Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 79–94
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Practical Reflection for Sequent Logics

Jason Hickey, Aleksey Nogin, Xin Yu and Alexei Kopylov

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

Abstract

It is well-known that adding reflective reasoning can tremendously increase the power of a proof assistant. In order for this theoretical increase of power to become accessible to users in practice, the proof assistant needs to provide a great deal of infrastructure to support reflective reasoning. In this paper we explore the problem of creating a practical implementation of such a support layer.

Our implementation takes a specification of a logical theory (which is identical to how it would be specified if we were simply going to reason within this logical theory, instead of reflecting it) and automatically generates the necessary definitions, lemmas, and proofs that are needed to enable the reflected meta-reasoning in the provided theory.

One of the key features of our approach is that the *structure* of a logic is preserved when it is reflected. In particular, all variables, including meta-variables, are preserved in the reflected representation. This also allows the preservation of proof automation—there is a structure-preserving one-to-one map from proof steps in the original logic to proof step in the reflected logic.

To enable reasoning about terms with sequent context variables, we develop a principle for context induction, called *teleportation*.

This work is fully implemented in the MetaPRL theorem prover.

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

1 Introduction

By reflection, we mean the ability to use one logic to reason about another, or the ability to use a logic to reason about itself. At its core, a reflection system has two parts. There is a *representation* function, written $\ulcorner t \urcorner$, that defines the representation or "quotation" of a logical formula t. Then, there is a *provability* operator, written $\Box\, q$, which is a predicate specifying that q is a quotation of a provable formula.

An implementation of a reflection system needs to have two corresponding parts: a specific representation function, and a *mechanized* reflective reasoning (including a definition of $\Box\cdot$ and some degree of reasoning automation)?

The issue of representation is central, and far from trivial. For example, while

1571-0661
doi:10.1016/j.entcs.2007.01.019

it is conceptually easy to define a representation function using a Gödel numbering [10], such schemes are impractical as the *structure* of a reflected term (a number) is so different from the original formula. Any plan to re-use mechanized reasoning methods on reflected terms would be extremely difficult.

The challenge is an instance of a general canonical problem—that of using mechanized reasoning to reason about meta-properties of systems, languages, or logics. Our goal is to develop a canonical solution that can be used for meta-reasoning *in general*. In our approach, we use reflection to implement a framework where meta-reasoning is higher-order. For example, one can develop theorems of the form, "Any system that has meta-property P also has meta-property Q," or "Every meta-property of system A is also a meta-property of system B."

However, mechanized reflection is not easy. The general issue is that, if one wants to talk about provability, then it seems necessary to formalize or emulate the theorem prover and its meta-logic. This naïve approach is not only difficult, but it would also require reimplementing the theorem prover within itself. Following Barzilay [4], we aim at reusing the theorem prover instead of reimplementing it.

We present an approach to practical reflection as part of a logical framework, where the representation function $\ulcorner\cdot\urcorner$ is defined over a logic, as well as the formulas, inferences, and theorems that it contains. That is, to develop an account of system $\mathcal{L}$ and its meta-properties, one first defines the system $\mathcal{L}$ as a primitive logic, using the exact same syntax and definition mechanism that are used in not-reflective case. Then, to develop an account of the meta-properties of $\mathcal{L}$, the logic is (automatically) reflected *en masse* to $\ulcorner\mathcal{L}\urcorner$, where each theorem $\mathcal{T}$ in $\mathcal{L}$ is reflected as $\Box_{\mathcal{L}}\ulcorner\mathcal{T}\urcorner$ in $\ulcorner\mathcal{L}\urcorner$, and any proof of $\mathcal{T}$ is reflected to form a proof of $\Box_{\mathcal{L}}\ulcorner\mathcal{T}\urcorner$. In our system, it is not necessary to prepare for reflection. One may develop a theory in the usual way, calling upon reflection if/when it is necessary to perform meta-reasoning.

Of course, this would still not be practical if reasoning in the reflected logic is difficult. The fundamental reason that our approach is practical is that the representation function preserves structure *exactly* in this sense: all variables, including both object and meta-variables, are preserved by the representation. One might call this meta-higher-order abstract syntax. In particular, since we are working with logics that use sequents to express their judgments, the representation function preserves sequent context variables. To do so, we develop a weak induction principle for sequent contexts, called *teleportation*.

The benefit of preserving the term structure is that mechanized reasoning works transparently. That is, there is a one-to-one correspondence from proof steps in the original logic $\mathcal{L}$ to proof steps in $\ulcorner\mathcal{L}\urcorner$. In fact the translation is direct and mechanical, which means that proof automation in the original logic $\mathcal{L}$ also applies in the reflected logic $\ulcorner\mathcal{L}\urcorner$.

This work is implemented in the MetaPRL logical framework [14, 17], and is available at `http://www.metaprl.org/`. The following is a summary of the contributions.

- A representation function $\ulcorner e\urcorner$ that preserves the structure of formula e, specifically preserving object and meta-variables, and all binding structure.

$$
\begin{array}{rll}
t & ::= & x & \text{object (first-order) variables} \\
 & | & z[t_1; \cdots ; t_n] & \text{second-order meta-variables} \\
 & | & \Gamma \vdash t & \text{sequents} \\
 & | & op\{b_1; \cdots ; b_n\} & \text{concrete terms} \\
b & ::= & x_1, \ldots, x_n.t & \text{bound terms} \\
\Gamma & ::= & h_1; \cdots ; h_n & \text{sequent contexts} \\
h & ::= & X[t_1; \cdots ; t_n] & \text{context meta-variables}\ [1] \\
 & | & x : t & \text{hypothesis bindings and terms} \\
\mathcal{L} & ::= & R_1; R_2; \cdots ; R_n & \text{a logic} \\
R & ::= & t_1 \longrightarrow \cdots \longrightarrow t_n & \text{an inference rule } (t_i \text{ are closed w.r.t. object variables})
\end{array}
$$

Fig. 1. Syntax of formulas and logics

- A one-to-one map from proofs in $\mathcal{L}$ to proofs in the reflected logic $\ulcorner \mathcal{L} \urcorner$.
- A new induction principle, called teleportation, for induction on sequent contexts.
- A practical implementation in the MetaPRL system.

The organization of the paper is as follows. In Section 2 we develop the syntax and language of logics. This then allows the formal definition of the representation function in Section 3, as well as the definition of provability $\Box t$ in Section 4. In order to work with sequent context variables, we develop the teleportation induction principle in Section 5. The final step in Section 6 is to develop methods for proof induction in reflected logics. We present related work in Section 7, and we conclude with a discussion of our approach to reflection in Section 8.

2 Terminology

We assume we are working in a meta-language with sequents, second-order meta-variables, and terms, as shown in Fig. 1. A term t is a formula containing variables, concrete terms, or sequents. A concrete term $op\{b_1; \cdots ; b_n\}$ has a name op, and some subterms $b_1, \ldots, b_n$ that have possible binding occurrences of variables. For example, a term for representing the sum $i + j$ might be defined as $\texttt{add}\{.i; .j\}$ (normally we will omit the leading . if there are no binders, writing it as $\texttt{add}\{i; j\}$). A lambda-abstraction $\lambda x.t$ would include a binding occurrence $\texttt{lambda}\{x.t\}$. Note that here the primitive binding construct is the bound term b, and λ-binders are a defined term. An alternate choice would be to use a single primitive λ binder (for example, as is done in LF [11]).

A sequent $\Gamma \vdash t$ includes a sequent context Γ, 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). Note that sequents can be arbitrarily nested inside other terms and are not necessarily associated with judgments.

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

[1] Strictly speaking, context variables are *bindings* and meta-variables have context arguments in addition to term argument. This does not affect the presentation until we get to context induction (Section 5, and we omit context arguments for now.

substitution functions, and are implicitly universally quantified for each rule in which they appear [19]. For example, a second-order variable $z[]$ represents all closed terms (we will normally omit empty bracket, writing simply z). 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.

$$\frac{\Gamma, x\colon t_3, \Delta \vdash t_1 \in t_2 \quad \Gamma, \Delta \vdash s \in t_3}{\Gamma, \Delta \vdash t_1[x \leftarrow s] \in t_2}$$

In our more concrete notation, s, t_1, t_2, t_3 are all represented with second-order variables, and Γ, Δ with context variables. Substitutions are defined using the term arguments; rules are defined using the meta-implication $\cdot \longrightarrow \cdot$, 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 $\texttt{member}\{s;t\}$, and z_i are second-order meta-variables).

$$\begin{array}{l}(X; x\colon z_3; Y \vdash z_1[x] \in z_2) \longrightarrow \\ (X; Y \vdash z_0 \in z_3) \longrightarrow \\ (X; Y \vdash z_1[z_0] \in z_2)\end{array} \tag{2.I}$$

In the final sequent, the term $z_1[z_0]$ specifies substitution of z_0 for x in z_1.

Note how the term arguments are used to specify binding precisely—the variable x is allowed to occur free in z_1, but in no other term. The reason we adopt this second-order notation is for this precision. All rule schemas representable with substitution notation are also representable as second-order schemas, but not vice-versa.

For the final part, a logic $\mathcal{L}$ is an ordered sequence of rules. Each rule may be an axiom, or it may be derived from the previous rules in the logic.

3 Representation of reflected terms

We will assume that we are working in the context of a logical framework, so there are at least three logics in consideration—$\mathcal{L}$: the object logic, $\mathbb{M}$: the meta-logic in which reasoning about the object logic 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 reflection process is to define a *representation* of formulas, judgments, rules and theorems of $\mathcal{L}$ in terms of formulas, propositions, and sentences in $\mathbb{M}$.

The representation function $\ulcorner\cdot\urcorner$ produces a quoted form of its argument. As we have mentioned previously, to preserve a one-to-one correspondence between proofs in an original logic $\mathcal{L}$ and its reflected logic $\ulcorner\mathcal{L}\urcorner$, it is important that $\ulcorner\cdot\urcorner$ preserve the structure of the term, including variables, meta-variables, and binding structure. Note that the representation function itself is not a part of the language of the logical framework; it is only a symbol of the "on-paper meta-meta-language"

Terms

$\ulcorner t \urcorner$:	$\ulcorner x \urcorner$	$\equiv x$
	$\ulcorner z[t_1; \cdots ; t_n] \urcorner$	$\equiv z[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$
	$\ulcorner \Gamma \vdash t \urcorner$	$\equiv \ulcorner \Gamma \urcorner \ulcorner \vdash \urcorner \ulcorner t \urcorner$
	$\ulcorner op\{b_1; \cdots ; b_n\} \urcorner$	$\equiv \ulcorner op \urcorner \{\ulcorner b_1 \urcorner; \cdots ; \ulcorner b_n \urcorner\}$
$\ulcorner b \urcorner$:	$\ulcorner x_1, \ldots, x_n.t \urcorner$	$\equiv \lambda_b x_1 \ldots . \lambda_b x_n . \ulcorner t \urcorner$

Sequent contexts

$\ulcorner \Gamma \urcorner$:	$\ulcorner h_1; \cdots ; h_n \urcorner$	$\equiv \ulcorner h_1 \urcorner; \cdots ; \ulcorner h_n \urcorner$
$\ulcorner h \urcorner$	$\ulcorner x : t \urcorner$	$\equiv x : \ulcorner t \urcorner$
	$\ulcorner X[t_1; \cdots ; t_n] \urcorner$	$\equiv X[\ulcorner t_1 \urcorner; \cdots ; \ulcorner t_n \urcorner]$

Rules and logics

$\ulcorner \mathcal{L} \urcorner$:	$\ulcorner R_1; \cdots ; R_n \urcorner$	$\equiv \ulcorner R_1 \urcorner; \cdots ; \ulcorner R_n \urcorner$
$\ulcorner R \urcorner$:	$\ulcorner t_1 \longrightarrow \cdots \longrightarrow t_n \urcorner$	$\equiv (Z \vdash \Box_{\mathcal{L}} \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box_{\mathcal{L}} \ulcorner t_n \urcorner)$

Fig. 2. The definition of the representation function

that we use for describing our implementation. Only for operators, $\ulcorner op \urcorner$ refers to some concrete way of reflecting the operator *op* within the system itself [21].

The representation function is shown in Fig. 2. The parts of interest are the quotations for concrete terms, sequents, and inference rules. The quoted representation of a concrete term, $\ulcorner op\{b_1; \cdots ; b_n\} \urcorner$, produces a new term with a quoted name $\ulcorner op \urcorner$, and the quotation is carried out recursively on the subterms $\ulcorner b_1 \urcorner; \cdots ; \ulcorner b_n \urcorner$.[2] The quotation of a sequent, $\ulcorner \Gamma \vdash t \urcorner$, is similar: the "turnstile operator" is quoted, and the parts are quoted recursively.

The quotation of bound terms introduces a binder, written $\lambda_b x.t$, that represents each binding in quoted form.[2] Note that the binding variable itself is unchanged; the variable is preserved as a binding, but each binding is explicitly coded as a λ_b.

Finally, the quotation of an inference rule, $\ulcorner t_1 \longrightarrow \cdots \longrightarrow t_n \urcorner$ becomes a judgment about provability $(Z \vdash \Box_{\mathcal{L}} \ulcorner t_1 \urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box_{\mathcal{L}} \ulcorner t_n \urcorner)$. The context variable Z is fresh, and each sequent $Z \vdash \Box_{\mathcal{L}} \ulcorner t_i \urcorner$ is a judgment in the *meta-logic* about provability.

Informally, the reflected rule states that if each premise $t_1, \ldots, t_{n-1}$ is provable in logic $\mathcal{L}$, then so is t_n. A key goal is that the reflected rule $\ulcorner R \urcorner$ must be automatically derivable from the definition of $\mathcal{L}$. For clarity, when reasoning about a single logic we will normally omit the subscript $\Box_{\mathcal{L}}$ and just write $\Box$.

The choice of meta-logic is somewhat arbitrary. 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 [16]. In other words, our meta-logic $\mathbb{M}$is CTT and our framework logic $\mathbb{F}$ is the one provided by MetaPRL. Note that in CTT, the reflected rules $\ulcorner R \urcorner$ are sometimes required to

[2] Further discussion on quotations of names and concrete terms can be found in [21]. The encoding we use is an essential foundation for this work, however the specific encoding details have little effect on the presentation here.

include additional well-formedness constraints on the typing of the meta-variables.

Returning to our example, the quoted form of the substitution lemma (2.I) is as follows, where we write $s \ulcorner\in\urcorner t$ for $\ulcorner member\urcorner\{s; t\}$.

$$\begin{array}{l} Z \vdash \Box(X; x\colon z_3; Y \ulcorner\vdash\urcorner z_1[x] \ulcorner\in\urcorner z_2) \longrightarrow \\ Z \vdash \Box(X; Y \ulcorner\vdash\urcorner z_0 \ulcorner\in\urcorner z_3) \longrightarrow \\ Z \vdash \Box(X; Y \ulcorner\vdash\urcorner z_1[z_0] \ulcorner\in\urcorner z_2) \end{array} \tag{3.I}$$

The operators have been quoted (in this case $\ulcorner\vdash\urcorner$ and $\ulcorner\in\urcorner$), and the theorem is now a judgment about provability stated in the meta-logic as $Z \vdash \Box\cdots$. 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, the binder x is translated to a meta-binder with λ_b.

$$\left[\begin{array}{l} X; x\colon z_1 \vdash z_2[x] \longrightarrow \\ X \vdash \forall x\colon z_1.z_2[x] \end{array}\right] \quad = \quad \left(\begin{array}{l} Z \vdash \Box(X; x\colon z_1 \ulcorner\vdash\urcorner z_2[x]) \longrightarrow \\ Z \vdash \Box(X \ulcorner\vdash\urcorner \ulcorner\forall\urcorner\{z_1; \lambda_b x.z_2[x]\}) \end{array}\right)$$

3.1 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 $\mathcal{L}$ of some theorem $t_1 \longrightarrow \cdots \longrightarrow t_n$. 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 $\ulcorner\mathcal{L}\urcorner$ of the reflected theorem $(Z \vdash \Box\ulcorner t_1\urcorner) \longrightarrow \cdots \longrightarrow (Z \vdash \Box\ulcorner t_n\urcorner)$. 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 $\ulcorner R_1\urcorner, \ulcorner R_2\urcorner, \ldots, \ulcorner R_n\urcorner$.

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,[3] 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.

3.2 Syntax and reasoning

Reflected rules have an important property—the quoted terms are syntactical expressions, and they can be manipulated. There are constructors and destructors

[3] It isn't clear to us whether a similar mechanism might work for non-foundational provers (those with "trusted" decision procedures).

for quoted terms, and more importantly there is an inductively-defined type that contains all quoted terms. The specific details of the encoding have been published previously [21]. For our current purposes it simply matters that there *is* a type, so that meta-properties can be expressed.

For example, one may wish to prove a formal cut-elimination property. Using the type `Context` for sequent contexts, and the type `BTerm` for quoted terms, a cut-elimination theorem can be written as the following predicate.

$$\forall X : \texttt{Context}. \forall a, b : \texttt{BTerm}. \Box(X \ulcorner\vdash\urcorner a) \Rightarrow \Box(X, a \ulcorner\vdash\urcorner b) \Rightarrow \Box(X \ulcorner\vdash\urcorner b)$$

In addition, we have yet to define the provability predicate $\Box\, t$, where it will again be necessary to give a type to the quoted term t. Provability is the topic of the next section.

4 Defining provability

So far, we have postponed the treatment of the provability predicate $\Box_{\mathcal{L}}\, t$, which specifies that the quoted formula t is provable in logic $\mathcal{L}$. To define provability properly, we take the following steps.

- First, for each rule $R \in \mathcal{L}$, 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 $R \in \mathcal{L}$.
- A formula t is *provable* in logic $\mathcal{L}$ 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 $\mathcal{L}$ 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* is a node in the proof tree that corresponds to a concrete inference $t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_n$. 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 *iff* 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 *arity* of a meta-variable is the number of arguments, so a variable $z[t_1; \cdots; t_n]$ has arity n. Let $\texttt{BTerm}\{i\}$ be the type of quoted terms of arity i, corresponding to the space of substitution functions $\texttt{BTerm}^i \to \texttt{BTerm}$. Similarly, let $\texttt{Context}\{i\}$ be the type of contexts of arity i (the contexts correspond to lists of quoted terms).

Consider a rule R with free context variables $\{X_1^{i_1}, \ldots, X_m^{i_m}\}$ and free second-order variables $\{z_1^{j_1}, \ldots, z_n^{j_n}\}$, where the superscripts i_k and j_k indicate the arities of the variables. [4] Then a concrete inference r is a valid instance of rule R *iff* the following holds.

$$\begin{array}{l}\exists X_1^{i_1} : \texttt{Context}\{i_1\}, \ldots, X_m^{i_m} : \texttt{Context}\{i_m\}. \\ \exists z_1^{j_1} : \texttt{BTerm}\{j_1\}, \ldots, z_n^{j_n} : \texttt{BTerm}\{j_n\}. r = R \in \texttt{ProofStep}\end{array} \tag{4.I}$$

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

For the purposes of proof checking, the existential witnesses are assembled into a proof witness term, and passed as explicit arguments to the checker. A proof witness is defined to be an element of the `Witness` type, which in turn is defined as `Context list` $\times$ `BTerm list`. Returning to the example of the substitution lemma (3.I), the corresponding proof checker is defined as follows, where r is the concrete proof step to be checked.

$$\begin{array}{l}\texttt{checks}(\texttt{subst\_lemma}, r, \langle [X; Y], [z_1; z_2; z_3; z_0] \rangle) \equiv \\ r = \left(\begin{array}{c} [(X; x\colon z_3; Y \ulcorner\vdash\urcorner z_1[x] \ulcorner\in\urcorner z_2); (X; Y \ulcorner\vdash\urcorner z_0 \ulcorner\in\urcorner z_3)], \\ (X; Y \ulcorner\vdash\urcorner z_1[z_0] \ulcorner\in\urcorner z_2) \end{array} \right) \in \texttt{ProofStep}\end{array} \tag{4.II}$$

In general, the "rule checker" predicate $\texttt{checks}\{R; r; w\}$ takes three arguments, where R is a rule, $r \in \texttt{ProofStep}$ is a concrete inference, and $w \in \texttt{Witness}$ is the witness for the rule instantiation. Given a logic $\mathcal{L}$ 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.

$$\texttt{checks}\{r; w\} \equiv \exists R \in \{R_1, \ldots, R_n\}.\texttt{checks}\{R; r; w\}$$

Since proof step equality is decidable, and each logic has a finite number of rules, the $\texttt{checks}\{r; w\}$ predicate is decidable as well.

4.2 Derivations

Now that we have defined proof step checking, the next part is to define the valid derivations, or proof trees. The type D of all derivations is defined inductively in the usual way.

$$\begin{array}{rcl} D_0 & \equiv & \texttt{void} \\ D_{i+1} & \equiv & \Sigma \mathit{premises}\colon D_i\ \texttt{list}.\Sigma \mathit{goal\_term}\colon \texttt{BTerm}\{0\}.\Sigma w\colon \texttt{Witness}. \\ & & \quad \texttt{checks}\{(\texttt{goal}\{\mathit{premises}\}, \mathit{goal\_term}); w\} \\ D & \equiv & \bigcup_{i \in \mathbb{N}} D_i \end{array} \tag{4.III}$$

In this definition, the term $\texttt{goal}\{[d_1; \cdots; d_n]\}$ is the list of goal terms for derivations $d_1, \ldots, d_n$.

[4] In a setting where context variables are treated as binders, the variable arities are expressions that depend on the lengths $|X_k|$.

This definition also allows us to prove an induction principle, which will form the basis for proof induction.

$$\begin{array}{l}\forall P.(\forall \mathit{premises}\colon D\ \mathtt{list}.\forall g\colon \mathtt{BTerm}\{0\}.\forall w\colon \mathtt{Witness}.\\ \quad \mathtt{checks}\{(\mathtt{goal}\{\mathit{premises}\}, g); w\}\\ \quad\quad \Rightarrow (\forall p \in \mathit{premises}.P[p]) \Rightarrow P[(\mathit{premises}, g, w)])\\ \quad \Rightarrow (\forall d\colon D.P[d])\end{array}$$

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

$$\Box\, t \equiv \exists d\colon D.(\mathtt{goal}\{d\} = t \in \mathtt{BTerm}\{0\})$$

5 Sequent context induction

At this point, we now have a representation function where rules are reflected into statements of provability, and in addition we have a proof-checking predicate for establishing proof correctness. The next step is to prove that the reflected rules are valid using the definition of provability. For example, consider the substitution lemma example. From the proof-checking predicate (4.II), we must prove the reflected rule (3.I).

However, there is a substantial gap between the two forms. We have glossed over the fact that the proof-checking predicates are defined using standard existential quantifiers (4.I, 4.III). For a quantifier of the form $\exists X\colon \mathtt{Context}\{i\}.\cdots$ the variable X is a *first-order* variable in the meta-logic $\mathbb{M}_{\mathrm{CTT}}$. In contrast, the reflected rules preserve meta-variables, and are expressed using context and second-order meta-variables (variables of the framework logic $\mathbb{F}_{\mathsf{MetaPRL}}$).

Second-order variables can be modeled with functions on `BTerm`, so the object quantifiers are expressive enough to represent second-order quantification. The question remains, how does one derive a formula involving context variables from a similar formula that does not? In general, sequent context variables are bindings, sequent contexts are not terms, and they cannot be modeled directly in the object logic.

Since the framework meta-logic we are using (the $\mathbb{F}_{\mathsf{MetaPRL}}$ meta-logic) does not include context quantifiers, one option is to add them and use them in the proof-checking predicate. However, this is undesirable in part because the framework's meta-logic would become extremely expressive and powerful, but also because the extension is perilous and difficult to get right.

Instead, we extend the framework's meta-logic with a weak theory of sequent context induction that we call *teleportation*. The central logical property is that contexts are finite and inductively defined. Note that this represents a strengthening of the meta-logic by effectively including Peano arithmetic.

5.1 Teleportation

The concept behind teleportation is deceptively simple. Since contexts are inductively defined, contexts can be "migrated," one hypothesis at a time, from one point

in a rule to another. Scoping must be preserved, including *context variable* scoping, but beyond that the migration locations are unconstrained.

To formalize this more precisely, we introduce the notion of teleportation contexts, written $R[\![\Gamma]\!]$, which represents a term or a rule with exactly one occurrence of the context Γ. We will use the symbol ϵ to denote the empty context. These definitions are for presentation purposes; they are not part of the meta-logic. Teleportation is specified using a pair of nested teleportation contexts, which we will write as $F[\![\cdot; G[\![\cdot]\!]]\!]$. Here $F[\![\Gamma; G[\![\Delta]\!]]\!]$ must be a rule that has exactly one occurrence of each of the Γ, Δ and G; in addition G must be in scope of Γ.

The simplest teleportation rule hoists the context from G to F.

$$\frac{\begin{array}{ll}\text{(base)}\ \forall X. & F[\![\epsilon; G[\![X]\!]]\!] \\ \text{(step)}\ \forall X, Y, z. & F[\![X; G[\![x\colon z; Y[x]]\!]]\!] \longrightarrow F[\![X; x\colon z; G[\![Y[x]]\!]]\!]\end{array}}{\forall X. \quad F[\![X; G[\![\epsilon]]\!]]\!]}$$

For clarity, we have written explicit universal quantifiers for the meta-variables to emphasize that meta-variables are quantified for each clause/rule. Again, these do not exist explicitly in the meta-logic, and we will omit them in the remaining rules. As usual, it is assumed that the schema language of the teleportation contexts would alpha-rename the bound variables as needed to avoid capture.

For generality, it is frequently useful to transform the hypotheses during migration. In the following rule f is an arbitrary function.

$$\frac{\begin{array}{ll}\text{(base)} & F[\![\epsilon; G[\![X]\!]]\!] \\ \text{(step)} & F[\![X; G[\![x\colon f(z); Y[x]]\!]]\!] \longrightarrow F[\![X; x\colon z; G[\![Y[x]]\!]]\!]\end{array}}{F[\![X; G[\![\epsilon]]\!]]\!]}$$

There is a corresponding reverse-hoisting rule.

$$\frac{\begin{array}{ll}\text{(base)} & F[\![X; G[\![\epsilon]]\!]]\!] \\ \text{(step)} & F[\![X; x\colon f(z); G[\![Y[x]]\!]]\!] \longrightarrow F[\![X; G[\![x\colon z; Y[x]]\!]]\!]\end{array}}{F[\![\epsilon; G[\![X]\!]]\!]}$$

We add the teleportation rules as new primitive rules in our framework logic $\mathbb{F}_{\mathsf{MetaPRL}}$. The conservativity theorem for sequent schema [19], which states that the language of framework meta-variables is a conservative extension of the meta-theory, can be extended to include teleportation rules. The central observation here is that for any particular finite concrete context Γ, any proof using the teleportation rules can be transformed into a proof without teleportation by posing a finite sequence of lemmas, one for each of the intermediate steps.

5.2 A simple example

For a fairly natural example, consider the problem of context exchange. That is, we are given an exchange rule for hypotheses, and we wish to derive a rule for

exchanging contexts.

$$\frac{X; y\colon z_2; x\colon z_1; Y[x;y] \vdash z_3[x;y]}{X; x\colon z_1; y\colon z_2; Y[x;y] \vdash z_3[x;y]} \Longrightarrow \frac{X; Z_2; Z_1; Y \vdash z}{X; Z_1; Z_2; Y \vdash z}$$

The proof in this case can be posed as a nested induction. To begin, we propose to migrate Z_2 left, where the $\bullet$ denotes the target: $X; \bullet; Z_1; Z_2; Y \vdash z$. The base case follows by assumption, and the step case presents us with the following subproblem.

$$(X; Z_3; x\colon z'; Z_1; Z_2; Y \vdash z) \longrightarrow (X; Z_3; Z_1; x\colon z'; Z_2; Y \vdash z).$$

The proof is concluded by migrating Z_1 past the hypothesis $x\colon z'$.

5.3 Computation on sequent terms

The sequent induction scheme also introduces a sequent induction combinator for computation over a sequent context. We introduce two new terms to the meta-logic. The $\texttt{sequent\_ind}\{x, y.step[x;y]; s\}$ performs computation over a sequent term s. The reduction rules for sequent computation are as follows.

$$\texttt{sequent\_ind}\{x, y.step[x;y]; (\vdash t)\} \rightarrow t$$

$$\texttt{sequent\_ind}\{x, y.step[x;y]; (z\colon t_1; X[z] \vdash t_2[z])\} \rightarrow$$

$$step[t_1; \lambda z.\texttt{sequent\_ind}\{x, y.step[x;y]; (X[z] \vdash t_2[z])\}]$$

To illustrate, suppose we wish to develop a "vector" universal quantifier. That is, a sequent with the following definition, given that the logic has a "scalar" quantifier $\forall x\colon t_1.t_2[x]$.

$$x_1\colon t_1; \cdots; x_n\colon t_n \vdash_\forall t_{n+1} \quad \equiv \quad \forall x_1\colon t_1, \ldots, x_n\colon t_n.t_{n+1}$$

The definition is implemented in terms of sequent induction.

$$\Gamma \vdash_\forall t \quad \equiv \quad \texttt{sequent\_ind}\{x, y.\forall z\colon x.(yz); (\Gamma \vdash t)\}$$

We get the following reductions.

$$\begin{array}{rcl} \vdash_\forall z & \rightarrow & z \\ x\colon z_1; X[x] \vdash_\forall z_2[x] & \rightarrow & \forall x\colon z_1.(X[x] \vdash_\forall z_2[x]) \end{array}$$

The simple introduction rule can be derived directly.

$$\frac{Z; x\colon z_1 \vdash (X[x] \vdash_\forall z_2[x])}{Z \vdash (x\colon z_1; X[x] \vdash_\forall z_2[x])}\ \text{vall-intro-single}$$

A general introduction rule is also derivable using the teleportation rules.

$$\frac{Z; X \vdash z}{Z \vdash (X \vdash_\forall z)}\ \text{vall-intro}$$

Using similar methods, it is possible to define a logic of vector operators, quantifiers, and a vector lambda calculus.

Note that in these rules, the variable X is a context variable, and the rules are valid for any instance of X.

5.4 Sequent induction and reflection

With this new tool in hand, let us return to the topic of reflection, where the issue was that we need to derive proofs of the reflected rules (with context variables) from the proof-checking predicates (no context variables).

At this point, the plan is conceptually easy. There are 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 the non-canonical form that *includes 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 $\ulcorner\lambda_H\urcorner x\colon t_1.t_2$ is a quoted term that represents a hypothesis, its binding, and the rest of the sequent; and $\ulcorner\texttt{concl}\urcorner\{t\}$ represents the conclusion of the sequent. The proof-checking predicates operate directly on quoted terms with this representation.

$$x_1\colon t_1;\cdots;x_n\colon t_n\ulcorner\vdash\urcorner t_{n+1} \quad\equiv\quad \ulcorner\lambda_H\urcorner x_1\colon t_1.\ldots.\ulcorner\lambda_H\urcorner x_n\colon t_n.\ulcorner\texttt{concl}\urcorner\{t_{n+1}\}$$

For the second part, we define a function using `sequent_ind` that computes the canonical representation from its non-canonical form. This function, written $\vdash_B$, is defined as follows.

$$X\vdash_B t \quad\equiv\quad \texttt{sequent\_ind}\{x,y.\ulcorner\lambda_H\urcorner z\colon x.(y\ z);(X\vdash\ulcorner\texttt{concl}\urcorner\{t\})\}$$

The original reflected form of a rule $R = (\Gamma_1 \vdash t_1) \longrightarrow \cdots \longrightarrow (\Gamma_n \vdash t_n)$ is $\ulcorner R\urcorner = Z \vdash \Box(\ulcorner\Gamma_1\urcorner\ \ulcorner\vdash\urcorner\ \ulcorner t_1\urcorner) \longrightarrow \cdots \longrightarrow Z \vdash \Box(\ulcorner\Gamma_n\urcorner\ulcorner\vdash\urcorner\ulcorner t_n\urcorner)$. Using the non-canonical forms, the new representation is as follows.

$$\ulcorner R\urcorner \quad=\quad (Z\vdash\Box(\ulcorner\Gamma_1\urcorner\vdash_B\ulcorner t_1\urcorner))\longrightarrow\cdots\longrightarrow(Z\vdash\Box(\ulcorner\Gamma_n\urcorner\vdash_B\ulcorner t_n\urcorner))$$

The right-hand-side is now proved by reducing the $\vdash_B$ sequents to canonical form, then proving that the reduced form passes the proof–checking predicate for all instances of the meta-variables. Note that contexts and context variables are not terms, and so it remains impossible to quantify over them directly. However, the reduced form of a non-canonical $\vdash_B$ sequent with context variables does contain sequent subterms with context variables. With teleportation it is possible to show that these embedded terms are well–defined.

These correspondence between a reflected rule and its proof-checking predicate is very close. In our implementation, the reflected rule and the proof checking definitions are created mechanically, and the proof is completely automated.

6 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 crucial piece missing—induction on the provability predicate.

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

Now, for illustration, assume the logic $\mathcal{L}$ contains three rules, $\mathcal{L} = t_{11},\ t_{21} \longrightarrow t_{22},\ t_{31} \longrightarrow t_{32} \longrightarrow t_{33}$. Then the induction form has the following shape.

$$\frac{\begin{array}{l}\text{(rule sketch)}\\ \Gamma; \Box\, t_{11} \vdash P[t_{11}]\\ \Gamma; \Box\, t_{21}; \Box\, t_{22}; P[t_{21}] \vdash P[t_{22}]\\ \Gamma; \Box\, t_{31}; \Box\, t_{32}; \Box\, t_{33}; P[t_{31}]; P[t_{32}] \vdash P[t_{33}]\end{array}}{\Gamma; \Box\, x \vdash P[x]}$$

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. As we explained in Section 5, explicit quantification of meta-variables is not expressible in our meta-logic.

However, here it is acceptable to use object-quantifiers. 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 $t_{ij}[\boldsymbol{X}]$ to represent a term that may contain any of the variable $\boldsymbol{X}$ but is otherwise free of context variables.

$$\frac{\begin{array}{l}\Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{11}[\boldsymbol{X}] \vdash P[t_{11}[\boldsymbol{X}]]\\ \Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{21}[\boldsymbol{X}]; \Box\, t_{22}[\boldsymbol{X}]; P[t_{21}[\boldsymbol{X}]] \vdash P[t_{22}[\boldsymbol{X}]]\\ \Gamma; \boldsymbol{X} : \texttt{Context}; \Box\, t_{31}[\boldsymbol{X}]; \Box\, t_{32}[\boldsymbol{X}]; \Box\, t_{33}[\boldsymbol{X}]; P[t_{31}[\boldsymbol{X}]]; P[t_{32}[\boldsymbol{X}]] \vdash P[t_{33}[\boldsymbol{X}]]\end{array}}{\Gamma; \Box\, x \vdash P[x]}$$

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 provable term t and the predicate P. The actual theorem has the form $\Gamma; u\colon t_1; \Box\, t_2[u] \vdash P[t_3[u]]$, 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 $\mathcal{L}$, 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.

This mechanism establishes the principle of proof induction. The principle of structural induction is reducible to proof induction by specifying the syntax of a language as a logic of type-checking.

For every object logic, the corresponding induction principle is not only automatically formulated by our system, but is also automatically derived. Since the proof induction principle implies soundness, this means that while we do not prove

the soundness of our formalization *in general*, for each particular object logic, it will be established automatically.

7 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 [12] has written an excellent survey and critique of a broad range of approaches to reflection. We give another broad survey in a previous paper [21].

Our approach to representing the syntax with bindings has some similarities to the HOAS implemented in Coq by Despeyroux and Hirschowitz [6] and to the modal λ-calculus [9, 7, 8].

In 1931 Gödel used reflection to prove his famous incompleteness theorem [10]. A modern version of the Gödel's approach was used by Aitken *et.al.* [3, 1, 2, 5] 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. Rueß [23] has implemented a computation reflection mechanism. Hendriks [13] formalized natural deduction for first-order logic in the proof assistant Coq, using de Bruijn indices for variable binding. O'Connor [22] constructively proved the Gödel–Rosser incompleteness theorem using the natural numbers to encode formulas and proofs.

8 Conclusion

The goal of this work is to develop a *practical* theory of logical reflection. We claim that doing so requires preserving the structure of a theory when it is reflected, including variables, meta–variables, and bindings. We presented a structure-preserving representation, building on previous work with the representation of logical terms [21]. Besides, we developed a new account of sequent context induction, called *teleportation*, to allow reasoning and computation over terms that include sequent context variables. 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 startlingly 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. In particular, the formalization of

context induction required several man-months of effort, mainly due to the need to develop a logical infrastructure for reasoning about terms containing context variables.

We are currently using reflection to develop an account of $F_{<:}$ type theory, which has acted both as a challenge and a guide [15]. For work in the near future, we are considering alternate ways to pose the proof induction principle. Induction is, by nature, not modular. However, we believe that significant practical advances can be made through improved automation and hierarchical decomposition.

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, the teleportation principle is needed. 4) A *function image* type [20].

References

[1] Aitken, W. and R. L. Constable, *Reflecting on NuPRL : Lessons 1–4*, Technical report, Cornell University, Computer Science Department, Ithaca, NY (1992).

[2] Aitken, W., R. L. Constable and J. Underwood, *Metalogical Frameworks II: Using reflected decision procedures*, Journal of Automated Reasoning **22** (1993), pp. 171–221.

[3] Allen, S. F., R. L. Constable, D. J. Howe and W. Aitken, *The semantics of reflected proof*, in: *Proceedings of the 5th Symposium on Logic in Computer Science* (1990), pp. 95–197.

[4] Barzilay, E., "Implementing Reflection in NuPRL," Ph.D. thesis, Cornell University (2006).

[5] Constable, R. L., *Using reflection to explain and enhance type theory*, in: H. Schwichtenberg, editor, *Proof and Computation*, NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F **139**, Springer, Berlin, 1994 pp. 65–100.

[6] Despeyroux, J. and A. Hirschowitz, *Higher-order abstract syntax with induction in Coq*, in: *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, Lecture Notes in Computer Science **822** (1994), pp. 159–173, also appears as INRIA research report RR-2292.

[7] Despeyroux, J. and P. Leleu, *A modal lambda calculus with iteration and case constructs*, in: T. Altenkirch, W. Naraschewski and B. Reus, editors, *Types for Proofs and Programs: International Workshop, TYPES '98, Kloster Irsee, Germany, March 1998*, Lecture Notes in Computer Science **1657**, 1999, pp. 47–61.
URL http://www.springerlink.com/link.asp?id=984f76cm6b6qv0a4

[8] Despeyroux, J. and P. Leleu, *Recursion over objects of functional type*, Mathematical Structures in Computer Science **11** (2001), pp. 555–572.
URL http://citeseer.ist.psu.edu/despeyroux00recursion.html

[9] Despeyroux, J., F. Pfenning and C. Schürmann, *Primitive recursion for higher–order abstract syntax*, in: Hindley [18], pp. 147–163, an extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[10] Gödel, K., *Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I*, Monatshefte für Mathematik und Physik **38** (1931), pp. 173–198, english version in [24].

[11] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery **40** (1993), pp. 143–184, a revised and expanded version of the 1987 paper.

[12] Harrison, J., *Metatheory and reflection in theorem proving: A survey and critique*, Technical Report CRC-53, SRI International, Cambridge Computer Science Research Centre, Millers Yard, Cambridge, UK (1995).
URL http://www.cl.cam.ac.uk/users/jrh/papers/reflect.html

[13] Hendriks, D., *Proof reflection in Coq*, Journal of Automated Reasoning **29** (2002), pp. 277–307.

[14] Hickey, J., A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. N. Krupski, L. Lorigo, S. Schmitt, C. Witty and X. Yu, *MetaPRL — A modular logical environment*, in: D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, Lecture Notes in Computer Science **2758** (2003), pp. 287–303.
URL http://nogin.org/papers/metaprl.html

[15] Hickey, J., A. Nogin, X. Yu and A. Kopylov, *Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection*, Accepted to the International Conference on Functional Programming (ICFP) (2006).

[16] Hickey, J. J., B. Aydemir, Y. Bryukhov, A. Kopylov, A. Nogin and X. Yu, *A listing of MetaPRL theories*.
URL http://metaprl.org/theories.pdf

[17] Hickey, J. J., A. Nogin, A. Kopylov et al., *MetaPRL home page*.
URL http://metaprl.org/

[18] Hindley, R., editor, "Proceedings of the International Conference on Typed Lambda Calculus and its Applications (TLCA'97)," Lecture Notes in Computer Science **1210**, Springer-Verlag, Nancy, France, 1997.

[19] Nogin, A. and J. Hickey, *Sequent schema for derived rules*, in: V. A. Carreño, C. A. Muñoz and S. Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, Lecture Notes in Computer Science **2410** (2002), pp. 281–297.
URL http://nogin.org/papers/derived_rules.html

[20] Nogin, A. and A. Kopylov, *Formalizing type operations using the "Image" type constructor*, Accepted to Workshop on Logic, Language, Information and Computation (WoLLIC) (2006).

[21] Nogin, A., A. Kopylov, X. Yu and J. Hickey, *A computational approach to reflective meta-reasoning about languages with bindings*, in: *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding* (2005), pp. 2–12, an extended version is available as California Institute of Technology technical report CaltechCSTR:2005.003.

[22] OConnor, R., *Essential incompleteness of arithmetic verified by Coq*, in: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Lecture Notes in Computer Science **3603**, 2005, pp. 245–260.

[23] Rueß, H., *Computational reflection in the calculus of constructions and its application to theorem proving*, in: Hindley [18].

[24] van Heijenoort, J., editor, "From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931," Harvard University Press, Cambridge, MA, 1967.