The Semantics of Reflected Proof*

Stuart F. Allen
Robert L. Constable
Douglas J. Howe
William E. Aitken

Cornell University
Ithaca, NY 14853

1990 (corrections 1998)

Abstract

We begin to lay the foundations for reasoning about proofs whose steps include both invocations of programs to build subproofs (*tactics*) and references to representations of proofs themselves (reflected proofs). The main result is the definition of a single type of proof which can mention itself, using a new technique which finds a fixed point of a mapping between metalanguage and object language. This single type contrasts with hierarchies of types used in other approaches to accomplish the same classification. We show that these proofs are valid, and that every proof can be reduced to a proof involving only primitive inference rules. We also show how to extend the results to proofs from which programs (such as tactics) can be derived, and to proofs that can refer to a library of definitions and previously proven theorems. We believe that the mechanism of reflection is fundamental in building proof development systems, and we illustrate its power with applications to automating reasoning and describing modes of computation.

1 Introduction

Formal proofs of interesting ideas tend to be large objects, so in practice some mechanism is used to shorten and abbreviate proofs. One of the most effective is the use of *tactics* (LCF[17], Nuprl [10], λProlog [24], HOL [16] and Oyster [9]). Proofs then are *tactic-trees.* These are proofs represented as trees in which certain steps are justified by programs (tactics) that compute subtrees.

In all heretofore existing systems, the tactic trees use two languages, one the *object language*, the other a *metalanguage*. In LCF the object language is the polymorphic predicate calculus (PPλ), in Nuprl and Oyster it is constructive type theory and in HOL it is a simple type theory. The metalanguage includes a programming language: ML in LCF, Nuprl and HOL, Prolog in Oyster. We want to consider the situation in which the object language has a notion of computation adequate for expressing tactics and to examine the possibility of using a single language by reflecting proofs from the metalanguage into the object language. This question is especially interesting in the case of systems like Nuprl in which programs are extracted from proofs, for then the language of proofs is itself a metalanguage for proofs.

*Supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409.

We show that there is such a class of proofs. This requires a new kind of argument which uses the reflection of the proof concept in a formal theory to define the notion of proof in that theory. There is an interesting closure principle revealed here based on a fixed point construction over a class of proof-like trees which interleave objects from the object language and the metalanguage.

The ensuing concept of reflected proof is noteworthy because it is encapsulated in a single recursive type, in contrast to other approaches which result in an infinite tower of types or languages [22, 27] or would result in an infinite tower if extended to provide the same closure [28]. Moreover, the issues raised in other uses of reflection are clearly present here, for instance the notions of *stance* and *connection* in the *procedural reflection* of Smith in [26] where he says "when you take a step backward to reflect, you need a place to stand, with just the right combination of connection and detachment". But these notions are easier to pick out in this setting because they occur in a single type rather than in Smith's "infinite tower of procedural self-models."

Our notion of a reflected proof turns out to be more useful than the unreflected kind. We know of at least seven applications of the concept in the areas of theorem proving, type definition and functional program evaluation. First, we know how to shorten proofs and speed-up the generation and checking of proofs by proving that the tactics are correct. Second, we can define a kind of polymorphism in theorems involving universe types. Third, it is possible to define evaluation mechanisms provably equivalent to the built-in evaluator yet more efficient on certain terms. Fourth, as a special case of the third application, expressions can be detected that can be destructively updated, thereby allowing a procedural evaluation for them. Fifth, reflection gives one approach to expressing computational complexity. Sixth, new systems ideas can be experimented with in a mathematically controlled way by running code from incomplete proofs of metatheorems. Finally, proof transformations can be used to build new logics on top of the built-in logic of a theory.

The informal concept of proof that we start with is called a *primitive proof*; it is essentially a finite tree of sequents which are either justified by rules or taken as premises. Below is an example; *Pr* is a type of propositions.

$\vdash \forall A, B\colon Pr.(A \vee B \Rightarrow B \vee A)$ by intro
$A, B\colon Pr \vdash (A \vee B \Rightarrow B \vee A)$ by intro
$A, B\colon Pr, A \vee B \ \vdash\ B \vee A$ by elim $A \vee B$

$A, B\colon Pr, A \ \vdash B \vee A$	$A, B\colon Pr, B \ \vdash B \vee A$
by intro right	by intro left.

A primitive rule is basically a function from a sequent s_0 to a list of sequents $s_1, \ldots, s_n$ such that if $s_1, \ldots, s_n$ are true, then s_0 is true. We say that a primitive rule justifies s from l provided the rule applied to s yields the sequents at the roots of the subproofs in the list l. A premise of a proof is an unproved sequent at a leaf.

One way to think of proofs containing tactics is to allow another kind of justification. It could be an expression in the metalanguage which builds a primitive proof.[1] We would say that a tactic, *tac*, justifies a sequent s provided that *tac* evaluates to a proof whose goal is s and whose premises are the roots of the subproofs. As noted above, the usual way to do this is using two languages, L and *ML*; one way to achieve a unification is to use the computational nature of L and express tactics in L itself, exploiting an internal representation of terms and proofs.

The tactic will be a term that represents a proof. This represented proof must have the goal as its root and the subgoals as its premises.

In building a tactic tree, there are different ways to check that a tactic produces the right thing. One way is to run it, another is to type check it, and a third is to prove that it has the right type. A proof (or type checking which is a special case) would save the expense of building the proof tree; we discuss this point in Section 2.2. The proof might be critical to understanding the tactic, and

[1] It is significant that we use an expression here because we want to be able to analyze the object intensionally.

indeed we may want to extract the tactic from a formal proof. Statements about the tactic refer to the internal notion of a proof. So in order to argue that the tactic is correct we must know that if there is an element of the reflected type, then there is an element of the real type. This is exactly what a reflection rule will guarantee. So we ask whether there is a notion of proof allowing this kind of reflection. In Section 2.2 we show that there is.

The notion of proof we define should be applicable to a class of *type theories* in a broad sense of the term. We do not specify this class, but instead present the main ideas somewhat abstractly, relegating most of the details specific to Nuprl's type theory to an appendix. To simplify the presentation of the main ideas we have taken advantage of a feature of Nuprl: programs are untyped. This allows us to write a program which applies to members of an as-yet undefined type. The dependency on this feature is straightforward to eliminate.

In the next section of the paper is a precise definition of reflected proof for a class of languages meeting certain computational properties. In Section 2.1 we present some preliminaries concerning representation. In section 2.2 we give the definition of proofs. We show that these proofs are valid and that any (complete) proof can be reduced to a proof with no uses of the tactic rule or reflection rule. Section 2.3 considers issues arising in constructive theories, where programs can be derived from proofs, and Section 2.4 deals with matters important for implemented systems. Section 3 discusses applications based on our experience with Nuprl.

2 Semantics of Proofs

2.1 Representation

The context of our work is that we are standing outside of a formal language L (traditionally called the *object language*). The syntax of L is given by an inductively defined class of terms. This is a type in the observer's language or *metalanguage*, *ML*. We are especially interested in those languages L with a computable evaluation relation on terms [14, 10, 17, 19, 25].

We are concerned with languages L in which types can be defined. From outside of L we can see a type T of L as the class of terms which have that type, and we require that for any term t, t and its value t' have the same types, *i.e.,* if $t \in T$ then $t' \in T$; furthermore, we require that if $t \in T$ then t and t' are equal as members of T, *i.e.,* $t = t'$ *in* T. To be specific, these results will apply to the *type systems* defined in [2].

A basic concept for all that follows is the notion that a type of the language L represents a mathematical class S.

Definition 1 *Given a relation t reps s between terms t and members s of a class of objects S, we say that a type T of the language L* represents S via reps *if*

1. *for any term t and any $s, s' \in S$, if t reps s and t reps s' then s is s'.*
2. *for any $s \in S$ there is a term t such that t reps s.*
3. *for any terms t and t', $t = t'$ in T if and only if there is an $s \in S$ such that t reps s and t' reps s.*

The first two conditions are general requirements for calling something a representation relation. The third is specific to the conventional use of a type as a system of notations for objects of some class. If t is a member of T then $\downarrow(t)$ denotes the member of S that t represents. Note that a given object may have many representatives. In the representation schemes used below we pick out for each s a *standard* representation denoted $\lceil s \rceil$; the function taking s to $\lceil s \rceil$ will be computable.

In the case of the class of terms itself, there is a natural representation relation that can be designed into the language. Let us assume that the language L has a particular term structure. In particular, let the set of terms be the least set containing variables and some set of operator names, and such that if op is an operator name, $t_1, \ldots, t_n$ are terms, $n \geq 0$, and $\overline{v_1}, \ldots, \overline{v_n}$ are lists of variables, then $op(\overline{v}_1.t_1; \ldots; \overline{v}_n.t_n)$ is a term. We say that t_i are the *subterms*; the $\overline{v}_i$ are *binding variables* with *scope* t_i and they bind the variables occurring in t_i. Operator names are included among the terms in anticipation of the need for a term to represent each operator name.

Let us suppose that there are types *Var* and *Op* in L that represent variables and operator names, respectively. Also, suppose that L has type constructors (and associated operations) for cartesian product and disjoint union, and that recursive types can be formed. Then a natural representation of a term $op(\overline{v}_1.t_1; \ldots, \overline{v}_n.t_n)$ is a pair

$$\langle OP, (\langle \overline{V}_1, T_1 \rangle, \ldots, \langle \overline{V}_n, T_n \rangle) \rangle$$

where OP represents op, $\overline{V}_i$ represents $\overline{v}_i$ and T_i represents t_i. Using this representation scheme we can show that the type *Term* defined as

$$rec\,(T.\ Op \,|\, Var \,|\, Op \times ((Var\ list \times T)\ list))$$

represents the set of terms of L.

The proofs we deal with involve *sequents*, which are objects of the form

$$x_1 \colon A_1, \ldots, x_n \colon A_n \ \vdash\ B$$

where each $x_i \colon A_i$ is a *hypothesis* consisting of a variable x_i and a term A_i, and where B is a term called the *conclusion* of the sequent. There is a natural representation of this class based on the representation of terms, namely,

$$Sequent \equiv (Var \times Term)\ list \times Term$$

Reflection of terms and evaluation is studied in more depth in [3].

2.2 Proofs

The design of our class of proofs is intended to exploit their representability in two ways. Tactics are internally represented and a reflection rule is included.

First let us consider prior constraints we have accepted. We have retained from the Nuprl design the general form of proof as a finite tree whose nodes are labelled with a sequent and some discrete justification "text" which is intended to indicate how the inference of the sequent from the sequents labelling the offspring is justified. The sequent at the root is called the *goal* of the tree. The leaves of a proof tree that are unjustified are the *premises* of the proof. Next, we require that the sequent and justification text completely (and effectively) determine the list of immediate *subgoals*, that is, the goals of the immediate subtrees. Finally, we require that proofs be effectively recognizable. However, we do not require that proofhood be effectively decidable; this is a necessary result of allowing arbitrary computable tactics, since one cannot then decide which (untyped) programs will produce proofs.

To simplify our presentation, we shall develop the basic ideas without incorporating extraction of programs from proofs. Also, we shall avoid accommodating libraries at first. Each of these topics will be addressed separately afterwards with explanations of how to extend the method to these more complex situations.

The special forms of rule exploiting proof representation are the tactic rule and the reflection rule. An application of a tactic rule can be written

$$\begin{array}{ll} s & \text{By tactic t} \\ \quad s_1 & \\ \quad \vdots & \\ \quad s_n & \end{array}$$

where s is the goal and the s_i are the subgoals. The justification text here contains a term t which represents a proof with goal s and premises $s_1, \ldots, s_n$. An application of the reflection rule can be written

$$\begin{array}{ll} s & \text{By refl } i{,}t \\ & \vdash \mathit{provable}(\lceil s \rceil, t, \lceil i \rceil) \\ & s_1 \\ & \vdots \\ & s_n \end{array}$$

where t represents the list $s_1, \ldots, s_n$, $\lceil s \rceil$ is the standard representative of sequent s, and $\lceil i \rceil$ is the standard representative of the number i. The number i is called the *level* of the reflection rule. $\mathit{provable}(\lceil s \rceil, t, \lceil i \rceil)$ is a type representing the proposition that there is a proof with goal (root) s, premises $s_1, \ldots, s_n$, and in which every use of reflection is of level less than i.

Another kind of justification is a primitive rule. We do not need to specify the form of justification text for applications of primitive inference rules. If $g, s_1, \ldots, s_n$, $n \geq 0$, are sequents and r is the justification text for an instance of a primitive rule then define $\mathit{instance}(r, g, s_1, \ldots, s_n)$ to be true if g is the conclusion and $s_1, \ldots, s_n$ are the premises of the instance of the rule named by r. Below we will refer to r simply as a primitive rule. More generally, we will not refer to "justification text" and instead only speak of justifications and rules.

Definition 2 *A* justification *is either a primitive rule, a* tactic rule *consisting of a single term, a* reflection rule *consisting of a nonnegative integer and a term, or the* empty justification.

Our first problem is to devise a (non-circular) definition of our self-referential class of proofs and to demonstrate their validity. It is convenient to deal with representation before defining proofs, so we introduce *preproofs*, which are trees having the same components as proof trees but having no restrictions on how the components fit together.

Definition 3 *A* preproof *is a tuple*

$$(s, r, \Delta_1, \ldots, \Delta_n)$$

where s is a sequent, r is a justification, $n \geq 0$ and for each i, $1 \leq i \leq n$, Δ_i is a preproof.

The *root* and *leaves* of a preproof are defined in terms of the obvious interpretation of preproofs as trees. The *premise list* of a preproof is the list of the sequents, in left-to-right order, occurring at leaves that have the empty justification. We will usually refer to the premise list simply as the *premises* of the preproof. The root of a preproof is also called the *goal* of the preproof.

Above we defined types for the terms that represent terms and sequents. We can similarly define types representing justifications and preproofs (we assume that the justification text for primitive rules is representable—this is easy to arrange). Details are given in the appendix.

We now proceed to define proofs. The definition involves a syntactic fixed-point construction, so we need to introduce a parameter through which to take the fixed point. Thus for any term q we will define q-proofs; these will be just like proofs except for the reflection rule. The purpose of the reflection rule is to allow us to make an inference step by proving that there is a proof justifying the step; the sequent which expresses the existence of such a proof must involve a type representing proofs. In a q-proof we use the term q in place of this as-yet unknown type of proofs. We will also need a particular fixed term T in the definition of q-proofs. We will not define T at this point; later we will specify the property required of T for validity of proofs, and make it plausible that T can be constructed (details are given in the appendix). The important point at this stage is that T is simply a particular term; exactly what term we pick is relevant *only* to the validity argument, since it enters in the definition of proof only as essentially a piece of text in a proof tree.

Definition 4 *Let q be any term. We inductively define the set $pf(q)$ of q-*proofs *as a subset of the set of preproofs. A q-proof is a preproof*

$$(s, r, \Delta_1, \ldots, \Delta_n)$$

where s is a sequent, r is a justification and

1. *$\Delta_1, \ldots, \Delta_n$ are q-proofs,*
2. *if r is the empty justification then $n = 0$,*
3. *if r is a primitive rule then*
$$instance(r, s, g_1, \ldots g_n)$$
where $g_1, \ldots, g_n$ are the goals of the preproofs $\Delta_1, \ldots, \Delta_n$,
4. *if r is a tactic rule consisting of a term t then t represents a preproof Δ such that Δ is a q-proof, the goal of Δ is s and the premises of Δ are the goals of $\Delta_1, \ldots, \Delta_n$, and*
5. *if r is a reflection rule consisting of a number i and term t then $n \geq 1$, t represents the sequent list consisting of the goals of $\Delta_2, \ldots, \Delta_n$, and the goal of Δ_1 is the sequent*
$$\vdash\ T(q)(\lceil s \rceil)(t)(\lceil i \rceil).$$

The *immediate subproofs* of a q-proof

$$(s, r, \Delta_1, \ldots, \Delta_n)$$

are $\Delta_1, \ldots, \Delta_n$ and also, in the case that r is a tactic rule consisting of a term t, the preproof represented by t. A q-proof has *level* k if any number appearing in a reflection rule is less than k. More precisely,

$$\Delta = (s, r, \Delta_1, \ldots, \Delta_n)$$

has level k if each immediate subproof of Δ has level k and in the case that r is a reflection rule r's number is less than k.

We now need to argue that most of the above can be internalized in type theory. We have given a definition of a type *Term* which represents our class of terms. Given a term t, we can easily determine the member $\lceil t \rceil$ of *Term* which represents t. It is straightforward to write a function in type theory which reflects this. In particular, we can construct a term *Rep* of type $\mathit{Term} \rightarrow \mathit{Term}$ such that for every member t of *Term*, the term $\mathit{Rep}(t)$ represents $\lceil u \rceil$ where u is the term that t represents.

We also need a type that represents the class of q-proofs. This is straightforward to construct since the forms of inductive definition we have used are directly supported in Nuprl's type theory. In the appendix we define a term *Pf* such that if Q is a term representing a term q then $Pf(Q)$ represents $pf(q)$.

We can now construct our fixed-point. Let d be the term

$$\lambda x.\, Pf(Ap(x, Rep(x)))$$

where Ap is a term which is a type theoretic function such that $Ap(\lceil a \rceil, \lceil b \rceil)$ evaluates to $\lceil a(b) \rceil$. Finally, let p be the term $d(\lceil d \rceil)$.

Definition 5 *The set pf of* proofs *is $pf(p)$.*

Note that $Pf(\lceil p \rceil)$ represents the class of all proofs. If we reduce the term $p \equiv d(\lceil d \rceil)$ we get $Pf(\lceil p \rceil)$, so in Nuprl's type theory p is a type equal to $Pf(\lceil p \rceil)$. Hence p is also a type representing the class of all proofs.

Before we can prove the validity of proofs, we need to assume a certain property of the term T that was used in the definition of q-proofs. Suppose that S represents a sequent s, L represents a list l of sequents, and I represents a number i. We assume that the term $T(p)(S)(L)(I)$ is a type representing the class of all members Δ of *pf* such that Δ has level i, the goal of Δ is s, and the premise list of Δ is l.

T is actually quite straightforward to construct. We want $T(p)(S)(L)(I)$ to be a type consisting of all members of p with level I, goal S and premises L. This requires functions in type theory which compute the premises and goal of a proof, and a predicate for the level of a proof. Although it may appear that we need to know what proofs are in order to define T, functions in Nuprl are untyped and can be defined to work on (representations of) preproofs. For level, the preproof represented by a tactic rule must be computed, and this can be done by a simple recursive algorithm. The level predicate will be partial on preproofs but total on proofs.

A proof is *valid* if the goal is true when the premises are. The proof of validity is mostly abstract with respect to the definition of truth for sequents. We assume, of course, that the primitive rules are valid. The only property of truth we will need is that if the sequent $\vdash A$ is true, then A is a type that has a member.

Theorem 1 *Proofs are valid.*

Proof. We show by induction on n that proofs of level n are valid. For each n we argue by induction on the immediate-subproof relation. Suppose, then, that

$$\Delta = (s, r, \Delta_1, \ldots, \Delta_n)$$

is a proof whose premises are true. Let $\overline{\Delta}$ denote the list $\Delta_1, \ldots, \Delta_n$. By induction the goal of each Δ_i is true. We now consider the cases for r.

If r is the empty justification then s is a premise hence true.

If r is a primitive rule then s is true by our assumption of the validity of primitive rules and the fact that $instance(r, s, g_1, \ldots, g_n)$ where g_i is the goal of Δ_i.

If r is a tactic rule with term t then t represents a subproof Δ' of Δ, so by induction Δ' is valid. The premises of Δ' are the goals of $\overline{\Delta}$, so they are true, hence the goal of Δ' is true, and this is s.

If r is a reflection rule with integer i and term t, then the goal of Δ_1

$$\vdash \ T(p, \lceil s \rceil, t, \lceil i \rceil)$$

is true, so the term in this sequent is a type that has a member, call it t. From our assumption about T it follows that t represents a proof Δ' of level less than i. By the induction hypothesis (of the induction on level) Δ' is valid. The premises of Δ' are the goals of $\Delta_2, \ldots, \Delta_n$ (by the assumption about T) and so are true. The goal of Δ' must then be true, and this is s. $\Box$

Tactic rules and reflection rules are eliminable from complete proofs. A *primitive* proof of s is a proof with goal s where every node is either a premise or is justified by a primitive rule. A *complete* proof is one with no premises. By an inductive proof of the same structure as the one just given for validity of proofs, we can show the following

Theorem 2 *If s has a complete proof then it has a complete primitive proof.*

In fact, given the extraction property to be discussed in the next subsection, there is a simple procedure for computing the primitive proof.

2.3 Extraction

For Nuprl, we are not satisfied with the validity of proofs. We want to be able to *extract* programs from proofs. For example, if we have proven a sequent of the form

$$\vdash \forall x.\, \exists y.\, R(x, y)$$

then we expect to be able to derive from the proof a program which when given an x will produce a y such that $R(x, y)$ is true.

We need to refine our notion of truth of sequents somewhat. For the discussion here, say that t *satisfies* a sequent

$$x_1{:}\, A_1, \ldots, x_n{:}\, A_n \vdash B$$

if the free variables of t are among $x_1, \ldots, x_n$ and if for all members $x_1, \ldots, x_n$ of $A_1, \ldots, A_n$ respectively, t is a member of B.[2]

We require an automatic procedure for producing a satisfying term from a complete proof; we call this an *extraction* procedure. In analogy with our definition of validity for proofs, we say a partial term-valued function F of proofs and term lists is valid if for any term list $b_1, \ldots, b_m$ and any proof Δ with $n \leq m$ premises, if $b_1, \ldots, b_n$ satisfy the premises of Δ then $F(\Delta, b_1, \ldots, b_m)$ satisfies the goal of Δ. We allow the list $b_1, \ldots, b_m$ to be longer than need be in order to make the definition of extraction given below easier.

As an implicit parameter for constructing our extraction procedure we assume a procedure $\mathit{prextr}(r, s, b_1, \ldots, b_m)$ which is used to extract from proofs justified by a primitive rule.

We can now recursively define extraction as a partial function *extr*. To define

$$b = \mathit{extr}((s, r, \Delta_1, \ldots, \Delta_n), b_1, \ldots, b_m),$$

let $g(i)$ be one plus the sum of the lengths of the premise lists of $\Delta_1, \ldots, \Delta_i$ and let h_i be $\mathit{extr}(\Delta_i, b_{g(i-1)}, \ldots, b_m)$. If r is the empty justification then $b = b_1$. If r is a primitive rule then $b = \mathit{prextr}(r, s, h_1, \ldots, h_n)$. If r is a reflection rule then

$$b = \mathit{extr}(\downarrow(\mathit{fst}(\mathit{extr}(h_1))), h_2, \ldots, h_n)$$

where $\mathit{fst}(t)$ is the first component of the pair term that t evaluates to. Finally, if r is a tactic rule consisting of a term t then

$$b = \mathit{extr}(\downarrow(t), h_1, \ldots, h_n).$$

Now, recall that the reflection subgoal is designed to represent the existence of a proof of a low enough reflection level and with a certain goal and premises. The obvious representation is to make this subgoal a sigma type over (or a subtype of) the type representing the proofs. Then the satisfying terms of such a goal are pairs whose first components represent proofs of appropriate reflection level and with the specified goal and premises. With this form for the reflection subgoal, and given that *prextr* is valid (*i.e.*, $\mathit{prextr}(r, s, b_1, \ldots, b_n)$ returns a term satisfying s if $b_1, \ldots, b_n$ satisfy $s_1, \ldots, s_n$ and $\mathit{instance}(r, s, s_1, \ldots, s_n)$), then by a proof analogous to that of Theorem 1, one can show that *extr* is valid.

It is a corollary that *extr* is total on complete proofs. Moreover, if, as is easy to arrange, one can represent the procedure *prextr*, then one can represent *extr* with a term E in the type of functions from proofs with no premises to *Term*. But there is no hope of proving internally that E has this type since it depends on the semantic validity of extracting from reflection subgoals, *i.e.*, the fact that the reflection subgoal *means* there is a proof meeting certain conditions is used in this argument. It is possible, but we are not sure, that we will want to reflect the fact that extraction is total, which means we would have to abandon the elegant procedure just given.

[2] This is only an approximation of Nuprl's satisfaction relation. For more details, see [1].

Suppose that *prextr* were designed to be total, but do not suppose all the primitive rules to be valid. Then, given a proof and a term list of the same length as the premise list of the proof, the only way that *extr* could fail to have a value would be for some term extracted from a reflection subgoal somewhere in the proof to not have a first component representing a preproof (and recall that it is not decidable whether a term represents anything). So, if we could avoid having to extract this preproof representative, we could guarantee totality and even type the extractor internally.

The direct solution would be to alter the reflection rule to include a specification of the extract function to be used. Thus a reflection rule would now consist of a triple (i, l, f) where i is number, l a term, and f is of a class not yet determined. The reflection subgoal would be strengthened to represent not only the condition that some proof Δ has the right goal and premises, and the right reflection level, but also that f specifies the extract function for Δ. These alterations would allow us to define an extract function for arbitrary proofs and the totality of this function would be independent of validity and the semantics of sequents; the semantics for sequents only enters into the proof that this total procedure is valid. Thus, we could type this extraction function internally.

Now, what might specifications for extraction look like if we use this method for making extraction total on proofs (not merely complete ones)? Since we want proofs to be effectively recognizable, we cannot permit the specification of arbitrary total procedures for extraction since one cannot index the class of all total procedures by an effectively recognizable index set. This is the limitation of requiring extraction to be total on all proofs. In practice this may be tolerable (in fact we expect to be satisfied with using second-order substitution templates as extraction specifications). Using this approach, the form of extraction on proof Δ with term list l is

$$\mathit{extrstep}\,(\mathit{extrix}\,(\Delta), l)$$

where $\mathit{extrix}(\Delta)$ produces a specification for an extraction step and $\mathit{extrstep}\,(f, l)$ applies it to term list l. To define extraction this way we need only be able to compose specifications.

2.4 Libraries

In practice, when using Nuprl, we will produce proofs in the context of a library of definitions and previously proved theorems. A library may contain definitions of new operators which explain how instances of an operator are to be expanded. The library is constrained so that new operators must fully expand to terms built only of original operators, and such that full expansion will be unique modulo change of bound variables. (There are also restrictions on expansion which allow the operators to serve as abstractions for their expansions, for example variables free in the expansion are also free in the unexpanded term, and substitution commutes with full expansion.) Thus full expansion must be factored into representation relations and into the truth (and satisfiability) of sequents. A sequent is *true under a library* L if its full expansion (that is, the sequent obtained by fully expanding under L its hypotheses and consequent) is true. Representation relations must also be parameterized by libraries; a term represents a certain object under L if its full expansion under L represents it.

In order to simplify this discussion of library use, we will ignore extraction. Since representation is used in the concept of proofs, we must parameterize our concept of proof by libraries. And since in practice we intend to continually extend the libraries in use, this library parameter will be permanent and explicit, and $\mathit{pf}(L)$, the set of proofs over L, should be monotonic in L. The library also enters into the primitive rules; two primitive rules we particularly intend to include which are "about" the library are an expansion rule, for inferring one sequent from another when they both fully expand to the same sequent, and a lemma rule which allows one to infer $H \vdash A$ when there is a proof of $\vdash A$ in the library. Since we intend to use the notion of proof with many different libraries, we must reflect $\mathit{pf}(L)$ with its library parameter, hence we must also reflect libraries; below we shall address the danger of the infeasibility of reflecting libraries in practice, which could require huge terms.

Parameterizing proofs by libraries requires more than simply parameterizing representation and *instance*. The reflection subgoal of a proof, since it is supposed to represent the proposition that there is a proof meeting certain conditions, must now be parameterized by a library representation; to insure the validity of the reflection rule, we naturally want the represented library to be a sublibrary of L. But where do we get the library representation to plug into the reflection subgoal? It must be derived from the justification, and since proofs are supposed to be effectively recognizable, we must be able to recognize when the justification of the reflection rule determines (a representation of) a sublibrary of L. This will be discussed further.

The incorporation of libraries complicates the notion of proof in the ways indicated above, but it can be used to avoid the fixed point construction used earlier. Instead of using the parameter called q above, we simply choose a fixed term PF using a new nonprimitive operator with no subterms. Then we can find a term p representing the family pf (so that if Γ represents L then the term $p(\Gamma)$ represents $pf(L)$) without taking a fixed point. Notice that representing pf does not depend on what PF represents; it is just validity that depends on PF representing pf. It will then turn out that proofs are valid under L if *instance* (L) is valid under L and PF represents pf under L. This is easily achieved by making PF expand under L to p. In practice, one would establish a standard prelude including this definition of PF as well as any other operator definitions one might want in order to make the design of p itself easier.

Now we consider library representation, but we shall be rather abstract about the library structure. We assume, as is quite plausible, that some representation relation can be found for libraries that consist of some easily indexed collection of operator definitions and preproofs. We also assume some standard-representation function for libraries. The standard representations of libraries would be intolerably large for use in actual proofs, and normally the libraries we represent in actual proofs are sublibraries of the actual library in use when those proofs were developed. To avoid ever having to produce the gigantic representations of libraries, we will extend our notion of full expansion under a library to include special operators which expand to library representations. This expansion is for semantic use only and is not to be implemented, for example, in the proposed expansion rule of inference mentioned above.

One might at first consider a single special nonprimitive LIB which under a library L expands to the standard representation of L. But then, the truth of sequents would not be preserved under library extension as we demand. For example,

$$\vdash\ LIB = t \ in\ Library,$$

where t has only primitive operators, would be true under the library represented by t but not under proper extensions of that library. Our solution is to represent only segments of the library, where segments are indexed by some discrete class of "library limiters". Let α and β range over library limiters. We require that there be an effectively recognizable relation $\alpha \leq \beta$, a limiter-valued function $|L|$ of libraries, and a "slice" operation $L[\alpha]$ on libraries, such that:

- $|L[\alpha]| = \alpha$ if $\alpha \leq |L|$, and
- L' is an extension of L if and only if $|L| \leq |L'|$ and $L'[|L|] = L$.

A plausible, but practically too simple, solution is to:

- let library items be indexed by numbers less than the number of items in the library,
- let limiters be numbers,
- let $|L|$ be the number of items in the library,
- let $L[n]$ be the initial segment of L of length n, and
- let $\leq$ be the obvious relation.

Now, let LIB_α be a family of nullary nonprimitive operators. The limiters are parameters to this family of operators the way that numbers are the parameters to the family of universes U_i. Define full expansion under a library L to expand LIB_α, for $\alpha \leq |L|$, to the standard representation of $L[\alpha]$, and to expand other operators in accordance with the operator definitions contained in L. Expansion under L is monotonic in L.

We are now in a position to avoid actually using gigantic terms to represent libraries, and $pf(L)$ can be made monotonic in L. (In order to make the *LIB* representations usable, we include primitive rules for accessing individual items.) Let us look at $pf(L)$. A member of $pf(L)$ is a preproof

$$(s, r, \Delta_1, \ldots, \Delta_n)$$

where

1. $\Delta_1, \ldots, \Delta_n \in pf(L)$,
2. if r is the empty justification then $n = 0$,
3. if r is a primitive rule then
$$instance\,(L, r, s, s_1, \ldots s_n)$$
where $s_1, \ldots, s_n$ are the goals of the preproofs $\Delta_1, \ldots, \Delta_n$,
4. if r is a tactic rule consisting of a term t then t represents under L a preproof Δ such that $\Delta \in pf(L)$, the goal of Δ is s and the premises of Δ are the goals of $\Delta_1, \ldots, \Delta_n$, and
5. if r is a reflection rule consisting of a number i, term t and limit α then $n \geq 1$, t represents under L the sequent list consisting of the goals of $\Delta_2, \ldots, \Delta_n$, the goal of Δ_1 is the sequent
$$\vdash\ T(LIB_\alpha)(PF)(\lceil s \rceil)(t)(\lceil i \rceil),$$
and $\alpha \leq |L|$.

Note that *instance* and the representation relations have been parameterized by L and that T must be redesigned to make use of the extra argument. The standard representation functions need not be parameterized since we can make them return terms having only primitive operators. The term T should now be designed so that it satisfies the following property. Let L be a library. Suppose that Γ' represents (under L) some sublibrary L' of L, U represents a list u of sequents, I represents a number i, and p represents pf (that is, for any library L, if Γ represents L then the term $p(\Gamma)$ represents $pf(L)$). Then the term $T(\Gamma')(p)(S)(U)(I)$ represents the class of all members Δ of $pf(L')$ such that Δ has level i, the goal of Δ is s, and the premise list of Δ is u.

If $instance\,(L)$ is monotonic in its library argument, then so is $pf(L)$. Clearly, as was indicated above, we can find a term p which represents the family of classes pf, and start off our library with a definition of the fixed nonprimitive term PF that makes it expand to p, and therefore represent pf. Then, for any library L under which PF represents pf and T satisfies the property given above, and under which the primitive rules are valid, one can show by induction on reflection level, then on proof structure, that any member of $pf(L)$ is valid under L. The interesting case is for a proof by reflection. The reflection subgoal is of form

$$\vdash\ T(LIB_\alpha)(PF)(\lceil s \rceil)(t)(\lceil i \rceil).$$

If this is a true sequent than there is a member of $pf(L[\alpha])$, hence a member of $pf(L)$, satisfying the appropriate conditions. This allows the completion of the inductive proof as was done earlier.

2.5 Negative Results

Theorem 2 says that tactic rules and reflection rules can be eliminated from (complete) proofs. This fact can be represented internally, but we have no hope of proving it internally since the case for a proof by reflection depends on semantics[3].

Actually, it is possible to employ Löb's theorem [6] somewhat indirectly to show the unprovability of this and a number of other interesting truths as well as the underivability of some valid rules. However, showing that the addition of certain valid rules to the proof system makes Löb's theorem apply is not elementary, so we shall not do it here. The difficult part is showing that there is an internal proof of

$$\vdash \; Pvbl(\lceil s \rceil) \rightarrow Pvbl(\lceil Pvbl(\lceil s \rceil) \rceil)$$

for all s, where $Pvbl(x)$ is a predicate asserting the existence of a complete proof with goal (represented by) x. (The conclusion of Löb's theorem in this setting would be that $\vdash \; A$ whenever $\vdash \; Pvbl(\lceil \vdash \; A \rceil) \rightarrow A$.) Still, the possibility of extending the proof system to make this true entails

- the internal unprovability of the primitive provability of all provable sequents,
- the underivability of the level k reflection rule via a level k proof, and
- the necessity of stratifying the reflection rule by levels.

3 Applications

3.1 Theorem Proving

In systems like Nuprl [10, 4, 20], tactics are used to build significant parts of a proof tree. Sometimes these tactics are costly to run because they build a large piece of data; this often happens in circumstances in which it is easy to discover beforehand whether or not the tactic will succeed. For example, there is a type checking tactic which will prove all sequents of the form $H \; \vdash \; T \; \mathit{in} \; \mathit{Type}$ as long as the type T is of restricted form, such as when it corresponds to a formula of first-order logic. In the reflected system we can express this as a metatheorem like the following.

Theorem 1

$$\begin{aligned} &\forall\, s\!: \mathit{Sequent}.\ \mathit{first\text{-}order}(s) \Rightarrow \mathit{wf\text{-}goal}(s) \\ &\qquad \Rightarrow \exists\, p\!: \mathit{Proof}.\ \mathit{goal}(p) = s \;\&\; \mathit{complete}(p) \end{aligned}$$

Now in the course of any argument in which the well-formedness goal $H \; \vdash \; T \; \mathit{in} \; \mathit{Type}$ appears, we can prove it by using the reflection rule. The cost of checking that a type is first order and recognizing a well-formedness goal will often be much simpler than proving the theorem. The reflection mechanism will enable users of systems like Nuprl to provide efficient type checking for those subclasses of terms for which a good algorithm is discovered, *e.g.*, for ordinary logic.

3.2 Evaluation

Evaluation of programs, as defined in the semantics of Nuprl, is (roughly) call-by-name reduction. However, when we know that all the functions used in a term are strict, we can instead call functions by value, and thus evaluate the term more efficiently. There are syntactic subclasses of the terms which have members that use only strict functions—for example, any class of terms that is strongly normalizing.

[3]One might want to take Theorem 2 as a reducibility axiom for proofs. But in fact adding it as an axiom makes it false!

In a system in which term structure and evaluation have been reflected, it is possible to write alternate evaluators and to prove them correct. An alternate evaluator *Alt* is written as a partial function from *Term* to *Term*. To show that it is correct one must prove that for any term t satisfying a predicate Φ, $Val(t)$ is equivalent to $Alt(t)$ in some equivalence relation R. The definitions of Φ and R vary somewhat between applications. We shall make some general comments about them and then sketch a simple example. Φ must imply that $Val(t)$ and $Alt(t)$ exist. Frequently it will place further restrictions on t to ensure that the alternate evaluator behaves correctly; in particular, it will often restrict t to a subtype of *Term* containing only representations of terms built using a limited, enumerated set of operators (Nuprl's class of operators is open-ended). R can be equality in the type *Term*, although usually, because Nuprl's evaluator reduces only until the outermost operator is canonical, this will be too fine a relation. Typically, instead, some approximation to computational equality will be used [21].

Consider a simply typed programming language in which the terms are built from integer constants, variables and operators for λ-abstraction, application, pairing and projection. Type inference is decidable for this language and all terms are strongly normalizing. Furthermore, the terms of this language are a subclass of Nuprl's terms and so are represented by a subtype of *Term*. Let $IL(t)$ be a predicate on terms that is satisfied if and only if t represents a term in this language. Let $TI(t)$ be the reflection of type inference procedure for this language. Both of these are easily defined by structural induction on terms. Let *Int* be the value of $TI(t)$ when t represents a term of integer type. Let $CBVE(t)$ be a call–by–value evaluator for this language. This too is easily definable by structural induction on terms. Then, the following theorem can be stated and proven:

$$\begin{aligned}\forall t\colon \mathit{Term}.\ IL(t) \wedge (TI(t) = \mathit{Int} \in \mathit{Type}) \Rightarrow \\ \mathit{Val}(t) = \mathit{CBVE}(t) \in \mathit{Term}.\end{aligned}$$

Removing the condition that t represents a term of integer type requires that the equivalence relation used be coarsened since neither evaluator necessarily fully normalizes t.

References

[1] S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[2] S. F. Allen. A non-type-theoretic definition of Martin-Löf's types. *Proc. of Second Symp. on Logics in Computer Science, IEEE*, pages 215–224., June 1987.

[3] S. F. Allen, R. L. Constable, and D. J. Howe. Reflecting the open-ended computation system of constructive type theory. In *Working Material, International Summer School on Logic, Algebra and Computation*, Marktoberdorf, West Germany, 1989.

[4] D. Basin. *Building Problem Solving Environments in Constructive Type Theory*. PhD thesis, Cornell University, 1990.

[5] J. Batali. Computational introspection. Technical report, MIT, 1983. AIM-TR-701.

[6] G. Boolos and R. Jeffery. *Computability and Logic*, volume Third edition. Cambridge University Press, 1988.

[7] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science.*, pages 103–84. NY:Academic Press, 1981.

[8] A. Bundy. A broader interpretation of logic in logic programming. In *Proc. 5th Sympo. on Logic Programming*, 1988.

[9] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In *International Joint Conference on Artificial Intelligence*, pages 359–365, Detroit, Michigan, 1989.

[10] R. L. Constable and et al. *Implementing Mathematics with the Nuprl Development System.* NJ:Prentice-Hall, 1986.

[11] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120., 1988.

[12] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. with Applications*, 5:217–230, 1979.

[13] S. Feferman. Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis. In *Proc. Conf. Intuitionism and Proof Theory*, pages 303–326, Buffalo, NY, 1970. North-Holland.

[14] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1988.

[15] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In *Meta-Programming in Logic Programming*, chapter 6, pages 123–140. MIT Press, 1989.

[16] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical report, Cambridge University, 1985. TR 68.

[17] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[18] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University, 1988.

[19] S. Hayashi and H. Nakano. *PX: A Computational Logic.* Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[20] D. J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[21] D. J. Howe. Computational metatheory in Nuprl. *CADE-9*, pages 238–257, May 1988.

[22] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Annual Symp. on Logic in Computer Science.* IEEE., 1986.

[23] J. McCarthy. A basis for a mathematical theory of computation. In *Comput. Program. and Formal Syst.*, pages 33–70. Amsterdam:North-Holland, 1963.

[24] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. on Logic Programming.* ACM, 1987.

[25] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. *Proc. Fourth Symp. Logic in Computer Science, IEEE*, pages 198–203, June 1989.

[26] B. C. Smith. Reflection and semantics in lisp. *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[27] A. S. Troelstra. Metamathematical investigation of intuitionistic mathematics. *Lecture Notes in Mathematics*, 344, 1973.

[28] R. Weyrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170., 1980.

Appendix

In the following formalization of the reflection of proofs, we use the notation $\overline{x_{i\ldots j}}$ to denote a list containing the elements $x_i,\ x_{i+1},\ \ldots,\ x_j$. The *case* expression is used as destructor for the *Justification* type. It has the form *case* j *of* c where c is a sequence of clauses of the form $t_i x_i \rightarrow r_i$. If j evaluates to an nth injection of v, then the value of the expression is the value of r_n with x_n bound to v. Note that t_n is merely a mnemonic tag. The version of the *rec* type constructor used in the definition of *IsAPf*, $rec\big(z, x.\ t; a\big)$, is used to define an instance of a parameterized family of recursive types. t is a term that defines the type. In it z stands for the family of types being defined, and x for the parameter. a is the actual parameter. As a simple example of the form's use, the proposition that a function f over the natural numbers eventually takes on the value 0 can be represented by the type $rec\big(P, x.\, f(x) = 0 \vee P(x+1); 0\big)$.

The only function we will define on preproofs is *level*; *premises* and *root* are similar inductions over *PPf* (the type representing preproofs). The term T used in Section 2.2 is $\lambda qsXi.\, Pvbl(q; i; s; X)$. The *recind* form used below is a primitive form for constructing recursive functions over members of a recursive type.

$$Term \equiv rec\,(t.\ Op \,\big|\, Var \,\big|\, Op \times ((Var\ list \times t)\ list))$$

$$Sequent \equiv Term\ list \times Term$$

$$Justification \equiv \{1\} \,\big|\, Prim \,\big|\, int \times Term \,\big|\, Term$$

$$PPf \equiv rec\,(s.\ Sequent \times s\ list \times Justification)$$

$$\begin{array}{l}
level(p; l) \equiv recind\big(p; \langle s, j, \overline{p_{1\ldots n}}\rangle, h.\ \bigwedge_{i=1}^{n} h(p_i) \wedge \\
\quad case\ j\ of \\
\qquad PREMISE\ x \rightarrow true \\
\qquad PRIM\ x \rightarrow true \\
\qquad REFL\ \langle n, t\rangle \rightarrow n < l \\
\qquad TACTIC\ t \rightarrow \exists p{:}\, PPf.\ t\ RepsPPf\ p \wedge h(p)\big)
\end{array}$$

$$\begin{array}{l}
Pvbl(Q; i; s; X) \equiv \exists p{:}\, Q. \\
\quad level(p; i) \wedge \\
\quad root(p) = s \in Sequent \wedge \\
\quad premises(p) = X \in Sequent\ list
\end{array}$$

$$T \equiv \lambda qsXi.\, Pvbl\,(q; i; s; X)$$

$$roots(x) \equiv map\ root\ x$$

$$\begin{array}{l}
IsAPf(p; Q) \equiv recind\big(p; \langle s, j, \overline{p_{1\ldots n}}\rangle, z.\bigwedge_{i=1}^{n} z(p_i) \wedge \\
\quad case\ j\ of \\
\qquad PREMISE\ x \rightarrow n = 0 \in int \\
\qquad PRIM\ r \rightarrow instance\,(r; s; roots\,(\overline{p_{1\ldots n}})) \\
\qquad REFL\ \langle l, t\rangle \rightarrow \\
\qquad\quad n > 0 \wedge \\
\qquad\quad t\ RepsSequentList\ roots(\overline{p_{2\ldots n}}) \wedge \\
\qquad\quad p_1 = \langle nil, Ap(\lceil T \rceil, Q, Repsequent(s), t, repint(l))\rangle \\
\qquad\qquad\quad \in Sequent \\
\qquad TACTIC\ t \rightarrow \exists x{:}\, PPf. \\
\qquad\quad root(x) = s \in Sequent \wedge \\
\qquad\quad premises(x) = roots(\overline{p_{1\ldots n}}) \in Sequent\ list \\
\qquad\quad t\ RepsPPf\ x \wedge z(x)\big)
\end{array}$$

$$Pf(Q) \equiv \{\, p{:}\, PPf \mid IsAPf(p; Q)\,\}$$