Mechanized Meta-Reasoning Using a Hybrid HOAS/de Bruijn Representation and Reflection

Jason Hickey  Aleksey Nogin  Xin Yu  Alexei Kopylov

Department of Computer Science, 256-80
California Institute of Technology
Pasadena, CA 91125
{jyh,nogin,xiny,kopylov}@cs.caltech.edu

Abstract

We investigate the development of a general-purpose framework for mechanized reasoning about the meta-theory of programming languages. In order to provide a standard, uniform account of a programming language, we propose to define it as a logic in a logical framework, using the same mechanisms for definition, reasoning, and automation that are available to other logics. Then, in order to reason about the language’s meta-theory, we use reflection to inject the programming language into a (usually richer and more expressive) meta-theory.

One of the key features of our approach is that structure of the language is preserved when it is reflected, including variables, meta-variables, and binding structure. This allows the structure of proofs to be preserved as well, and there is a one-to-one map from proof steps in the original programming logic to proof steps in the reflected logic. The act of reflecting a language is automated; all definitions, theorems, and proofs are preserved by the transformation and all the key lemmas (such as proof and structural induction) are automatically derived.

The principal representation used by the reflected logic is higher-order abstract syntax (HOAS). However, reasoning about terms in HOAS can be awkward in some cases, especially for variables. For this reason, we define a computationally equivalent variable-free de Bruijn representation that is interchangeable with the HOAS in all contexts. The de Bruijn representation inherits the properties of substitution and alpha-equality from the logical framework, and it is not complicated by administrative issues like variable renaming.

We further develop the concepts and principles of proofs, provability, and structural and proof induction. This work is fully implemented in the MetaPRL theorem prover. We illustrate with an application to $\text{F}_\epsilon$, as defined in the POPmark challenge.

Categories and Subject Descriptors

F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages—Operations on languages; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Mechanical theorem proving; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms

Languages, Theory, Verification

Keywords

Reflection, Higher-Order Abstract Syntax, Meta-Theory, Type Theory, MetaPRL, NuPRL, Languages with Bindings, Mechanized Reasoning

1. Introduction

When designing a framework for mechanized meta-reasoning about the theory of programming languages, one is confronted with a myriad of questions and choices.

We will assume that we are working in the context of a logical framework, so there are at least three logics in consideration—$\mathbb{P}$: the programming language (also called the “object logic”); $\mathbb{M}$: the meta-logic in which reasoning about the programming language is to be performed; and $\mathbb{F}$: the meta-meta-logic, or framework logic, in which the meta-logic $\mathbb{M}$ is defined. The first step in the process is to define a representation of programs and judgments in $\mathbb{P}$ in terms of formulas, propositions, and sentences in $\mathbb{M}$. This leads to the following questions.

- What meta-logic $\mathbb{M}$ will be used?
- How are the terms and judgments of the programming language $\mathbb{P}$ to be represented in $\mathbb{M}$?
- What are the induction principles on which reasoning will be based?
- Are the framework logic $\mathbb{F}$ and meta-logic $\mathbb{M}$ expressive enough to state (and prove) the expected meta-properties, or are extensions needed?
- What logical facts and automation can be re-used? Will any additional automation be needed?

The choice of framework logic $\mathbb{F}$ and meta-logic $\mathbb{M}$ are in some ways the easier choices. These logics are based mainly on the choice of the prover, which then restricts the set of choices for framework and meta-logics (in many cases $\mathbb{F}$ and $\mathbb{M}$ are one and the same).

However, the choice of representation is central, and choices abound. In particular, the issue of naming and binding is a critical choice that has a pervasive effect on the entire endeavor. For examples, one may use nameless or de Bruijn encodings for variables [7], nominal representations [21,5], or higher-order abstract syntax [20] (HOAS), with various tradeoffs. The de Bruijn encoding provides a very concrete representation with a clear induction principle, but reasoning is cluttered by superfluous artifacts like the need to perform name shifting, and one gets very little built-in help from the prover for these issues. At the other extreme, HOAS provides a clean abstract representation with excellent support from
the prover, but variable names are inaccessible and the induction scheme can be hard to formulate. Nominal approaches are in between; names are accessible, the representation is mildly cluttered by explicit renamings, but frequently the existing framework logic $\mathcal{F}$ or meta-logic $\mathcal{M}$ must be extended to include explicit naming contexts.

In this paper we explore a hybrid approach using a combined HOAS/de Bruijn representation for terms, and using reflection to provide a uniform representation map $\mathcal{F} + P \rightarrow \mathcal{F} + \mathcal{F} + \mathcal{P}$. In our approach, a programming language $P$ is defined in the logical framework $\mathcal{F}$ as a particular logic in the usual way, then reflected *en masse* to form a subtheory of the meta-logic $\mathcal{M}$. A key property of the reflected representation is that it preserves structure—that is, there is a one-to-one mapping from proof steps in the logic $P$ to reflected proof steps in the meta-logic $\mathcal{M}$. In the manner of HOAS, binding structure is also preserved by the transformation, both for formulas and for (sequent) judgments.

To address the issue of computation and induction over terms, each reflected term has two equivalent forms. One is a HOAS representation, where variables in the programming language $P$ are represented by variables in the meta-language $\mathcal{M}$, and binding is preserved. On top of it there is a de Bruijn representation layer, where binders are specified by arity, and variables are denoted with numerical indices. These two representations are formally and computationally indistinguishable, which allows the appropriate representation to be selected at the appropriate time. For example, the HOAS representation is normally the preferred form for users because of its clarity, but the de Bruijn representation is more appropriate for computations that involve induction or computation on variables (for example, computing the free variables of a term).

While the issues of representation and reflection can be factored (for example, it is possible to use the hybrid representation without also requiring reflection), reflection provides a context that is convenient for talking about representations and specifying properties of proofs in a uniform way that applies to all programming languages $P$ and other logical theories that can be specified as a logic in $\mathcal{F}$. Furthermore, the act of reflecting a language $P$ is automated, and the principles for structural and proof induction are automatically derived for every reflected theory. We have carried out a complete formal account of this work in the MetaPRL logical framework[12, 15]; all formalizations are available online[15, 14]. An important feature of this work is that no extensions to the meta-logic $\mathcal{M}$ or framework logic $\mathcal{F}$ are needed.

The development proceeds as follows. First, we develop the hybrid HOAS/de Bruijn representation for simple terms (Section 2), and then we extend it to sequent judgments (Section 3), where we rely on the use of "teleportation" to perform sequent context induction. Once the representation for terms is defined, we proceed to develop a representation of proofs (Section 4) and the corresponding principle for proof induction (Section 5). We proceed with some examples relating to the POPlmark challenge[4] (Section 7) where we also develop the principle for structural induction (Section 7.4). We finish with discussion and related work (Sections 8 and 9).

2. Hybrid representation

By reflection, we mean the ability to use one logic to reason about another, in this case using the meta-logic $\mathcal{M}$ to reason about the programming language $P$. At its core, a reflection system has two parts. There is a representation function, written $\Gamma^\gamma$, which computes the representation or "quotation" of a logical formula $t$, where $t$ is a formula in $P$ and $\Gamma^\gamma$ is the corresponding formula in $\mathcal{M}$. Then, there is a provability operator, written $\square t$, which is a predicate specifying that $t$ is the quotation of a provable formula.

$\begin{align*}
t & ::= x \quad \text{object (first-order) variables} \\
& | z[t_1; \cdots ; t_n] \quad \text{second-order meta-variables} \\
& | \Gamma \vdash t \quad \text{sequents} \\
& | op\{b_1; \cdots ; b_m\} \quad \text{concrete terms} \\
b & ::= x_1, x_2, \cdots , x_n \quad \text{bound terms} \\
\Gamma & ::= h_1; \cdots ; h_m \quad \text{sequent contexts} \\
h & ::= X[t_1; \cdots ; t_n] \quad \text{context meta-variables} \\
& | x : t \quad \text{hypothesis bindings and terms} \\
\mathcal{P} & ::= R_1; R_2; \cdots ; R_n \quad \text{a logic} \\
R & ::= t_1 \rightarrow \cdots \rightarrow t_n \quad \text{inference rule} \\
& \quad \text{(t_i are closed w.r.t. object variables)}
\end{align*}$

Figure 1. Syntax of formulas and logics

2.1 Terminology

We assume that the language of the logical framework $\mathcal{F}$ contains sequents, second-order meta-variables, and terms, as shown in Figure 1. A term $t$ is a formula containing variables, concrete terms, or sequents. A concrete term $op\{b_1; \cdots ; b_m\}$ has a name $op$, as well as some subterms $b_1, \ldots , b_m$ that have possible binding occurrences of variables. For example, a term for representing the sum $i + j$ might be defined as $add\{i; j\}$ (normally, we will omit the leading , if there are no binders, writing it as $add\{i\}$). A lambda-abstraction $\lambda x.t$ would include a binding occurrence $\lambda x.\{x; t\}$. Note that in this case, the primitive binding construct is the bound term $h$, and $\lambda$-binders are a defined term. An alternate choice would be to use a single primitive $\lambda$ binder (for example, as is done in LF[9]).

A sequent $\Gamma \vdash t$ includes a sequent context $\Gamma$, which is a sequence of dependent hypotheses $h_1; \cdots ; h_m$, where each hypothesis is a binding $x : t$ or a context variable $X[t_1; \cdots ; t_n]$ ($x$ and $X$ bind to the right). Second-order meta-variables $z[t_1; \cdots ; t_n]$ and context variables $X[t_1; \cdots ; t_n]$ include zero-or more term arguments $t_1, \ldots , t_n$. These meta-variables represent closed substitution functions, and are implicitly universally quantified for each rule in which they appear[16]. For example, a second-order variable $z[t]$ represents all closed terms (we will normally omit empty brackets, writing simply $t$). The second-order variable $z[x]$ represents all terms with zero-or more occurrences of the variable $x$ (that is, any term where $x$ is the only free variable).

To illustrate, consider the "substitution lemma" that is valid in many logics. In textbook notation, it might be written as follows, where $t_1[x \leftarrow s]$ represents the substitution of $s$ for $x$ in $t_1$, and $x \notin bv(\Delta)$ is a side-condition of the rule.

$$\begin{align*}
\Gamma , x : t_1; \Delta \vdash t_1 \in t_2 \\
\Gamma , \Delta \vdash s \in t_3 \\
\Delta \vdash t_1[x \leftarrow s] \in t_2
\end{align*}$$

In our more concrete notation, $s, t_1, t_2, t_3$ are all represented with second-order variables, and $\Gamma , \Delta$ with context variables. Substitutions are defined using the term arguments; rules are defined using the meta-implication $\rightarrow$ , and we consider all meta-variables to be universally quantified in a rule. The concrete version is written as follows (where we use $s \in t$ as a pretty form for a term member$\{s; t\}$, and $z_i$ are second-order meta-variables).

$$\begin{align*}
(X ; x ; z_3 ; Y \vdash z_1[x] \in z_2) & \longrightarrow \\
(X ; Y \vdash z_0 \in z_3) & \longrightarrow \\
(X ; Y \vdash z_1[z_0] \in z_2)
\end{align*}$$

1 Strictly speaking, context variables are *bindings* and meta-variables have context arguments in addition to term arguments. This does not affect the presentation here, and we omit them. Further discussion can be found in Hickey et.al. [13].
In order to preserve a one-to-one correspondence between proofs for this purpose, the explicit binder \( \Gamma \ \vdash \ t \) becomes a statement about provability in \( \Gamma \vdash \Box \ t \). The context variable \( Z \) is fresh, and each sequent \( Z \vdash \Box \ t \) is a judgment in the meta-logic \( M \) about provability in \( P \). As we will see in Section 4, \( \Box \cdot \) is a specific binary predicate defined in meta-logic \( M \) using the standard definitional mechanisms provided by the logical framework \( F \). In formally, the reflected form of the rule states that if each premise \( t_1 \), \( \ldots \), \( t_n \) is provable in logic \( P \), then so is \( t \). A key goal is that the reflected rule \( \Gamma \vdash \Box \ t \) must be automatically derivable from the definition of \( P \). For clarity, when reasoning about a single logic we will normally omit the subscript \( P \) and simply write \( \Box \).

The choice of meta-logic is arbitrary, but it must be expressive enough that the term encodings can be stated and reasoning can be performed. For our purposes, we have chosen to use computational type theory (CTT), which is a variant of Martin-Löf intuitionistic type theory as implemented in the MetaPRL logical framework [13]. Note that in this meta-logic, the reflected rules \( \Gamma \vdash \Box \ t \) are sometimes required to include additional well-formedness constraints on the typing of the meta-variables.

Returning to our example, the quoted form of the substitution lemma (2.1) is as follows, where we write \( t \vdash \Box x : r \ t \).

\[
\begin{align*}
Z &\vdash \Box (X; x : z_1 ; X \vdash \Box x : z_1 x : \forall \ t) \quad (2.1I)
\end{align*}
\]

The operators have been quoted (in this case \( \vdash \Box \cdot \) and \( \vdash \Box \in \cdot \)), and the theorem is now a statement about provability expressed in the meta-logic as \( Z \vdash \Box \cdot \). Only the operator names have been changed, otherwise the structure, including variables and binding, has not changed.

For an example with binding, consider the rule for universal-introduction, shown below with the translated version. In this case,
Concrete representations

\[ op \{ r_1; \ldots; r_n \} \equiv \top \{ op \{ r_1; \ldots; r_n \} \} \equiv \text{inr} \{ \{ op \}, \{ r_1; \ldots; r_n \} \} \]

\[ \lambda_p \cdot x \cdot r \equiv \text{inl} \{ \lambda x \cdot r \} \]

Term operations

\[ \text{subterm} \{ r_1; r_2 \} \equiv \text{match} \ r_1 \ \text{with} \]

short hand: \[ r_1 \od r_2 \]

\[ \{ \text{int} (f) \rightarrow f (r_2) \} \]

\[ \{ \text{inr} \{ o, s \} \rightarrow \bullet \} \]

\[ \text{nth} \ _\text{subterm} \{ r; i \} \equiv \text{match} \ r \ \text{with} \]

\[ \{ \text{int} (f) \rightarrow \lambda_p \cdot x \cdot \text{nth} \ _\text{subterm} \{ f (x); i \} \} \]

\[ \{ \text{inr} \{ o, s \} \rightarrow \text{nth} \{ x; i \} \} \]

\[ \text{num} \ _\text{subterms} \{ r \} \equiv \text{match} \ r \ \text{with} \]

\[ \{ \text{int} (f) \rightarrow \text{num} \ _\text{subterms} \{ f (\bullet) \} \} \]

\[ \{ \text{inr} \{ o, s \} \rightarrow \text{length} \{ x \} \} \]

\[ \text{bdepth} \{ r \} \equiv \text{match} \ r \ \text{with} \]

short hand: \[ | r | \]

\[ \{ \text{int} (f) \rightarrow \text{bdepth} \{ f (\bullet) \} + 1 \} \]

\[ \{ \text{inr} \{ o, s \} \rightarrow 0 \} \]


Figure 4. Concrete representation in the meta-logic \( \mathcal{M} \)

the binder \( x \) is translated to a meta-binder with \( \lambda_p \).

\[
\begin{align*}
X \cdot x \cdot z_1 \vdash z_2 [x] & \rightarrow \\
X \vdash \forall x \cdot z_1 \vdash z_2 [x]
\end{align*}
\]

\[
Z \vdash \Box (X \cdot x \cdot z_1 \vdash \gamma \gamma z_2 [x]) \rightarrow \\
Z \vdash \Box (X \vdash \gamma \gamma \gamma \gamma z_2 [z_1 \cdot \lambda_p \cdot x \cdot z_2 [x]])
\]

2.4 Concrete representation in the meta-logic \( \mathcal{M} \)

Let us postpone discussion of sequents for the moment, and focus on the essential term representatives \( \overline{op} \{ r_1; \ldots; r_n \} \) and \( \lambda_p \cdot x \cdot r \). We must decide on a concrete representation for these terms in the meta-logic \( \mathcal{M} \). As a requirement, the two kinds of terms should be distinguishable from one another, and they should be defined in such a way that concrete term operations are computable.

The concrete definitions are shown in Figure 4, and are based on the term representatives introduced in Nogin et al. [18]. A term \( \overline{op} \{ r_1; \ldots; r_n \} \) is defined as the right-injection of a pair of the operator and the list of subterms (we will also use the shorthand \( \overline{T} \{ \overline{op} \{ r_1; \ldots; r_n \} \} \)). The \( \lambda_p \cdot x \cdot r \) is defined as the left injection of a lambda abstraction. Since we are working with computational type theory as the meta-logic, we need not give types for these terms yet, though that is of course among our eventual goals.

Figure 4 also shows a few terms operations that are defined over reflected terms. Here, the symbol \( \bullet \) is used to denote a dummy or error term. That is, a substitution is defined only for binders, with the following computational equivalence (where we use \( \leftrightarrow \) to denote computational equivalence).

\( \lambda_p \cdot x \cdot r \{ x_1 \} \od r_2 \leftrightarrow r_1 \od r_2 \)

The \( \text{nth} \ _\text{subterm} \{ r; i \} \) is another interesting computation in that it preserves bindings. An example computation is as follows.

\( \lambda_p \cdot x_1; \ldots; x_n \cdot T \{ x; [\ldots; s_i [x_1; \ldots; x_n]; \ldots]; i \} \leftrightarrow \lambda_p \cdot x_1; \ldots; x_n \cdot s_i [x_1; \ldots; x_n] \)

2.5 de Bruijn representation

At this point, the representation function has been defined, and we have also defined concrete representatives for reflected terms. We have yet to define a type and induction principle for these terms. As mentioned in the introduction, defining an induction principle for the HOAS representation can be difficult. The issue is how to deal with bindings. We have hinted at a solution with the

de Bruijn terms

\[
\begin{align*}
V \{ n; m \} & \equiv \lambda_p \cdot x_1; \ldots; x_n \cdot \lambda_p \cdot x \cdot \lambda_p \cdot y_1; \ldots; y_m \cdot z \\
B \{ n; op; [s_1; \ldots; s_m] \} & \equiv \lambda_p \cdot x_1; \ldots; x_n \cdot T \{ op; [s_1 \od x_1 \od \ldots \od x_n; \ldots; s_m \od x_1 \od \ldots \od x_n] \}
\end{align*}
\]

Equivalences for variables

\[
\begin{align*}
\lambda_p \cdot x \cdot V \{ n; m \} & \leftrightarrow V \{ n + 1; m \} \\
V \{ 0; m \} & @ r \leftrightarrow r \\
V \{ n + 1; m \} & @ r \leftrightarrow V \{ n; m \}
\end{align*}
\]

Equivalences for terms

\[
\begin{align*}
B \{ 0; op; [s_1; \ldots; s_m] \} & \leftrightarrow \top \{ op; [s_1; \ldots; s_m] \}
\end{align*}
\]

\[
\begin{align*}
B \{ n + 1; op; [s_1; \ldots; s_m] \} & @ r \quad (\text{sub-term-commutes}) \\
& \leftrightarrow B \{ n; op; [s_1 @ r; \ldots; s_m @ r] \}
\end{align*}
\]

\[
\begin{align*}
\lambda_p \cdot B \{ n; op; [s_1; \ldots; s_m] \} & \leftrightarrow B \{ n + 1; op; \lambda_p \cdot x \cdot s_1 [x]; \ldots; \lambda_p \cdot x \cdot s_m [x] \}
\end{align*}
\]

\[
\lambda_p \cdot x \cdot (r @ x) \leftrightarrow r \quad (\text{if} | r | \neq 0) \quad (\text{eta})
\]

Figure 5. Definitions for the de Bruijn representation
Type definitions
\[
\begin{align*}
\text{dom}[T] & \equiv \mathbb{N} \times \mathbb{N} + (\Sigma n : \mathbb{N}. \Sigma o : f \cdot (s ; T \text{ list} \mid \text{compatible} \{ n ; o ; s \})) \\
f(t) & \equiv \text{match} \; \text{with} \\
& \begin{cases} 
\text{inl} \{ (n ; m) \} & \rightarrow \mathbb{V} \{ (n ; m) \} \\
\text{inr} \{ (o ; s) \} & \rightarrow B \{ n ; o ; s \}
\end{cases} \\
T_0 & \equiv \text{void} \\
T_{i+1} & \equiv \text{img} \{ \text{dom}[T_i] ; x . f(s) \} \\
B \text{Term} & \equiv \bigcup_{i \in \mathbb{N}} T_i
\end{align*}
\]

Depth compatibility
\[
\text{compatible}[n;o;s] \equiv \\
\begin{cases} 
\forall i \in \{0..\text{length}(s) - 1\}, \\
bdepth(nth\{s;i\}) = n + \text{arity}\{o;i\}
\end{cases}
\]

Figure 6. Definition of the type BTerm

HOAS form corresponds closely to the original expression, while the de Bruijn form is variable-free.

It should be noted that the de Bruijn representatives \(\mathbb{V}[n;m]\) and \(B[n;o;s]\) and the corresponding destructors are defined in terms of the HOAS operations. They do not add any additional properties that were not already provable in the meta-logic—in other words, the extension is conservative. In our implementation, all equivalences have been formally verified [18].

2.6 The type of reflected terms

Naturally, since we wish to reason about programs in type theory, we should give a type that specifies the collection of reflected terms. For this we rely on the function image type, generated by Nogin and Kopylov [17]. The image type has the form \(\text{img} \{ A ; x \cdot f[x] \}\) where \(f[x]\) is an arbitrary function, \(A\) is its domain, and the type includes exactly those terms \(f[a]\) for each \(a \in A\).

While it would be straightforward to define a type containing all de Bruijn-style terms, we plan to be a bit more careful and include only those where the binding depths (defined by bdepth in Figure 4) are “sensible.” For example, suppose our language contains a term called let with the expected form.

\[
\begin{align*}
\text{let } x = t_1 \text { in } t_2[x] \\
T[\text{let } x = \cdot ; t_1 \cdot t_2[x] ]
\end{align*}
\]

In this case, the term \(T[\text{let } x = \cdot ; t_1 \cdot t_2[x] ]\) makes sense only if the subterm \(\lambda x . t_2[x]\) contains exactly one more binder than the subterm \(\lambda x . t_1\). In other words, for any reflected term \(B \{ n ; \text{let } x = \cdot ; r_1 ; r_2 \}\) we require that the binding depths are well-formed.

\[
\begin{align*}
bdepth(r_1) & = n \\
bdepth(r_2) & = n + 1
\end{align*}
\]

So far, we have assumed that the type \(f\) of all operators was just a set of constants with a decidable equality. We now strengthen that by requiring that \(f\) must be equipped with a binding signature [22] and the arity of an operator \(op\) must be computable. The term \(\text{arity}\{op\}\) represents a list of natural numbers specifying the binding arities of the corresponding subterms. The following are some examples.

\[
\begin{align*}
\text{arity}(\lambda \text{ lambda}) & = [1] \\
\text{arity}(\lambda \text{ let}) & = [0 ; 1] \\
\text{arity}(\lambda \text{ decide}) & = [0 ; 1 ; 1]
\end{align*}
\]

With this technicality in place, we define the type \(B\text{Term}\) of reflected terms in Figure 6. In this formulation, the function \(f(t)\) produces a variable or bound term in de Bruijn form. The type \(T_i\) represents a quoted term tree with maximal subterm depth \(i\). Finally, the desired type \(B\text{Term}\) is the type of all term trees with finite depth.

Since the type \(B\text{Term}\) is defined inductively, we obtain an induction principle and term destructors, shown in Figure 7. Induction has the expected form. The destructors are included for completeness. While somewhat technical, they illustrate typical operations like including a case analysis, collecting the subterms of a term, and computing binding depths. More detail may be found in Nogin et al. [18].

2.7 Exotic terms and decidability

Further discussion is in order here. First, we should note that what we call the “HOAS” representation is in fact a very restricted form of HOAS, where the \(\text{img}\{\cdot\}\) type constructor is used to restrict the function spaces. For instance, consider the standard HOAS representation of a universal quantifier \(\forall x : t_1 t_2[x]\). In traditional HOAS, this might be represented with a term constructor taking two arguments.

\[
\forall : \text{Type} \rightarrow (\text{Term} \rightarrow \text{Prop}) \rightarrow \text{Prop}
\]

There are two potential problems with this representation. First, how should alpha-equality of terms be defined? Strictly speaking, the second argument to a \(\forall\) term is a function. Since function equality is undecidable in general, if alpha-equality is to be computable, this cannot be an arbitrary function as we normally think of it! Second, (again) if the function is unrestricted, the issue of exotic terms
arises, where the function analyzes the structure of its argument inappropriately. The approach taken in LF to the first problem is to syntactically restrict the function space by disallowing term destructors. In LF the \( \lambda \) terms are strongly normalizable and the alpha-equality is therefore decidable. In LF, the exotic terms are simply not expressible.

Our approach is similar, but also somewhat different. Our HOAS allows us to express exotic terms, however since the \( \text{BTerm} \) type is formalized as the image of the de Bruijn representation (and the formalization does not refer to function spaces), the exotic terms are left out and \( \text{BTerm} \) type is restricted to only those HOAS-style expressions that are not exotic.

**Theorem 1.** The type \( \text{BTerm} \) contains only those terms that have a de Bruijn representation.

**Proof.** The induction principle (Figure 7), which was mechanically checked, establishes that the only terms in \( \text{BTerm} \) are \( \forall [n;m] \) and \( \mathbb{B}\{n;o;s\} \).

**Corollary 2.** The type \( \text{BTerm} \) does not contain exotic terms.

For example, consider the following exotic term, for some arbitrary fixed term \( t \).

\[
\lambda b, y. \text{if } x = t \text{ then } x \text{ else } y
\]

This term is in canonical form, but it does not have type \( \text{BTerm} \) because it is not computationally equivalent to any \( \forall [n;m] \) or \( \mathbb{B}\{n;o;s\} \).

Similarly, the mechanically-checked introduction rules for the \( \text{BTerm} \) type [14] establish that all the terms that have a de Bruijn representation belong to this type. For those HOAS-style expressions that are members of the \( \text{BTerm} \) type, the corresponding de Bruijn representation is computable [18] and therefore the alpha-equality is decidable (as it can be reduced to straightforward comparisons of de Bruijn representations).

### 2.8 A small example: computing free variables

To illustrate the utility of the de Bruijn representation, let’s write a simple function to compute the free variables of a term (as might be used in closure conversion for example). That is, suppose we have a term \( t \) with binding depth \( n \). In the HOAS representation, this term is computationally equivalent to a term of the form \( t = \lambda b, x_1, \ldots, x_r, r \), and we wish to determine which of the variables \( x_1, \ldots, x_r \) are free in \( r \).

Now suppose we work directly with the HOAS representation. Calculating the free variables seems simple enough. First, we choose two distinct terms \( r_1 \neq r_2 \). Then, \( x_i \) occurs free in \( r \) iff the substitutions are not equal \( r[r_1/x_i] \neq r[r_2/x_i] \)[2]. This leads to the following definition.

\[
f_{\text{HOAS}}(t) = \{ i | r[\ldots r_1 @ r_2] \neq r[\ldots r_1 @ r_2] \}
\]

However, this definition is somewhat unsatisfying. It involves \( n \) substitutions and \( n \) equality tests. Most of all, it does not conform to the kind of naive model we would expect, where the term is traversed in the usual way. In contrast, the de Bruijn destructor representation allows the more naive algorithm.

In the following program, \( S \) is the set of free variables, \( t \) is the term being analyzed, and \( \text{fold} \) is the list fold function that calls \( f_{\text{DB}} \) for each of the terms in the list \( l \). The key part is the variable \( ^5 \)

\[
f_{\text{DB}}(S, t) = \text{match } r \text{ with } \begin{align*}
| \forall [i;j] & \rightarrow \\
\text{if } i < n & \text{ then } S \cup \{i\} \text{ else } S \\
| \mathbb{B}\{n;o;l\} & \rightarrow \\
f_{\text{DB}} S \{l\}
\end{align*}
\]

We claim that the de Bruijn-style computation \( f_{\text{DB}} \) is clearer that the corresponding HOAS computation \( f_{\text{HOAS}} \). has a more straightforward computational flavor, and is perhaps somewhat easier to reason about. Of course, there is no magic here. The two functions are extensionally equivalent. Furthermore, since the de Bruijn representation is defined in terms of HOAS, it merely codifies a style of computation that is reducible to operations on the HOAS.

### 3. Sequent representation

So far, we have delayed the discussion of the sequent representation. Our goal here is to reduce the sequent representation to the terms that already exist. We do this in two parts. First, we develop a canonical representation of concrete sequents without context variables. For the second part, we define a (formal) function that computes the canonical representation from non-canonical forms. As we will see later, functions of this form can also be used to represent HOAS-style schemas that contain context variables.

The first part is an issue of coding, where the goal is to define a representation that preserves the structure of concrete sequents. We choose the following representation, where \( r : \lambda H x_1:t_1 t_2 \) is a quoted term that represents a hypothesis, its binding, and the rest of the sequent; and \( r \text{ concl} \{t\} \) represents the conclusion of the sequent.

\[
x_1 : t_1 ; \ldots ; x_n : t_n ; \text{ concl} \{t\} \equiv r : \lambda H x_1:t_1 \ldots \lambda H x_n:t_n \text{ concl} \{t\}
\]

where

\[
r : \lambda H x_1 : t_1 t_2 \equiv T(\text{hlambda};[t_1 ; \lambda b, t_2])
\]

\[
r \text{ concl} \{t\} \equiv T(\text{concl};[t])
\]

### 3.1 Sequent context induction

For the second part, we wish to define a function over arbitrary quoted sequents that computes a representation in canonical form. The issue here is that arbitrary sequents, especially those that are used to define judgments in the programming language \( P \), usually include context variables. For example, consider a rule that might be used to define typing of a lambda abstraction.

\[
\frac{X : x : A \vdash \text{z}[x] \in B}{X \vdash \lambda x : A, \text{z}[x] \in A \rightarrow B} \lambda \text{lambda-intro}
\]

\[
\frac{Z \vdash \text{z}[x] \in B}{Z \vdash \lambda x : A \vdash x \in A \rightarrow B} \lambda \text{lambda-intro}^\gamma
\]

Here, \( X \) and \( Z \) are sequent context variables, \( A, B \), and \( \text{z}[x] \) are second-order variables, and \( x \) is a first-order variable. How can we reduce a sequent with a context variable to canonical form?

For this purpose, we use a weak form of sequent context induction called teleportation [13] postulating that contexts stand for finite sequences of hypotheses. While the teleportation mechanism is an essential foundation for this work, the specific details have little effect on the presentation here.

### 3.2 Computation on sequent terms

The sequent induction scheme also introduces a sequent induction combinator for computation over a sequent context, with two
The sequent \text{ind}\{x, y.\text{step}[x; y]; t\} performs computation over a sequent term \(s\). The reduction rules for sequent computation are as follows.

\[
\text{sequent\_ind}\{x, y.\text{step}[x; y]; t\} \rightarrow t
\]

\[
\text{sequent\_ind}\{x, y.\text{step}[x; y]; z. t_1; t_2[z]\} \rightarrow \text{step}\{t_1; \lambda z.\text{sequent\_ind}\{x, y.\text{step}[x; y]; [X[z] \mapsto t_2[z]]\}\}
\]

3. Computing canonical sequent representations

To conclude the sequent representation, we define a function using \text{sequent\_ind} that computes the canonical representation from its non-canonical form. This function, written \(\vdash_B\), is defined as follows.

\[
X \vdash_B t \equiv \text{sequent\_ind}\{x, y.\lambda y.x; z; \vdash x(y, z); \vdash X \vdash \text{conc\_ind}\{t\}\}
\]

Using this definition, the representation of a reflected rule is computed using \(\vdash_B\) in place of \(\vdash\). For example, the reflection form of the lambda-typing rule is as follows.

\[
Z \vdash \Box (X; x.\vdash B \vdash y.z[x] \in B) \Gamma \rightarrow \text{lambda-intro}\n\]

Note that occurrences of \(\vdash_B\) represent actual sequents. Since the reflection translation \(\vdash\) preserves variables, including context variables and bindings, the reflected form of the rule has the same binding structure as the original rule in the programming language \(P\).

Theorem 3. The refinement relation for rules \([16]\) is preserved when reflected. Specifically, let \(R\) be a rule, and \(\sigma\) be a refinement of \(R\) (by second-order and context substitutions), producing \(R'\). The following diagram is commutative for an appropriate refinement \(\sigma'\).

\[
R \quad [\sigma] \quad R'
\]

\[
\vdash \Box (X; x.\vdash B \vdash y.z[x] \in B) \Gamma \rightarrow \text{lambda-intro}\n\]

Proof sketch. Define \(\sigma'\) to be the reflected form of \(\sigma\). Namely, for each substitution in \(\sigma\) define a corresponding substitution in \(\sigma'\), where each term operator is reflected. Because the \(\vdash\) mapping preserves binding and free variable structure, \(\sigma'\) will be a valid refinement function \([16]\). By construction, \(\sigma'([\Gamma]) = [\Gamma']\) is valid.

In fact, the reflected rule is in a form that can be used directly as a proof step in the reflected logic \(\Gamma P\), and there is a one-to-one correspondence from proof steps in \(P\) to proof steps in \(\Gamma P\). Furthermore, as we discussed in the next section, the reflected rules (such as \(\lambda\text{-intro}\)) are automatically derivable in the meta-logic \(\Gamma\).

4. Defining provability

So far, we have postponed the treatment of the provability predicate \(\Box\), which specifies that the quoted formula \(t\) is provable in logic \(P\). In fact, a concept we have glossed over is the truth of reflected rules—if logic \(P\) defines a rule \(R\), the truth of \(\Gamma R\) does not follow axiomatically. It must be proved, and the proof relies on the definition of the provability predicate. To define provability properly, we take the following steps.

- First, for each rule \(R \in P\), we define a proof checking predicate that specifies whether a proof step is a valid application of rule \(R\).
- Next, we define the (legal) derivations to be the proof trees where each proof step in the tree is validated by some rule \(R \in P\).

A formula \(t\) is provable in logic \(P\) if, and only if, there is a derivation with root \(t\).

The usual properties hold: proof checking is decidable, provability is not decidable in general.

4.1 Proof checking

A logic \(P\) is an ordered list of inference rules \(R_1, \ldots, R_n\). A proof is a tree of inferences, and it is legal only if each proof step corresponds to an inference using some rule \(R_i\). A proof step \(r\) is a node in the proof tree that corresponds to a concrete inference \(t_1 \rightarrow \cdots \rightarrow t_{n-1} \rightarrow t_n\). We call the terms \(t_1, \ldots, t_{n-1}\) the premises, and the term \(t_n\) the goal.

In general, a rule \(R\) defines a schema, where each second-order meta-variable stands for a term, and each context meta-variable stands for a context. A concrete proof step is a valid inference of a rule \(R\) if, for each second-order meta-variable in \(R\) there is an actual term, and for each context meta-variable in \(R\) there is an actual context, such that the concrete inference is an instance of the rule.

Let us state this more formally. The \textit{arity} of a meta-variable is the number of its arguments, so a variable \(z_1; \ldots; z_n\) has arity \(n\). Let \(BTerm\{i\}\) be the type of quoted terms of arity \(\text{bdepth}\ i\), corresponding to the space of substitution functions \(BTerm^m \rightarrow BTerm\). Similarly, let \(Context\{i\}\) be the type of contexts of arity \(i\) (the contexts correspond to lists of quoted terms).

Consider a rule \(R\) with free context variables \(\{X_1, \ldots, X_m\}\) and free second-order variables \(\{x_1, \ldots, x_n\}\), where the superscripts \(i_k\) and \(i_j\) indicate the arities of the variables. Then a concrete inference \(r\) is a valid instance of rule \(R\) if the following holds.

\[
\exists X_1^i: Context\{i_1\}, \ldots, X_m^i: Context\{i_m\}, x_1^i: \text{BTerm}\{j_1\}, \ldots, x_n^i: \text{BTerm}\{j_n\}.
\]

\[
r = R \in \text{ProofStep}
\]

That is, the concrete inference \(r\) is equal to an instance of rule \(R\). The type \(\text{ProofStep}\) is the type of proof steps \(\text{BTerm}\{\text{list}\} \times \text{BTerm}\) containing the pairs \(\langle \text{premises}, \text{goal}\rangle\).

For the purposes of proof checking, the existential witnesses are assembled into a proof witness term, and passed as explicit arguments to the checker. Returning to the example of the substitution lemma \((2.11)\), the corresponding proof checker is defined as follows, where \(r\) is the concrete proof step to be checked.

\[
\text{checks}(\text{subst\_lemma}, r, \{X: \text{Context}\{0\}, z_1: \text{BTerm}\{1\}, z_0, z_2, z_3: \text{BTerm}\{0\}\}) =
\]

\[
r = \left(\left(\left(\left(\left(\left(X; x.\vdash B \vdash y.z[x] \in B) \Gamma \rightarrow \text{lambda-intro}\right)\right)\right)\right)\right) \in \text{ProofStep}
\]

(4.11)

The checks predicate takes three arguments \(\text{checks}(R; r; w)\), where \(R\) is a rule, \(r\) is a concrete inference, and \(w\) is the witness for the rule instantiation. Given a logic \(P\) with rules \(R_1, \ldots, R_n\), a proof step is valid \(iff\) it is an instance of one of the rules in the logic.

\[
\text{check}_P[R: \text{ProofStep}; w: \text{Witness}] \equiv \exists R \in \{R_1, \ldots, R_n\}. \text{checks}(R; r; w)
\]

Since proof step equality is decidable, and each logic has a finite number of rules, the \(\text{check}_P[R; r; w]\) predicate is decidable as well.

\[\text{In a setting where context variables are treated as binders, the variable arities are expressions that depend on the lengths } |X_i|\]
4.2 Derivations

Now that we have defined proof step checking, the next part is to define the valid derivations, or proof trees. For this section, we will assume we are referring to a specific logic \( \mathcal{P} \), using the proof-checking predicate \( \text{checkp} \) for logic \( \mathcal{P} \).

The type \( D \) of all derivations is defined inductively in the usual way. A derivation is a finite tree of proof steps; the derivation is valid iff each proof step is valid. The derivations are defined inductively over the length of the derivation. There are no derivations of length 0, and a derivation of length \( i \) has a goal term and a list of derivation premises, in the following form, where \( D_i \) is the type of derivations of length at most \( i \), and each premise \( d_j \) has length at most \( i - 1 \).

\[
D_i \ni d_1^{d_i} \cdots d_{i-1}^{d_i-1}
\]

Given a derivation \( d \), we define \( \text{goal}(d) \) to be the goal term of \( d \). Extending this to lists, we define \( \text{goal}(\{d_1; \cdots ; d_n\}) \) to be the list of goal terms for derivations \( d_1, \ldots, d_n \).

The type \( D_i \) of derivations of length at most \( i \) can be defined formally as follows, where \( \text{void} \) is the empty type. The type \( D \) of all derivations is the union of all derivations of finite length.

\[
\begin{align*}
D_0 & \equiv \text{void} \\
D_{i+1} & \equiv \Sigma \text{premises} : D_i \text{list.} \\
& \quad \Sigma \text{goal term} : \text{BTerm}(0) \\
& \quad \Sigma \text{witness} : \text{Witness} \\
D_i & \equiv \bigcup_{w \in D_i} D_i \\
\end{align*}
\]

This definition also yields an induction principle, which will form the basis for proof induction.

\[
\forall D. (\forall \text{premises} : D \text{ list.} \forall w : \text{Witness.} \text{ checkp}((\text{goal}(\text{premises}), \text{goal term}); w)) \\
\Rightarrow (\forall p \in \text{premises.} \text{ P}(p)) \\
\Rightarrow \text{ P}(\text{premises, w})) \\
\Rightarrow (\forall d : D. \text{ P}(d))
\]

At this point, the definition of the provability predicate \( \square t \) is straightforward. A quoted term \( t \) is provable iff there is a derivation where \( t \) is the goal term.

\[
\square t \equiv \exists d : D. \text{ goal}(d) = t \in \text{BTerm}(0)
\]

4.3 Proof reflection and automation

One important consequence of structure-preservation is that proofs can be reflected as well. Consider a proof in the original logic \( \mathcal{P} \) of some theorem \( t_1 \rightarrow \cdots \rightarrow t_n \). In a foundational prover, the proof is expressed as a tree of inferences that can be linearized to a finite sequence of rule applications \( R_1, R_2, \ldots, R_n \).

Since the structure of each inference is preserved, there is a corresponding proof in the reflected logic \( \overset{\sim}{\! \mathcal{P}} \) of the reflected theorem \( (Z \vdash \square t_1) \rightarrow \cdots \rightarrow (Z \vdash \square t_n) \). In fact, the proof is a one-to-one map of the original theorem, using reflected justifications in place of the original. That is, the reflected proof is \( R_1, R_2, \ldots, R_n \).

While this might seem quite straightforward, the important property here is that the prover internals do not need to be reflected. It is not necessary to formalize the inference mechanics of the theorem prover, because the original mechanism works without change in the reflected theory.

Proof automation is similar. Again, in a foundational prover, each run of a heuristic or decision procedure is justified by a sequence of inferences \( R_1, R_2, \ldots \). The existing automation may be used for reasoning in the reflected logic, provided that rule selection for reflected proofs uses the reflected rules rather than the original ones.

5. Reflection and induction

So far, we have presented a structure-preserving representation function, a mechanism for formalizing reflected logics, and a procedure for deriving reflected provability rules. This system is already powerful enough to express and prove meta-properties over reflected systems. However, it remains impractical. There is a piece missing, namely induction on the provability predicate \( \square t \).

What exactly is the induction principle for provability? Suppose we wish to prove a theorem of the form \( \square x \Rightarrow P[x] \), where \( x \) is a variable, and \( P \) is a predicate on quoted terms. Since \( x \) is provable, that means that there is a derivation with root \( x \), and we can apply induction on the length of the derivation.

Now, for illustration, assume that the logic \( \mathcal{P} \) contains three rules, \( \mathcal{P} = R_1 = t_{11}, R_2 = t_{21} \rightarrow t_{22}, R_3 = t_{31} \rightarrow t_{32} \rightarrow t_{33} \). Then the induction form has the following shape.

\[
\begin{align*}
\Gamma ; \square t_{11} & \Rightarrow P[t_{11}] \\
\Gamma ; \square t_{21} ; \square t_{22} ; P[t_{21}] & \Rightarrow P[t_{22}] \\
\Gamma ; \square t_{31} ; \square t_{32} ; \square t_{33} ; P[t_{31}] ; P[t_{32}] & \Rightarrow P[t_{33}] \\
\Gamma & \Rightarrow P[x]
\end{align*}
\]

However, this rule is not quite right. The issue is that the terms \( t_{ij} \) will in general contain meta-variables, and the meta-variables must be separately universally quantified for each induction case.

Explicit quantification of meta-variables is not expressible in our meta-logic \( \text{MetaPRL} \). However, here it is acceptable to use object-quantifiers provided by the meta-theory \( \text{MCTT} \). There is no appreciable effect on proof automation as long as the first-order form is compatible with the automatically-generated reflected rules. The correct form of the rule explicitly quantifies over the meta-variables, re-using the mechanism for generating the proof-checking rules. For the current example, we introduce explicit quantifiers. In this case we write \( t_{ij}[x] \) to represent a term that may contain any of the variable \( X \) but is otherwise free of context variables.

\[
\begin{align*}
\Gamma ; X : \text{Context.} & \Rightarrow \square t_{11}[x] \Rightarrow P[t_{11}[x]] \\
\Gamma ; X : \text{Context.} & \Rightarrow \square t_{21}[x] ; \square t_{22}[x] ; P[t_{21}[x]] \Rightarrow P[t_{22}[x]] \\
\Gamma ; X : \text{Context.} & \Rightarrow \square t_{31}[x] ; \square t_{32}[x] ; \square t_{33}[x] ; P[t_{31}[x]] ; P[t_{32}[x]] \Rightarrow P[t_{33}[x]] \\
\Gamma ; & \Rightarrow P[x]
\end{align*}
\]

In our implementation, we generate a variant of this rule that allows for induction over terms, not just variables. This is done by introducing a "shared" term \( u \) that establishes a connection between the provable term \( t \) and the predicate \( P \). The actual theorem has the form \( \Gamma ; u : \square t_{ij}[u] \Rightarrow P[t_{ij}[u]] \), where \( u \) is the shared part. The new form is derivable from the previous case for provability on variables, and we omit it here. In fact, the size of the rule is one of the main drawbacks. In practice, even for fairly small logics \( \mathcal{P} \), the statement of the elimination rule is already several pages long, and it is difficult to use the rule interactively. We are expecting to address this in future work.

6. Preliminary discussion

At this point, all the pieces are in place. We have defined 1) the type \( \text{BTerm} \) of reflected terms, 2) the reflection function \( \overset{\sim}{\! \mathcal{P}} \), 3) the provability predicate \( \square \) (for \( r \in \text{BTerm} \)), and 4) we have derived a proof induction principle. The following are the key steps when using this methodology.

---

4 It isn’t clear to us whether a similar mechanism might work for non-foundational provers (those with “trusted” decision procedures).
To reason about a specific programming language $P$:
(A) Formalize the syntax and axioms/typing rules $R_1, \ldots, R_n$ as a primitive logic $\mathcal{P}$ in the framework $\mathcal{F}$.
(B) State any relevant theorems $R'_1, \ldots, R'_m$ of interest in $\mathcal{P}$. Prove those that do not require meta-theoretical reasoning.
(C) Instruct the framework to reflect the logic $\mathcal{P}$. The framework will create a subtheory $\mathcal{P}'$ of the meta-logic $\mathcal{M}$ having the following parts,
- a definition of $\Box_{\mathcal{P}}$,
- inference rules $\Gamma_1, \ldots, \Gamma_n$,
- theorem statements $\Gamma'_1, \ldots, \Gamma'_m$,
- a proof induction principle for $\Box_{\mathcal{P}} \mathcal{P} r$, where $r$ is a reflected formula in $\mathcal{P}'$.
The following are proved automatically by the framework:
- Each inference rule $\Gamma_i$ is derived as a theorem of $\mathcal{M}$.
- The induction principle is derived as a theorem of $\mathcal{M}$.
- Any proofs of the theorems $R'_1, \ldots, R'_m$ are reflected to form the proofs of $\Gamma'_1, \ldots, \Gamma'_m$.
- At no point are new axioms added to $\mathcal{F}$ and $\mathcal{M}$: the subtheory $\mathcal{P}'$ is fully derived.

As we have mentioned, the structure of terms and rules is preserved by the transformation $\mathcal{P}'$. The practical consequence is that reasoning in the original logic $\mathcal{P}$ maps directly to reasoning in the reflected logic $\mathcal{P}'$; any approach to constructing a proof of the theorem $R'_i$ in $\mathcal{P}$ equally applies to proving $\Gamma'_i$ in $\mathcal{P}'$.

If a theorem $R'_i$ is not proved in $\mathcal{P}$, the framework will state the corresponding $\Gamma'_i$ as a theorem to be proved in the reflected logic $\mathcal{P}'$ by the user. The reflected logic $\mathcal{P}'$ affords an induction principle, and in addition permits the use of quantifiers from $\mathcal{M}$ that are not available in $\mathcal{P}$. These two properties together enable the statement (and proof) of meta-theoretical principles that could not be proved in $\mathcal{P}$ itself. Such meta-theoretical principles can then be used in the proof of $\Gamma'_i$.

It should be noted that this is only a fragment of reflection—we do not add a reflection rule $\Box_{\mathcal{P}} \mathcal{P} t \Rightarrow t$. Any meta-theorem that is proved in $\mathcal{P}'$ remains a fact solely of $\mathcal{P}'$; the original theory $\mathcal{P}$ remains open and unconstrained. While a reflection rule could be added as a new axiom to $\mathcal{F}$, we believe it is unnecessary and that the consequent strengthening of the framework logic $\mathcal{F}$ is undesirable and dangerous.

7. An example: $F_{c\downarrow}$
To illustrate reflection in practice, we have formalized an initial part of the programming language/system $F_{c\downarrow}$, as defined by the POPLmark challenge. This work (as well as the reflection mechanism that has been discussed), is implemented in the MetaPRL logical framework. In this system, the framework logic $\mathcal{F}$ is the logic of sequent-schema [16], which is essentially a logic of second-order Horn formulas; the meta-logic $\mathcal{M}$ is Computational Type Theory [14], a variant of Martin-Lof type theory.

Following the methodology described in this section, the first step is to define $\mathcal{P}_{F_{c\downarrow}}$ as a primitive logic in $\mathcal{F}$. This immediately brings us to the issue of syntax and representation. In most textbook accounts, the syntax of a logic receives little discussion; the syntax is frequently described with a context-free grammar, and then dismissed so that the discussion can proceed to more interesting issues. In a formal account, the process cannot be dismissed, and in fact is quite important, as it provides the basis for structural induction.

7.1 Defining the syntax of $F_{c\downarrow}$
The syntax of $F_{c\downarrow}$ is shown in Figure 8 in the usual textbook form as well as in the concrete syntax for defining the terms of a logic in MetaPRL. The textbook form is standard. The MetaPRL form introduces each term as a declaration. The “typeclass” declarations, such as declare typeclass $\texttt{Ty}$, define classes of syntax (this should not be confused with the word typeclass as it is used in Haskell or other languages, the meaning here is simply that the term $\texttt{Ty}$ denotes a syntactic collection of terms). The term declarations are in a
Syntax judgments
\[ J_M \overset{\text{def}}{=} \Gamma \vdash_M t \in_M \mathcal{G} \] syntax judgment
\[ \Gamma_M \overset{\text{def}}{=} \chi_1 : \mathcal{G}_1 \cdots \chi_n : \mathcal{G}_n \] syntax contexts

Syntax declaration
\[ \text{TyAll}\{t_1 : \text{Ty}; X : \text{Ty}; t_2[2X] : \text{Ty}\} : \text{Ty} \]

\[ \text{lambda}\{t : \text{Ty}; x : \text{Exp}; e[x] : \text{Exp}\} : \text{Exp} \]

sequent \( \{\text{Exp} : \text{Ty} | \text{Ty} : \text{TyBound} \vdash \text{Prop}\} : \text{Judgment} \)

\[ \Gamma_M \vdash_M t \in_M \text{Ty} \]
\[ \Gamma_M \vdash_M (x : t; Y[x] = P[x]) \in_M \text{Judgment} \]

Rules
\[ \Gamma_M ; x : \mathcal{G} ; \Gamma_M [x] \vdash_M x \in_M \mathcal{G} \]

Syntax judgment
\[ \Gamma_M \vdash_M t_1 \in_M \text{Ty} \quad \Gamma_M ; X : \text{Ty} \vdash_M t_2[2X] \in_M \text{Ty} \]
\[ \Gamma_M \vdash_M \text{TyAll}\{t_1 : \text{Ty}; X : \text{Ty}; t_2[2X] : \text{Ty}\} \in_M \text{Ty} \]

\[ \Gamma_M \vdash_M t \in_M \text{Ty} \quad \Gamma_M ; X : \text{Exp} \vdash_M e[x] \in_M \text{Exp} \]
\[ \Gamma_M \vdash_M \text{lambda}\{t : \text{Ty}; x : \text{Exp}; e[x] : \text{Exp}\} \in_M \text{Exp} \]

\[ \Gamma_M \vdash_M P \in_M \text{Judgment} \]
\[ \Gamma_M \vdash_M (\neg P) \in_M \text{Judgment} \]

\[ \Gamma_M \vdash_M t \in_M \text{TyBound} \]
\[ \Gamma_M ; X : \text{Ty} \vdash_M (Y[x] = P[x]) \in_M \text{Judgment} \]
\[ \Gamma_M \vdash_M (x : t; Y[x] = P[x]) \in_M \text{Judgment} \]

Figure 9. A fragment of the syntax judgments

restricted form of HOAS. For example, the declaration \( \text{TyFun}\{t_1 : \text{Ty}; t_2 : \text{Ty}\} : \text{Ty} \) specifies that the term \( \text{TyFun} \) represents a type (\( \text{Ty} \)) with two subterms, both of which must be types. The declaration \( \text{TyAll}\{t_1 : \text{Ty}; X : \text{Ty}; t_2[2X] : \text{Ty}\} : \text{Ty} \) is similar, but it introduces a binding \( X \), and the subterm \( t_2[2X] \) must be a type if \( X \) is a type.

Sequent declarations introduce a new issue. In \( \text{F}_\mathcal{C} \), sequent hypotheses have two forms, one that introduces an expression binder, and another that introduces a type binder. For example, in the \( \text{F}_\mathcal{C} \) fragment, \( \text{X} < : \text{Top} : x \in X \), the variable \( X \) has type \( \text{Ty} \), and \( x \) has type \( \text{Exp} \). The corresponding \( \text{MetaPRL} \) framework logic \( \mathcal{F} \) includes only a single binding form, so we introduce a new term \( \text{BoundedBy}\{t : \text{Ty}\} : \text{TyBound} \). The \( \text{F}_\mathcal{C} \) hypothesis \( X < : \text{Top} \) takes the form \( \text{X} : \text{BoundedBy}\{\text{Top}\} \).

7.2 Reflected syntax

When reflected, a formula \( t \in F_{\mathcal{F}_\mathcal{C}} \) becomes a formula \( \Gamma \vdash t \in \text{BTerm} \). However, we should be careful here, because the transformation \( \Gamma \vdash t \in \text{BTerm} \) is defined over many more terms than those in \( F_{\mathcal{F}_\mathcal{C}} \). For example, we have \( \Gamma \vdash \text{lambda}\{\text{Top} ; x : \text{Top}\} \in \text{BTerm} \), even though \( \text{lambda}\{\text{Top} ; x : \text{Top}\} \) is not a well-formed formula in \( \mathcal{F}_\mathcal{C} \). The type \( \text{BTerm} \) includes all the well-formed formulas; what is needed is to define subtype of \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \) where \( \mathcal{G} \) is a syntactic class (for example \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \)).

However, naive definitions of \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \) quickly run aground. For example, suppose we wish to define the type \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \) using a type-theoretic form of set comprehension

\[ \text{BTerm}_{\mathcal{F}_\mathcal{C}} \overset{\text{def}}{=} \{ t : \text{BTerm} | \text{isty}(t) \} \]

where \( \text{isty} : \text{BTerm} \rightarrow \mathbb{B} \) is a predicate and \( \text{isty}(t) \) is true if \( t \) is a term that represents a type in \( \mathcal{F}_\mathcal{C} \). The pseudo-code is as follows.

\[
\text{isty}(t) =
\begin{cases}
\text{match } t \text{ with } \\
| \text{Top} \rightarrow \text{true}
| \text{TyFun}\{t_1 ; t_2\} \rightarrow \text{isty}(t_1) \wedge \text{isty}(t_2)
| \text{TyAll}\{t_1 ; t_2[x]\} \rightarrow \\
\text{isty}(t_1) \wedge x : \text{BTerm}_{\mathcal{F}_\mathcal{C}} ; \text{isty}(t_2[x])
\end{cases}
\]

However, the final clause for \( \text{TyAll} \) includes a quantification over \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \), so the type definition must at least be recursive. Furthermore, the occurrence is negative; this approach is unlikely to work. In fact, this kind of negativity is just an instance of the general problem of naming in formal meta-theory.

However, there is a simple and easy way out, which is to define the syntax as a logic. That is, each term declaration is viewed as a syntactical judgment. We introduce a syntactical judgment \( \vdash_M \) (we call it a “meta-type” judgment) that defines syntactic well-formedness. Some examples are shown in Figure 9. The syntax rules are expressed in pre-reflected form. For the most part, these rules are entirely straightforward. Each declaration defines a corresponding type-checking rule in the logic \( \vdash_M \). Sequent declarations result in two-or-more type-checking rules, where there is one rule for each of the kinds of hypotheses in the sequent.

If we wish, we can now give a formal definition to the type \( \text{BTerm}_{\mathcal{F}_\mathcal{C}} \) as follows.

\[ \text{BTerm}_{\mathcal{F}_\mathcal{C}} \overset{\text{def}}{=} \{ t : \text{BTerm} | (\Gamma \vdash_M t \in_M \text{Ty}) \} \]

Note however that this type includes only the closed terms \( t \). For this reason, in our implementation, we use the predicate \( (\Gamma \vdash_M t \in_M \text{Ty}) \) directly.

7.3 The \( \mathcal{F}_\mathcal{C} \) logic proper

We now proceed to define the logical (as opposed to syntax) rules of the \( \mathcal{F}_\mathcal{C} \) type system. The rules themselves are standard, we show a small fragment in Figure 10. Here, the letters \( e, s, S \), and \( T \) stand for second-order meta-variables. As before, hypothesis \( x : \text{BoundedBy}(t) \) represents the form \( x < t \).

The remainder of the logic now proceeds much as it did for the syntax. When reflected, the rules define a logic of provability in \( \mathcal{F}_\mathcal{P} \). The reflected sublogic \( \mathcal{F}_\mathcal{P} \) enables reasoning both by structural induction (based on the syntax rules) and proof induction (based on the logical rules), and the user can draw on either principle as need be.

One point to note is that some meta-properties are expressible in the original logic, and it is usually desirable to do so in these cases. For example, the property of reflexivity of subtyping is expressible (though not provable) in the original logic, as a theorem of the following form.

\[ \Gamma \vdash t < t \]

When reflected in the context of the \( \mathcal{F}_\mathcal{C} \) syntax, we require that the theorem be a valid judgment of the logic \( \mathcal{F}_\mathcal{K} \), and the theorem takes the form \( \Gamma \vdash_M (\Gamma \vdash t < t) \in_M (\Gamma \vdash \text{Judgment}) \Rightarrow (\Gamma \vdash (\Gamma \vdash t < t) \in_M \text{Judgment}) \) where quotations have been added in the appropriate places by the reflection mechanism.
7.4 Structural induction

Since the syntax is expressed as a type-checking logic, structural induction reduces to proof induction. That is, a type-checking judgment has the form \( \Box (\Gamma \vdash t : \tau) \), for some term \( t \) and syntax class \( \tau \). Induction on the provability yields the possible cases for the term \( t \).

To illustrate, suppose we wish to prove reflexivity of subtyping. The proof proceeds as follows. For clarity, we have omitted occurrences of the quotation symbol \( \Box \), it should be understood that every non-variable term is quoted.

- \( \Box (\Gamma \vdash x : x) \subseteq M \) (Judgment) \( \Rightarrow \) \( \Box (\Gamma \vdash x : x) \)

This is the goal to be proved. We forward-chain, using the assumption as follows.

- \( \Box (\Gamma \vdash x : x) \subseteq M \) (Judgment)

This is the assumption. Proof induction on the Judgment judgment leads to the following fact, where the \( [\Gamma] \) denotes the reduction of the context \( \Gamma \) to a syntactic context with hypotheses of the form \( x : \tau \).

- \( \Box ([\Gamma] \vdash x : x) \subseteq M \) (Prop)

From the declaration for subtype, we can infer from \( (x : x) \) that \( x \) is a type by proof induction.

- \( \Box ([\Gamma] \vdash x : x) \subseteq M \) (Ty)

We now perform structural induction on \( \Box ([\Gamma] \vdash x : x) \subseteq M \) (Ty), to obtain the following subgoals. By assumption \( ([\Gamma] \vdash x : x) \subseteq M \) (Ty) has a proof, and that proof must end with one of the Ty rules, so \( t \) is either a variable or one of \( \text{top}, \text{tyFun}, \text{or tyAll} \).

- \( \Box (\Gamma \vdash x : T, \Delta x \vdash t_1 : x) \)

- \( \Box (\Gamma \vdash \text{top} : \text{top}) \)

- \( \Box (\Box ([\Gamma] \vdash t_1 : t_1) \Rightarrow (\Box ([\Gamma] \vdash t_2 : t_2) \Rightarrow (\Box ([\Gamma] \vdash (t_1 \rightarrow t_2) : (t_1 \rightarrow t_2))) \)

Each of these subgoals is now provable from the rules in \( F_{\text{K}} \).

In other cases, the meta-theorems are not expressible in the original logic. For example, the proof of transitivity of subtyping requires the proof of two properties by simultaneous induction.

(A) \( \Gamma \vdash t_1 : t_2 \) and \( \Gamma \vdash t_2 : t_3 \), then \( \Gamma \vdash t_1 : t_3 \).

(B) \( \Gamma, x : t_2, \Delta x \vdash t_3 : x \) and \( \Gamma \vdash t_6 : t_2 \), then \( \Gamma, x : t_6, \Delta x \vdash t_4 : x \).

Since the theorem is the simultaneous proof of two rules, it cannot be expressed in \( F_{\text{K}} \). In the reflected system it simply becomes a conjunction.

\[
\begin{align*}
\Box (\Box \Gamma \vdash t_1 : t_2) ) \Rightarrow \\
\Box (\Box \Gamma \vdash t_2 : t_3) ) \Rightarrow \\
\Box (\Box \Gamma \vdash t_1 : t_3) \\
\Box (\Box \Gamma, x : t_2, \Delta x \vdash t_3 : x ) \Rightarrow \\
\Box (\Box \Gamma \vdash t_6 : t_2 ) \Rightarrow \\
\Box (\Box \Gamma, x : t_6, \Delta x \vdash t_4 : x ) \Rightarrow \\
\end{align*}
\]

The proof proceeds in the usual way, by structural induction on \( t_2 [4] \) Appendix A).

8. Related work

This work build upon a very large number of related efforts. In fact, the number of such efforts is so big that we are unable to give an adequate overview in this limited space. Harrison [10] has written an excellent survey and critique of a broad range of approaches to reflection. We give another broad survey in a previous paper [18].

There has been much interest in the POPMARK challenge [4]. Challenge submissions take a wide variety of approaches, including representations with de Bruijn indices (Vouillon, Sallieiros), nominal approaches (Fairbairns), nameless approaches (Leroy), and HOAS (Ashley-Rollman, Crary, and Harper).

In 1931 G"odel used reflection to prove his famous incompleteness theorem [8]. A modern version of the G"odel’s approach was used by Aitken et al. [3] to implement reflection in the NuPRL theorem prover. A large part of this effort was essentially a reimplementation of the core of the NuPRL prover inside NuPRL’s logical theory.

A number of approaches to logical reflection were explored in the Coq proof assistant. Rieu [23] has implemented a computation reflection mechanism. Hendriks [11] formalized natural deduction for first-order logic in the proof assistant Coq using de Bruijn indices for variable binding. O’Connor [19] constructively proved the G"odel–Rosser incompleteness theorem using the natural numbers to encode formulas and proofs.

9. Discussion and future work

The goal of this work is to develop a formal framework for programming language meta-theory. We claim that, as a framework, there should be a general, uniform way to define, view, and manipulate programming languages, and the tool of choice is reflection. While reflection can be defined over many different representations, we believe that a practical approach requires re-using existing automated methods, and doing so requires that the structure of a theory be preserved, including variables, meta-variables, and bindings. We presented a structure-preserving representation, building on previous work with the representation of syntax [18] and logical terms [13]. This led to a formalization of proofs, proof-checkers, and derivations, together with automated generation of reflected rules and induction forms in the reflected theory.

In some ways, the result seems simple. When a logic is reflected, its presentation changes only slightly, and the existing reasoning methods and proof procedures continue to work. The difference is, of course, that reasoning about meta-properties of the logic becomes possible.
It was important to us that the development of the theory of reflection be accompanied by its implementation. This makes it more useful of course, but an additional reason is that the theory of reflection is rife with paradoxes, and it is easy to fall into false thinking. While we have tried to simplify the account in this paper, the actual formalization was demanding.

Part of the power of our approach comes from the availability of first-order forms (including the de Bruijn representation) that are used to build the formal foundations. The goal, however, is to provide users with a convenient HOAS-based high-level interface.

Currently, we are using reflection to develop an account of the $\forall$; type theory as defined by the POPLmark challenge. Using our environment for the task of formalization is straightforward; the routine proofs are fully automated. In addition, for simple proofs, the first-order forms are entirely hidden. One challenge, however, is to simplify the use of proof induction, which, as discussed in Section 4, relies on the first-order representation, including the meta-logic quantifiers.

We believe that our results may be generalized to other provers and frameworks. The non-standard properties of the logical framework that we rely upon are the following. 1) Programs may be expressed without first giving them a type; in addition, programs may have more than one type. 2) Computation defines a congruence; any two programs that are computationally (beta) equivalent can be interchanged in any formal context. 3) For reasoning about sequents, a weak induction principle [13] is needed on sequent context variables [16]. 4) A function image type [17].

References


