This case follows that for the previous case.
a valid context, \( \Gamma, x_1 : B_1, \ldots, x_n : B_n \vdash A' : \text{Type} \) has a derivation. Then by Lemma 5 if \( x_1 \) has a strict occurrence in \( A' \) there must be an LF term \( t_i \) such that \( t_i = (t'_i) \). Since \( A' \) must then also be a base type, let it be of the form \( u \cdot N_1 \cdot \ldots \cdot N_k \). Then by the definition of backchain there are a collection of derivations \( \{[\Gamma]^1 \vdash F_i \}_1 \) where \( F_i = \top \) if \( x_i \) appears strictly and
\[
F_i = \left( [B_i]^i \vdash (x_i) \right) [t_1/x_1 \ldots t_i/x_i]
\]
otherwise.

We continue by showing an inner induction on \( i \) that if for each \( j < i \), \( t_j = (t'_j) \) for some LF object \( t'_j \) then there is \( t_i \) such that \( t_i = (t'_i) \).

Suppose \( F_i = \top \).

Then it must be that \( x_i \) appears rigidly and such a \( t'_i \) exists by the argument stated previously.

Suppose \( F_i = \left( [B_i]^i \vdash (x_i) \right) [t_1/x_1 \ldots t_i/x_i] \).

Since \( \Gamma \) is a valid context, \( \Gamma \vdash \Pi x_1 : B_1, \ldots, \Pi x_n : B_n \cdot A' : \text{Type} \) must be derivable. Then for each \( i \), there must also be a derivation of \( \Gamma, x_1 : B_1, \ldots, x_i : B_i \vdash \Pi \cdot \Pi x_1 : B_1, \ldots, \Pi x_n : B_n \cdot A' : \text{Type} \). Because \( t_i = (t'_i) \) for each \( t_j \), we can substitute these LF objects to obtain a derivation of \( \Gamma \vdash B_i [t'_1/x_1 \ldots t'_{i-1}/x_{i-1}] : \text{Type} \), and so by the outer induction, there must be a derivation of
\[
\delta \left( [\Gamma] \vdash \left( B_i [t'_1/x_1 \ldots t'_{i-1}/x_{i-1}] \right) \right) t_i.
\]

By the correctness of the naive translation, \( t_i = (t'_i) \) for some LF object \( t'_i \).

We now construct a derivation of
\[
\delta \left( [\Gamma] \vdash \left( B_i [t'_1/x_1 \ldots t'_{i-1}/x_{i-1}] \right) \right) t_i
\]
for each \( i \).

Suppose \( F_i = \left( [B_i]^i \vdash (x_i) \right) [t_1/x_1 \ldots t_i/x_i] \).

Then the derivation was found in the previous argument.

Suppose \( F_i = \top \).

Then there is a derivation of \( x_1, \ldots, x_i ; x_i \vdash _{\top} A' \). We also have a derivation of \( \Gamma \vdash \Pi x_1 : B_1, \ldots, \Pi x_n : B_n \cdot A' : \text{Type} \). Both \( A \) and \( A' \) are base types, and so since \( A \sim A' \), we know that \( \langle A \rangle \sim \langle A' \rangle \). We have shown that for each \( i \), \( t_i = (t'_i) \) for an LF object \( t'_i \). And thus
\[
\langle A \rangle [t_1/x_1 \ldots t_n/x_n] = \langle A' [t'_1/x_1 \ldots t'_n/x_n] \rangle
\]

By the correctness of \( \langle \cdot \rangle \) then, \( A = A'[t'_1/x_1 \ldots t'_n/x_n] \) and by assumption \( \{[\Gamma]^1 \} \) we have a derivation of \( \Gamma \vdash A'[t'_1/x_1 \ldots t'_n/x_n] \).

Applying Lemma 3 we now obtain a derivation of
\[
\delta \left( [\Gamma] \vdash A[t'_1/x_1 \ldots t'_{n-1}/x_{i-1}] \right) t_i.
\]

Finally, the correctness of the naive translation allows us to conclude there is a derivation of \( \left( [\Gamma] \vdash A[t'_1/x_1 \ldots t'_{n-1}/x_{i-1}] \right) \) as desired.

Now we have a collection of derivations which we compose using backchain to obtain a derivation of
\[
\delta \left( [\Gamma] \vdash A \right) (y \cdot t_1 \ldots t_n).
\]

The key to this proof is that we are able to reconstruct typing derivations for instantiations of strict variables by leveraging our assumption that \( A \) is a well formed term in the context \( \Gamma \). By removing the redundant formulas from implications we simplify the translated signature in a way that more closely resembles the original LF types. Recall the signature for append presented in Figure 2 and its translation into hohh shown in Figure 6. Using this new translation, we refine the clauses generated for append and appCons, as seen in Figure 10. The clauses shown in the figure are presented in a simplified form where the obviously satisfiable goals generated by the translation are removed for clarity.

The result of this example is exactly what is produced by the translation based on the former notion of strictness in [9], since every strict variable actually appears in the target type. Instead consider the example deceleration for \( f \) from Figure 8. Under the previous notion of strictness this would become the hohh clause
\[
\forall x. (hastype (x :: 1 :: x) \bullet hastype (x :: 1 :: x) \bullet
\forall y. (hastype (f x y) \bullet (d y \bullet (w y x y x y y))).
\]

Using the extension to strictness presented in this work, we can further reduce this clause by removing the formula related to typing \( x \) resulting in the clause
\[
\forall x. (hastype (f x y) \bullet (d y \bullet (w y x y x y y))).
\]

In Lemma 5 we are able to take the term \( t \), for which we have some knowledge of the structure, and construct an LF term \( t' \) such that \( t = (t') \). In next section we look at how we might encode general hohh terms, for which some type information is known, back into LF.

6. Translating back to LF terms

The previous sections have defined a translation which provides a means for taking LF specifications to XProlog programs. We would like to use this translation as a vehicle for efficiently executing LF specifications, but to do this we need a method for getting back to LF expressions once the execution in the hohh setting is completed. As we have presented it up to this point, the main objective of proof search is to identify an LF object corresponding to a given type. Thus, to make this approach to implementing Twelf work, what we need is a way to map an hohh term back into an LF object.

From the term encoding rules in figure 3 it is clear that through type erasure it is possible for a single hohh term to be the encoding of multiple objects in LF. This lack of uniqueness poses a problem for defining a general reverse encoding into LF. However, if the LF typing information is retained there is in fact a unique LF expression that the hohh term can be identified with. In the implementation context we are looking at, the relevant typing information is available directly from the original Twelf query. Thus after execution, the type can be used in guiding a reverse encoding.

To spell out the inverse translation process more precisely, we will assume that it takes place in a setting where the type of the LF object that is to be produced is known. Moreover, we will assume that each hohh constant corresponds to an object level constant with the same name and a known type; the type information will be provided by a signature \( \Sigma \). The hohh term may contain free variables and we will also assume that the LF types corresponding to these variables are given by a context \( \Gamma \). In the beginning, there will be no free variables so \( \Gamma \) can be empty and and our rules
The term is of the form \( \langle \cdot \rangle \) as desired.

\[ \Gamma \vdash M \colon T \]

for each \( i \). So by the induction hypothesis

\[ \Gamma ; \langle M_i \rangle \vdash_{\Sigma} M_i : T[M_i/y_1 \ldots M_{i-1}/y_{i-1}] \]

is derivable for each \( i \). We can now compose these by inv-\text{var} for a derivation of

\[ \Gamma ; (x \langle M_1 \rangle \ldots \langle M_n \rangle) \vdash_{\Sigma} (x M_1 \ldots M_n) : T. \]

But \( \langle c M_1 \ldots M_n \rangle = (c \langle M_1 \rangle \ldots \langle M_n \rangle) \), and so we really have one of \( \Gamma ; (\langle c M_1 \ldots M_n \rangle) \vdash_{\Sigma} (c M_1 \ldots M_n) : T \) as desired.

We now have a method of re-encoding closed \textit{hohh} terms back into LF, which complements the term encoding in Figure 5. This reverse encoding provides the ability to run Twelf programs containing such queries efficiently via Tejyus and then return the results in a form which aligns with the specification formed in Twelf.

\section{Towards Treating Existential Variables}

So far we have only considered the problem of finding inhabitants for closed types. In practice, Twelf also allows free variables to appear in types that constitute queries. These variables are considered to be existentially quantified in the sense that answering the query requires finding substitutions for these variables that make the type well-formed in addition to providing an inhabitant for the resulting type.

Let us consider an example of the use of such variables based on the signature for \textit{append} from Section 2. Suppose that we want to find the list \( L \) that is the result of appending \( \text{cons} \ (s \ z) \) nil to \( \text{cons} \ z \) nil. We can have Twelf determine this list by posing the query

\[ M : \text{append} \ (s \ z) \text{nil} \ (\text{cons} \ z) \text{nil} \ L. \]

Now, Twelf must find a substitution for \( L \) that yields a well-formed type and simultaneously determine a value for \( M \) that constitutes an object of the resulting type. To solve this query, Twelf will match it with the clause for \textit{appCons}, resulting in the new goal of constructing an inhabitant \( M' \) of the type \text{append} \text{nil} \ (\text{cons} \ z) \text{nil} \ L'; this match will also produce the binding \( \text{cons} \ (s \ z) \text{nil} \) for \( L \). At this stage, Twelf will use the \textit{appNil} clause, resulting in a solution to the overall goal with the binding \( \text{cons} \ z \text{nil} \) for \( L' \) and, therefore, of \( \text{cons} \ (s \ z) \text{nil} \) for \( L \). The inhabitant found for the original query will correspondingly be

\[ \text{appCons} \ (s \ z) \text{nil} \ (\text{cons} \ z) \text{nil} \ (\text{appNil} \ (\text{cons} \ z) \text{nil}). \]
The translation that we have described for LF types works without modification also for types with free variables. However, there is potential for the computational behavior of the translated form to be different from that for Twelf because of the way in which variables in types are instantiated. Before we can discuss this in detail, it is important to understand the way Twelf treats dependencies in types. Let us assume the declarations introduced for representing natural numbers and provide the following additions to the signature

\[
\begin{align*}
  i & : \text{type} \\
  \text{bar} & : \text{nat} \to \text{type} \\
  \text{foo} & : \{X : i\}\text{bar} \: z
\end{align*}
\]

Now we consider the query \( T : \text{bar} \: z \). Twelf will fail in this query under the following rationale: The only way to construct an object of this type is by finding an object of type \( i \) and using this as an argument of \( \text{foo} \). However, there is no way to construct an object of type \( i \).

An interesting thing happens if we change the definition of \( \text{foo} \) to make the argument variable actually appear in the target type. More specifically suppose that the definition of \( \text{foo} \) is replaced with the following:

\[
\text{foo} : \{X : i\}\text{bar} \: X.
\]

Now the query \( T : \text{bar} \: Y \) returns the substitution \( T = \{X : i\}\text{foo} \: X : \{X : i\}\text{bar} \: X \).

This solution can be interpreted as telling the user that if he/she can provide something of type \( i \), then there would be a term of the required type. From the specification we know that there cannot in fact be any terms of type \( i \), but Twelf does not try to find any terms of type \( i \) and instead leaves it as a constraint.

Twelf’s behavior on dependent types can then be summed up as follows. If the argument variable does not appear in the target type, thereby signaling the absence of any real dependencies, proof search will force the finding of an inhabitant for the corresponding type. However, if the dependency is real, then the variable will be instantiated only to the extent needed by other parts of the proof search procedure; he actual finding of an inhabitant will not be required for a successful solution.

Returning now to the comparison with the translated form, we see that the behaviors are convergent in the case where the argument variable does not appear in the target type. This is because the variable does not appear strictly in the target type and hence the translation produces a \( \text{hohh} \) clause that forces the search for an inhabitant. We can see this, for example, by looking at the clause produced for the first version of \( \text{foo} \) which would be

\[
\forall X (\text{hastype} \: X \: i \supset \text{hastype} \: (\text{foo} \: X) \: (\text{bar} \: z)).
\]

When the variable does occur in the target type, however, we have two different situations. If the variable has at least one strict occurrence, then the clause that is produced will not check for the type of the term instantiating the variable and hence also will not force the search of an inhabitant if the variable is uninstantiated. This is seen, for example, from the clause resulting from the second definition of \( \text{foo} \), which will be

\[
\forall X \: \text{hastype} \: (\text{foo} \: X) \: (\text{bar} \: X).
\]

However, if the variable does not have even one strict occurrence in the type, then the translated version will force the search for an inhabitant and the behavior corresponding to it will diverge from that under Twelf.

We have conjectured that the computational behavior of Twelf on an LF specification and of \( \lambda \text{Prolog} \) on a translated form of the LF specification will be closely related if all the argument variables have at least one strict occurrence in their types in the specification.

In proving this conjecture, however, it is necessary also to take into account unification behavior. A complicating factor here is that unification in \( \lambda \text{Prolog} \) will take place on “type erased” forms of terms in LF. We believe, however, that this will not matter: the unifying substitutions will be related also via a type erasure. In particular we believe that the following claim holds.

**Claim 1.** Suppose \( M \) and \( M' \) are terms of equivalent type. Then \( \sigma \) is a unifier for \( M \) and \( M' \) if and only if \( \sigma' = \langle \sigma \rangle \) is a unifier for \( \langle M \rangle \) and \( \langle M' \rangle \).

If we can show this claim, this will allow us to move between unification of LF expressions and \( \text{hohh} \) terms freely because they will be essentially equivalent. Using this fact, we should be able to relate the operational semantics of Twelf and \( \lambda \text{Prolog} \). This would be done by recursively looking at each step and maintaining a correspondence between the two. From this we should be able to prove a strong form of correspondence between the original LF specification satisfying the strictness restriction and the translated version with respect to proof search even in the presence of existential variables.

When the restriction is not satisfied, the behaviors diverge as we have noted earlier. To understand the nature of the divergence, let us consider the following example signature.

\[
\begin{align*}
  \text{nat} & : \text{type} \\
  z & : \text{nat} \\
  s & : \text{nat} \to \text{nat} \\
  \text{bar} & : \text{nat} \to \text{type} \\
  \text{foo} & : \{Y : \text{nat}\}\{F : \text{nat} \to \text{nat}\}\text{bar} \: (F \: Y)
\end{align*}
\]

Consider the query \( T : \text{bar} \: z \). Twelf is unable to resolve the equation \( z = F \: X \) and so cannot supply a solution for this query. However, under the translation we can determine the obvious solutions in a systematic way. The clause for \( \text{foo} \) under the translation would have the form

\[
\forall Y (\text{hastype} \: Y \: \text{nat} \supset \forall F (\forall X_1 (\text{hastype} \: X_1 \: \text{nat} \supset \text{hastype} \: (F \: X_1) \: \text{nat}) \supset \text{hastype} \: (\text{foo} \: F \: Y) \: (\text{bar} \: (F \: Y))))
\]

and the query is \( \text{hastype} \: T \: (\text{bar} \: z) \). Backchaining on the clause for \( \text{hastype} \) to solve this query still requires us to unify \( z \) and \( F \: Y \). However, this time we have a method for finding valid substitutions to consider for \( F \) and \( Y \) in the course of solving the unification problem. First a term of type \( \text{nat} \) must be determined for \( Y \) using a query such as \( \text{hastype} \: Y \: \text{nat} \) and once fixed, a substitution for \( F \) will be formed using

\[
\forall X_1 (\text{hastype} \: X_1 \: \text{nat} \supset \text{hastype} \: (F \: X_1) \: \text{nat}).
\]

If these substitutions satisfy \( F \: Y = z \) they can be considered a valid solution. Working through a few examples, if we first fix \( Y = z \) by matching with the generated clause for \( z \) in the translated signature, both \( \lambda x. \: x \) and \( \lambda x. \: z \) will be substitutions for \( F \) for which \( (F \: Y) = z \). Next we would fix \( Y = (s \: z) \) as a substitution, and so \( \lambda x. \: x \) no longer satisfies our constraint. Thus there is only one substitution for \( F \) which is valid, \( F = \lambda x. \: z \). In fact, for every successive natural number only \( F = x \: z \) will satisfy \( F \: Y = z \). In this systematic manner we are able to generate the possible substitutions using the \( \lambda \text{Prolog} \) program.

**8. Conclusion**

This paper continues the work in [3] on translating LF specifications into \( \text{hohh} \) formulas. We have extended that work by defining a richer notion of strictness that can potentially lead to the elimination of more type checking and can thereby be the basis for a more efficient and clearer translation. We have also presented a procedure for translating \( \text{hohh} \) terms back to LF objects in the context
of the original LF specifications. This inverse translation provides a means for taking terms generated by running the translated specification and constructing the corresponding LF term. Finally, we have analyzed the situation where existential variables appear in LF types; this situation leads to type reconstruction in the Twelf system. Although we have still to formalize our observations, our analysis suggests that when the types in an LF specification are such that every dependently quantified variable appears strictly in the target type, then the behavior of AProlog over the 
\texttt{hohh} translation will mimic that of Twelf over the LF specification.

In future work, we would like to translate the results of this paper into practical applications. We have already modified the code in the Parinati system \cite{8} to take into account the extended strictness check described in Section \cite{5}. We are currently in the process of testing the resulting system and assessing whether the new cases it covers are ones that have real efficiency benefits. In a related direction, we would like to use our ideas to provide an alternative implementation of the logic programming part of the Twelf system. This new system would take LF specifications and queries, translate them into their AProlog counterparts and then use the inverse translation to return the results in a form understandable in the LF setting.

Another direction for continued work concerns the treatment of existential variables in types. One task is to formalize the observations made in Section \cite{7} This would involve, for example, providing a proof of Claim \cite{1}. In Section \cite{7} we also noted that when existential variables are present there are situations in which the behavior under the translation would be different from that under the original LF specification. We would like to understand these situations better: we feel that the translated form might actually give us better control over computational behavior than the LF version in these cases and would like to substantiate this aspect if it is actually true.

The work in this paper has dealt exclusively with reasoning from the specifications written in LF. A completely different direction to pursue is that of reasoning about LF specifications. To understand the difference, we might consider the specification of natural numbers and the plus relation that was provided in Section \cite{2}. Based on these specifications, we might want to show something about the plus relation. For example, we may want to show that given two natural numbers \(m\) and \(n\) there is always another natural number \(k\) that is \(m\) plus \(n\). This clearly cannot be shown by solving any query from the specification. Rather, it involves proving something about all queries that can be made against the specification. The Twelf system allows such reasoning to be realized through tools for showing the totality of specifications written in it. For example, the property in question about \texttt{plus} can be established by showing that for any \(M\) and \(N\) there is always a value \(K\) for which the goal \texttt{plus}\(M\cdot N\cdot K\) will succeed. Totality checking does not provide an explicit proof of the property since there is no explicit logic supporting this style of reasoning. The translation from LF to the \texttt{hohh} logic suggests an alternative path: we can think of also translating totality checking into an explicit proof in the Abella system \cite{3} that supports the capability of logic based reasoning about \texttt{hohh} specifications. As a continuation of this work, we would like to explore the extension of the translation to this kind of meta-theoretic reasoning. Another direction that is much more challenging is to see if the Abella logic can be used to directly reason over LF-style specifications rather than having to do this via translations.

Acknowledgments

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