A Two-Level Logic Approach to Reasoning about Typed Specification Languages

Mary Southern[1] and Kaustuv Chaudhuri[2]

1 **University of Minnesota, USA**
marys@cs.umn.edu
2 **INRIA and LIX/École polytechnique, France**
kaustuv.chaudhuri@inria.fr

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.

1998 ACM Subject Classification F.4.1 Mathematical Logic: Proof theory

1 Introduction

Higher-order abstract syntax (*HOAS*) [14], also known as *λ-tree syntax* (*λTS*) [9], 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* [10] or *Twelf* [15], which lets us directly animate the specifications by means of logic programming interpreters and compilers such as *Teyjus* [12].

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 [21, 20].

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 $\mathcal{G}$ 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 [18, 19], 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 [21]. More details, including the full proof systems and their meta-theory, can be found in the following sequence of papers: [20, 6, 8].

2.1 The Reasoning Logic $\mathcal{G}$

The reasoning logic of *Abella*, $\mathcal{G}$, 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 $\texttt{true}, \texttt{false} : \texttt{prop}$, binary connectives $\wedge, \vee, \supset$: $\texttt{prop} \to \texttt{prop} \to \texttt{prop}$, and an infinite family of quantifiers $\forall_\tau, \exists_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$ for types τ that do not contain `prop`. For every type τ not containing `prop`, we also add an atomic predicate symbol $=_\tau : \tau \to \tau \to \texttt{prop}$ to reason about intensional (*i.e.*, up to $\alpha\beta\eta$-conversion) equality. Following usual conventions, we write $\wedge$,$\vee$, $\supset$, 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 λ-terms, which is necessary when reasoning about *HOAS* representations, $\mathcal{G}$ also supports *generic reasoning*. This is achieved by adding, for each type τ not containing `prop`, an infinite set of *nominal constants* and a generic quantifier $\nabla_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$. We also add a weaker form of intensional equality called *equivariance* that equates two terms whose free nominal constants may be systematically permuted to each other. Note that equivariance is only used to match conclusions to hypotheses in the $\mathcal{G}$ proof system; $=$ continues to have the standard λ-conversion semantics. The *support* of a term t, written $\mathrm{supp}(t)$, 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, $\mathcal{G}$ is parameterized by sets of *definitional clauses*. Each such clause has the form $\forall \vec{x}.\, (\nabla \vec{z}.\, A) \triangleq B$ where A (the *head*) is an atomic formula whose free variables belong to $\vec{x}$ or $\vec{z}$, while B (the *body*) is any arbitrary formula that can only mention the variables in $\vec{x}$, 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 $\mathrm{supp}(\nabla \vec{z}.\, A)$ and $\mathrm{supp}(B)$ be both empty, and that recursive predicate occurrences satisfy a *stratification* condition [5].[1] Finally, some of these definitions in $\mathcal{G}$ can be marked as *inductive* or *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 *Abella* by means of *size annotations*, which are formally defined and proved correct in [4].

[1] Roughly, stratification prevents definitions such as $p \triangleq \neg p$, which would lead to inconsistency.

$$\texttt{seq}\ L\ (G_1\ \&\ G_2) \triangleq \texttt{seq}\ L\ G_1 \wedge \texttt{seq}\ L\ G_2 \qquad \texttt{bch}\ L\ (F_1\ \&\ F_2)\ A \triangleq \texttt{bch}\ L\ F_1\ A \vee \texttt{bch}\ L\ F_2\ A$$
$$\texttt{seq}\ L\ (F \Rightarrow G) \triangleq \texttt{seq}\ (F :: L)\ G \qquad \texttt{bch}\ L\ (G \Rightarrow F)\ A \triangleq \texttt{seq}\ L\ G \wedge \texttt{bch}\ L\ F\ A$$
$$\texttt{seq}\ L\ (\texttt{pi}\ F) \triangleq \nabla x.\, \texttt{seq}\ L\ (F\ x) \qquad \texttt{bch}\ L\ (\texttt{pi}\ F)\ A \triangleq \exists t.\, \texttt{bch}\ L\ (F\ t)\ A$$
$$\texttt{seq}\ L\ A \triangleq \texttt{mem}\ F\ L \wedge \texttt{bch}\ L\ F\ A \qquad \texttt{bch}\ L\ A\ A \triangleq \texttt{true}$$
$$\texttt{seq}\ L\ A \triangleq \texttt{prog}\ F \wedge \texttt{bch}\ L\ F\ A$$

Figure 1 Encoding *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.

2.2 The Specification Language: *HOHH*

The essence of the *2LL* 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 *HOHH* language. To encode *HOHH* formulas, we use a new basic type $\texttt{o}$, and formula constructors $\Rightarrow, \& : \texttt{o} \to \texttt{o} \to \texttt{o}$ (written infix), and an infinite family of specification-level quantifiers $\texttt{pi}_\tau : (\tau \to \texttt{o}) \to \texttt{o}$ (standing for universal quantification, written prenex) for types τ that do not contain $\texttt{o}$. To prevent circularity, we disallow the type $\texttt{prop}$ and the reasoning level formula constructors from occurring inside specification level types and terms.

The proof system for *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 *backchaining* [20]. This proof system is implemented in $\mathcal{G}$ using two predicates, $\texttt{seq} : \texttt{olist} \to \texttt{o} \to \texttt{prop}$ and $\texttt{bch} : \texttt{olist} \to \texttt{o} \to \texttt{o} \to \texttt{prop}$, standing for *goal reduction* and *backward chaining* respectively, with the definitional clauses shown in figure 1. Here, specification contexts have the type $\texttt{olist}$, the type of lists of $\texttt{o}$, with constructors $\texttt{nil} : \texttt{olist}$ and $(\texttt{::}) : \texttt{o} \to \texttt{olist} \to \texttt{olist}$ (written infix), and a membership relation $\texttt{mem} : \texttt{o} \to \texttt{olist} \to \texttt{prop}$ that has the obvious inductive definition. In *Abella*, these two relations are displayed using the more evocative notation $\{L \vdash G\}$ and $\{L, [F] \vdash G\}$ for $\texttt{seq}\ L\ G$ and $\texttt{bch}\ L\ F\ G$. The final clause for $\texttt{seq}$ uses a separate predicate $\texttt{prog} : \texttt{o} \to \texttt{prop}$ 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.

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 λ-calculus, itself specified as an object logic in *HOHH*. The simple type system is represented using a new basic type $\texttt{ty}$ with two constructors, $\texttt{i} : \texttt{ty}$ (a basic sort), and $\texttt{arr} : \texttt{ty} \to \texttt{ty} \to \texttt{ty}$ for constructing arrow types. The λ-terms are typed using a different basic type $\texttt{tm}$ with two constructors: $\texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}$ and $\texttt{abs} : \texttt{ty} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}$. The λ-term $\lambda x{:}\texttt{i}.\, \lambda f{:}\texttt{i} \to \texttt{i}.\, fx$ would be represented as $\texttt{abs}\ \texttt{i}\ (\lambda x.\, \texttt{abs}\ (\texttt{arr}\ \texttt{i}\ \texttt{i})\ (\lambda f.\, \texttt{app}\ f\ x))$. The relation between terms (of type $\texttt{tm}$) and types (of type $\texttt{ty}$) is usually expressed in the form

of an inference system such as:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash (\lambda x{:}A.\, M) : A \to B}\,{\to_I} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\,{\to_E} \qquad \frac{}{\Gamma, x{:}A \vdash x : A}\,\mathsf{var}$$

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

$\texttt{pi}\ a{:}\texttt{ty}.\ \texttt{pi}\ b{:}\texttt{ty}.\ \texttt{pi}\ m{:}\texttt{tm}.\ \texttt{pi}\ n{:}\texttt{tm}.\ \texttt{of}\ m\ (\texttt{arr}\ a\ b) \Rightarrow \texttt{of}\ n\ a \Rightarrow \texttt{of}\ (\texttt{app}\ m\ n)\ b.$

$\texttt{pi}\ a{:}\texttt{ty}.\ \texttt{pi}\ b{:}\texttt{ty}.\ \texttt{pi}\ r{:}\texttt{tm} \to \texttt{tm}.\ (\texttt{pi}x{:}\texttt{tm}.\texttt{of}\ x\ \texttt{tm} \Rightarrow \texttt{of}\ (r\ x)\ \texttt{tm}) \Rightarrow \texttt{of}\ (\texttt{abs}\ a\ r)\ (\texttt{arr}\ a\ b).$

Note that there is no clause for `var`; rather, it is folded into an assumption in the body of the `abs` case, which delimits its scope. It is generally easier to read such clauses when they are written using the standard *λProlog* syntactic convention of using capital letters for universally closed variables, writing implications in the reverse direction with the head first, and separating assumptions by commas rather than repeated implications. Thus, the above clauses correspond to:

```
of (app M N) B ⇐ of M (arr A B), of N A.
of (abs A R) (arr A B) ⇐ pi x\ of (R x) B ⇐ of x A.
```

The formula `pi` $(\lambda x.\, F)$ is rendered as `pi x\ F` in the concrete syntax, and the scope of `x` extends as far to the right as possible. Note that all the types are inferred.

In the reasoning mode of *Abella*, the above *λProlog* specification is *imported* by reflecting all specification constants and types in the reasoning signature, and by generating a definition for `prog` that is true only for the two clauses for `of`. The typing judgement $x{:}A, y{:}B \vdash M : C$ in the inference system (2.3), for instance, would be represented by `seq` $(\texttt{of}\ y\ B :: \texttt{of}\ x\ A :: \texttt{nil})\ (\texttt{of}\ M\ C)$. As an example of reasoning on this specification, we can prove that `of` is deterministic in its second argument:

```
forall M A B, {⊢ of M A} → {⊢ of M B} → A = B.
```

This theorem is proved by induction on the derivation of one of the `seq` assumptions, such as the first one. This induction would repeatedly *match* the form `seq nil` $M\ A$ against the left hand sides of the definitional clauses in figure 1; for every successful match, the corresponding right hand side of the clause would give us new assumptions, which may then be used in the inductive hypothesis.

Initially, the only clause that matches is the final one for `seq` corresponding to backchaining on a program clause. In the case where the clause for `abs` is selected, the corresponding `bch` clause for it would in turn call `seq` with a different list of assumptions. Thus, the inductive argument cannot proceed with empty dynamic specification contexts (the first argument to `seq` and `bch`) alone: we must also allow for reasoning under an abstraction. This is achieved in the reasoning logic by inductively characterizing all such dynamic context extensions with a new atom, say $\texttt{ctx} : \texttt{olist} \to \texttt{prop}$, with the following inductive definitional clauses:

$\texttt{ctx}\ \texttt{nil} \triangleq \texttt{true}.$

$\forall A.\, \forall G.\, (\nabla x.\, \texttt{ctx}\ (\texttt{of}\ x\ A :: G)) \triangleq \texttt{ctx}\ G.$

We can then prove a stronger lemma:

```
forall G M A B, ctx G → {G ⊢ of M A} → {G ⊢ of M B} → A = B.
```

Now, when the dynamic context does grow when backchaining on the clause for `abs`, it will grow exactly by the form in the head of the second clause of `ctx`, *i.e.*, with a formula of

$$\phi\,(\Pi x{:}A.\,P) := \phi\,(A) \to \phi\,(P) \qquad \langle c\rangle := c \qquad \langle \lambda x{:}A.\,M\rangle := \lambda x{:}\phi\,(A).\,\langle M\rangle$$
$$\phi\,(a\ M_1\ \cdots\ M_n) := \texttt{lfobj} \qquad \langle x\rangle := x$$
$$\phi\,(\texttt{type}) := \texttt{lftype} \qquad \langle M_1\ M_2\rangle := \langle M_1\rangle\ \langle M_2\rangle$$

Figure 2 Encoding of *LF* types and kinds as simple types and *LF* objects as simply typed λ-terms.

the form `of` n A where n is a nominal constant that does not occur in A nor in the original context `G`. Thus, when we in turn backchain on the dynamic clauses (using the penultimate clause for `ctx` in figure 1), we will know the precise form of the selected clause.

3 An Adequate Translation of *LF* to *HOHH*

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

Kinds	$K ::= \texttt{type} \mid \Pi x{:}A.\,K$
Types	$A, B ::= a\ M_1 \ldots M_n \mid \Pi x{:}A.\,B$
Objects	$M, N ::= c \mid x \mid \lambda x{:}A.\,M \mid M\ N$

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 $A \to B$ 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 λ-calculus and its associated typing relation used in section 2.3. The λ-terms are encoded using the following signature:

```
ty  : type.                  tm  : type.
i   : ty.                    app : tm → tm → tm.
arr : ty → ty → ty.          abs : ty → (tm → tm) → tm.
```

For the typing relation `of`, in *LF* we declare it as a dependent type rather than as a predicate as in *HOHH*. The clauses of the `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 $\Pi x{:}A.\,B$.

```
of    : tm → ty → type.
ofApp : {A:ty} {B:ty} {M:tm} {N:tm}
          of M (arr A B) → of N A → of (app M N) B.
ofAbs : {A:ty} {B:ty} {R:tm → tm}
          ({x:tm} of x A → of (R x) B) → of (abs A R) (arr A B).
```

The *LF* type system is formally defined in [7] and will not be repeated here. Instead, we will directly give an adequate encoding of the *LF* type system in terms of *HOHH*, based on the variant of the encoding in [3] defined in [18], with the inverse mapping defined in [19].

The encoding proceeds in two steps. First, we transform our dependently typed terms into simply typed *HOHH* terms. The encoding of types and kinds is defined as a mapping, written $\phi\,(-)$. These types indicate that the term is an encoding of an *LF* type and an *LF* object, respectively. For each constant $c : P$ in the *LF* signature, we generate a simply typed term c of type $\phi\,(P)$. Using this mapping, the dependently typed λ-terms are converted into simply typed λ-terms using the mapping $\langle -\rangle$. Figure 2 contains the rules for both $\langle -\rangle$ and $\phi\,(-)$. Note that $\phi\,(-)$ erases not just the type dependencies but also the identities

$$\begin{aligned}
\{\{\Pi x{:}A.\, P\}\} &:= \lambda m{:}\phi\,(\Pi x{:}A.\, P).\, \texttt{pi}\ x{:}\phi\,(A).\, \{\{A\}\}x \Rightarrow \{\{P\}\}(m\ x) \\
\{\{a\ M_1\ \cdots\ M_n\}\} &:= \lambda m{:}\texttt{lfobj}.\, \texttt{hastype}\ m\ (a\ \langle M_1\rangle\ \cdots\ \langle M_n\rangle) \\
\{\{\texttt{type}\}\} &:= \lambda m{:}\texttt{lftype}.\, \texttt{istype}\ m
\end{aligned}$$

Figure 3 Encoding of *LF* types and kinds using the `hastype` and `istype` predicates.

of the types. For an atomic type $A = a\ M_1\ \cdots\ M_n$, we further write $\langle A\rangle$ to stand for $a\ \langle M_1\rangle\ \cdots\ \langle M_n\rangle$.

The second pass uses two new predicates, $\texttt{hastype} : \texttt{lfobj} \to \texttt{lftype} \to \texttt{o}$ and $\texttt{istype} : \texttt{lftype} \to \texttt{o}$, to encode the type and kind judgements of *LF*. Whenever $M : A$ is derivable in *LF* under a given signature, it must be the case that $\{\{A\}\}\langle M\rangle$ is derivable in *HOHH* from the clauses for `hastype` and `istype` produced from encoding the signature. Likewise, when $A : K$ is derivable, it should be the case that $\{\{K\}\}\langle A\rangle$ is derivable. The rules for this encoding are shown in figure 3.

▶ **Theorem 1** (Adequacy, [18]). *The LF hypothetical judgement $x_1 : P_1, \ldots, x_n : P_n \vdash M : A$ is derivable in the LF type theory [7] from an LF signature Σ if and only if the HOHH formula $\{\{P_1\}\}x_1 \Rightarrow \cdots \Rightarrow \{\{P_n\}\}x_n \Rightarrow \{\{A\}\}\langle M\rangle$ is derivable from the HOHH encoding of Σ according to the rules in figures 2 and 3.* ◀

Because this encoding is adequate, it is possible to define a right-inverse that maps a *HOHH* formula in the image of the translation in figure 3 back to an *LF* judgement. This inverse will be very useful in the next section where we will use the encoding of *LF* to extend the *2LL* via translations. The user of the system will not need to be aware of the details of the encoding as the *HOHH* formulas will be inverted into their corresponding *LF* judgements.

Defining such an inverse requires a small amount of care. We obviously cannot invert every *HOHH* formula, just those that correspond to a given signature. However, even for formulas constructed using the encodings of an *LF* signature, we may not necessarily be able to invert them; for instance, the formula may be the translation of a malformed or ill-typed *LF* judgement. This inverse will also not necessarily recover exactly the *LF* judgement used to construct the *HOHH* formula in the first place; rather, the inversion will only produce a unique inverse (if one exists) up to $\beta\eta$-conversion.

The inversion operation is defined in terms of the following four sequent forms:

$\Gamma \vdash \texttt{hastype}\ m\ a \longrightarrow M : A$	inverting typing; M, A output
$\Gamma \vdash \texttt{istype}\ a \longrightarrow A : \texttt{type}$	inverting kinding; A output
$\Gamma \vdash m : A \rightsquigarrow M$	inverting canonical terms; M output
$\Gamma \vdash m \rightsquigarrow M : A$	inverting atomic terms; M, A output

The rules are shown in figure 4. In each case, Γ 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* types, M and N over *LF* terms, F and G over *HOHH* formulas and a, m and n over simply typed λ-terms produced by $\langle - \rangle$. 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 [19] (in a slightly more general form).

▶ **Theorem 2** (Right inverse). *If $\{\{P\}\}\langle U\rangle = F$ under the translation of Γ and $\Gamma \vdash F \longrightarrow U' : P$, then $\Gamma \vdash U =_{\beta\eta} U' : P$ in LF.*

Proof. By structural induction on the inversion derivation. Note the requirement for η-contraction of the term to a variable in the second premise of inv-nest is necessary, for

$$\frac{(a{:}\Pi x_1{:}A_1.\ \cdots \Pi x_k{:}A_k.\ B) \in \Gamma \qquad \begin{array}{c}\Gamma \vdash m_1 : A_1 \rightsquigarrow M_1 \\ \cdots \\ \Gamma, x_1{:}A_1, \ldots, x_{k-1}{:}A_{k-1} \vdash m_k : A_k \rightsquigarrow M_k \\ \Gamma \vdash m : a\ M_1\ \cdots\ M_k \rightsquigarrow M\end{array}}{\Gamma \vdash \texttt{hastype}\ m\ (a\ m_1\ \cdots\ m_k) \longrightarrow M : a\ M_1\ \cdots\ M_k}\ \text{inv-has}$$

$$\frac{(a{:}\Pi x_1{:}A_1.\ \cdots \Pi x_k{:}A_k.\ \texttt{type}) \in \Gamma \qquad \begin{array}{c}\Gamma \vdash m_1 : A_1 \rightsquigarrow M_1 \\ \cdots \\ \Gamma, x_1{:}A_1, \ldots, x_{k-1}{:}A_{k-1} \vdash m_k : A_k \rightsquigarrow M_k\end{array}}{\Gamma \vdash \texttt{istype}\ (a\ m_1\ \cdots\ m_k) \longrightarrow a\ M_1\ \cdots\ M_k : \texttt{type}}\ \text{inv-is}$$

$$\frac{\Gamma \vdash F \longrightarrow R : A \qquad R =_\eta x \qquad \Gamma, x{:}A \vdash G \longrightarrow M : B}{\Gamma \vdash \texttt{pi}\ x{:}a.\ F \Rightarrow G \longrightarrow (\lambda x{:}A.\ M) : \Pi x{:}A.\ B}\ \text{inv-nest}$$

$$\frac{\Gamma, x{:}A \vdash m : B \rightsquigarrow M}{\Gamma \vdash (\lambda x{:}T.\ m) : (\Pi x{:}A.\ B) \rightsquigarrow \lambda x{:}A.\ M}\ \text{inv-lam} \qquad \frac{\Gamma \vdash m \rightsquigarrow M : \Pi x{:}A.\ B \qquad \Gamma \vdash n : A \rightsquigarrow N}{\Gamma \vdash m\ n \rightsquigarrow M\ N : B[N/x]}\ \text{inv-app}$$

$$\frac{\Gamma \vdash m \rightsquigarrow M : A}{\Gamma \vdash m : A \rightsquigarrow M}\ \text{inv-switch} \qquad \frac{(x{:}A) \in G}{\Gamma \vdash x \rightsquigarrow x : A}\ \text{inv-hyp} \qquad \frac{(c{:}A) \in G}{\Gamma \vdash c \rightsquigarrow c : A}\ \text{inv-const}$$

Figure 4 Inverting the *LF* encoding of judgements ($\longrightarrow$) and terms ($\rightsquigarrow$).

otherwise the rule would produce an unsound abstraction. If the formula F was generated from the translation of figures 2 and 3, then this η-contraction check will always succeed. ◀

4 Translational Two-level Logic Approach

We will now use both the translation of *LF* signatures to *HOHH* formulas and its inverse to extend the *2LL* in such a way that we can reason about *LF* signatures just as we were able to reason on *λProlog* specifications as shown in the example of section 2.3.

4.1 Importing the *LF* Specification

As the type system of *LF* and $\mathcal{G}$ are different, we cannot directly reflect the constants and types of the *LF* specification logic like we did with *HOHH* in section 2. Instead, for every *LF* constant c of *LF* type or kind P in the *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 `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 β-normal. They are also η-long, because the definition of $\{\{-\}\}$ traverses the type or kind until it is atomic.

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

```
ty  : lftype.                    tm  : lftype.
i   : lfobj.                     app : lfobj → lfobj → lfobj.
arr : lfobj → lfobj.             abs : lfobj → (lfobj → lfobj) → lfobj.

of    : lfobj → lfobj → lftype.
ofApp : lfobj → lfobj → lfobj → lfobj → lfobj → lfobj → lfobj.
ofAbs : lfobj → lfobj → (lfobj → lfobj) →
            (lfobj → lfobj → lfobj) → lfobj.
```

The following clauses are then added to `prog` by step (2) described above:

```
lfisty ty.
lfhas i ty.
lfhas (arr Z1 Z2) ty ⇐ lfhas Z1 ty, lfhas Z2 ty.
```

```
lfisty tm.
lfhas (app Z1 Z2) tm ⇐ lfhas Z1 tm, lfhas Z2 tm.
lfhas (abs Z1 Z2) tm ⇐
  lfhas Z1 ty, (pi x\ lfhas x tm ⇒ lfhas (Z2 x) tm).

lfisty (of Z1 Z2) ⇐ lfhas Z1 tm, lfhas Z2 ty.
lfhas (ofApp A B M N Z1 Z2) (of (app M N) B) ⇐
  lfhas A ty, lfhas B ty, lfhas M tm, lfhas N tm,
  lfhas Z1 (of M (arr A B)), lfhas Z2 (of N A).
lfhas (ofAbs A B R Z1) (of (abs A (x\ R x)) (arr A B)) ⇐
  lfhas A ty, lfhas B ty,
  (pi x\ lfhas x tm ⇒ lfhas (R x) tm),
  (pi x\ lfhas x tm ⇒ pi z\ lfhas z (of x A) ⇒
            lfhas (Z1 x z) (of (R x) B)).
```

The variables named $\mathtt{Z}_i$ are generated by the translator for those variables that are omitted from the input signature by the use of $\to$ instead of Π. We write clauses using standard *λProlog* 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 *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 `of`. For example, consider the case of `ofApp`. Two of the formulas, `hastype` Z_1 (`of` M (`arr` A B)) and `hastype` Z_2 (`of` N A) are already present in nearly this form in the *HOHH* specification. The additional assumptions are just repetitions of the typing assumptions for the arguments to `ofApp`; indeed, many of them are redundant since the `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 [18, 19].

4.2 Representing *LF* Hypothetical Judgements

We use the concrete syntax $\langle M : A\rangle$ or $\langle A : K\rangle$ to depict $\{\{A\}\}M$ 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: $\langle x_1 : P_1, ..., x_n : P_n \vdash U : P\rangle$ as an abbreviation for: `seq` $(\langle x_1 : P_1\rangle$ `::` $\cdots$ `::` $\langle x_n : P_n\rangle)$ $(\langle U : P\rangle)$. As an example,[2] the uniqueness theorem for the `of` relation is (eliding types):

$$\forall G, M, A, B, P_1, P_2, \mathtt{ctx}\ G \supset \langle G \vdash P_1 : \mathtt{of}\ M\ A\rangle \supset \langle G \vdash P_2 : \mathtt{of}\ M\ B\rangle \supset A = B. \qquad (1)$$

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

Of course, in order to prove this theorem we would require a suitable `ctx` definition. Unlike in the simply typed case, the recursive case for λ-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.

$$\mathtt{ctx}\ \mathtt{nil} \triangleq \mathtt{true}.$$

$$\nabla x{:}\mathtt{lfobj}.\, \nabla p{:}\mathtt{lfobj}.\, \mathtt{ctx}\ (\langle x : \mathtt{tm}\rangle :: \langle p : \mathtt{of}\ x\ A\rangle :: G) \triangleq \mathtt{ctx}\ G.$$

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:

$$\nabla x{:}\mathtt{lfobj}.\, \nabla p{:}\mathtt{lfobj}.\, \mathtt{ctx}\ (\langle p : \mathtt{of}\ x\ A\rangle :: \langle x : \mathtt{tm}\rangle :: G) \triangleq \mathtt{ctx}\ G.$$

[2] The full *Abella*/*LF* development may be interactively browsed online at `http://abella-prover.org/lf`.

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.

```
forall G, nabla p x,
  ctx (G x p) → mem ⟨x : tm⟩ (G x p) →
    exists A, mem ⟨p : of x A⟩ (G x p) ∧ fresh p A ∧ fresh x A.
```

The dependency of `G` on `x` and `p` is indicated explicitly using application. For `A`, this dependency is implicit, because the `exists` occurs in the scope of the corresponding `nabla`s, so we use the predicate `fresh : lfobj → lfobj → prop` 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: $\forall A.\, (\nabla x.\, \texttt{fresh}\ x\ A) \triangleq \texttt{true}$.

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

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

The rest of the proof is fairly systematic, and largely identical in structure to that of the *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).

```
forall G M A B P1 P2, ctx G →
  ⟨G ⊢ P1 : of M A⟩ → ⟨G ⊢ P2 : of M B⟩ → P1 = P2.
```

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 `lf` branch of the *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 *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 $\langle -\rangle$, 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 *Abella* itself.

[3] Details for downloading and building this branch can be found in `http://abella-prover.org/lf`.

- 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 *Twelf* system [15], 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 *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 *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 *simple* translation from [18], which is itself based on earlier work [3], while the inversion on terms is similar to the definition in [19], omitting meta-variables. Various optimized versions of this translation have been used to use *λProlog* as an engine for logic programming with *LF* specifications; in particular, the *Parinati* system [18] and its extension to meta-variables in [19]. 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 *Abella*/*LF* variant in the future. The combination of *Parinati* and *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 *Abella* ecosystem as *HOHH* and *λProlog* have traditionally been.

There are many other systems designed to reason with and about *LF* specifications. The most mature implementation is *Twelf* [15], which has a very efficient type-checker incorporating sophisticated term and type reconstruction. As mentioned in section 4.3, we use *Twelf* to type-check and elaborate the *LF* specifications we import in *Abella*. In addition to the type-checker, *Twelf* has a suite of meta-theoretic tools that can verify certain properties of *LF* specifications, such as that a declared relation determines a total function. *Twelf* is, however, not powerful enough to reason inductively on arbitrary *LF* derivations. For example, although *Twelf* can check coverage, it cannot express the logical formula that corresponds to the coverage property.

Some of these expressive deficiencies of *Twelf* have been addressed in the *Delphin* [17] and *Beluga* [16] systems that add a functional programming language that can manipulate and reason inductively on *LF* syntax. The *Beluga* system, in particular, extends the *LF* type theory with *contextual modal types* [13] that give a type-theoretic treatment for meta-variables and explicit substitutions; in more recent work, *Beluga* also allows abstraction over contexts and substitutions [2]. The type-checker of *Beluga* is therefore very sophisticated and performs many kinds of reasoning on contexts automatically that must be done manually in *Abella*. On the flip-side, *Abella* has a small trusted core based on the logic $\mathcal{G}$ with a well-understood and—importantly!—stable proof system [11, 5]. It would be interesting to formally compare

the representational abilities of *Abella*/LF and *Beluga*. Moreover, *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.

Acknowledgements

Thanks to Dale Miller and Gopalan Nadathur for many discussions about all aspects of this work and for many useful comments on earlier drafts. Thanks also to Taus Brock-Nannestad for useful comments on the final draft. This work has been partially supported by the NSF Grant CCF-0917140 and by the ERC Advanced Grant *ProofCert*. Opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the European Research Council.

References

1 O. S. Bélanger and K. Chaudhuri. Automatically deriving schematic theorems for dynamic contexts. In *LFMTP '14*, pages 9:1–9:8. ACM, 2014.

2 A. Cave and B. Pientka. Programming with binders and indexed data-types. In *POPL*, pages 413–424. ACM, 2012.

3 A. Felty and D. Miller. Encoding a dependent-type λ-calculus in a logic programming language. In *CADE*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.

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

5 A. Gacek, D. Miller, and G. Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

6 A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

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

8 R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.

9 D. Miller and G. Nadathur. A computational logic approach to syntax and semantics. 10th Symp. of the Mathematical Foundations of Computer Science, May 1985.

10 D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

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

12 G. Nadathur and D. J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In *CADE*, number 1632 in LNAI, pages 287–291. Springer, 1999.

13 A. Nanevski, F. Pfenning, and B. Pientka. Contextual model type theory. *ACM Trans. on Computational Logic*, 9(3):1–49, 2008.

14 F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

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

16 B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR*, pages 15–21. LNCS 6173, 2010.

17 A. Poswolsky and C. Schürmann. System description: Delphin - A functional programming language for deductive systems. In *LFMTP*, volume 228, pages 113–120, 2008.

18 Z. Snow, D. Baelde, and G. Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *PPDP*, pages 187–198, 2010.

19 M. Southern and G. Nadathur. A λProlog based animation of Twelf specifications, July 2014. Available at `http://arxiv.org/abs/1407.1545`.

20 Y. Wang, K. Chaudhuri, A. Gacek, and G. Nadathur. Reasoning about higher-order relational specifications. In *PPDP*, pages 157–168, Madrid, Spain, Sept. 2013.

21 The Abella web-site. `http://abella-prover.org/`, 2013.