Reflection and Propositions-as-Types*

Sergei Artemov, Eli Barzilay, Robert L. Constable, and Aleksey Nogin

Department of Computer Science
Cornell University
Ithaca, NY 14853
{artemov,eli,rc,nogin}@cs.cornell.edu

Abstract. Reflection is the ability of a deductive system to internalize aspects of its own structure and thereby reason to some extent about itself. In this paper we present a theoretical framework for exploring reflection in type theories that use the "Propositions-as-Types" principle, such as Martin-Löf style theories. One of the main results is that it is unnecessary to build a complete Gödel style "reflection" layer on top of the logical theory. This makes it possible to use our framework for an efficient implementation of reflection in theorem provers for such type theories. We are doing this for the NuPRL and MetaPRL systems.

1 Introduction

We use the term "reflection" to refer to the capability of a logical system to talk about itself, say about its consistency, provability properties, syntax, semantics and proofs. There are various "degrees of reflection" depending on how much the system can say about itself. We are concerned mainly with constructive formal systems that define a notion of computation; there is more to say about these systems than about non-computational ones. Moreover, such systems are important in computer science. One reason is that they are relevant to the foundations of computer science; and another is because their implementations, such as NuPRL [11], MetaPRL [15, 19], Coq [6] and Alf [18] have demonstrated that such systems are critical to the challenge of creating reliable software as well as to the new enterprise of formalizing computational mathematics. We refer to reflection for implemented proof development systems based on open constructive logics as *practical reflection*. According to our definitions, practical reflection allows a system to talk about its computability relation.

It turns out that practical reflection is a subtle business. First there is a theoretical issue, namely the most obvious (but naive) proof rules are inconsistent because of Lob's theorem and related phenomena. Second, there is a practical challenge in that the traditional Gödel approach to reflection entails, in a sense, a complete reimplementation of a system within itself. While this approach is theoretically sound, it leads to an expensive and unnecessary blowup when applied to real computer systems.

* This work was partially supported by DARPA grant LPE-F30602-98-2-0198 (Open Logical Programming Environments).

This paper addresses these two problems. We show how to accomplish practical reflection in a way that gradually incorporates more and more of a logic. We then show how to derive reflection rules from the base logic, and finally we show how to approach the problem of validity for the reflected system by proving properties of it and relating them to other logical systems. In particular, in Section 8 we show how to relate a simple abstract reflected propositional logic of Artemov [2] to our type theory.

To implement any form of reflection, the first thing that we need is to identify the level of objects we are interested in, and then find a way to represent them. Section 2 explains how a theory that uses the "Propositions-as-types" principle allows us to internalize and reason about provability (using an implicit form of the Curry-Howard isomorphism and higher-order functionality of type theory). However this approach is too abstract for many practical needs: we would like, among other things, to be able to talk about syntactical objects — *terms*.

The most direct ways of implementing concrete syntactical reasoning (such as computing Gödel numbers) can lead to exponential space complexity. On the other hand, with a theorem prover we have a computer system that already has an encoding of the syntax. In Section 3 we show how this encoding can be *reused* by *exposing* it to the logic rather than *reimplementing* it[1]. This is the approach that have been used in programming languages (e.g., Scheme [17]) as a natural implementation technique. Boyer and Moore used this approach to talk about the notion of computability in NqThm [8].

Once the internal implementation of syntax is exposed to the logic, we need to establish the relation between these *syntactic* objects and the *semantical* objects they should represent. One way of achieving this would be to copy all existing semantical rules into the syntactical domain, getting a separate syntactical provability theory, artificially-linked to the actual one [1]. We present a more direct approach which, again, follows the design principle of Section 3 — instead of duplicating this information, we expose it. In a sense, we "let the theory know" how the syntactical object correspond to semantical object they represent. To do this, in Section 4 we add a function to our theory that takes a syntactic object of type *Term* and returns the semantical object it represents.

Surprisingly, it turns out that such a function (together with a simple description of its properties) is the only thing that we need for most of our theorem-proving needs. In Section 5 we demonstrate how that is possible and, in particular, how for each application we can choose which objects (including proof objects) we want to treat purely semantically and which objects we want to treat both syntactically and semantically.

While our main goal is to implement practical reflection, we cannot say that we have achieved this goal until we have a good understanding of the logical power of our

[1] Sometimes in addition to "proper" reflection of a logical system A *in itself*, we may want to be able to reason about some other logical system B inside A. In that case, obviously, we have no other choice than to reimplement in A the relevant structures from B that are absent in A. However we still can apply the approach outlined in this paper to avoid reimplementing the structures that are common to both.

system and in particular, how it compares to traditional reflection approaches. This reveals an additional need to be able to reason generally about logical properties of reflective computer systems while abstracting away from the specifics of the reflection implementation. To address this need, we present in Section 7 a calculus designed for studying reflection in an implementation-independent way, and we show how to interpret it in our type theories.

2 Internalizing provability via "Propositions-as-types"

Before we can figure out what we need to add to a type theory in order for it to be able to reflect itself, we need to understand what is it that it is capable of doing on its own. In the case of a type theory based on the "Propositions-as-types" principle, the answer turns out to be — "a lot".

One of the common reasons for theorem provers' users to want to have reflection capabilities is so that they can express and prove a theorem that formulas from certain syntactic class are true and then be able to apply that theorem to particular formulas. A common "toy example" for such approach is proving that for any $n \geq 1$, $1 \leq k \leq n$ and propositions C_i $(i = 1 \dots n)$, the formula $C_n \Rightarrow (C_{n-1} \Rightarrow \dots (C_1 \Rightarrow C_k) \dots)$ (which we denote as $F_{C,n,k}$) is provable. We can use the higher-order functions present in type theory to express $F_{C,n,k}$ as $\textsc{primrec}\big(n;\ C(k);\ \lambda P : \texttt{Prop}.\ \lambda i : \mathbb{N}.\ C(i) \Rightarrow P\big)$ [2].

Using the "Propositions-as-types" nature of our type theory, we can also state that $F_{C,n,k}$ is always *provable*:

$$\forall n : \mathbb{N}^{+}.\ \forall k : \{1..n\}.\ \forall C : \big(\{1 \dots n\} \to \texttt{Prop}\big).\ F_{C,n,k} \tag{1}$$

and then prove the above statement by induction over n using the fact that when we unfold the definition of PRIMREC for positive n, we get a formula with an "$\Rightarrow$" as its top-level operator which allows us to prove the formula using ordinary logical rules for implication.

In order to better understand how close the above is to "real reflection", consider the following. After the proof of (1) is complete, we can automatically extract a *witness* from it and convert the proof into a proof of

$$\forall n : \mathbb{N}^{+}.\ \forall k : \{1..n\}.\ \forall C : \big(\{1 \dots n\} \to \texttt{Prop}\big).\ w_{C,n,k} \in F_{C,n,k} \tag{2}$$

which means that for each particular C, n and k, $w_{C,n,k}$ computes a Curry-Howard encoding of a proof of $F_{C,n,k}$, which in turn means that (2) explicitly shows that $F_{C,n,k}$ is always provable while (1) is doing exactly the same implicitly. Note that from this point of view, the "reflection" rule is just the (*witness*) rule

$$\frac{\Gamma \vdash t \in A}{\Gamma \vdash A} \quad (\textit{witness})$$

that was always present in type theory.

[2] Where PRIMREC is a primitive recursion operator s.t. $\textsc{primrec}(0;\ \textit{base};\ \textit{step}) = \textit{base}$, and $\textsc{primrec}(i;\ \textit{base};\ \textit{step}) = \textit{step}(\textsc{primrec}(i-1;\ \textit{base};\ \textit{step});\ i)$ (when $i > 0$), and $\texttt{Prop}$ is a type of propositions.

Of course, this is not a real reflection yet. One feature that is noticeably missing is the ability to reason about syntactical properties of formulas and syntactical operations on formulas. This feature is addressed in Sections 3 through 5.

3 Internalizing syntax: the *Term* type

To add *practical* reflection to a logical system, we need to be able to reason about its syntax, be able to do structural syntactic induction, reason about evaluation, etc. For this, we must begin by allowing the system to express its own syntax: adding term quotations. For an extended discussion, see [7].

All theorem prover implementations have some implementation of its syntax datatype, commonly called *term*. We want to have a *Term* type of syntactic *user* objects in our theory that represents this internal *term* type of implementation objects in our logical environment. The traditional "Gödel-like" approach is to use existing logical capabilities to create *Term* as an encoding for *term* objects, Gödel numbers in the case of an arithmetic-oriented logic. In type theories like NuPRL's, we can use a recursive type with tuples and lists, which has the advantage of producing a declaration of *Term* that is analogous to the internal *term* definition. There are two fundamental problems with this solution:

- This representation is by its nature exponential in its quotation level.
- Information that is part of the implementation is repeated unnecessarily.

To demonstrate the first problem, take a simple NuPRL term: "$\mathrm{pair}(1;2)$" — its quotation using this naive definition is "$\mathrm{pair}(\ulcorner \mathrm{pair} \urcorner; \mathrm{list}(\ulcorner 1 \urcorner; \ulcorner 2 \urcorner))$", and quoting this yields:

$$\mathrm{pair}(\ulcorner \mathrm{pair} \urcorner; \mathrm{list}(\ulcorner\ulcorner \mathrm{pair} \urcorner\urcorner; \mathrm{pair}(\ulcorner \mathrm{list} \urcorner; \mathrm{list}(\ulcorner\ulcorner 1 \urcorner\urcorner; \ulcorner\ulcorner 2 \urcorner\urcorner)))){}^{3}.$$

A similar problem occurs in programming languages that implement reflection, like Scheme. There, the common solution is for the implementation to *expose* its internal syntax representation to user-level code. Both problems above are solved instantly by this approach: there is no blowup, and there is no repetition of structure definitions, there is even no need for verifying that the reflected part is equivalent to the implementation since they are the *same*. Most Scheme implementations take this even further: the `eval` function is the internal function which is exposed to the user-level; [20] shows how this approach can get an infinite tower of processors.

This solution is the general principle that guides our implementation of reflection: *never repeat information* — in our case, we expose internal machinery instead of reimplementing it. Translating this principle to term quotation, we wish to expose internal *term* objects — make *Term* be *term*. Theorem provers behave differently than programming languages and the straightforward solution does not work. For example, NuPRL uses equational reasoning so it assumes that terms can be substituted with equal terms, therefore, a quotation context like Scheme's `quote` special form cannot be used. Instead, we introduce a mechanism for *shifting* operator names: a tagged

[3] This is simplified: quoted primitives are themselves terms.

term "$\underline{t(}\ldots\underline{)}$" is a value term[4] that stands for "$t(\ldots)$". To quote a complete term, we tag its operator and continue recursively, for example, the quotation "$\mathrm{pair}(1;2)$" is "$\underline{\mathrm{pair}(1;2)}$" — the second is a member of the *Term* type, that stands for the first as an internal *term* instance. The ability to locally choose terms to quote gives us several advantages:

- no need for a special evaluation contexts like Scheme's `quote`;
- we can still mix constant quotations with descriptions: terms that evaluate to quotations, no quasiquote mechanism is needed;
- linear number of tags instead of an exponential blowup.

Another difficulty with NuPRL's syntax is that *term* objects contain *bound subterms* — terms with binding positions. This problem has a surprisingly simple solution: binding occurrences are left intact — they are still binding positions, the quotation of "$\lambda x.x+1$" is "$\underline{\lambda x.}x\underline{+1}$". This enables us to use internal *term* objects for representing terms, but it has a strong implication: since quoted bound variables are still variables, we have no access to their *names*. This might sound as a restriction, but this is not the case. The interface that gets exposed as *Term* (e.g., its equality relation) is the internal *term modulo alpha-equality* which is a big win. We need not formalize anything related to alpha-renaming and valid substitutions — the system handles this for us the same as all terms. For example, "$\underline{\lambda x.x}$" is automatically equal to "$\underline{\lambda y.y}$". This mechanism is similar in its nature to some designs for a low-level Scheme macro mechanism to support hygienic rewrite rules, like the identifiers of [9].

The higher system abstraction layer that gets exposed using this technique eliminates a lot of work, but we still need term management capability. An approach that is again similar to high-level Standard Scheme macros [17] is to use rewrite rules[5]. The MetaPRL system [15, 16] has demonstrated that the mechanism of rewrite rules is a powerful term management tool, so exposing it should be sufficient. Some other useful operations like pattern matching and destructing terms can be expressed via rewrites. Even for extreme cases where special functionality is needed, we always follow the "exposing instead of reimplementing" approach.

4 Linking syntax with semantics

The *Term* type allows us to reason about *syntactic* objects, however, we have no connection so far to the *semantic* objects they should represent. In order to establish such a connection, we add an operation to our theory that takes a syntactic object of type *Term* and returns the semantic object it represents. We call this function "a *reference* operator" (see [13]), notated as "$\downarrow$", which "strips" quotation tags yielding the represented object. For example, the following terms all evaluate to "$\mathrm{pair}(1;2)$":

- $\downarrow\underline{\mathrm{pair}(1;2)}$
- $\downarrow\underline{\mathrm{pair}(1;1+1)}$
- $\downarrow((\lambda x.\underline{\mathrm{pair}(1;}x\underline{)})(\underline{1+1}))$

[4] A value term is a term that evaluates to itself.

[5] Note that we must have these implemented internally and exposed to the theory.

In order to have partial evaluation semantics for "$\downarrow$", it is natural to define "$\downarrow$" of a term with a quoted name as stripping the quote and continuing recursively through its subterms: "$\downarrow \underline{t(}t_1\underline{;} \ldots \underline{;}t_n\underline{)} \mapsto t(\downarrow t_1; \ldots ; \downarrow t_n)$". This definition works fine with the above example, including the last case.

However, this does not work for quoted terms that have binding positions, for example we expect "$\downarrow \underline{\lambda}x\underline{.}x \underline{+ 1}$" to evaluate to "$\lambda x.x + 1$", but following the above reduction scheme, we get "$\lambda x.(\downarrow x) + 1$". The problem lies in the fact that when we quote the original "$\lambda x.x + 1$" term, we leave the binding position x and its bound occurrences as variables, and when we push the "$\downarrow$" operator to the subterm, we must not try to strip the non-existent quotation tag from bound occurrences. The way we can still push "$\downarrow$" into bound subterms is to wrap the corresponding bound variables by an operator that cancels the effect of "$\downarrow$", say "$\uparrow$", using standard substitution. So "$\downarrow \underline{\lambda}x\underline{.}x \underline{+ 1} \mapsto \lambda x.\downarrow ((\uparrow x) \underline{+ 1})$," and evaluation continues with "$\lambda x.(\downarrow \uparrow x) + 1$", and finally the "$\downarrow\uparrow$" cancel.

The formal partial evaluation rule for "$\downarrow$" is therefore simple:

$$\downarrow \underline{T(}\boldsymbol{x_1}\underline{.}t_1\underline{;} \ldots \underline{;}\boldsymbol{x_n}\underline{.}t_n\underline{;)} \mapsto T(\boldsymbol{x_1}.\downarrow t_1[\uparrow x_1/x_1]; \ldots ; \boldsymbol{x_n}.\downarrow t_n[\uparrow x_n/x_n]) \tag{3}$$

$$\downarrow\uparrow x \mapsto x \tag{4}$$

This makes "$\uparrow$" an auxiliary constructor for evaluating "$\downarrow$". "$\uparrow$" is a value term that can be considered as a "promise" to cancel a future "$\downarrow$" application[6].

A concept related to "$\downarrow$", is that of subtypes of *Term* that classify them according to their denotation types. Informally, we define Term_A as $\{x : \mathit{Term} \mid \downarrow x \in A\}$. For an extended discussion, see [13, Section 3].

5 Implementing full reflection

In our approach to reflection, we get to choose which objects we want to treat purely *semantically*, referring only to properties of the object itself, and which objects we want to treat both *syntactically and semantically*, referring to properties of both the object itself and to properties of the term it represents[7].

Note that in the conventional Gödel-style approach, when we discuss the provability of some formula, we treat the formula itself purely syntactically, and we treat proofs both semantically (when reasoning about the provability of certain propositions) and syntactically (when, for example, doing induction on proofs). In our case, since the way we have internalized provability is not directly connected to syntax in any way, we have the freedom of choosing whether we want to treat something

[6] Another view on the interplay of "$\downarrow$" and "$\uparrow$" then we use both to keep bound variables at the same quotation level.
Note that "$\uparrow$" cannot be defined as a function as there is no deterministic way to get a syntactic representation for a semantical value.

[7] For example, in the case of a λ-term, the properties of the function it computes would be the semantical properties and the syntactical structure of the term itself would be, obviously, its syntactical property.

syntactically on a per-proof and per-object basis. Also, when we treat something syntactically, the $\downarrow$ operation described above gives us the ability to switch to reasoning about semantic properties of an object.

For example, suppose A and B are syntactic representations of propositions, in other words, they have the type $Term_{\texttt{Prop}}$. In that case $A \underline{\Rightarrow} B$ is a syntactic implication from A to B. Using $\downarrow$, we can easily express the statement that A is a syntactic representation of a provable formula — $\downarrow A$. We can also express something very similar to modal normality principle ($\Box A \Rightarrow \Box(A \Rightarrow B) \Rightarrow \Box B$):

$$\forall A : Term_{\texttt{Prop}}.\ \forall B : Term_{\texttt{Prop}}.\ (\downarrow A) \Rightarrow (\downarrow(A \underline{\Rightarrow} B)) \Rightarrow (\downarrow B) \tag{5}$$

Using the partial evaluation rule for $\downarrow$ (3), we can easily prove that $\downarrow(A \underline{\Rightarrow} B)$ is the same as $(\downarrow A) \Rightarrow (\downarrow B)$ after which (5) can be proven using the modus ponens rule. Similarly, we can prove an explicit version of (5):

$$\forall A : Term_{\texttt{Prop}}.\ \forall B : Term_{\texttt{Prop}}.\ \forall a : (\downarrow A).\ \forall f : (\downarrow(A \underline{\Rightarrow} B)).\ (f \circ a) \in (\downarrow B) \tag{6}$$

where $f \circ a$ is a notation for $f(a)$. Clearly, in (5) we treat proofs purely semantically. However, if we use another instance of (3), namely the fact that $\downarrow(f \underline{\circ} a)$ is the same as $(\downarrow f) \circ (\downarrow a)$, we can also prove the more syntactical version of (6):

$$\forall A : Term_{\texttt{Prop}}.\ \forall B : Term_{\texttt{Prop}}.\ \forall a : Term_{\downarrow A}.\ \forall f : term_{\downarrow (A \underline{\Rightarrow} B)}.\ (f \underline{\circ} a) \in Term_{\downarrow B} \tag{7}$$

For completeness sake, we can also consider the case where we only want to treat proofs syntactically, but only treat formulas semantically. In that case we can state and prove the following:

$$\forall A : \texttt{Prop}.\ \forall B : \texttt{Prop}.\ \forall a : Term_A.\ \forall f : Term_{A \Rightarrow B}.\ (f \underline{\circ} a) \in Term_B \tag{8}$$

It is worth mentioning that among all of the examples above, (7) is closest to the conventional way of dealing with provability (where everything is considered syntactically). If we denote $\Box A\ ==\ Term_{\downarrow A}$ and then take the implicit version of (7), we get the normalization principle:

$$\forall A : Term_{\texttt{Prop}}.\ \forall B : Term_{\texttt{Prop}}.\ \Box A \Rightarrow \Box(A \underline{\Rightarrow} B) \Rightarrow \Box B \tag{9}$$

While the examples above illustrate that we can treat anything syntactically that we want to treat that way, they do not show why we would want to do it. Consider now the following example — suppose we want to write, say, a normalization procedure for polynomials with integer coefficients and we want to be able to prove some properties of it. In this case, obviously, we need to be able to reason about syntactic properties of the polynomials (since normalization is a syntactical procedure) as well as about their semantical properties (we want to be able to say that a normalized polynomial is always equal to the original one). To do this, we define a type `Poly` (a subtype of $Term_{\mathbb{Z} \to \mathbb{Z}}$)[8] of polynomials with integer coefficients and we write our normalization

[8] Writing such a definition is a straightforward process, but a little lengthy when doing it "from scratch", so we omit it.

procedure $\mathtt{norm} : \mathtt{Poly} \to \mathtt{Poly}$. Now we can state some properties of the normalization algorithm:

$$\begin{array}{l} \forall p : \mathtt{Poly}.\ {\downarrow}(\mathtt{norm}(p)) = {\downarrow}p \in (\mathbb{Z} \to \mathbb{Z})\ \ \& \\ \forall p, p' : \mathtt{Poly}.\ ({\downarrow}(p) = {\downarrow}(p') \in (\mathbb{Z} \to \mathbb{Z})) \Leftrightarrow (\mathtt{norm}(p) = \mathtt{norm}(p') \in \mathtt{Poly}) \end{array} \tag{10}$$

where the ternary relation $a = b \in T$ means that a is equal to b as elements of type T, in particular $a = b \in \mathtt{Poly}$ means that a and b are syntactically equal[9], while $a = b \in (\mathbb{Z} \to \mathbb{Z})$ means that a and b are semantically equal (as functions from integers to integers). We can prove (10) by structural induction on p and p' and using, as usual, evaluation rule (3). For example, when proving that reordering summands does not change the result, we use

$${\downarrow}(a\underline{+}b) \;=\; ({\downarrow}a) + ({\downarrow}b) \;=\; ({\downarrow}b) + ({\downarrow}a) \;=\; {\downarrow}(b\underline{+}a)$$

where the first and last equalities follow from the properties of $\downarrow$ and the second one — from the properties of "normal" semantical $+$.

6 Reflection by trust

A limitation of our method is that it models the particular implementation that we have chosen, making it impossible to talk about alternatives without resorting to the kind of duplication we are striving to avoid. For example, our *Term* quotations of primitive *term* objects makes quotation easy to manage within the system, but it also makes it impossible to express different binding rules than the ones we have implemented. This restriction is acceptable because we expose our system exactly, and that is what reflection is about. However, there might be cases where we want to use the mechanisms for reflection to get a little more than proper reflection, namely, when we want to use meta-functionality to translate external proofs to NuPRL proofs automatically, or to translate between MetaPRL modules that use different axiom sets.

Thus our approach creates a form of reflection that is not what is usually studied in logic: we don't have any formal definition of the representation etc — instead we have a reflection of our logic as it is currently implemented. This is valid only if we trust our implementation. This validity issue becomes more acute as we expose more of the implementation. Suppose we want to talk about proof structure and even theory structure. For example, the implemented proof systems we have mentioned all use the data-type *proof* to define rules and tactics. We can axiomatize this using an internal type *Proof* — providing internal functions that find the hypothesis list, the goal, the justification, and the subproofs. The key properties and operations can be read off from the system implementation (which uses ML abstract data-types). We can also use the concrete definition from and explicit coding of proofs already done in [1]. We will not discuss the technical details in this paper; but the approach raises a potential concern that we do discuss next.

[9] Note that everywhere we say that two objects are equal in *Term*, we actually mean alpha-equal, according to the discussion in Section 3

As we expose more and more of the implementation, we end up adding many more axioms to the system, those about *Term*, those about *Eval*, *Proof* and so forth. These axioms are essentially saying that the implementation is logically correct, and we add new knowledge by "trusting the implementation," leading to characterizing our approach as "reflection by trust." But don't we risk making the logic invalid if the internals don't actually work as expected? The answer is that if the internals are not working correctly, the system can not be trusted anyway; and if they do work correctly, then the axioms we have added to the logic are valid. This means that adding these axioms does not make the system less trustworthy. Furthermore, the axioms about the implementation provide a starting point for documenting the system and for demonstrating the logical properties required of the implementation.

One approach to these concerns about correctness is to prove properties of the reflected system and relate them to standard facts about logic. We can start by considering simple abstractions of the full system and proving properties of these abstract logics. We illustrate this method last by showing how to connect an abstract propositional logic of reflection to our type theoretic system.

7 Some theory of reflection

From the point of view of proof theory, the Curry-Howard isomorphism may be regarded as a mapping from natural derivations from hypotheses

$$A_1, A_2, \ldots, A_n \vdash B$$

in intuitionistic logic to well defined typed λ-term $t(x_1, x_2, \ldots, x_n) : B$ with a dual meaning of $t(x_1, x_2, \ldots, x_n)$ as

(i) a term having type B provided its variables $x_1, x_2, \ldots, x_n$ have types $A_1, A_2, \ldots, A_n$ respectively,
(ii) a proof of B provided $x_1, x_2, \ldots, x_n$ are proofs of $A_1, A_2, \ldots, A_n$ respectively.

In this respect one can view the λ-calculus as propositional level calculus of formal derivations (proofs), where a derivation of a full form λ-term

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \;\vdash\; t(x_1, x_2, \ldots, x_n) : B$$

is a step-by-step internal replica of a corresponding intuitionistic derivation of

$$A_1, A_2, \ldots, A_n \;\vdash\; B.$$

Modal logic provides an alternative way of representing reflection where $\Box F$ is interpreted as "F is provable" or equivalently "there exists a proof of F" (cf. [4, 5, 14, 21]). In particular, the sequent $\Box A_1, \Box A_2, \ldots, \Box A_n \vdash \Box B$ can be read as

"if there exists a proof of $A_1, A_2, \ldots, A_n$, then there exists a proof of B".

Unlike λ-terms, the modal language does not represent proofs directly, but rather via provability, i.e., an assertion that proof exists. The modal approach to reflection met serious semantical difficulties which have been finally resolved by introducing a special kind of typed λ-terms ([2, 3, 5]).

In this section we present the calculus $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ (first introduced in [2]) of reflexive λ-terms which is the basic abstract model of reflected proofs. The main idea is that we want to be able to formally reason with statements of the form "t_n is a proof that t_{n-1} is a proof that ... is a proof that t_1 proves A." We denote such statement by $t_n : t_{n-1} : \ldots : t_1 : A$, which is abbreviated as $\boldsymbol{t} : A$. In addition there is an operator, "$\Uparrow$" that reflects a proof, and its inverse, "$\Downarrow$" that interprets a reflected proof. The system $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ is a joint calculus of propositions (types) and proofs (λ-terms). For the sake of brevity we follow a well-established tradition in typed λ-calculus and first consider types with intuitionistic $\{\rightarrow, \wedge\}$ logic on the background. The language of $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ contains

propositional letters (atomic types) $p_1, p_2, p_3, \ldots$
variables $x_1, x_2, x_3, \ldots$
connectives $\rightarrow, \wedge$
functional symbols: unary ! , $\Uparrow^n$, $\Downarrow^n$, $\boldsymbol{\pi}_0^n$, $\boldsymbol{\pi}_1^n$; binary $\circ^n$, $\mathbf{p}^n$,
operator symbols :, λ^n, $n = 1, 2, 3, \ldots$.

Terms τ are build from variables x by functional symbols and λ^n-operators in the usual manner:

$$\tau = x \mid ! \, \tau \mid \Uparrow^n \tau \mid \Downarrow^n \tau \mid \boldsymbol{\pi}_0^n \tau \mid \boldsymbol{\pi}_1^n \tau \mid \tau \circ^n \tau \mid \mathbf{p}^n(\tau, \tau) \mid \lambda^n x.\tau.$$

We refer to those terms as *reflexive λ-terms*. Formulas (types) φ are built from propositional letters p and terms τ by connectives and operator ':':

$$\varphi = p \mid \varphi \rightarrow \varphi \mid \varphi \wedge \varphi \mid \tau : \varphi.$$

For the sake of brevity we refer to formulas as terms (of depth 0).

In $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ we use a concise sequent style notation for derivations in $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ by reading $\Gamma \vdash F$ as a $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$-derivation of F with the set of open assumptions Γ.

We identify *well-defined terms* with derivations in the calculus $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$ of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash t(x_1, x_2, \ldots, x_n) : B.$$

This may be also read in the usual λ-term manner: term $t(x_1, x_2, \ldots, x_n)$ has type B provided each variable x_i has type A_i for all $i = 1, 2, \ldots, n$.

Within the current definition below we assume that $n = 0, 1, 2, \ldots$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$. We also agree on the following vector-style notations:

$\boldsymbol{t} : A$ denotes $t_n : t_{n-1} : \ldots t_1 : A$ (e.g. $\boldsymbol{t} : A$ is A, when $n = 0$),
$\lambda^n \boldsymbol{x}.\boldsymbol{t} : B$ denotes $\lambda^n x_n.t_n : \lambda^{n-1} x_{n-1}.t_{n-1} : \ldots \lambda^1 x_1.t_1 : B$,
$(\boldsymbol{t} \circ^n \boldsymbol{s}) : B$ denotes $(t_n \circ^n s_n) : (t_{n-1} \circ^{n-1} s_{n-1}) : \ldots (t_1 \circ s_1) : B$,
$\Uparrow^n \boldsymbol{t} : B$ denotes $\Uparrow^n t_n : \Uparrow^{n-1} t_{n-1} : \Uparrow^1 t_1 : B$,
likewise for all other functional symbols of $\boldsymbol{\lambda}^{\boldsymbol{\infty}}$.

Derivations (i.e. well-defined terms) are generated by the following clauses. Here A, B, C are formulas, Γ a finite set of formulas, $n = 0, 1, 2, \ldots$, $\boldsymbol{s}, \boldsymbol{t}$ are n-vectors of terms, $\boldsymbol{x}$ is an n-vector of variables.

(Ax) $\quad \Gamma, \boldsymbol{x}{:}A \vdash \boldsymbol{x}{:}A$

($\boldsymbol{\lambda}$) $$\frac{\Gamma, \boldsymbol{x}{:}A \vdash \boldsymbol{t}{:}B}{\Gamma \vdash \lambda^n \boldsymbol{x}.\boldsymbol{t}{:}(A \to B)},$$

where none of $\boldsymbol{x}$ occurs free in the conclusion sequent.

(App) $$\frac{\Gamma \vdash \boldsymbol{t}{:}(A \to B) \quad \Gamma \vdash \boldsymbol{s}{:}A}{\Gamma \vdash (\boldsymbol{t} \circ^n \boldsymbol{s}){:}B}$$

(p) $$\frac{\Gamma \vdash \boldsymbol{t}{:}A \quad \Gamma \vdash \boldsymbol{s}{:}B}{\Gamma \vdash \mathbf{p}^n(\boldsymbol{t}, \boldsymbol{s}){:}(A \wedge B)}$$

($\boldsymbol{\pi}$) $$\frac{\Gamma \vdash \boldsymbol{t}{:}(A_0 \wedge A_1)}{\Gamma \vdash \boldsymbol{\pi}_i^n \boldsymbol{t}{:}A_i}\ (i = 0, 1)$$

($\Uparrow$) $$\frac{\Gamma \vdash \boldsymbol{t}{:}u{:}A}{\Gamma \vdash \Uparrow^n \boldsymbol{t}{:}!u{:}u{:}A}$$

($\Downarrow$) $$\frac{\Gamma \vdash \boldsymbol{t}{:}u{:}A}{\Gamma \vdash \Downarrow^n \boldsymbol{t}{:}A}$$

Remark 1. It is clear that the intuitionistic propositional logic, with rules for implication and conjunction only, is contained in $\boldsymbol{\lambda}^\infty$. Indeed, if we require, for a derivation in $\boldsymbol{\lambda}^\infty$, that $n = 0$, and use the corresponding rules only, we have the system $\boldsymbol{Ni}_{\to\wedge}$ (cf. [21]). Similarly, the usual typed λ-calculus over $\to, \wedge$ corresponds to the level $n = 1$.

Along with the natural versions of β-contraction and projections contractions, $\boldsymbol{\lambda}^\infty$ admits up-down contraction $\Downarrow^n \Uparrow^n \boldsymbol{t}{:}A \ \triangleright\ \boldsymbol{t}{:}A$.

Theorem 1 (Cf. [2]). *$\boldsymbol{\lambda}^\infty$ is strongly normalizable and confluent. Each well-defined term of $\boldsymbol{\lambda}^\infty$ has a unique normal form.*

Remark 2. The strong normalization and confluence theorems for $\boldsymbol{\lambda}^\infty$ provide a generalization of those for the intuitionistic logic and λ-calculus. All these make $\boldsymbol{\lambda}^\infty$ the basic propositional logic of reflection.

8 Interpretation of $\boldsymbol{\lambda}^\infty$, soundness

One of the main reasons we are interested in $\boldsymbol{\lambda}^\infty$ is that it can be interpreted in the type theory in a very natural way. Depending on whether we want to be able to treat t or F (or none, or both) syntactically, we can interpret "$t : F$" as $t \in Term_F$, $t \in \downarrow F$,

$t \in F$ or $t \in Term_{\downarrow F}$. In each of the four cases, we interpret $\circ$ as application (quoted application when t is treated syntactically), λ as the one in type theory (again, the syntactical $\underline{\lambda}$ when t is syntactical), $\Uparrow t$ and the proof checker $!u$ as the universal proof checker `Ax`.

The only connective that is not present in type theory as naturally is $\Downarrow$. Under the natural interpretation we only get a version of the ($\Downarrow$) rule for $n = 0$ (e.g. $u : A \Rightarrow A$). However we can easily fix it by using $:'$ defined, for example, as

$$\langle t, x\rangle \;:'\; F \;:=\; t : A \wedge (\downarrow x = F)$$

and redefining all the connectives accordingly.

References

1. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.
2. J. Alt and S. Artemov. Reflective λ-calculus. Technical Report CFIS 2000-06, Cornell University, 2000.
3. S. Artemov. On explicit reflection in theorem proving and formal verification. In *Automated Deduction - CADE-16. Proceedings of the 16th International Conference on Automated Deduction, Trento, Italy, July 1999*, pages 267–281. Springer-Verlag, 1999. LNAI Vol. 1632.
4. S. Artemov. Uniform provability realization of intuitionistic logic, modality and lambda-terms. *Electronic Notes on Theoretical Computer Science*, 23(1), 1999. `http://www.elsevier.nl/entcs/`.
5. S. Artemov. Explicit provability and constructive semantics. *The Bulletin for Symbolic Logic*, 6(1), 2001. to appear, `http://www.math.cornell.edu/~artemov/BSL`.
6. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.
7. Eli Barzilay. Quotation and reflection in NuPRL and Scheme. Technical Report 2001–1832, Cornell University, Ithaca, New York, January 2001.
8. R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–84. Academic Press, New York, 1981.
9. W. Clinger. Hygienic macros through explicit renaming. LISP Pointers, 4(4), 1991.
10. Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F*, pages 65–100. Springer, Berlin, 1994.
11. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.
12. Robert L. Constable, Stuart F. Allen, and Douglas J. Howe. Reflecting the open-ended computation system of constructive type theory. In H. Schwichtenberg, editor, *Logic, Algebra and Computation,* NATO ASI Series, Vol. F79, pages 265–280. Springer-Verlag, 1990.

13. Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In *Preprint*, 1998.
14. D. de Jongh and G. Japaridze. Logic of provability. In S. Buss, editor, *Handbook of Proof Theory*, pages 475–546. Elsevier, 1998.
15. Jason Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, January 2001.
16. Jason Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
17. R. Kelsey, Clinger W., J. Rees, et al. Revised^5 report on the algorithmic language scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, 1998.
18. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of Lecture Notes in Computer Science, pages 213–237. Springer-Verlag, 1994.
19. MetaPRL home page. `http://metaprl.org/`.
20. B.C. Smith. Reflection and semantics in Lisp. *Principles of Programming Languages*, pages 23–35, 1984.
21. A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Amsterdam, 1996.