A Two-Level Logic Approach to Reasoning about Typed Specification Languages
Mary Southern, Kaustuv Chaudhuri

To cite this version:
Mary Southern, Kaustuv Chaudhuri. A Two-Level Logic Approach to Reasoning about Typed Specification Languages. 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014), Dec 2014, New Delhi, India. 10.4230/LIPIcs.FSTTCS.2014.557. hal-01091544

HAL Id: hal-01091544
https://inria.hal.science/hal-01091544
Submitted on 5 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NoDerivatives 4.0 International License
A Two-Level Logic Approach to Reasoning about Typed Specification Languages

Mary Southern
University of Minnesota, USA
marys@cs.umn.edu

Kaustuv Chaudhuri
INRIA and LIX/École polytechnique, France
kaustuv.chaudhuri@inria.fr

December 5, 2014

Abstract

The two-level logic approach (2LL) to reasoning about computational specifications, as implemented by the Abella theorem prover, represents derivations of a specification language as an inductive definition in a reasoning logic. This approach has traditionally been formulated with the specification and reasoning logics having the same type system, and only the formulas being translated. However, requiring identical type systems limits the approach in two important ways: (1) every change in the specification language’s type system requires a corresponding change in that of the reasoning logic, and (2) the same reasoning logic cannot be used with two specification languages at once if they have incompatible type systems. We propose a technique based on adequate encodings of the types and judgements of a typed specification language in terms of a simply typed higher-order logic program, which is then used for reasoning about the specification language in the usual 2LL. Moreover, a single specification logic implementation can be used as a basis for a number of other specification languages just by varying the encoding. We illustrate our technique with an implementation of the LF dependent type theory as a new specification language for Abella, co-existing with its current simply typed higher-order hereditary Harrop specification logic, without modifying the type system of its reasoning logic.

1 Introduction

Higher-order abstract syntax (HOAS) [13], also known as λ-tree syntax (λTS) [8], has become a standard representational style for data structures with variable binding. Such data are pervasive in the syntax of programming languages, proof systems, process calculi, formalized mathematics, etc. Variable binding issues are a particularly tricky aspect of the meta-theory of computational systems given in the form of structural operational semantics (SOS). Such specifications are nearly always formulated as relations presented in the form of an inference system; for instance, the typing judgement for the simply typed λ-calculus is a relation between λ-terms and their types, usually defined in terms of a natural deduction proof system. Such relations on higher-order data can then be systematically formalized as higher-order logic programs in languages such as λProlog [9] or Twelf [14], which lets us directly animate the specifications by means of logic programming interpreters and compilers such as Tejus [11].

In this paper, we are concerned with proving properties about such higher-order relational specifications. For example, if the specification is of the typing relation for simply typed λ-terms, then we may want to prove that a given λ-term has exactly one type (type-uniqueness), or that the type of a λ-term remains stable during evaluation (type-preservation). This kind of reasoning proceeds by induction on the derivations of the specified relations, so we need a formalism that supports both inductive definitions and reasoning by induction. The two-level logic approach (2LL) is a general scheme for such reasoning systems, where the specification language derivations are viewed as an inductively defined object in a reasoning logic. In this reasoning logic, the specification language derivations are given a closed world reading, which is to say that that derivability in the specification language is completely specified: it can not only establish that certain specification formulas or judgements are derivable, but also that others are not derivable, or that two specification derivations are (bi)similar. We focus on the Abella implementation of the 2LL, which is an interactive tactics-based theorem prover designed to reason about a subset of higher-order λProlog programs seen as the logic of higher-order hereditary Harrop (HOHH) formulas [20, 19].

We consider an extension of the 2LL that can use a single reasoning logic to reason about a number of different specification languages in the same development. The HOHH language has only simple types, which makes both the specifications and the reasoning somewhat verbose because structural invariants must be separately specified and explicitly invoked in theorems. Richer type systems can often encode such invariants intrinsically in the types; to illustrate, dependent types can be used to define a type of (representations of) well-typed λ-terms,
which is not possible with just simple types. Moreover, with such richer type systems one can often use the
inductive structure of the terms themselves to drive the inductive argument rather than using auxiliary relations.

Unfortunately, the 2LL as currently defined [6] does not sufficiently address these desiderata. In particular,
the specification and reasoning languages are required to have the same type system because the specification-
level constants and their types are directly lifted to reasoning-level constants with the same type. Thus, if we
required a version of Abella based on a dependent type theory as a specification language, we would need to also
change its reasoning logic G to be dependently typed. This goes against the 2LL philosophy where the reasoning
logic is seen as common, static, and eternal. More importantly, it both breaks portability of developments and
causes duplication of effort.

Our position is that we should extend the 2LL in such a way that the reasoning-level and specification-level
type systems are separated. Indeed, the specification types and judgements must themselves be encoded as terms
and formulas of the reasoning logic. This encoding must be coherent with that of specification-level terms and
formulas, both of which are encoded as reasoning-level atomic formulas. This is achieved by guaranteeing that
our encoding of the type systems is adequate; that is, the encoding of the specification-level type system must be
able to represent all specification-level typing derivations, and that reasoning about the specification-level type
system should be reducible to reasoning, by induction, on the encoding. An essential ingredient of adequacy is a
right-inverse of the encoding that extracts a specification-level typing judgement from a reasoning-level formula
when the formula is in the image of the encoding.

To be concrete, we illustrate the extended 2LL in this paper by giving an encoding of the LF dependent
type theory, which is then implemented as a translation layer in Abella. The reasoning logic of Abella is left
untouched, as is the existing HOHH specification language for reasoning about λProlog programs. Our encoding
of LF is based on that of [17, 18], suitably modified to the context of interactive theorem proving rather than
logic programming. Since both LF and HOHH are based on intuitionistic logic, our extension of Abella uses a
core implementation of an intuitionistic specification language that is shared by both the HOHH and the LF
languages. Interestingly, the details of the encoding into this core language can almost entirely be obscured
for the user; in particular, to use the system the user does not need to know how the specification language is
encoded, since the system uses the right inverse mentioned above to present the types, terms, and judgements
of the specification language in their native forms.

The rest of the paper is organized as follows. Section 2 presents the two-level logic approach (2LL) and its
implementation in the Abella theorem prover. Section 3 presents LF and its adequate translation into a simply
typed higher-order logic programming language. This is then used in section 4 to explain our extension of the
2LL by means of adequate translations. Related work is surveyed in section 5.

2 Background

This section sketches the two-level logic approach (2LL) as implemented in the Abella theorem prover [20]. More
details, including the full proof systems and their meta-theory, can be found in the following sequence of papers:
[19, 6].

2.1 The Reasoning Logic G

The reasoning logic of Abella, G, is a predicative and intuitionistic version of Church’s Simple Theory of Types.
Types are built freely from primitive types, which includes the type prop of formulas, using the function type
constructor \( \to \). Intuitionistic logic is introduced into this type system by means of the constants \( \text{true}, \text{false} : \text{prop} \),
binary connectives \( \land, \lor, \supset : \text{prop} \to \text{prop} \to \text{prop} \), and an infinite family of quantifiers \( \forall \tau, \exists \tau : (\tau \to \text{prop}) \to \text{prop} \)
for types \( \tau \) that do not contain \( \text{prop} \). For every type \( \tau \) not containing \( \text{prop} \), we also add an
atomic predicate symbol \( =_\tau : \tau \to \tau \to \text{prop} \) to reason about intensional (i.e., up to \( \alpha\beta\gamma \)-conversion)
equality. Following usual conventions, we write \( \land, \lor, \supset \), and \( =_\tau \) infix, and write \( \forall x : \tau. A \) for \( \forall \tau.(\lambda x. A) \) (and similarly for \( \exists \)). We also omit the type subscripts and type-ascription on variables when unambiguous.

To provide the ability to reason on open \( \lambda \)-terms, which is necessary when reasoning about HOAS represen-
tations, \( G \) also supports generic reasoning. This is achieved by adding, for each type \( \tau \) not containing \( \text{prop} \),
an infinite set of nominal constants and a generic quantifier \( \forall \tau : (\tau \to \text{prop}) \to \text{prop} \). We also add a weaker form
of intensional equality called equi-invariance that equates two terms whose free nominal constants may be system-
atically permuted to each other. Note that equi-invariance is only used to match conclusions to hypotheses in the
\( G \) proof system; \( = \) continues to have the standard \( \lambda \)-conversion semantics. The support of a term \( t \), written
\( \text{supp}(t) \), is the multiset of nominal constants that occur in it; whenever we introduce a new eigenvariable, such
as using the \( \forall \)-right or \( \exists \)-left rules, we raise the eigenvariables over the support of the formula. This raising is
needed to express permitted dependencies on these nominal constants.

To accommodate fixed-point definitions, \( G \) is parameterized by sets of definitional clauses. Each such clause
has the form \( \forall x.(\forall \exists A) \triangleq B \) where \( A \) (the head) is an atomic formula whose free variables belong to \( \bar{x} \) or
\[
\begin{align*}
\text{seq } L (G_1 \land G_2) &\triangleq \text{seq } L G_1 \land \text{seq } L G_2 \\
\text{bch } L (F \land G) &\triangleq \text{bch } L F \lor \text{bch } L G \\
\text{bch } L (F \rightarrow G) &\triangleq \text{seq } (F :: L) G \\
\text{seq } L (\pi_1 F) &\triangleq \forall x. \text{seq } L (F x) \\
\text{seq } L A &\triangleq \text{mem } F L \lor \text{bch } L F A \\
\text{seq } L A &\triangleq \text{prop } F \lor \text{bch } L F A
\end{align*}
\]

Figure 1: Encoding \textit{HOHH} using definitional clauses in \mathcal{G}. \(F\) and \(G\) range over arbitrary specification formulas, while \(A\) ranges over atomic specification formulas. All clauses are implicitly universally closed over their capitalized variables.

\(\vec{z}\), while \(B\) (the \textit{body}) is any arbitrary formula that can only mention the variables in \(\vec{z}\), and can additionally have recursive occurrences of the predicate symbol in the head. Each clause partially defines a relation named by the predicate in the head. We additionally require that \(\text{supp}(\forall \vec{z}. A)\) and \(\text{supp}(B)\) be both empty, and that recursive predicate occurrences satisfy a \textit{stratification} condition [5].\(^1\) Finally, some of these definitions in \mathcal{G} can be marked as \textit{inductive} or \textit{co-inductive}, in which case the set of definitional clauses for that relation are given least or greatest fixed-point semantics. This is approximated in \textit{Abella} by means of \textit{size annotations}, which are formally defined and proved correct in [4].

\subsection{2.2 The Specification Language: HOHH}

The essence of the \textit{SLL} is to encode the deductive formalism of the specification language in terms of an inductive definition. However, before this can be done, the terms and formulas—and types!—of the specification language must be represented in the reasoning logic. This is trivial if the specification and reasoning logics have the same term and type language, which is the case for the \textit{HOHH} language. To encode \textit{HOHH} formulas, we use a new basic type \(\text{o}\), and formula constructors \(\Rightarrow, \&: \text{o} \rightarrow \text{o} \rightarrow \text{o}\) (written infix), and an infinite family of specification-level quantifiers \(\pi_\text{x}:(\text{r} \rightarrow \text{o}) \rightarrow \text{o}\) (standing for universal quantification, written prenex) for types \(\text{r}\) that do not contain \(\text{o}\). To prevent circularity, we disallow the type \text{prop} and the reasoning level formula constructors from occurring inside specification level types and terms.

The proof system for \textit{HOHH} is a standard focused sequent calculus for this fragment of the logic, assuming that all atoms have negative polarity; this is equivalent to saying that the proof system implements \textit{backward chaining} [19]. This proof system is implemented in \mathcal{G} using two predicates, \text{seq} : \text{olist} \rightarrow \text{o} \rightarrow \text{prop} and \text{bch} : \text{olist} \rightarrow \text{o} \rightarrow \text{prop}, standing for \textit{goal reduction} and \textit{backward chaining} respectively, with the definitional clauses shown in figure 1. Here, specification contexts have the type \text{olist}, the type of lists of \text{o}, with constructors \text{nil} : \text{olist} and \(::) : \text{o} \rightarrow \text{olist} \rightarrow \text{olist}\) (written infix), and a membership relation \text{mem} : \text{o} \rightarrow \text{olist} \rightarrow \text{prop} that has the obvious inductive definition. In \textit{Abella}, these two relations are displayed using the more evocative notation \([L \vdash G]\) and \([L, [F] \vdash G]\) for \text{seq} \(L G\) and \text{bch} \(L F G\). The final clause for \text{seq} uses a separate predicate \text{prop} : \text{o} \rightarrow \text{prop} that is true exactly for the clauses in the specification program. It is easy to see that with this syntax, the definitional clauses of figure 1 are precisely the inductive definition of a backchaining proof system.

\subsection{2.3 Example: Type Uniqueness}

The need for the two kinds of specification sequents and the mechanism for proving properties about the specification logic are best described with an example. Consider the simply typed \(\lambda\)-calculus, itself specified as an object logic in \textit{HOHH}. The simple type system is represented using a new basic type \(\text{ty}\) with two constructors, \(\text{i} : \text{ty}\) (a basic sort), and \text{arr} : \text{ty} \rightarrow \text{ty} \rightarrow \text{ty}\) for constructing arrow types. The \(\lambda\)-terms are typed using a different basic type \text{tm} with two constructors: \text{app} : \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} and \text{abs} : \text{ty} \rightarrow (\text{tm} \rightarrow \text{tm}) \rightarrow \text{tm}. The \(\lambda\)-term \(\lambda x. f x\) would be represented as \text{abs} i ((\lambda x. \text{abs} (\text{arr} i) (\lambda f. \text{app} f x))\). The relation between terms (of type \text{tm}) and types (of type \text{ty}) is usually expressed in the form of an inference system such as:

\[
\begin{align*}
\Gamma, x : A &\vdash M : B \\
\Gamma, x : A &\vdash M : B
\end{align*}
\]

This relation is succinctly expressed as a pair of \textit{HOHH} program clauses for the predicate \text{of} : \text{tm} \rightarrow \text{ty} \rightarrow \text{o}, which is used both for assumptions of the form \(x : A\) and for conclusions of the form \(M : A\) in the inference system above.

\[
\begin{align*}
\text{pi } a : \text{ty} &\vdash \text{pi } b : \text{ty} \rightarrow \text{pi } m : \text{tm} \rightarrow \text{tm}. \text{of } m (\text{arr } a) \Rightarrow \text{of } n a \Rightarrow \text{of } (\text{app } m n) b. \\
\text{pi } a : \text{ty} &\vdash \text{pi } b : \text{ty} \rightarrow \text{tm}. (\text{pi } x . \text{tm} \Rightarrow \text{of } (r x) \text{tm}) \Rightarrow \text{of } (\text{abs } a r) (\text{arr } a b).
\end{align*}
\]

\(^1\)Roughly, stratification prevents definitions such as \(p \triangleq \neg p\), which would lead to inconsistency.
The formula \( \pi \) we can prove that match

The Edinburgh Logical Framework (LF) is a dependently typed \( \lambda \)-calculus which is used for specifying formal systems. Terms of this language belong to one of the following three syntactic categories:

Kinds: \( K ::= \text{type} | \Pi x: A. K \)

Types: \( A, B ::= a | M_1 \ldots M_n | \Pi x: A. B \)

Objects: \( M, N ::= c | x | \lambda x: A. M | M N \)

Types, sometimes called families, classify objects and kinds classify types. Here \( a \) represents a type-level constant, \( c \) an object level constant, and \( x \) an object level variable. Following standard convention, we will write \( A \rightarrow B \) as a shorthand for \( \Pi x: A. B \) when \( x \) does not appear free in \( B \). We will use \( U \) to denote both types and objects and \( P \) for both kinds and types, so \( U : P \) will stand either for a typing or a kinding judgement. We will 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 \( x_1, \ldots, x_n \) respectively.

An LF specification is a list of object or type constants together with their types or kinds, called a signature. Let us revisit the example of the simply typed \( \lambda \)-calculus and its associated typing relation used in section 2.3.

The \( \lambda \)-terms are encoded using the following signature:

\[
\begin{align*}
ty & : \text{type}. \\
\text{i} & : ty. \\
\text{app} & : tm \rightarrow tm \rightarrow tm. \\
\text{arr} & : ty \rightarrow ty \rightarrow ty. \\
\text{abs} & : ty \rightarrow (tm \rightarrow tm) \rightarrow tm.
\end{align*}
\]

For the typing relation \( \text{of} \), in LF we declare it as a dependent type rather than as a predicate as in \textit{HOHH}. The clauses of the \text{of} type are then viewed as constructors for the dependent type, and are therefore also given names. Here, the concrete syntax \( \{x:A\} B \) denotes \( \Pi x: A. B \).
The rules for this encoding are shown in figure 3. In each case, \( \Gamma \) contains the type and kind information for the signature constants and the typing assumptions for the bound variables in the input terms. A and B range over \( LF \) terms, \( F \) and \( G \) over \( HOHH \) formulas and \( a, m \) and \( n \) over simply typed \( \lambda \)-terms produced by \( \langle \cdot \rangle \). The rules for inverting typing and kinding are novel, but those for inverting terms are standard from bidirectional type-checking, and have already been developed in [18] (in a slightly more general form).

**Theorem 2** (Right inverse). If \( \{ \{ P \} \} \{ U \} = F \) under the translation of \( \Gamma \) and \( \Gamma \vdash F \longrightarrow U' : P \), then \( \Gamma \vdash U =_{\beta \eta} U' : P \) in \( LF \).
The following clauses are then added to \( \text{prog} \) are added to the \( \text{LF} \) signatures just as we were able to reason on \( \lambda \text{Prolog} \) specifications as shown in the example of section 2.3.

4 Translational Two-Level Logic Approach

We will now use both the translation of \( \text{LF} \) signatures to \( \text{HOHH} \) formulas and its inverse to extend the 2LL in such a way that we can reason about \( \text{LF} \) signatures just as we were able to reason on \( \lambda \text{Prolog} \) specifications as shown in the example of section 2.3.

4.1 Importing the \( \text{LF} \) Specification

As the type system of \( \text{LF} \) and \( \mathcal{G} \) are different, we cannot directly reflect the constants and types of the \( \text{LF} \) specification logic we did with \( \text{HOHH} \) in section 2. Instead, for every \( \text{LF} \) constant \( c \) of \( \text{LF} \) type or kind \( P \) in the \( \text{LF} \) signature, we do the following: (1) add a constant \( c : \phi(P) \) to the \( \mathcal{G} \) signature; and (2) add the clause \([\{P\}]c\) to the \( \text{prog} \) definition. Note that because the clauses are always of the form \([\{P\}]c\) for a constant \( c \), there will never be any redexes in the clauses, i.e., the generated clauses are \( \beta \)-normal. They are also \( \eta \)-long, because the definition of \([\{\cdot\}]\) traverses the type or kind until it is atomic.

Example 3. Consider again the \( \text{LF} \) signature in section 3. When it is imported into \( \mathcal{G} \), the following constants are added to \( \mathcal{G} \) signature by step (1):

\[
\begin{align*}
\text{ty} & : \text{lf}\text{type}. \\
\text{tm} & : \text{lf}\text{type}. \\
\text{arr} & : \text{lfobj} \rightarrow \text{lfobj}. \\
\text{app} & : \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj}. \\
\text{abs} & : \text{lfobj} \rightarrow (\text{lfobj} \rightarrow \text{lfobj}) \rightarrow \text{lfobj}. \\
of & : \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lf}\text{type}. \\
of\text{app} & : \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj}. \\
of\text{abs} & : \text{lfobj} \rightarrow \text{lfobj} \rightarrow (\text{lfobj} \rightarrow \text{lfobj}) \rightarrow (\text{lfobj} \rightarrow \text{lfobj}) \rightarrow \text{lfobj} \rightarrow \text{lfobj} \rightarrow \text{lfobj}.
\end{align*}
\]

The following clauses are then added to \( \text{prog} \) by step (2) described above:

\[
\begin{align*}
\text{lfstty} & \text{ty}, \\
\text{lfhas} & \text{ty}, \\
\text{lfhas} & (\text{arr} \text{Z1 Z2}) \text{ty} \equiv \text{lfhas Z1 ty}, \text{lfhas Z2 ty}. \\
\text{lfsity} & \text{tm}, \\
\text{lfhas} & (\text{abs} \text{Z1 Z2}) \text{tm} \equiv \text{lfhas Z1 tm}, \text{lfhas Z2 tm}. \\
\end{align*}
\]

\[
\begin{align*}
\text{lfstty} (\text{of} \text{Z1 Z2}) & \equiv \text{lfhas Z1 tm}, \text{lfhas Z2 ty}. \\
\text{lfsity} (\text{of} \text{A B}) & \equiv \text{of}\text{app} \text{A B N}, \text{of}\text{app} \text{A B N}, \text{of}\text{app} \text{A B N}. \text{(of} \text{app N R \text{B})} \equiv \text{lfhas a ty}, \text{lfhas A ty}, \text{lfhas B ty}, \text{lfhas M tm}, \text{lfhas N tm}. \text{lfhas Z1 (of N (arr A B))}, \text{lfhas Z2 (of N A)}. \\
\text{lfsity} (\text{of} \text{arr A B}) & \equiv \text{lfhas a ty}, \text{lfhas B ty}, \text{lfhas A ty}, \text{lfhas B ty}, \text{lfhas Z1 ty}, \text{lfhas a tm} \equiv \text{lfhas (R a) tm}, \\
\text{(ps a \text{Z1 Z2})} & \equiv \text{lfhas (Z2 a) tm}, \\
\end{align*}
\]
\[(\text{pl z} \leftarrow \text{lhas z tm} = \text{pl z} \leftarrow \text{lhas z (of z A)} = \text{lhas (Z1 z a) (of (R x) B)})\).

The variables named \(Z_i\) are generated by the translator for those variables that are omitted from the input signature by the use of \(\rightarrow\) instead of \(\Pi\). We write clauses using standard \(\lambda\)Prolog syntax for clarity; it is simple to take the output of translation into this form.

It is instructive to compare these clauses to those for the pure \(\text{HOHH}\) version in section 2.3. Although, on the surface, these two look quite different, there are similarities in the kinds of subgoals that are produced for the three constructors of \(\text{of}\). For example, consider the case of \(\text{ofApp}\). Two of the formulas, \(\text{hastype Z1 (of M (arr A B)})\) and \(\text{hastype Z2 (of N A)}\) are already present in nearly this form in the \(\text{HOHH}\) specification. The additional assumptions are just repetitions of the typing assumptions for the arguments to \(\text{ofApp}\); indeed, many of them are redundant since the \(\text{ofApp}\) term is already assumed to be type-correct. This kind of redundancy analysis can be used to further improve the translation, making it nearly identical to the simply typed specification [17, 18].

### 4.2 Representing LF Hypothetical Judgements

We use the concrete syntax \(\langle M : A \rangle\) or \(\langle A : K \rangle\) to depict \(\{[A]\}\) or \(\{[K]\}\) \(A\), respectively. In fact, since the LF type system is given in terms of hypothetical derivations, we generalize this syntax to the form: \(\langle x_1 : P_1, \ldots, x_n : P_n \upharpoonright U : P \rangle\) as an abbreviation for: \(\text{seq} ((x_1 : P_1) : \cdots : (x_n : P_n)) ((U : P))\). As an example,\(^2\) the uniqueness theorem for the \(\text{of}\) relation is (eliding types):

\[
\forall G, M, A, B, P_1, P_2. \text{ctx} G \supset (G \vdash P_1 : \text{of} M A) \supset (G \vdash P_2 : \text{of} M B) \supset A = B. \tag{1}
\]

Here, \(P_1\) and \(P_2\) are (encodings of) the LF proof-terms for the judgements of \(M\) \(A\) and \(M\) \(B\) respectively; these proof terms are built out of the constructors for the \(\text{of}\) relation, \(\text{viz.}\) \(\text{ofApp}\) and \(\text{ofAbs}\).

Of course, in order to prove this theorem we would require a suitable \(\text{ctx}\) definition. Unlike in the simply typed case, the recursive case for \(\lambda\)-abstractions not only introduces a new variable but also a proof that it has a given LF type at the same time. This gives us the following definitional clauses.

\[
\text{ctx nil} \triangleq \text{true}.
\]

\[
\forall x : \text{lfobj}. \forall p : \text{lfobj}. \text{ctx} \langle (x : \text{tm}) :: (p : \text{of} x A) :: G \rangle \triangleq \text{ctx} G.
\]

It is interesting to note that, because variables are introduced (bound) in a different place than their typing assumptions, it would be just as valid to use the following clause instead for the second clause above:

\[
\forall x : \text{lfobj}. \forall p : \text{lfobj}. \text{ctx} \langle (p : \text{of} x A) :: (x : \text{tm}) :: G \rangle \triangleq \text{ctx} G.
\]

This reordering of the context that is not strictly allowed in the LF type system poses no problems for us. Indeed, when we reason about the elements of the context, we can always recover these two assumptions that are always simultaneously added to the context.

\[
\text{forall G, nbla p x,}
\]

\[
\text{ctx} \langle G x p \rangle \rightarrow \text{mem} \langle x : \text{tm} \rangle \langle G x p \rangle \rightarrow
\]

\[
\text{exists A, mem} \langle p : \text{of} x A \rangle \langle G x p \rangle \land \text{fresh} p A \land \text{fresh} x A.
\]

The dependency of \(G\) on \(x\) and \(p\) is indicated explicitly using application. For \(A\), this dependency is implicit, because the \(\text{exists}\) occurs in the scope of the corresponding \(\text{nbl}\), so we use the predicate \(\text{fresh} : \text{lfoj} \rightarrow \text{lfoj} \rightarrow \text{pro}\) to further assert that its first arguments are nominal constants that do not occur in its second arguments. This is definable with the single clause: \(\forall A, (\forall x. \text{fresh} x A) \rightarrow \text{true}\).

The proof of (1) proceeds by induction on the second assumption, \(G \vdash P_1 : \text{of} M A\), using the clauses added to \(\text{prog}\) when importing the specification. There are exactly three backchaining possibilities for \(\text{prog}\) clauses, corresponding to the \(\text{ofApp}\) and \(\text{ofAbs}\) cases, respectively. Finally, when backchaining on the dynamic clauses in \(G\), we use the \(\text{ctx}\) definition to characterize the shape of the selected clause: if the selected clause is \(\langle x : \text{tm} \rangle\), then the branch immediately succeeds since \(\langle x : \text{tm} \rangle\) will not unify with \(\langle P_1 : \text{of} M A \rangle\). Thus, the only backchaining case worth considering is when the selected dynamic clause is of the form \(\langle p : \text{of} x A' \rangle\). In this case, we continue by case-analysis of the second derivation, \(\langle P_2 : \text{of} M B \rangle\), in which case again the only possibility that is not immediately ruled out by unification is the case of \(\langle p' : \text{of} x B' \rangle\) being selected from \(G\).

In this case, we appeal to a uniqueness lemma [1] of the following form:

\[
\forall G, X, A, B, P_1, P_2. \text{ctx} G \supset \text{mem} \langle P : \text{of} X A \rangle G \supset \text{mem} \langle P_2 : \text{of} X B \rangle G \supset A = B.
\]

The rest of the proof is fairly systematic, and largely identical in structure to that of the \(\text{HOHH}\) case. It is also worth remarking that once we have shown that the types \(A\) and \(B\) are identical in (1), we can then also show that the proof terms \(P_1\) and \(P_2\) must also be equal (up to \(\alpha\beta\eta\), of course).

\(^2\)The full Abella/LF development may be interactively browsed online at \url{http://abella-prover.org/lf}. jag
\[
\text{forall } G \times A \times B \times P_1 \times P_2, \quad \text{ctx } G \rightarrow \left(G \vdash P_1 : \text{of } M \Rightarrow A\right) \rightarrow \left(G \vdash P_2 : \text{of } M \Rightarrow B\right) \rightarrow P_1 \equiv P_2.
\]

This is expected from the \( LF \) type theory, but would be difficult to state in \( LF \) itself because of the lack of equality as a built-in relation.

### 4.3 The Implementation

The implementation of the translational \( 2LL \) can be found in the \( 1f \) branch of the \textit{Abella} repository.\(^3\) This implementation also comes with a few examples of reasoning on \( LF \) specifications that can be browsed online without needing to run \textit{Abella}. We have made the following observations about these developments:

- The user of the system never needs to look at the encoding of \( LF \) in \( HOHH \) directly. The system always translates \( LF \) judgements, written using \((\cdot)\), transparently to \( HOHH \), and also inverts any \( HOHH \) formulas in the image of the translation back into an \( LF \) (hypothetical) judgement. Hence, the only domain knowledge the user needs to use the system is the tactics-based proof language of \textit{Abella} itself.

- Our implementation currently does not perform type-checking on the \( LF \) judgements written by the user, either in the specification itself or as part of reasoning. This is not as such a problem, since we can never prove anything false about well-typed judgements. However, without type checking we have no way to verify that the theorem which has been proved is really meaningful since we are allowed to reason about ill-formed \( LF \) judgements. It would also be useful for users to have a type-checker as a sanity check.

For the time being, we run the input specification through the \textit{Twelf} system \([14]\), both to type-check it and to get an explicit form of the specification.

### 5 Related Work and Conclusion

We have proposed here a translational extension to the two-level logic approach for reasoning about specifications. By adding a translation layer to the \textit{Abella} theorem prover we have been able to reason over dependently typed \( LF \) specifications without needing to change the reasoning logic, and allowing \( LF \) to co-exist with the \( HOHH \) specification logic. We are already in the process of extending this implementation to arbitrary pure type systems instead of just \( LF \); in particular, extending the type system with polymorphism, which is the most common feature request for \textit{Abella}, should be encodable via our translation that realizes specification types as reasoning terms.

The translation of \( LF \) to \( HOHH \) used in this work is a minor variant of the \textit{simple} translation from \([17]\), which is itself based on earlier work \([3]\), while the inversion on terms is similar to the definition in \([18]\), omitting meta-variables. Various optimized versions of this translation have been used to use \textit{\lambda Prolog} as an engine for logic programming with \( LF \) specifications; in particular, the \textit{Parinati} system \([17]\) and its extension to meta-variables in \([18]\). The meta-theory of the optimized translation is not as immediate as for the simple translation, but it would be interesting to investigate its use for the \textit{Abella}/\( \lambda \text{Prolog} \) variant in the future. The combination of \textit{Parinati} and \textit{Abella}/\( LF \) gives us both an efficient execution model for dependently typed logic programs and a mechanism to reason about the meta-theory of such specifications in the extended \( 2LL \). In effect, \( LF \) becomes as much a first class citizen of the \textit{Abella} ecosystem as \( HOHH \) and \textit{\lambda Prolog} have traditionally been.

There are many other systems designed to reason with and about \( LF \) specifications. The most mature implementation is \textit{Twelf} \([14]\), which has a very efficient type-checker incorporating sophisticated term and type reconstruction. As mentioned in section 4.3, we use \textit{Twelf} to type-check and elaborate the \( LF \) specifications we import in \textit{Abella}. In addition to the type-checker, \textit{Twelf} has a suite of meta-theoretical tools that can verify certain properties of \( LF \) specifications, such as that a declared relation determines a total function. \textit{Twelf} is, however, not powerful enough to reason inductively on arbitrary \( LF \) derivations. For example, although \textit{Twelf} can check coverage, it cannot express the logical formula that corresponds to the coverage property.

Some of these expressive deficiencies of \textit{Twelf} have been addressed in the \textit{Delphin} \([16]\) and \textit{Beluga} \([15]\) systems that add a functional programming language that can manipulate and reason inductively on \( LF \) syntax. The \textit{Beluga} system, in particular, extends the \( LF \) type theory with \textit{contextual modal types} \([12]\) that give a type-theoretic treatment for meta-variables and explicit substitutions; in addition, \textit{Beluga} also allows abstraction over contexts and substitutions \([2]\). The type-checker of \textit{Beluga} is therefore very sophisticated and performs many kinds of reasoning on contexts automatically that must be done manually in \textit{Abella}. On the flip-side, \textit{Abella} has a small trusted core based on the logic \( \mathcal{G} \) with a well-understood and—importantly!—stable proof system \([10, 5]\). It would be interesting to formally compare the representational abilities of \textit{Abella}/\( LF \) and \textit{Beluga}. Moreover, \textit{Abella} has recently acquired a Plugin architecture that allows arbitrary (but soundness-preserving) user-written extensions to its automation capabilities \([1]\), which might help us add more automation in the future.

---

\(^3\)Details for downloading and building this branch can be found in http://abella-prover.org/lf.
References


