A Meta-Programming Approach to Realizing Dependently Typed Logic Programming

Zach Snow
Computer Science and Eng.
University of Minnesota
snow@cs.umn.edu

David Baelde
Computer Science and Eng.
University of Minnesota
dbaelde@cs.umn.edu

Gopalan Nadathur
Computer Science and Eng.
University of Minnesota
gopalan@cs.umn.edu

ABSTRACT

Dependently typed lambda calculi such as the 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 provide the benefits of a Twelf-like system for encoding type and proof-and-formula dependencies. 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 resembles the original specification, thereby allowing LF specifications to be viewed as *hohh* meta-programs. Using the Teyjus implementation of λProlog, we show that our translation provides an efficient means for executing LF specifications, complementing the ability that the Twelf system provides for reasoning about them.

Keywords

logical frameworks, dependently typed lambda calculi, higher-order logic programming, translation

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. Dependently typed lambda calculi such as the Logical Framework (LF) [11] provide many conveniences from a specification perspective in this context: such calculi facilitate the use of a higher-order approach to describing the syntax of formal objects, they allow relationships between terms to be captured in an elegant way through type dependencies, and they formalize the authentication of proofs through type-checking. Such calculi can also be assigned a logic programming interpretation by exploiting the well-known isomorphism between formulas and types [12]. The Twelf system [17] that we consider in this paper exploits this possibility relative to LF. As such, it has been used successfully in specifying and prototyping varied formal systems and mechanisms have also been built into it to reason about these specifications.

Predicate logics provide the basis for logic programming languages that are also capable of encoding rule-based specifications. Within this framework, the logic of higher-order hereditary Harrop (*hohh*) formulas [13] that underlies the language λProlog [15] provides a builtin ability to treat binding notions in syntax and thus has particular usefulness in representing formal systems. However, unlike LF, this logic cannot reflect dependencies between objects into types and also does not directly represent the relationship between formulas and their proofs. While such correspondences can always be encoded in auxiliary predicate definitions, it is of interest to understand if a more systematic treatment is possible. A specific form to this question is if Twelf specifications can be viewed as λProlog "meta-programs." There are benefits to such a possibility: the convenience of writing specifications using dependent types can be combined with the ability to execute them via an efficient λProlog implementation as well as perhaps to reason about them using logics and systems meant for analyzing *hohh* descriptions [2, 7, 10, 14].

A partial answer to the question raised above has been provided by Felty, who described a translation of LF specifications to *hohh* formulas and then showed that LF derivations correspond exactly to *hohh* derivations of the translated LF judgment [5, 6]. The focus on matching derivations allows Felty to assume the existence of a complete LF judgment, and, in particular, of an LF object in her translation. However, this assumption is inappropriate in our context, given that we are interested in *constructing* proof terms that show particular types are inhabited, *i.e.*, in *proof search* that plays a fundamental role in the logic programming setting. We therefore refine the earlier mapping to remove this assumption and show that the resulting translation preserves derivability in a sense relevant to the logic programming interpretation; an important part of our proof is showing how to extract an LF object satisfying a type from a derivation constructed using the *hohh* version of the specification. Our first encoding may include redundant type-checking judgments which obscure the translated specification and can

PPDP 2010

result in poor execution behavior. We design conditions for eliminating some of these judgments, resulting in an improved translation that corresponds closely to the intention of the orginal LF specification. This part of our work relies on an analysis of the structure of LF expressions and also has relevance, for example, to providing compact representations of proof terms. Finally, we demonstrate that the execution of the translated form by means of the Teyjus implementation [9] of λProlog [15] provides an effective means for animating Twelf programs.

In the next two sections, we describe a relevant fragment of the *hohh* logic and the Twelf specification language. Section 4 then presents our first translation; a full proof of its correctness appears in Appendix A. In the following section, we describe and exploit a property of LF expressions and type-checking to refine the earlier translation to produce a more efficient and transparent version. Section 6 provides experimental data towards supporting the use of this translation as a means for executing Twelf programs. We conclude the paper with a discussion of related work and possible future directions.

2. A HIGHER-ORDER PREDICATE LOGIC FOR DESCRIBING COMPUTATIONS

The logic of *hohh* formulas is based on an intuitionistic version of Church's simple theory of types [4]. Both logics are built over a typed form of the λ-calculus. The types are constructed using →, the infix, right associative function type constructor, starting from a finite collection of atomic types that includes o, the type of propositions, and at least one other type.[1] We assume that we are given sets of variables and constants, each with an associated type. 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 between terms that is generated by β- and η-reduction. It is well-known that every term has a unique normal form under these reduction operations in this simply-typed setting. All terms are to be converted into such a form prior to their consideration in any context. 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 *nonlogical* ones that constitute a *signature* and logical ones. We do not permit o to appear in the type of the arguments of nonlogical constants and variables. The logical constants are restricted to $\top$ of type o, $\supset$ of type $o \to o \to o$ that is written in the customary infix form and, for each type α, Π of type $(\alpha \to o) \to o$. 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 nonlogical 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:

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

A specification or logic program is a finite collection of closed D-formulas that are also called *program clauses* and a goal or a query is a closed G-formula.

Computation corresponds to searching for a derivation of a sequent of the form $\Sigma; \Gamma \longrightarrow G$ where Σ is the initial (language) signature, Γ is a logic program and G is a goal. Figure 1 presents the rules for constructing such a derivation. Read in a proof search direction, the $\forall R$ rule leads to an expansion of the signature in the sequent whose derivation is sought and the $\supset R$ rule similarly causes an addition to the logic program. The expression "t is a Σ-term" in the $\forall L$ rule means that t is a closed term all of whose nonlogical 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 the *decide* rule 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.[2] 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 rule

$$\frac{\Sigma; \Gamma \longrightarrow F_1' \qquad \ldots \qquad \Sigma; \Gamma \longrightarrow F_n'}{\Sigma; \Gamma \longrightarrow A}\ \textit{backchain}$$

which 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 $F_i[t_1/x_1, \ldots, t_i/x_i]$.

The logic that we have described has been given an efficient implementation in the *Teyjus* system [9]. It is possible also to reason in sophisticated ways about specifications that are constructed using it. To begin with, the logic has strong meta-theoretic properties arising from the fact that derivability in it corresponds exactly to intuitionistic provability. Moreover, it is possible to construct logics incorporating mechanisms such as induction to reason powerfully about what does and does not follow from a given specification [1, 8, 10, 14]. In fact, systems such as Abella [7] and Tac [2] have been constructed to provide computer support for such reasoning.

3. LOGIC PROGRAMMING USING THE TWELF SPECIFICATION LANGUAGE

There are three categories of expressions in LF: *kinds*, *types* or *type families* that are classified by kinds and *objects*

[1]Other, non-interpreted type constructors can be added but are not discussed here for simplicity.

[2]For the reader unfamiliar with such presentations, the expression $\Sigma; \Gamma \xrightarrow{D} A$ corresponds essentially to the selection of the program clause D as the one to backchain on. This then leads to instantiations of universally quantified variables and to the solution of the "body" goals of the clause using the rules $\forall L$ and $\supset L$, culminating eventually in solving the atomic goal by matching it with the head of the clause using the *init* rule.

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

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

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

Figure 1: Derivation rules for the *hohh* logic

or *terms* that are classified by types. We assume two denumerable sets of variables, one for objects and the other for types. We use x and y to denote object variables, u and v to denote type variables and w to denote either. Letting K range over kinds, A and B over types, and M and N over object terms, the syntax of LF expressions is given by the following rules:

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

Expressions of any of these kinds will be denoted by P and Q. Here, Π and λ are operators that associate a type with a variable and bind its free occurrences over the expression after the period. Terms that differ only in the names of bound variables are identified. As with the *hohh* logic, $P[N_1/x_1,\ldots,N_n/x_n]$ denotes a simultaneous substitution with renaming to avoid variable capture. We write $A \rightarrow P$ for $\Pi x{:}A.P$ when x does not appear free in P. We abbreviate $\Pi x_1{:}A_1.\ldots\Pi x_n{:}A_n.P$ by $\Pi\overrightarrow{x{:}A}.P$.

LF expressions are equipped with a notion of β-reduction defined through the rule $(\lambda x{:}A.P)\ N \rightarrow_\beta P[N/x]$. All LF expressions that are well-formed in the sense formalized below are strongly normalizing under this reduction relation [11]. 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.

The type correctness of LF expressions is assessed relative to contexts that are finite collections of assignments of types and kinds to variables. Formally, contexts, denoted by Γ, are given by the rule

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

Here, $\cdot$ denotes the empty collection. We write $\mathit{dom}(\Gamma)$ to denote the variables with assignments in Γ, and we write $\Gamma\backslash y$ to indicate the context Γ with the binding for y removed. We are concerned with assertions of the following four forms:

$$\vdash \Gamma\ \mathit{ctx} \qquad \Gamma \vdash K\ \mathit{kind} \qquad \Gamma \vdash A : K \qquad \Gamma \vdash M : A$$

The first assertion signifies that Γ is a well-formed context. The remaining assertions mean respectively that, relative to a (well-formed) context Γ, K is a well-formed kind, A is a well-formed type of kind K and M is a well-formed object of type A. Figure 2 presents the rules for deriving such assertions. 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. The 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 $\Gamma \vdash A : K$.

A variable w that appears in an LF expression P that is well-formed with respect to a context Γ has a kind or type of kind *Type* associated with it through either an assignment in Γ or a binding operator. 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 w is *fully applied* if it appears in a subterm of the form $w\ M_1\ \ldots\ M_n$. 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. A well-formed type of the form $u\ M_1\ \ldots\ M_n$ 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.

In later sections we shall consider LF derivations in which all expressions in the end assertion are in normal form. Notice that every expression in the entire derivation must then also be in such a form. This in turn means that 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.

The following "transitivity" property for LF derivations that follows easily from the results in [11] will be useful later; here α stands for any judgment, and substitution and normalization over α and Γ corresponds to distributing these operations to the expressions appearing in them.

Proposition 1 (Substitution). *Let Γ_1, Γ_2 be canonical contexts, and A be a type in canonical 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 LF derivations, which follows from Proposition 1.

Proposition 2 (Renaming). *Let P be a canonical type or kind, $\Gamma = \Gamma_1, x : P, \Gamma_2$ be a canonical context, and α a canonical judgment. 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.*

The logic programming interpretation of LF is based on viewing types as formulas. More specifically, a specification or program in this setting is given by a context. This starting context, also called a *signature*, essentially describes the vocabulary for constructing types and asserts the existence

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

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

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

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

$$\frac{\vdash \Gamma\ \mathit{ctx} \quad u : K \in \Gamma}{\Gamma \vdash u : K^{\beta}}\ \textit{var-fam} \qquad \frac{\vdash \Gamma\ \mathit{ctx} \quad x : A \in \Gamma}{\Gamma \vdash x : A^{\beta}}\ \textit{var-obj}$$

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

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

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

Figure 2: Rules for Inferring LF Assertions

of particular inhabitants for some of these types. Against this backdrop, questions can be asked about the existence of inhabitants for certain other types. Formally, this amounts to asking if an assertion of the form $\Gamma \vdash M : A$ has a derivation. However, the object M is left unspecified—it is to be extracted from a successful derivation. Thus, the search for a derivation of the assertion is driven by the structure of A and the types available from the context.

A concrete illustration of the paradigm is useful for later discussions.[3] Consider a signature or context Γ comprising the following assignments in sequence:

$$\begin{array}{l}
\mathit{nat} : \mathit{Type} \\
z : \mathit{nat} \\
s : \mathit{nat} \rightarrow \mathit{nat} \\
\mathit{list} : \mathit{Type} \\
\mathit{nil} : \mathit{list} \\
\mathit{cons} : \mathit{nat} \rightarrow \mathit{list} \rightarrow \mathit{list} \\
\mathit{append} : \mathit{list} \rightarrow \mathit{list} \rightarrow \mathit{list} \rightarrow \mathit{Type} \\
\mathit{appNil} : \Pi K{:}\mathit{list}.\mathit{append}\ \mathit{nil}\ K\ K \\
\mathit{appCons} : \Pi X{:}\mathit{nat}.\Pi L{:}\mathit{list}.\Pi K{:}\mathit{list}.\Pi M{:}\mathit{list}. \\
\qquad\quad (\mathit{append}\ L\ K\ M) \rightarrow \\
\qquad\qquad\quad (\mathit{append}\ (\mathit{cons}\ X\ L)\ K\ (\mathit{cons}\ X\ M))
\end{array}$$

We can ask if there is some term M such that the judgment

$$\begin{array}{l}
\Gamma \vdash M : \mathit{append}\ (\mathit{cons}\ z\ \mathit{nil}) \\
\qquad\qquad\qquad\quad (\mathit{cons}\ (s\ z)\ \mathit{nil}) \\
\qquad\qquad\qquad\quad (\mathit{cons}\ z\ (\mathit{cons}\ (s\ z)\ \mathit{nil}))
\end{array}$$

is derivable. Assuming that Γ is given by the ambient environment, such a query can be posed in Twelf simply by presenting the type expression. The logic programming interpreter of Twelf will find that the proof term

$$\begin{array}{l}
(\mathit{appCons}\ z\ \mathit{nil}\ (\mathit{cons}\ (s\ z)\ \mathit{nil}) \\
\qquad\qquad\quad (\mathit{cons}\ (s\ z)\ \mathit{nil}) \\
\qquad\qquad\quad (\mathit{appNil}\ (\mathit{cons}\ (s\ z)\ \mathit{nil})))
\end{array}$$

inhabits this type and hence will succeed on the query. In reaching this conclusion, the interpreter will use the types involving *append* that are present in Γ. Further it will do this in a way that bears a close resemblance to the use of clauses in a Prolog-like setting, interpreting Π like a universal quantifier and $\rightarrow$ like an implication.

The simple example we have considered here will suffice to illustrate most of the later ideas in this paper but it does not bring out the richness of dependent types in specifications. We leave this demonstration to the many discussions already in the literature. We also note that Twelf has many additional features like allowing Π quantification in types to be left implicit and permitting instantiatable variables in queries whose values are to be found through unification. While these aspects are treated in our implementation, to keep the theoretical discussions focused, we shall assume that the only capability that is to be emulated is that of determining the derivability of an assertion of the form $\Gamma \vdash M : A$ in which Γ and A are in canonical form (and M is left unspecified). This assumption is easily justified: these will be "type-checked" prior to conducting a search and the Twelf system assumes equality under η-conversion.

4. FROM TWELF SPECIFICATIONS TO PREDICATE FORMULAS

Felty has previously shown how to translate LF specifications and judgments into *hohh* formulas [5, 6]. Her translation proceeds in two steps. First, she describes a coarse mapping of LF expressions into (simply typed) λ-terms. This mapping loses information about dependencies in types and kinds and also does not reflect the correspondences between objects and types and types and kinds. These relationships are encoded later through binary predicates over λ-terms.

The general structure of Felty's translation is applicable in the context of interest to us. However, the details of her

[3] The example of appending lists has been chosen here for its conciseness and because it allows for an easy connection with more traditional forms of logic programming. The primary application domain of Twelf is in specifying (and reasoning about) formal systems such as evaluators and interpreters for languages, type assignment calculi and proof systems. This orientation informs the choice of benchmarks used in Section 6.

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

$$\langle u\ M_1 \ldots M_n\rangle := u\ \langle M_1\rangle \ldots \langle M_n\rangle$$
$$\langle x\ M_1 \ldots M_n\rangle := x\ \langle M_1\rangle \ldots \langle M_n\rangle \qquad \langle \lambda x{:}A.M\rangle := \lambda^{\phi(A)} x.\langle M\rangle$$

$$\{\!\{\Pi x{:}A.B\}\!\} := \lambda M.\ \forall x.\ (\{\!\{A\}\!\}\ x) \supset (\{\!\{B\}\!\}\ (M\ x))$$
$$\{\!\{A\}\!\} := \lambda M.\ \mathit{hastype}\ M\ \langle A\rangle \text{ where } A \text{ is a base type}$$

Figure 3: Encoding of types, objects, and simplified translation of LF judgments to *hohh*

$\mathit{hastype}\ z\ \mathit{nat}$
$\forall n.\ \mathit{hastype}\ n\ \mathit{nat} \supset \mathit{hastype}\ (s\ n)\ \mathit{nat}$
$\mathit{hastype}\ \mathit{nil}\ \mathit{list}$
$\forall n.\ \mathit{hastype}\ n\ \mathit{nat} \supset \forall l.\ \mathit{hastype}\ \ l\ \mathit{list} \supset \mathit{hastype}\ (\mathit{cons}\ n\ l)\ \mathit{list}$
$\forall l.\ \mathit{hastype}\ l\ \mathit{list} \supset \mathit{hastype}\ (\mathit{appNil}\ l)\ (\mathit{append}\ \mathit{nil}\ l\ l)$
$\forall x.\ \mathit{hastype}\ x\ \mathit{nat} \supset \forall l.\ \mathit{hastype}\ l\ \mathit{list} \supset \forall k.\ \mathit{hastype}\ k\ \mathit{list} \supset \forall m.\ \mathit{hastype}\ m\ \mathit{list} \supset$
$\quad \forall a.\ \mathit{hastype}\ a\ (\mathit{append}\ l\ k\ m) \supset \mathit{hastype}\ (\mathit{appCons}\ x\ l\ k\ m\ a)\ (\mathit{append}\ (\mathit{cons}\ x\ l)\ k\ (\mathit{cons}\ x\ m))$

Figure 4: Simple translation of the LF specification for *append*

mapping do not quite fit our needs because of her focus on *derivations* in the LF and *hohh* logics. One manifestation of this is that her translation is not based exclusively on types, but assumes also the availability of the objects they are intended to qualify. This is not acceptable in the context of proof search where the task is precisely to determine the existence of those objects: we need a translation that is only based on the type, and which can be applied to an *hohh* metavariable to correspond to an LF query whose object is left unspecified as a metavariable. Second, the correctness result only states an equivalence between LF derivability and *hohh* derivability for *known* LF assertions, and does not consider, for example, whether it is possible for non-canonical or ill-formed objects to be produced in the course of searching for proofs from the *hohh* specification. In contrast, our completeness result will guarantee that after running a query with a metavariable standing for the (encoding of the) object, the only possible instantiations of that metavariable are actual encodings of terms.

The first step towards producing a translation into *hohh* that can be used to interpret Twelf specifications is to adapt Felty's translation in a way that makes it acceptable in logic programming discussions. Our translation shall only account for judgments of the form $\Gamma \vdash M : A$ since these are the only ones of interest in the logic programming setting described in the previous section. The adequacy of this restriction actually relies on an auxiliary, easily verified, observation: if $\Gamma \vdash A : \mathit{Type}$ is known to have a derivation and the last rule in a purported derivation of $\Gamma \vdash M : A$ is an *abs-obj*, then the left premise for the latter derivation must have a proof and hence does not need to be encoded by the translation.

Our translation is presented in Figure 3. This translation first encodes LF objects and types in *hohh* terms by dropping a lot of typing information; as mentioned already, this information will be recovered later in the encoding of LF judgments. Under this translation, 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 or base type Q is then given by $\langle Q\rangle$; note that in the process we assume a reuse of (LF) variable names with an appropriate type as part of the corresponding *hohh* signature. As an example, the LF signature at the end of the last section leads to the following *hohh* signature:

$\mathit{nat} : \textit{lf-type}$
$z : \textit{lf-obj}$
$s : \textit{lf-obj} \to \textit{lf-obj}$
$\mathit{list} : \textit{lf-type}$
$\mathit{nil} : \textit{lf-obj}$
$\mathit{cons} : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj}$
$\mathit{append} : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-type}$
$\mathit{appNil} : \textit{lf-obj} \to \textit{lf-obj}$
$\mathit{appCons} : \textit{lf-obj} \to \textit{lf-obj} \to$
$\qquad \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj}$

Further, 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 (types) of the same type (kind). Finally, we take up the translation of LF type assignments and judgments in the last two clauses in Figure 3. To emphasize reliance only on the structure of types, these clauses describe explicitly only the translation of an LF type A. Such a type is mapped into an *hohh* predicate denoted by $\{\!\{A\}\!\}$ that, intuitively, codifies the property of being a translation of an LF object of type A. This translation is defined on all canonical types and uses the *hohh* predicate *hastype* of type $\textit{lf-obj} \to \textit{lf-type} \to o$. If A is a base type, $\{\!\{\Pi x_1{:}B_1.\ldots\Pi x_n{:}B_n.A\}\!\}$ has type $\tau \to o$ where τ is $\textit{lf-obj} \to \ldots \to \textit{lf-obj} \to \textit{lf-obj}$ with n negative occurrences of *lf-obj*. Once the translation of LF types is in place, we define $\{\!\{M : A\}\!\}$ derivatively to be $(\{\!\{A\}\!\}\ \langle M\rangle)$.

Twelf specifications are encoded by dropping all kind assignments and translating each type assignment they contain. As an example, the Twelf specification of *append* translates into the clauses in Figure 4. From these clauses, we can, for example, derive the goal

$$\mathit{hastype}\ (\mathit{cons}\ (s\ z)\ \mathit{nil})\ \mathit{list}$$

and we could search for terms X satisfying the goal

$$\begin{array}{ll} \textit{hastype}\ X\ (\textit{append}\ (\textit{cons}\ z\ \textit{nil}) \\ \qquad\qquad\qquad\qquad (\textit{cons}\ (s\ z)\ \textit{nil}) \\ \qquad\qquad\qquad\qquad (\textit{cons}\ z\ (\textit{cons}\ (s\ z)\ \textit{nil}))). \end{array}$$

Let Γ' be the translation of an LF context Γ and α' be the translation of the LF judgment α. These translations are based on an implicit *hohh* signature Σ. In the case that all the free variables in α belong to $dom(\Gamma)$, then, in fact, Σ consists of an isomorphic copy of the symbols in $dom(\Gamma)$. Henceforth, we shall assume Σ to be just such an *hohh* signature and we shall write $\Gamma' \longrightarrow \alpha'$ as a shorthand for $\Sigma; \Gamma' \longrightarrow \alpha'$. The correctness of the (simple) translation is then the content of the following theorem.

THEOREM 1. *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 $\{\!\{\Gamma\}\!\} \longrightarrow (\{\!\{A\}\!\}\ M)$ has a derivation for any hohh term M of appropriate type, then there is a canonical LF object M' such that $M = \langle M' \rangle$ and $\Gamma \vdash M' : A$ has a derivation.*

Proof outline.
Completeness can be proved by a simple induction on the LF derivation, building an *hohh* derivation that mimics its structure. Soundness is more involved: we proceed by induction on the *hohh* derivation, gradually recovering the structure of M', maintaining the derivability of $\Gamma \vdash A : Type$ that allows us to build an LF derivation even in the case that *abs-obj* was the last rule used. The detailed proof is presented in Appendix A.

The simple translation presented in this section cannot be the basis of a practical implementation of logic programming in LF. Proof search using a program it 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 from the example in Figure 4: at every step of deriving an instance of *append*, the lists must be checked to be well-typed, which artificially introduces a quadratic complexity. An important point to note, however, is that 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 terms. It is this issue that we take up in the next section.

5. AN IMPROVED TRANSLATION OF TWELF SPECIFICATIONS

In order to make the translation of LF specifications into *hohh*, and, in particular, into λProlog, practical from an implementation standpoint, we make two optimizations.

The main optimization exploits the fact that we are considering derivations of the form $\Gamma \vdash M : A$ where Γ and A have already been type-checked. For example, suppose we are trying to determine whether the LF type

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

is inhabited. Before we embark on this task, we would have already determined that *append* (*cons z nil*) *nil* (*cons z nil*) is a valid type, which means, for instance, that we would

$$\frac{\Gamma; \cdot; x \sqsubset_o A_i \text{ for some } A_i}{\Gamma; x \sqsubset_t c\,\overrightarrow{A}}\ \mathrm{APP_t}$$

$$\frac{y_i \in \delta \text{ for each } y_i \qquad y_i \text{ distinct}}{\Gamma; \delta; x \sqsubset_o x\ \overrightarrow{y}}\ \mathrm{INIT_o}$$

$$\frac{\Gamma, y; x \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.B}\ \mathrm{PI_t}$$

$$\frac{y \notin \Gamma \text{ and } \Gamma; \delta; x \sqsubset_o M_i \text{ for some } i}{\Gamma; \delta; x \sqsubset_o y\ \overrightarrow{M}}\ \mathrm{APP_o}$$

$$\frac{\Gamma; \delta, y; x \sqsubset_o M}{\Gamma; \delta; x \sqsubset_o \lambda y{:}A.M}\ \mathrm{ABS_o}$$

Figure 5: Rigidly occurring variables in types and objects

have checked that (*cons z nil*) is a valid object of type *list*. Therefore, there is no need to show again that (*cons z nil*) has this property in the course of searching for an inhabitant of the displayed type. Our optimized translation tries to take advantage of this kind of observation to remove some checks of well-typedness in the translation of LF types. More specifically, our optimization is based on the following idea. Suppose that for any instantiation $t_1, \ldots, t_n$ of the type $\Pi x_1{:}A_1. \ldots \Pi x_n{:}A_n.B$ we can determine that a particular t_i must always appear in the type $B[t_1/x_1, \ldots, t_n/x_n]$. Then, the translation of this type need not include explicit type-checking over the instantiation of x_i. Now, one way to determine the requisite property of instantiations of x_i is to look at the occurrences of x_i in B. Formally, we use the notion of a *rigid occurrence* that is expressed by the judgment $\overrightarrow{x}; x_i \sqsubset_t B$ defined in Figure 5 to characterize some of these cases; the rules $\mathrm{APP_t}$ and $\mathrm{PI_t}$ in this figure act on LF types, and the rules $\mathrm{INIT_o}$, $\mathrm{APP_o}$, and $\mathrm{ABS_o}$ act on LF objects. By allowing some type checking to be elided, this enhancement to the earlier simple translation does not only yield an efficiency benefit, but also makes the specification more readable, and effectively makes the intended logic programming behavior of the produced *hohh* formula similar to the original LF type.

The second optimization is more transparent, not depending on deeper properties of dependent types. The essential observation is the following. Instead of producing predicates of the form *hastype X* (*append L K M*) and *hastype L list*, we can specialize them to *append X L K M* and *list L*. This results in a *hohh* program that is much clearer, and more closely related to the original LF specification. Moreover, this simple transformation can also lead to better performance in a logic programming setting because it allows for the exploitation of a common optimization, namely, the indexing on a predicate name that speeds up the determination of candidate clauses on which to backchain.

The resulting optimized translation is presented on Figure 6. 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. This translation is illustrated by its application to the example Twelf specification considered in Section 3 that yields the clauses shown in Figure 7. These clauses should be contrasted with the ones in Figure 4 that are produced by the earlier, naive translation.

We shall now establish the correctness of the optimized translation, by relating it to the simple translation. We first prove a fundamental lemma concerning the type checking of instantiations of rigidly occurring variables. In order to be able to use this observation in our correctness argument, we formulate the lemma in a rather technical way, talking about encoded types that are the result of instantiations of (a priori) arbitrary *hohh* terms in encoded types. We stress that these technical details concerning encodings are shallow, and that the real result is one about type checking and redundancy concerning LF expressions that the reader may find informative to extract.

DEFINITION 1. *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 $(M \sim N)[t_1/x_1 \dots t_n/x_n]$ when $\langle M\rangle = \langle N\rangle[t_1/x_1 \dots t_n/x_n]$. For LF types A and B, we write $(A \sim B)[t_1/x_1 \dots t_n/x_n]$ when the two types are equal up to $(\bullet \sim \bullet)[t_1/x_1 \dots t_n/x_n]$ 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 $P \sim Q$.*

LEMMA 1. *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 $\Gamma_0 = x_1 : B_1, \dots, x_n : B_n$.*

1. *Let Γ and Δ be LF contexts, M an LF object and A a type, all being assumed canonical. Let δ be dom(Δ). Suppose that there are derivations of $\overrightarrow{x}; \delta; x_i \sqsubset_o M$, $\Gamma, \Gamma_0, \Delta \vdash M : A$ and $\Gamma, \Delta' \vdash M' : A'$, 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 $\Gamma \vdash t'_i : B_i[t'_1/x_1, \dots, t'_{i-1}/x_{i-1}]$.*

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

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 INIT$_o$, $M = x_i\overrightarrow{y}$ where $\overrightarrow{y}$ are distinct bound variables from δ. The derivation $\mathcal{D}$ must consist of n *app-obj* rules and a *var-obj* rule on x_i, whose type B_i must be of the form $\Pi\overrightarrow{z{:}C}.D$, with $A = D[\overrightarrow{y}/\overrightarrow{z}]$. 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 subderivations 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 subderivations 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 is of the form $\lambda z_1 \dots \lambda z_n.u$. We have
$$\langle M'\rangle = t_i\overrightarrow{y} = u[\overrightarrow{y}/\overrightarrow{z}],$$
hence $u = \langle M'\rangle[\overrightarrow{z}/\overrightarrow{y}] = \langle M'[\overrightarrow{z}/\overrightarrow{y}]\rangle$ (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
$$t'_i = \lambda\overrightarrow{z{:}C'}.u' \text{ where } u' = M'[\overrightarrow{z}/\overrightarrow{y}]$$
We have
$$\begin{aligned}\langle t'_i\rangle &= \lambda z_1 \dots \lambda z_n.\ \langle M'\rangle[\overrightarrow{z}/\overrightarrow{y}]\\ &= \lambda z_1 \dots \lambda z_n.\ u = t_i.\end{aligned}$$
We also have $M' = t'_i\ y_1 \dots y_n = u'[\overrightarrow{y}/\overrightarrow{z}]$.

We know that $\mathcal{D}'$ derives $\Gamma, \Delta' \vdash M' : A'$. From this we obtain a derivation of
$$\Gamma, \Delta'[\overrightarrow{z}/\overrightarrow{y}] \vdash u' : A'[\overrightarrow{z}/\overrightarrow{y}]$$
by renaming variables $\overrightarrow{y}$ into $\overrightarrow{z}$, employing Proposition 2. The context $\Delta'[\overrightarrow{z}/\overrightarrow{y}]$ contains assignments $(z_i : C'_i)$ and the other variables in its domain do not occur in u' nor $A'[\overrightarrow{z}/\overrightarrow{y}]$ (since $A' \sim A$, $A = D[\overrightarrow{y}/\overrightarrow{z}]$ and D is a subterm of B_i which cannot contain any y_i). We then have
$$\Gamma \vdash (\lambda\overrightarrow{z{:}C'}.u') : (\Pi\overrightarrow{z{:}C'}.A'[\overrightarrow{z}/\overrightarrow{y}])$$
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 $B_i[t'_1/x'_1 \dots t'_{i-1}/x_{i-1}]$.

- In the ABS$_o$ case, we have $M = \lambda y{:}A_1.N$ and $\mathcal{D}$ ends with the *abs-obj* rule as follows:
$$\frac{\Gamma, \Gamma_0, \Delta \vdash A_1 : Type \quad \Gamma, \Gamma_0, \Delta, y : A_1 \vdash N : A_2}{\Gamma, \Gamma_0, \Delta \vdash (\lambda y{:}A_1.N) : (\Pi y{:}A_1.A_2)}$$
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.

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

Figure 6: Optimized translation of LF specifications and judgments to *hohh*

$$\begin{array}{l}
nat\ z \\
\forall n.\ nat\ n \supset nat\ (s\ n) \\
list\ nil \\
\forall n.\ nat\ n \supset \forall l.\ list\ l \supset list\ (cons\ n\ l) \\
\forall l.\ \top \supset append\ (appNil\ l)\ nil\ l\ l \\
\forall x.\ \top \supset \forall l.\ \top \supset \forall k.\ \top \supset \forall m.\ \top \supset \\
\qquad \forall a.\ append\ a\ l\ k\ m \supset append\ (appCons\ x\ l\ k\ m\ a)\ (cons\ x\ l)\ k\ (cons\ x\ m)
\end{array}$$

Figure 7: Optimized translation of the LF specification for *append*

- In the APP$_o$ 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 rule APP$_t$) 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 object is 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). $\square$

The definition of rigidity described above might seem restrictive. In particular, one might want to allow

$$\Gamma; \delta; x \sqsubset_o x\overrightarrow{N}$$

in INIT$_o$. However, with such a rule the rigidity lemma described above is no longer true. For example, in a signature with $num : nat \to \mathit{Type}$ and $num_n : \Pi n{:}nat.(num\ n)$, the object $t = num_n$ provides a counter-example to Lemma 1, part (1): we have $\Gamma \vdash (t\ z) : (num\ z)$ and

$$\Gamma, x : (nat \to num\ z) \vdash (x\ z) : (num\ z)$$

but not $\Gamma \vdash t : nat \to num\ z$.

THEOREM 2. *Let Γ be an LF context, A an LF type, both canonical, such that $\vdash \Gamma$ ctx and $\Gamma \vdash A : \mathit{Type}$ 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:

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

First, $\Gamma \vdash B : \mathit{Type}$, $\vdash (\Gamma, x : B)$ *ctx* and $\Gamma, x : B \vdash A' : \mathit{Type}$ must have derivations since Γ and A are well-formed. 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.*, on

$$\begin{array}{l} \forall x_1.\ ([\![B_1]\!]^-(x_1) \supset \ldots \supset \\ \qquad \forall x_n.\ ([\![B_n]\!]^-(x_n) \supset (u\ (y\ \overrightarrow{x})\ \overrightarrow{\langle N\rangle})))). \end{array}$$

In particular, this rule application has the form

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

where 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 $i \le n$, showing that for all $j \le i$, $t_j = \langle t'_j\rangle$ 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 $\Gamma \vdash t'_j : B_j[t'_1/x_1, \ldots, t'_{j-1}/x_{j-1}]$.

- 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 1, 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 by Theorem 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 1) 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 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

$$\{\!\{\Gamma\}\!\} \longrightarrow \mathit{hastype}\ (y\,\overrightarrow{t})\ (u\ \overrightarrow{\langle N \rangle})[\overrightarrow{t/x}]$$

Completeness is proved by an induction on the derivation of the simple translation. This direction is rather straightforward as it consists only of dropping information. Details can be found in Appendix A. $\square$

Therefore, by Theorems 1 and 2, intuitionistic provability under the optimized translation is equivalent to provability in LF, and the following is a theorem.

THEOREM 3 (OPTIMIZED TRANSLATION CORRECTNESS). *Let Γ be an LF specification such that $\vdash \Gamma\ ctx$ has a derivation, A an LF type such that $\Gamma \vdash A : \mathit{Type}$ 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.*

6. PERFORMANCE COMPARISONS

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 *append* specification presented in Figure 7, especially when one uses the easily applied simplification of a formula of the form $\top \supset F$ to F. Notice also the correspondence of the definition of the *append* predicate to the one that one might in, *e.g.*, Prolog, if one drops the first "proof term" argument of the predicate. To fully appreciate this benefit, it is necessary to consider larger examples that space does not allow us to do in this paper. However, such examples are available with the implementation [21]. We suggest that the reader look especially at the example of the evaluator for Mini-ML with terms that are not indexed by their type that is described below in the collection of benchmarks: the translation results in an *hohh* program that is what one might write in *hohh* directly.

To test the second claim, we have carried out performance comparisons between the Twelf implementation that interprets LF specifications directly via a Standard ML program and an implementation obtained by translating these specifications into *hohh* programs and then executing these using the Teyjus system. We present results here over programs that have a few different characteristics:

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

- The encoding of evaluators for various languages is a common usage of LF. We have therefore used an encoding of Mini-ML along with an encoding of addition as another sample program. This benchmark, referred to as *miniml*, consists of adding n to 10 using the encoding.

- 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 [19]. This benchmark, called *perm*, tests list permutation encoded in INCLL and run using the meta-interpreter for lists of length n.

- The last benchmark, referred to as *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.

The third through fifth columns of Figure 8 present data comparing the simple translation, the translation with redundant typing judgments removed, and the fully optimized translation against the standard of Twelf with default optimizations on these benchmarks.[4] As described in Figure 6, the fully optimized translation inserts the proof term as the first argument of the predicate generated. Since this term is to be determined by proof search, advantage cannot be taken of the capability Teyjus possesses of indexing on the first argument. The last column presents data for the case where we make the proof term the last argument instead. 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 leads to better performance in most cases, often significantly so. On the other hand, the simple 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. However, the simple translation is still comparable to Twelf on the first

[4]This setting with Twelf leads to the best performance on these examples.

Example	Twelf	Simple	Optimized	Typed Optimized	Indexing
reverse(10)	1.0	0.40	0.14	0.07	0.08
reverse(20)	1.0	0.57	0.19	0.12	0.11
reverse(30)	1.0	0.63	0.20	0.14	0.11
reverse(40)	1.0	0.41	0.13	0.10	0.07
reverse(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 8: Performance comparison results

three benchmarks. On the *perm* benchmark, Twelf does quite well and even out-performs Teyjus with the optimized translation on problems of large size. 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 that delays expensive unification computation till after simpler checks have been made may have something to do with this. The fact that term indexing causes significant improvement with Teyjus gives credence to this observation.

For problems of very large size with all the benchmarks, the performance of Twelf deteriorates quite dramatically; this is seen, for example, in the case of *num(n)* for a problem of size 512. This phenomenon is linked to the fact that Twelf consumes excessive amounts of memory. The ultimate source of this problem is perhaps the fact that Twelf is implemented in SML: 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 [3].

7. CONCLUSION AND FUTURE WORK

We have considered in this paper a translation of Twelf specifications into logic programs in the *hohh* language. An important part of our ideas is the recognition of certain situations in which type information is redundant in LF expressions and hence its checking can be avoided. Our eventual translation produces a program that corresponds closely to the original specification and we have argued that it can be the basis for an effective animation of Twelf descriptions.

The specific work undertaken here can be extended in a few different ways. As an extension to our notion of rigidity, we might observe that, when applying a variable of type $\Pi \overrightarrow{x{:}B}.A$, we could identify redundant type information, not only between a B_i and A, but also between a B_i and a different B_j. It would also be interesting to relate our work to the ideas of Reed [20] who describes a notion of *strictness* similar to rigidity, used for the different purpose of identifying sub-terms of LF objects that could be reconstructed if elided – in contrast, we avoid redundant type checking but still generate a complete LF object. Such an understanding might lead both to an improvement of our translation and to the ability to shorten LF terms that are needed in applications such as that of proof-carrying-code [16]. 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: experiments in this direction indicate in some cases a ten-fold performance improvement over the optimized translation.

We have focused here on realizing Twelf through a translation to λProlog. A different approach, worthy of investigation, is that of compiling Twelf specifications directly to bytecode for the virtual machine underlying the Teyjus system. Such an approach would make it possible to realize optimizations that have been developed for the direct implementation of Twelf [18, 19]. Of special note here are optimizations like the linear heads treatment of unification described by Pientka and Pfenning [19] for minimizing occurs checking, that could make a difference in examples such as the *perm* program considered in the previous section: direct compilation would allow us to regain opportunities for such improvements that might be lost by translating first to λProlog and then relying on its implementation that is not specially optimized to treat Twelf-specific programs.

A more ambitious line of development concerns metareasoning over specifications. Existing tools might be used to reason about LF programs via the translation, the transparency of the translation becoming essential. Anecdotal evidence suggests that this transparency is not only enabling, it is also elucidating: that 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.

8. ACKNOWLEDGEMENTS

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.

9. REFERENCES

[1] D. Baelde. *A linear approach to the proof-theory of least and greatest fixed points.* PhD thesis, Ecole Polytechnique, Dec. 2008.

[2] D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. Accepted for publication at IJCAR'10, 2010.

[3] P. Brisset and O. Ridoux. The architecture of an implementation of lambda-prolog: Prolog/mali. In *ILPS Workshop: Implementation Techniques for Logic Programming Languages*, 1994.

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

[5] A. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, Aug. 1989.

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

[7] A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[8] A. Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems.* PhD thesis, University of Minnesota, 2009.

[9] A. Gacek, S. Holte, G. Nadathur, X. Qi, and Z. Snow. The Teyjus system – version 2, Mar. 2008. Available from `http://teyjus.cs.umn.edu/`.

[10] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] W. A. Howard. The formulae-as-type notion of construction. 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.

[13] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[14] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[15] G. Nadathur and D. Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Aug. 1988. MIT Press.

[16] G. 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.

[17] F. Pfenning and C. 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.

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

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

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

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

APPENDIX

A. PROOFS OF THEOREMS

A.1 Correctness of the simplified encoding (Theorem 1)

A.1.1 Completeness

We use induction on the derivation of $\Gamma \vdash M : A$ to build one for $\{\!\{\Gamma\}\!\} \longrightarrow \{\!\{M : A\}\!\}$. We proceed by case analysis on the canonical type A.

If A is of the form $\Pi x{:}B.A'$ then M must be of the form $\lambda x{:}B.M'$ and the LF derivation must end with an *abs-obj* rule, *i.e.*, a rule of the form

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

The induction hypothesis gives us a derivation for

$$\{\!\{\Gamma, x : B\}\!\} \longrightarrow \{\!\{M' : A'\}\!\}.$$

By applying the rules $\forall R$ and $\supset R$ to this, we get a derivation for $\{\!\{\Gamma\}\!\} \longrightarrow \forall x.\ \{\!\{x : B\}\!\} \supset \{\!\{M' : A'\}\!\}$. The righthand side of this sequent is 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 virtue of η-conversion.

If A is a base type then M must be of the form $x\ N_1\ \ldots\ N_n$ and the canonical LF derivation must end with a chain of *app-obj* rules following a *var-obj* rule that reveals that

$$x : \Pi y_1{:}B_1.\ldots\Pi y_n{:}B_n.A' \in \Gamma.$$

Moreover, A must be $A'[N_1/y_1, \ldots, N_n/y_n]$ and, from looking at the right upper premise of the *app-obj* rules, there must be shorter derivations of

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

for $1 \leq i \leq n$. By the induction hypothesis we obtain derivations $\mathcal{D}_i$ of $\{\!\{\Gamma\}\!\} \longrightarrow \{\!\{N_i : B_i[N_1/x_1, \ldots, N_{i-1}/x_{i-1}]\}\!\}$. Further, $\{\!\{\Gamma\}\!\}$ must contain

$$\forall y_1.\ (\{\!\{B_1\}\!\}\ y_1) \supset \ldots \supset \\ \forall y_n.\ (\{\!\{B_n\}\!\}\ y_n) \supset \mathit{hastype}\ (x\ y_1\ \ldots\ y_n)\ \langle A'\rangle,$$

i.e., the encoding of $x : \Pi y_1{:}B_1.\ldots\Pi y_n{:}B_n.A'$. By applying *backchain* on that clause, choosing $\langle N_i\rangle$ for y_i and using the derivations $\mathcal{D}_i$, we obtain a derivation of

$$\{\!\{\Gamma\}\!\} \longrightarrow \mathit{hastype}\ (x\ \langle N_1\rangle\ \ldots\ \langle N_n\rangle) \\ (\langle A'\rangle[\langle N_1\rangle/y_1, \ldots \langle N_n\rangle/y_n]).$$

The right side of this sequent is precisely

$$\{\!\{(x\ N_1\ \ldots\ N_n) : A'[N_1/y_1, \ldots, N_n/y_n]\}\!\}.$$

A.1.2 Soundness

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 $M = \langle M'\rangle$ for some canonical object M' and we build a derivation of $\Gamma \vdash M' : A$. A case analysis on the structure of the canonical type A will guide us.

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

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

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 $M\ x = \langle M'\rangle$ and that $\Gamma, x : B \vdash M' : A'$ is derivable for some canonical object M'. 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 the derivation we are considering is that of $\{\!\{\Gamma\}\!\} \longrightarrow \mathit{hastype}\ M\ \langle A\rangle$. This derivation must end in a *backchain* rule that uses some clause in $\{\!\{\Gamma\}\!\}$ of the form

$$\forall y_1.\ (\{\!\{B_1\}\!\}\ y_1) \supset \ldots \supset \\ \forall y_n.\ (\{\!\{B_n\}\!\}\ y_n) \supset \mathit{hastype}\ (x\ y_1\ \ldots\ y_n)\ \langle A'\rangle;$$

note that the variables $y_1, \ldots, y_{i-1}$ can appear in $\{\!\{B_i\}\!\}$ here. Thus, for some *hohh* terms $N_1, \ldots, N_n$,

$$\langle A\rangle = \langle A'\rangle[N_1/y_1, \ldots, N_n/y_n],$$

$M = (x\ N_1\ \ldots\ N_n)$, and, for each i such that $1 \leq i \leq n$, there is a shorter derivation of

$$\{\!\{\Gamma\}\!\} \longrightarrow (\{\!\{B_i\}\!\}\ y_i)[N_1/y_1, \ldots, N_i/y_i],$$

i.e., of $\{\!\{\Gamma\}\!\} \longrightarrow (\{\!\{B_i\}\!\}[N_1/y_1, \ldots, N_{i-1}/y_{i-1}]\ N_i)$. Further, we know that $x : \Pi y_1{:}B_1.\ldots\Pi y_n{:}B_n.A' \in \Gamma$ for some x. We now claim that, for $1 \leq i \leq n$, $N_i = \langle N'_i\rangle$ for some canonical LF object N'_i and that $\Gamma \vdash N'_i : B_i[N'_1/y_1 \ldots N'_{i-1}/y_{i-1}]$ has a derivation. If this claim is true, then, we can use the *var-obj* rule to derive $\Gamma \vdash x : \Pi y_1{:}B_1.\ldots\Pi y_n{:}B_n.A'$ and follow this by a sequence of *app-obj* rule applications to prove $\Gamma \vdash (x\ N'_1\ \ldots\ N'_n) : A'[N'_1/y_1 \ldots N'_n/y_n]$. Now, evidently $M = \langle x\ N'_1\ \ldots\ N'_n\rangle$ and, since substitution permutes with encoding, $A = A'[N'_1/y_1, \ldots, N'_n/y_n]$. Thus, the desired result would be proven.

It only remains, then, to establish the claim. We actually strengthen it to include also the assertion that, for $1 \leq i \leq n$, $\Gamma \vdash B_i[N'_1/y_1 \ldots N'_{i-1}/y_{i-1}] : \mathit{Type}$ has a derivation. To prove it, we use an inner induction on i. Since Γ is a well-formed context, and $x : \Pi y_1{:}B_1.\ldots\Pi y_n{:}B_n.A' \in \Gamma$, there must be a derivation of

$$\Gamma, x_1 : B_1, \ldots, x_{i-1} : B_{i-1} \vdash B_i : \mathit{Type}$$

for $1 \leq i \leq n$. Using Proposition 1 and the induction hypothesis we see that there must be a derivation of

$$\Gamma \vdash B_i[N'_1/y_1 \ldots N'_{i-1}/y_{i-1}] : \mathit{Type}.$$

Noting that

$$\{\!\{B_i\}\!\}[N_1/y_1, \ldots, N_{i-1}/y_{i-1}] = \{\!\{B_i[N_1/y_1, \ldots, N_{i-1}/y_{i-1}]\}\!\},$$

the outer induction hypothesis and the shorter derivation of $\{\!\{\Gamma\}\!\} \longrightarrow (\{\!\{B_i\}\!\}[N_1/y_1, \ldots, N_{i-1}/y_{i-1}]\ N_i)$ allows us to conclude that $N_i = \langle N'_i\rangle$ for some canonical LF term N'_i and that there is a derivation of

$$\Gamma \vdash N'_i : B_i[N'_1/y_1 \ldots N'_{i-1}/y_{i-1}],$$

thus verifying the claim.

A.2 Completeness of the optimized encoding (Theorem 2)

If $\{\!\{\Gamma\}\!\} \longrightarrow \{\!\{A\}\!\}M$ has a derivation, then $[\![\Gamma]\!]^+ \longrightarrow [\![A]\!]^- M$ has a derivation as well. Note that for this direction of the proof we are simply dropping information (subderivations) 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 $\langle A\rangle = \langle A'\rangle[t_1/x_1 \ldots t_n/x_n]$:

$$\frac{\{\!\{\Gamma\}\!\} \longrightarrow F_1 \quad \ldots \quad \{\!\{\Gamma\}\!\} \longrightarrow F_n}{\{\!\{\Gamma\}\!\} \longrightarrow \{\!\{A\}\!\}\ (y\ \overrightarrow{t})}\ \mathit{backchain}$$

Here, $F_i = (\{\!\{B_i\}\!\}\ x_i)[t_1/x_1 \ldots t_n/x_n]$. 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 $F_i = \{\!\{B_i[t'_1/x_1 \ldots t'_n/x_n]\}\!\}(t_i)$. We shall build the derivation of $[\![\Gamma]\!]^+ \longrightarrow [\![A]\!]^-(y\ \overrightarrow{t})$ by using *backchain* on the optimized encoding of $(y : \Pi \overrightarrow{x{:}B}.A') \in \Gamma$, by choosing $\overrightarrow{t}$ for $\overrightarrow{x}$. The resulting premises are either

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

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 $\top R$.