

Using Reflection to Explain and Enhance Type Theory*

Robert L. Constable
Cornell University
Department of Computer Science
Ithaca, NY

January 20, 1994

Abstract

The five lectures at Marktoberdorf on which these notes are based were about the architecture of *problem solving environments* which use theorem provers. Experience with these systems over the past two decades has shown that the prover must be *extensible*, yet it must be kept *safe*. We examine a way to safely add new decision procedures to the Nuprl prover. It relies on a reflection mechanism and is applicable to any tactic-oriented prover with sufficient reflection. The lectures explain reflection in the setting of constructive type theory, the core logic of Nuprl.

Keywords: decision procedures, HOL, Löb's theorem, Nuprl, problem-solving environments, reflection, tactics, tautologies, theorem provers, type theory

*Work supported in part by NSF grant CCR-9244739 and ONR contract N00014-92-J-1764.

1 Introduction

1.1 Computation in Mathematics

The last two decades appear to be part of a new Golden Age of mathematics. We have seen the Four Color Problem fall, the Mordell Conjecture as well, and now this year Wiles has announced a proof of Fermat's Last Theorem, the most famous problem of them all. Soon the P=NP problem may be the most well-known open problem in all of mathematics, and like some among the old Hilbert problems, it deals with computation in a fundamental way.

As we reflect on these connections to computation, we might wonder whether we are in fact entering into a New Age of mathematics in which the computer plays a fundamental and striking role. In the solution of the Four Color Problem, the computer was an essential *tool* as it has been in numerical mathematics since mid century. For a variety of reasons, computers are becoming indispensable in many branches of mathematics.

These lectures will be about the theory and application of a kind of computer system, one that surely is a tool, but one whose designers and intimate users expect to become more, to become a *partner* in some sense. This is because these systems, which we call *proof development systems* or *problem solving environments* combine many tools into a single environment that provides coordinated and integrated assistance with a variety of tasks: formula and text entry and editing, numerical computation, symbolic computation, theorem and definition storage and retrieval from a data base, hypertext reading and browsing facilities for the data base, and proof checking and proof development assistance. There will also soon be communication facilities for monitoring activity at other development centers with their own systems and at repository centers for results in various branches of mathematics.

The combination of these tools, especially the proof checkers and theorem provers accessing the large data bases will create the feeling of a system that is an assistant in the enterprise of communicating mathematics, and in some cases even in the exciting part, discovering new results.

1.2 Technical Issues

The technical problem we solve in these lectures is related to questions that arise in the design and deployment of *problem solving environments* (PSE).

Since these systems interact with users at the limits of what they understand, there is a need to extend the problem solving mechanisms over time; new knowledge is added (say definitions and theorems) as well as new techniques (say new symbolic algorithms for simplifying algebraic expressions). We call these *open systems*.

In the case of theorem provers, but to some extent in all PSE's, extensions of the system require great care lest their reliability suffer. The earliest provers did not allow any user extension of their *methods*. Users could add only new knowledge. The Edinburgh LCF system pioneered the idea of tactics which allow new methods as well.

The LCF tactic mechanism enabled users to securely add new proof methods. The capabilities of the prover could be expanded by users and was not the exclusive province of the designers (“power to the people” is a slogan of this community). Many other provers have now adopted this approach [13, 15, 24, 41].

Experience from the 1970s and earlier showed also that decision procedures are very effective in building proofs. The Stanford Pascal Verifier [30], PL/CV [11], and EHD [44] among others used such procedures. In particular the PL/CV-arith [11], congruence closure [32], sup-inf [46] are from this period and all are actively used today. New decision procedures are always being studied, monotone closure [36], real-closed fields [33], etc.

Here are typical uses of `arith`.

$$x < x^2 \ \& \ x \neq 0 \Rightarrow 2 \leq x \vee x < 0 \text{ by } \mathbf{arith}$$

$$\text{or } (x - y \leq y \ \& \ y \leq x + 1 \ \& \ x - 1 \leq z \ \& \ z \leq x + 1 \ \& \ y - 1 \leq z \ \& \ z \leq y + 1) \Rightarrow \\ (x = y) \vee (y = z) \vee (z = x) \text{ by } \mathbf{arith}$$

Nuprl was the first tactic-oriented prover to incorporate decision procedures. Some people have worried that this will compromise the integrity of the system, because even though the `arith` and congruence closure algorithms were informally proved correct, adding these mechanisms is *dangerous*. They substantially increase the complexity of the prover and raise the burden of proof that the system is correct. This danger makes us very reluctant to add more procedures in this *raw* way,

Our experience has thus created a *design dilemma*: **How to securely add new decision procedures?** One approach is taken by HOL. Decision procedures are used freely when the system executes in *unsafe mode*, then those steps previously solved by the procedure are later reduced to primitive rules when the whole proof is done in *safe mode*. This is a secure method. But it does not represent what people do when they rely on such procedures, and it is costly in terms of machine resources. For Nuprl this approach would be especially costly since that system saves proof trees for user inspection and for control of the proof process.

Another approach would be to verify the decision procedure inside the system. The easiest way is to verify a purely “mathematical” algorithm and then recode it. But this violates the maxim that one should “execute exactly what was verified”. One could attempt to verify the algorithm exactly as coded in the implementation language. This approach is being

followed by E. Gunter for HOL90 where the implementation language is SML-NJ. This is a very large task which resembles the reflection work in Nuprl [3, 4, 6] on which this paper is based, but one must rely on the stability of a large programming language and must define a *substantially* more complex system.

1.3 Summary of a solution

A critical step in solving the design problem is to observe that for type theories with an identifiable computation system, as with *constructive type theories*, it is possible to write tactics and decision procedures directly in the logic itself. This is true for Nuprl whose computation system is a functional programming language in the spirit of ML [25], which is the language of choice for writing tactics. In this case the decision procedure is both proven correct and directly executable.

A critical part of writing tactics inside a type theory is to have a model of the logic inside it. Again this can be done in rich type theories and has been done for Nuprl.

Even with these two pieces in place, we still do not have a solution. At this stage one could write and verify decision procedures in the logic, and one may apply them to same logic. For example, in Nuprl we can define the syntax and proof rules of the Intuitionistic Propositional Calculus (IPC) abstractly. We can prove the decidability of provability of IPC in Nuprl and instantiate the result for the class of propositional Nuprl formulas, which form an instance of IPC. But *still* these decision procedures are beyond our reach; we need a way to *link* this result to the Nuprl logic itself.

The final step of the solution I am describing here [5] relies on the Nuprl *reflection rule*, a unique feature of the system [3, 4, 6, 31]. This rule says that to prove a goal G under hypothesis H , it suffices to show that the reflected sequent, $\ulcorner H \vdash G \urcorner$, is provable. That is,

$$H \vdash G \text{ if } \exists p : \text{Proof. } p \text{ proves } \ulcorner H \vdash G \urcorner.$$

Now we can apply the verified decision procedure to $H \vdash G$ as we show in section 5. This rule requires restrictions to avoid contradictions from Löb's theorem (see section 4.7).

1.4 Plan of the Paper

We will begin with an informal introduction to Type Theory. This is the theory underlying the logic we use to formalize decision procedures. We then look at a formalization of this theory, then at its reflection and finally at applications.

2 An Introduction to Naive Type Theory

The informal language of mathematics uses types and sets, but when mathematicians want to be rigorous about a concept, they tend to rely on the 100 year old tradition of reducing it to concepts in pure set theory. In these lectures, we will find the rigorous concepts in type theory. For pedagogical reasons, we will often mention the set theory version of a concept as well.

The idea of a type is built up inductively (as is the Zermelo hierarchical concept of set). We start with *primitive types* and *type constructors*.

2.1 Primitive Types

void is a type with no elements

N is the type of natural numbers; $0, 1, 2, \dots$

There will be other primitive types introduced later. Notice, in set theory we usually have only two primitive sets, \varnothing , the empty set, and some infinite set (usually ω).

2.2 Compound Types

We build new types using *type constructors*. These tell us how to construct various kinds of objects. (In set theory, there is only one kind, *sets*).

The type constructors we choose are motivated both by mathematical and computational considerations. So we will see a tight relationship to the notion of *type* in programming languages. The notes by C.A.R. Hoare, *Notes on Data Structuring* [27], make the point well.

2.2.1 Cartesian products

If A and B are types, then so is their product, written $A \times B$. There will be many *formation rules* of this form, so we adopt a simple convention for stating them. We write

$$\frac{A \text{ is a Type, } B \text{ is a Type}}{A \times B \text{ is a Type.}}$$

The elements of a product are pairs, $\langle a, b \rangle$. Specifically if a belongs to A and b belongs to B , then $\langle a, b \rangle$ belongs to $A \times B$. We abbreviate this by writing

$$\frac{a \in A, b \in B}{\langle a, b \rangle \in A \times B.}$$

In programming languages these types are generalized to n -ary products, say $A_1 \times A_2 \times \dots \times A_n$ and called *structures* or *records*.

We say that $\langle a, b \rangle = \langle c, d \rangle$ in $A \times B$ iff $a = c$ in A and $b = d$ in B .

In set theory, equality is uniform, but in type theory we define equality with each constructor.

There is essentially only one way to decompose pairs. We say things like, “take the first element of the pair P ,” symbolically we might say $first(P)$ or $1of(P)$. We can also “take the second element of P ,” $second(P)$ or $2of(P)$.

2.2.2 Function Space

We use the words “function space” as well as “function type” for historical reasons. If A and B are types, then $A \rightarrow B$ is the type of *computable functions* from A to B . These are given by rules which are defined on each a in A and which produce a *unique value*. We summarize by

$$\frac{A \text{ is a Type, } B \text{ is a Type}}{A \rightarrow B \text{ is a Type}}$$

The notation we use informally comes from mathematics texts, e.g. Bourbaki’s *Algebra*. We write expressions like $x \mapsto b$ or $x \xrightarrow{f} b$, the later gives a name to the function. For example, $x \mapsto x^2$ is the squaring function on numbers.

If b computes to an element of B when x has value a in A for each a , then we say

$$(x \mapsto b) \in A \rightarrow B.$$

If f, g are functions, we define their equality as

$$f = g \quad \text{iff} \quad f(x) = g(x) \quad \text{for all } x \text{ in } A.$$

If f is a function from A to B and $a \in A$, we write $f(a)$ for the value of the function.

2.2.3 Disjoint Unions (also called Discriminated Unions)

Forming the union of two sets, say $x \cup y$, is a basic operation in set theory. It is basic in type theory as well, *but* for computational purposes, we want to discriminate based on which

type an element is in. To accomplish this we put tags on the elements to keep them disjoint. Here we use tag_l and tag_r as the tags, but anything will do.

$$\frac{A \text{ is a Type, } B \text{ is a Type}}{A + B \text{ is a Type}}$$

The membership rules are

$$\frac{a \in A}{tag_l(a) \in A + B} \quad \frac{b \in B}{tag_r(b) \in A + B}$$

We say that $tag_l(a) = tag_l(a')$ iff $a = a'$ and likewise for $tag_r(b)$.

We can now use a case statement to detect the tags and use expressions like

$$\begin{array}{ll} \text{if } x = tag_l(z) & \text{then } \dots \text{ some expression in } z \dots \\ \text{if } x = tag_r(z) & \text{then } \dots \text{ some expression in } z \dots \end{array}$$

in defining other objects. The test for $tag_l(z)$ or $tag_r(z)$ is computable.

2.2.4 Inductive Types

Defining types in terms of themselves is quite common in programming, often pointers are used in the definition (in Pascal and Ada) but in languages like *ML*, direct recursive definitions are possible. For example, a list of numbers, L can be introduced by a definition like

$$\text{define type } L = N + (N \times L).$$

In due course, we will give conditions telling when such definitions are sensible, but for now let us understand how elements of such a type are created. Basically a type of this kind will be sensible just when we understand exactly what elements belong to the type.

Clearly elements like $tag_l(0), tag_l(1), \dots$, are elements. Given them it is clear that

$$tag_r(\langle 0, tag_l(0) \rangle), tag_r(\langle 1, tag_l(0) \rangle)$$

and generally

$$tag_r(\langle n, tag_l(m) \rangle)$$

are elements. But given these, we can also build

$$\text{tag}_r(\langle k, \text{tag}_r(\langle n, \text{tag}_l(m) \rangle) \rangle).$$

In general, we can build elements in any of the types

$$\begin{aligned} N + N \times Y \\ N + N \times (N + N \times Y) \\ N + N \times (N + N \times (N + N \times Y)) \end{aligned}$$

where we do not use elements of the type Y which is just used as a place holder.

The key question about this process is whether we want to allow *anything else* in the type L . Our decision is **no**, we want only elements obtained by this finite process.

In set theory, we can understand similar inductive definitions, say a *set* L such that $L = N \cup (N \times L)$, as least fixed points of monotonic operators $F : \text{Set} \rightarrow \text{Set}$. In general, given such an operator F , we say that the set inductively defined by F is the least F -closed set, call it $I(F)$. We define it as

$$I(F) = \cap \{Y \mid F(Y) \subseteq Y\}.$$

We use set theory as a *guide* to justify recursive type definitions. Let Type be the collection of (small) types. It is a type itself, but does *not* include Type , i.e. $\text{Type} \notin \text{Type}$. We take Type as a new primitive (large) type.

For the sake of defining monotonicity, we introduce a very simple *subtyping relation*, $S \subseteq T$. This holds just when the elements of S are also elements of T , and the equality on S and T is the same. For example,

$$\text{if } S \subseteq T \quad \text{then} \quad N + N \times S \subseteq N + N \times T.$$

Definition Given $F : \text{Type} \rightarrow \text{Type}$ such that if $T_1 \subseteq T_2$ then $F(T_1) \subseteq F(T_2)$, then write $\mu X.F(X)$ as the *type inductively defined by* F .

To construct elements of $\mu X.F(X)$, we basically just *unwind* the definition. That is,

$$\text{if } t \in F(\mu X.F(X)) \quad \text{then} \quad t \in \mu X.F(X).$$

We say that $t_1 = t_2$ in $\mu X.F(X)$ iff $t_1 = t_2$ in $F(\mu X.F(X))$.

The power of recursive types comes from the fact that we can define total computable functions over them very elegantly, and we can prove properties of elements recursively. (These are the same idea as we will show later.) Recursive definitions are given by this term,

$$\mu\text{-ind}(a; f, z.b)$$

called a *recursor* or *recursive-form*. It obeys the computation rule

$$\mu\text{-ind}(a; f, z.b) \text{ evaluates in one step to } b[a/z, (y \mapsto \text{ind}(y; f, z.b))/f].$$

(Note in this rule we use the notation $b[s/x, t/y]$ to mean that we substitute s for x and t for y in b .)

Typing

The way we figure out a type for this form is given by a simple rule. We say that

$$\mu\text{-ind}(a; f, z.b) \text{ is in type } B$$

provided that $a \in \mu X.F(X)$, and if when Y is a Type, and $Y \subseteq \mu X.F(X)$, and z belongs to $F(Y)$, and f maps Y to B , then b is of type B .

Induction

The principle of inductive reasoning over $\mu X.F(X)$ is just this.

μ -induction

Let $R = \mu X.F(X)$ and assume that Y is a subtype of R and that for all x in Y , $P(x)$ is true. (This is the induction hypothesis.) Then if we can show $\forall z : F(Y).P(z)$, we can conclude

$$\forall x : R.P(x).$$

With this principle and the form μ -ind, we can write programs over recursive types and prove properties of them. The approach presented here is quite abstract, so it applies to a large variety of specific programming languages. It also stands on its own as a mathematical theory of types.

2.2.5 Subset Types

Among the objects of mathematics we consider are *propositions*. For example, $0 =_N 0$ and $0 < 1$ are true propositions about the primitive type N . We also consider *propositional forms* such as $x =_N y$ or $x < y$. These are sometimes called *predicates*.

The type of all (small) propositions is Prop . *Propositional functions* on a type A are elements of the type $A \rightarrow \text{Prop}$.

We also need types that can be restricted by predicates such as $\{x = N \mid x = 0 \text{ or } x = 1\}$. This type behaves like the *Booleans*.

The general formation rule is this. If A is a type and $B : A \rightarrow \text{Prop}$, then $\{x : A \mid B(x)\}$ is a *subtype of A* .

The elements of $\{x : A \mid B(x)\}$ are those a in A for which $B(a)$ is true.

Sometimes we state the formation in terms of predicates, so if P is a proposition for any x in A , then $\{x : A \mid P\}$ is a subtype of A .

We clearly have $\{x : A \mid P\} \subseteq A$.

2.3 Dependent Types

2.3.1 Dependent Products

Suppose you are writing a business application and you wish to construct a type representing the date:

$$\text{Month} = \{1, \dots, 12\}$$

$$\text{Day} = \{1, \dots, 31\}$$

$$\text{Date} = \text{Month} \times \text{Day}$$

We would need a way to check for valid dates. Currently, $\langle 2, 31 \rangle$ is a perfectly legal member of $Date$, although it is not a valid date. One thing we can do is to define

$$\begin{aligned} Day(1) &= \{1, \dots, 31\} \\ Day(2) &= \{1, \dots, 29\} \\ &\vdots \\ Day(12) &= \{1, \dots, 31\} \end{aligned}$$

and we will now write our data type as

$$Date = m : Month \times Day(m).$$

We mean by this that the second element of the pair belongs to the type indexed by the first element. Now, $\langle 2, 20 \rangle$ is a legal date since $20 \in Day(2)$, and $\langle 2, 31 \rangle$ is illegal because $31 \notin Day(2)$.

Many programming languages implement this or a similar concept in a limited way. An example is Pascal's *variant records*. While Pascal requires the indexing element to be of scalar type, we will allow it to be of any type.

We can see that what we are doing is making a more general type. It is very similar to $A \times B$. Let us call this type $prod(A; x.B)$. We can display this as $x : A \times B$. The typing rules are:

$$\frac{E \vdash a \in A \quad E \vdash b \in B[a/x]}{E \vdash \langle a, b \rangle \in prod(A; x.B)}$$

$$\frac{E \vdash p : prod(A; x.B) \quad E, u : A, v : B[u/x] \vdash t \in T}{E \vdash spread(p; u, v.t) \in T}$$

Note that we haven't added any elements. We've just added some new typing rules.

2.3.2 Dependent Function Spaces

If we allow B to be a family in the type $A \rightarrow B$, we get a new type, denoted by $fun(A; x.B)$, or $x : A \rightarrow B$, which generalizes the type $A \rightarrow B$. The rules are:

$$\frac{E, y : A \vdash b[y/x] \in B[y/x]}{E \vdash (x \mapsto b) \in fun(A; x.B)} \text{ new } y$$

$$\frac{E \vdash f \in fun(A; x.B) \quad E \vdash a \in A}{E \vdash f(a) \in B[a/x]}$$

Example Back to our example **Dates**. We see that $m : Month \rightarrow Day[m]$ is just $fun(Month; m.Day)$, where Day is a family of twelve types. And $\lambda(x.maxday[x])$ is a term in it.

3 Formalizing Type Theory

Nuprl presents a particular formalization of type theory. There are several others [17, 22, 23, 24, 26, 42]. At the core are ideas from deBruijn [17] and Martin-Löf [34] extended with the notions of *inductive types* and *direct computation* which give Nuprl its unique architecture.

The formalization comes in two parts. First we lay down the structure of *terms* and then *evaluation*.

terms
evaluation

This is the functional programming core of the theory, on top of it, we define the notion of type and proof. These are the basis of logic and mathematics.

type
proof

3.1 Terms

The informal language we used in Naive Type Theory does not reveal any pattern to the notation. We see forms like $x \mapsto b$, $f(a)$, $A \rightarrow B$, $\forall x:A.B$, U_i consisting of infix operators, parenthesized arguments, prefix operators, subscripts, etc. We would like a regular pattern to the terms that is a basis for analyzing and manipulating them. A Lisp-like syntax, $(f a_1 \dots a_n)$ might be adequate, but does not provide the flexibility we want for dealing with binding structure; Lisp only allows $(lambda (x_1 \dots x_n) b)$ for binding. So the term structure in Nuprl has the form

$$op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$$

where op is an operator name, \bar{v}_i is a list of binding variables governing the subterm t_i which is then *scope*. Every occurrence of a variable from \bar{v}_i in t_i is called *bound*. The t_i are subterms.

For display purposes, if \bar{v}_i is empty, we do not show it, and if $n = 0$ we drop the parentheses. Here is a list of all the Nuprl terms we need

<u>term</u>	<u>display form</u>
<code>function(A; x.B)</code>	$x : A \rightarrow B[x]$
<code>function(A; B)</code>	$A \rightarrow B$
<code>lambda(x.b)</code>	$\lambda x.b$
<code>apply(f; a)</code>	$f(a)$
<code>product(a; x.B)</code>	$x : A \times B[x]$
<code>product(A; B)</code>	$A \times B$
<code>pair(a; b)</code>	$\langle a, b \rangle$
<code>spread(p; u, v.t)</code>	$p.u$
	$p.v$
<code>rec(t.F)</code>	$\text{let } \text{rec } t = F(t)$
<code>rec_ind(h; f, z.t)</code>	$Y(t)$
<code>void</code>	φ

Syntactic variety comes from the display forms, written in the right column. These are like \LaTeX print macros, but they are also used for input as well.

Syntactic Theory

To understand how to compute with terms, we first need a complete account of certain aspects of the syntax. The main idea we need is the notion of substitution of a term s for occurrences of a variable x in a term t , written

$$t[s/x].$$

We need a very precise definition of this concept. To get it we require the definition of α -equality and renaming of bound variables. We want an account that is mathematically rigorous, yet transparently clear to users. To illustrate how this is accomplished, I refer to the working material by W. Aitken *A Formal Introduction to the Lambda Calculus* [2]. This theory is easy to express in Naive Type Theory. Here are some highlights.

Definition The λ terms are the following inductively defined class.

λ terms

Variables x, y, z, \dots

If f, a are λ terms and v is a variable then

$ap(f; a)$

$\lambda(v.a)$

are λ terms

Induction Principle

The induction principle corresponding to this class is given by

for all f, a, b λ terms and x a variable

$$\frac{P(v) \text{ for } v \text{ a variable} \quad P(f) \& P(a) \Rightarrow P(ap(f; a)) \text{ and } P(b) \Rightarrow P(\lambda(x.b))}{\forall t : \lambda\text{term. } P(t)}$$

Inductive Definitions

Objects can be defined inductively over the class of λ terms using the same induction principle.

Here is the definition of the set of free variables, $FV(t)$, of a term t .

$$FV(t) = \text{case } t \text{ is a variable } v \text{ then } \{v\} \\ t \text{ is an application } ap(p; q) \text{ then } FV(p) \cup FV(q) \\ t \text{ is an abstraction } \lambda(v.b) \text{ then } FV(b) - \{v\}.$$

Based on the syntactic theory, we can define exactly the kind of substitution we want. The interesting feature is that it must rename bound variables to avoid capture and rename them in a minimal way, i. e. rename as few variables as possible. For example, in

$$\lambda(x.\lambda(y.ap(z; x)))$$

if we substitute x for z , then we must rename the bound occurrences of x , but we do not want to rename them to y , because this will cause y to be renamed in the scope of x .

Aitken gives a precise definition of all this. He uses the concept $t[s/x]t'$ defined as:

Definition We say that t' is a substitution of s for x in t , written $t[s/x]t'$,

iff

$$\begin{aligned}
& x[s/x]s \\
& v[s/x]v \quad \text{if } v \neq x \\
& ap(p; q)[s/x]ap(p'; q') \quad \text{if } p[s/x]p' \text{ and } q[s/x]q' \\
& \lambda(x.r)[s/x] \lambda(x'.r') \quad \text{if } \lambda(x.r) =_\alpha \lambda(x'.r') \\
& \lambda(v.r)[s/x] \lambda(v'.r') \quad \text{if } x \neq v, \ v' \notin FV(s) \text{ or } x \notin FV(r) \ \& \\
& \quad \exists r''. \lambda(v.r) =_\alpha \lambda(v'.r'') \text{ and } r''[s/x]r'.
\end{aligned}$$

Theorem: $\forall t, s : \text{LTerm}. \forall x : \text{Var}. \exists t' : \text{LTerm}$

$$t[s/x]t' \ \& \ t' \text{ is a minimal renaming}$$

From a (constructive) proof of this $\forall\exists$ formula, we can extract a function to compute t' from t and s .

3.2 Evaluation

Inductive Definition

The terms of Nuprl denote data, programs and types; but programs and types can be data too. So we must say how to evaluate every term. There are many ways to approach this mathematically. We can give a denotational semantics, or a reduction (rewrite) semantics, or a structural operational semantics or something else. We choose to base the account on inductively defined relations and partial functions in the style of [13, 35, 39, 43], sometimes called “natural semantics”. We use a lazy evaluator [1, 13, 35].

We define first a relation “evaluates to,” $t \text{ evals_to } t'$. Then we define val as a map

$$val : \{x : \text{term} \mid \exists t : \text{term} \ x \text{ evals_to } t\} \rightarrow \text{term}.$$

We give some cases of the evaluation relation to illustrate the method.

- $$\frac{f \text{ evals_to } \lambda(x.b) \quad b[a/x] \text{ evals_to } c}{ap(f;a) \text{ evals_to } c}$$
- $$\frac{}{\lambda(x.b) \text{ evals_to } \lambda(x.b)}$$
- $$\frac{}{inl(a) \text{ evals_to } inl(a)}$$
- $$\frac{d \text{ evals_to } inl(a) \quad t_1[a/u] \text{ evals_to } c}{decide(d; u.t_1; v.t_2) \text{ evals_to } c}$$
- $$\frac{d \text{ evals_to } inr(b) \quad t_2[b/v] \text{ evals_to } c}{decide(d; u.t_1; v.t_2) \text{ evals_to } c}$$

Properties of Evaluation

We can prove rather easily these properties of evaluation.

Theorem:

1. $t \text{ evals_to } t' \ \& \ t \text{ evals_to } t''$ implies $t' = t''$.
(deterministic)
2. $t \text{ evals_to } t'$ implies t' is a value.
(value producing)
3. If t is a value then $t \text{ evals_to } t$.
(idempotent)

Evaluation fragments

In order to allow for the addition of new operators in a systematic way, we index the evaluation process by the operator name. So the actual evaluation relation has the form $t \text{ evals_to}_{op}(f) t'$ where f is a parameter. We just gave these fragments for some of the operators. The evaluation relation is then defined as

Definition: $t \text{ evals_to } t'$ iff $t \text{ evals_to}_{opname(t)} (evals_to) t'$

3.3 Types and membership

typehood

Essentially types are collections of terms. This is how they look from “outside the theory.” Inside the theory we see types as methods of construction; to specify a type is to say how to build a certain category of mathematical object. In order to say anything interesting about these objects, we need also to say when we regard two of them as equal. We follow Martin-Löf’s method for doing this.

Definition: To specify a type

- Designate a canonical name for it (type is a value)
- Say how canonical elements are formed (a method of construction is specified)
- Say when two elements are equal

Definition: Membership

If A is a type, then $a \in A$ means that a *evals to* a' and a' canonical of type A . Since A is a type, we know what its canonical members are.

Examples of type construction

A new type

Let $\mathbf{1}$ be a type, the unit type

Let \bullet be a value in $\mathbf{1}$, equal to itself, $\bullet =_1 \bullet$ or write $\bullet = \bullet$ in $\mathbf{1}$.

$\text{product}(\mathbf{1}; \mathbf{1})$ is a type $\text{pair}(\bullet; \bullet)$ is an element

$\text{function}(\mathbf{1}; \mathbf{1})$ is a type $\lambda(x.x)$ $\lambda(x.\bullet)$ are elements.

Also, $\lambda(x.x) = \lambda(x.\bullet)$ in $\text{function}(\mathbf{1}; \mathbf{1})$

Membership

The notion of type membership, written $a \in A$ for A a type, is defined by the mutually recursive inductive definition of typehood given above. It is easy to see that membership is not decidable.

Let Φ_i be the *i*th Turing machine in some acceptable indexing,

Define

$$\begin{aligned} H_i(x) &:= \text{if } \Phi_i(i) \text{ halts in } x \text{ steps} \\ &\quad \text{then } \mathbf{void} \\ &\quad \text{else } \mathbf{1} \text{ fi} \end{aligned}$$
$$\begin{aligned} \lambda(x.\bullet) \in x : N &\rightarrow H_i(x) \\ \text{iff} \\ \Phi_i(i) &\text{ diverges} \end{aligned}$$

3.4 Derivations of membership judgments

We need some method of systematically showing that $a \in A$. Since the definition is inductive, we can resort to some standard presentation method for membership in inductively defined

types as discussed in section 2 on Naive Type Theory. The result is a calculus or logic for making what Martin-Löf calls *membership judgments*. The form of this judgment is

$$\vdash a \in A$$

which is read: we know that a belongs to type A .

The definition of

$$\vdash \lambda(x.b) \in A \rightarrow B$$

suggest that we also want *hypothetical judgments*, such as

$$x : A \vdash b \in B.$$

In general, a *sequent*

$$x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash g \in G$$

means: if A_1 is a type, x_1 an element, A_2 a type for x_1 in A_1, \dots , then G is a type with element g .

Rules

We present the inductive definition in terms of rules for sequents. We organize them *top-down*, e. g.

$\vdash \lambda(x.b) \in A \rightarrow B \quad \text{by } \underline{\hspace{2cm}}$ $x : A \vdash b \in B$
--

Nuprl supports a problem-solving style that is *top-down* or *goal-driven* (like Prolog and λ Prolog [38]). In this style, the goals is

$$\begin{array}{c} \vdash G \\ \text{thought of as} \\ \vdash? \in G \end{array}$$

starting with G we try to synthesize g .

Rules in this style look like

$\vdash A \rightarrow B \quad \text{by } D0[\text{new } x]$ $x : A \vdash B$
--

The rule name Di means decompose hypothesis i or if $i = 0$ decompose the conclusion.

Sample Proof

$$\begin{aligned} &\vdash A \rightarrow \mathbf{1} \text{ by } D0 \text{ [new } y\text{]} \\ & y : A \vdash \mathbf{1} \text{ by } \bullet \\ & \vdash \bullet \in \mathbf{1} \end{aligned}$$

Extract $\lambda(y.\bullet)$ as element.

Proof Trees

In Nuprl, derivations are organized into proof trees. Here is a diagram.

$$\begin{array}{c} \vdash G \text{ by } r_0 \\ \begin{array}{ccc} H_1 \vdash G_1 \text{ by } r_1 & & H_2 \vdash G_2 \text{ by } r_2 \\ H_{11} \vdash G_{11} & H_{12} \vdash G_{12} & H_{21} \vdash G_{21} \text{ by } r_{21} \end{array} \end{array}$$

We can define these trees inductively in Naive Type Theory as follows:

sequent = (term list) \times term
 justification = {rule names} + ...
 proof = sequent \times justification \times proof list

3.5 Completeness

The terms, evaluation fragments, and rules for type membership are intended to formalize Naive Type Theory. The display notation and the informal semantics we have given suggest the connections between formal and informal accounts. I want to just mention some of the highlights by referring to the presentation in section 2.

Primitive Types

Nuprl has `void` for the empty type and takes the integers, `int`, as primitive. But N can be defined inductively or as a subset of `int`, e. g.

$$N = \{z : \text{int} \mid 0 \leq z\}$$

Compound Types

The display forms indicate the correspondence between the type constructors and Nuprl terms, for example the Cartesian product, $A \times B$, is `product(A; B)`. The tags for discriminated unions are formalized as `inr(b)`, `inl(a)`. The informal function notation, $x \mapsto b$ is formalized as `$\lambda(x.b)$` of course.

The inductive type $\mu x.F(x)$ is formalized as $\text{rec}(x.F(x))$, and the recursor, $\mu\text{-ind}(a; f, z.b)$ is $\text{rec_ind}(a; f, z.b)$.

Universes

The formal type theory must make some distinctions not seen in the naive version. In order for types to be values, they need to belong to a type. We might try adding a single new type, say *Type*, for this purpose. But then it must belong to a type. We might try attaining closure by postulating.

$$\textit{Type} \in \textit{Type}.$$

But this causes many troubles. It allows us to inhabit every type [21] and permits non-terminating functions in the function spaces [28]. So following the traditions of predicative type theory [45, 34, 19, 20], we introduce a hierarchy of types. Following [35] we call them *universes*. They are denoted $\mathbf{U}_1, \mathbf{U}_2, \dots$, and we have that

$$\begin{aligned} \mathbf{U}_i &\in \mathbf{U}_{i+1} \quad \text{and} \\ \text{if } A \in \mathbf{U}_i &\text{ then } A \in \mathbf{U}_j \quad \text{for } i < j. \end{aligned}$$

Logic is introduced into type theory by taking *propositions-as-types*. This is well explained elsewhere, e. g. [12, 13, 17, 23, 34, 35]. According to this principle,

$$\text{Prop}_i = \mathbf{U}_i.$$

4 Reflecting Formal Type Theory

4.1 The Basic Ideas

We used the Naive Type Theory of section 2 to present the Nuprl type theory, e. g. *term* is an inductive type. Nuprl is an approximation to the naive theory. A natural question to ask is whether we could use Nuprl to “define itself.” We know from Gödel’s incompleteness theorem and Tarski’s truth theorem that we cannot define Nuprl completely inside itself. But how close can we come?

It is easy to see that the basic syntax and the evaluation semantics can be completely defined. We already know similar results from experience defining Lisp evaluation in Lisp.

It is also easy to imagine that we can define the notion of a proof internally and the concept of a tactic. This again is clear from the ability to define these concepts nicely in ML as actually done in LCF, HOL, and Nuprl.

What is new and exciting here is this:

- We can define *terms*, *evaluations*, and *proof* naturally by just transcribing the informal definition. The formal definition clarifies the logic.

- The internal definition allows us to write theorem-proving tactics as Nuprl programs and prove properties of them.
- The internal account of proof enables us to define a new proof rule, the *reflection rule*, which is the basis for solving the design problem posed by decision procedures (see the introduction).

In this section we will examine the first two points and treat the last one in the next section. Most of this is conceptually clear, but the details are critical. I refer you to other writings for most of those. There are however a few subtle points to notice.

In reflecting the class *term* we will define an internal inductive type, **Term**. We have shown elsewhere that **Term** represents *term*. One tricky point here is to avoid the following infinite regress. For each operator name, like *int* or *spread*, we add a new operator name that denotes it, say **int** and **spread**. (We actually use `opid{int:opname}` and `opid{spread:opname}`.) Now we have new operators, **int** and **spread**, and we need names for them; we might try **int1** and **spread1** or something. But then we would need **int2** and **spread2**, etc. How to stop the regress? First notice this; all reflected operators have the form `opid{o:opname}`, and this is the generic form of operators. So they get reflected like any other operator, essentially `op{i:family}` is represented as a *pair*, $\langle \text{reflected operator, reflection of the list of pairs of index and index family name} \rangle$.

Another subtle point arises with reflecting evaluation. We do this by adding new operators **Evals_To** and **Val**. But we need to define evaluation fragments for them which are self-referential. This is explained in Aitken's thesis.

One of the most subtle points comes up in the definition of *proof*. The reflected concept uses **PreProof** and **Proof**. We need to specify how tactics build proofs, and this must be part of the definition of *proof*. But a tactic is a function that builds elements of **Proof**. So the definition of the informal concept of *proof* requires that we anticipate the definition of **Proof** and refer to it. This requires a delicate fixed point construction that appears in [6].

Dealing with these points correctly and formally takes a lot of care and relies on many details. But using the mechanism is quite natural. It is much like replacing ML or SML in a *tactic-oriented* prover with Nuprl's programming language.

4.2 Terms

Recall that terms have the form

$$op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$$

where op is an operator. Operators actually have structure because some of them come in *families*. For example, universes (displayed as \mathbf{U}_i) are nullary operators indexed by natural numbers. Generally an operator has the form

$$o\{i : f\}$$

where o is an *operator name*, and $i : f$ is a *tagged parameter* consisting of *index family*, f . For universes, the operator is

$$\mathbf{univ}\{i : \mathbf{nat}\}.$$

To reflect operators, we need to reflect operator names and index families. The *index families* include \mathbf{opname} , \mathbf{ifname} , \mathbf{var} , \mathbf{nat} . There are others. In order to specify the type representing these families, we need the new type constructor $\mathbf{IFam}(x)$; if x represents an index family name, then $\mathbf{IFam}(x)$ is a type representing the corresponding family. For example, $\mathbf{IFam}(\mathbf{ifid}(\mathbf{nat}))$ is the type $\{z : \mathbf{int} \mid 0 \leq z\}$.

Notice, we need an operator to represent the index family name; this operator is \mathbf{ifid} (“index family id”). It produces an operator name inside Nuprl corresponding to the various index families. In the Nuprl literature, we use these definitions

$$\begin{aligned} \mathbf{IFName} &= \mathbf{IFam}(\mathbf{ifid}(\mathbf{ifname})) \\ \mathbf{OpName} &= \mathbf{IFam}(\mathbf{ifid}(\mathbf{opname})). \end{aligned}$$

The primitive type \mathbf{OpName} is used to represent the operator names. Its elements look like $\mathbf{opid}\{p\}$ for some tagged parameter p . For example, the representation of the \mathbf{opname} \mathbf{univ} is $\mathbf{opid}\{\mathbf{univ}:\mathbf{opname}\}$.

To illustrate the level of precision that we achieve with this machinery, let us look at the official definition of variables. The terms called variables are

$$\mathbf{var}\{v : \mathbf{var}\}$$

These are formed with the \mathbf{var} operator name indexed by elements from a mathematical class, an index family (also named \mathbf{var}). At this level, we could choose any discrete mathematical class for \mathbf{var} , say character strings. To be precise, we say that \mathbf{var} is the class of finite sequences of *variable characters*. We require a subset of the ASCII characters as specified in the Nuprl manual.

Now to represent the variables *inside* Nuprl, we choose one of the Nuprl types that will represent this index family, and axiomatize it to be equal to $\mathbf{IFam}(\mathbf{ifid}(\mathbf{var}))$. We have chosen to represent the variable characters by the type N_{256} , natural numbers from 0 to 256. Variables are lists of these. The definition and axioms are

$$\begin{aligned} \mathbf{VarChar} &= N_{256} \\ \mathbf{IFam}(\mathbf{ifid}(\mathbf{var})) &= \mathbf{VarChar} \text{ list}. \end{aligned}$$

Let us take a closer look at `OpName`. This is a new type. What are its elements? Intuitively the elements are the terms that represent the actual operator names. Here are some of those names.

<u>operator</u>	<u>representation</u>
<code>var</code>	<code>opid{var}</code>
<code>univ</code>	<code>opid{univ}</code>
<code>natnum</code>	<code>opid{natnum}</code>
<code>void</code>	<code>opid{void}</code>
<code>lambda</code>	<code>opid{lambda}</code>

We use the convention that `opid{x}` stands for `opid{x:opname}`. This can be done for any fixed number of operator names because we can specify in the definition which index family is associated with which parameter. So in practice, we can eliminate the tags. They are needed to deal with the open-ended nature of the class `OpName`. This type must be open-ended because the mathematical class of actual operator names is open-ended.

The operator `opid{x}` produces canonical values. These are terms that evaluate to themselves.

Here is the official definition of `Term`.

Definition

```
Parameter = i:IFName × IFam(i)
OP = OpName × Parameter list
VarName = VarChar list
Term = rec(t.OP × (VarName list × t) list)
```

This tells us everything we need to know about terms. From the inductive type definition we get a recursor defining objects inductively over `Term` and we get an induction principle.

The small theory of the lambda calculus can be seen as a special case of a theory of `Term` if we just take these definitions and display forms.

```
Display << opid{var}, [a1, ..., an] >, nil > as implode([a1, ..., an])
Define λ(v.t) as << opid{lambda}, nil >, [< [v], t >] >
      ap(f; a) as << opid{apply}, nil >, [< nil, f >, < nil, a >] >
```

where $[a_1, \dots, a_n]$ is a notation for lists and `implode` takes a list of characters to an atom.

4.3 Sequents

Once we have represented terms, it is quite an easy matter to represent a sequent $H \vdash G$. The goal G is a term and H is a list of terms. Actually, since some of the hypotheses are treated specially, we need to tag them with a boolean value, and hypotheses are labelled. So the actual definitions are these.

Definition

$$\begin{aligned}\text{Hypothesis} &= \text{VarName} \times (\text{Term} \times \text{Boolean}) \\ \text{Sequent} &= \text{Hypothesis list} \times \text{Term}\end{aligned}$$

4.4 Primitive Proofs

Essentially, a proof is a finite tree whose nodes are sequents and justifications connected in certain ways. The definitions can be given in the form

$$\begin{aligned}\text{PreProof} &= \text{rec}(p. \text{Sequent} \times (\text{Justification} \times p \text{ list})) \\ \text{Proof} &= \{p:\text{PreProof} \mid \text{IsAProof}(p)\}\end{aligned}$$

The justifications explain why the goal sequent follows from the subgoals. We allow incomplete proofs; they are recognizable because the justification is a special element and no subgoals are generated.

Another kind of justification is a primitive rule name. These are the simple justifications, and if we were not interested in automating the development of proofs, this would be all that is necessary to completely define the concept of a proof. The only condition to be checked by the predicate *IsAProof* would be that the right subgoals were generated by the rules. We call such proofs *primitive*.

4.5 Tactics and Automation

Nuprl uses *tactics* to automate parts of the proof development process. A tactic is a function that builds part of a proof; it is given a sequent, and it produces a proof whose goal is the sequent. The resulting proof might be incomplete (usually is), and it might not be possible to complete it if the approach is a dead end.

The general idea of *tactic-oriented theorem proving* was pioneered by Edinburgh LCF [25]. Nuprl has extended this method by introducing the concept of a *tactic-tree* and by introducing the concept of a transformation tactic. We will not discuss the later idea.

The idea of a tactic-tree is that a justification for a proof step can be a tactic. The subproof built is not displayed, but the frontier of the subproof is listed as the new subgoals to be proved. This makes the tactic look like a new proof rule. So, by writing tactics, users extend the logic, but in a totally *safe* way. The tactics actually build primitive proofs, so there is no increase in the axiomatic basis of the theory. When a proof is complete, there is a primitive proof built in the background.

Here is an example of the internal proof produced by a tactic called `ChainThruHyps`.

1. $A \Rightarrow B_1$ 2. $B_1 \Rightarrow B_2$ 3. $B_2 \Rightarrow B_3$

$\vdash A \Rightarrow B_3$ by `ChainThruHyps`

4. A

$\vdash B_3$ by $D1$

$\vdash A$ by Hyp

5. B_1

$\vdash B_3$ by $D2$

$\vdash B_1$ by Hyp

6. B_2

$\vdash B_3$ by $D3$

$\vdash B_2$ by Hyp

7. B_3

$\vdash B_3$ by Hyp

The rule used in chaining is

$ \begin{array}{l} H, f : A \Rightarrow B, H' \vdash G \text{ by } D \ f \\ H, f : A \Rightarrow B, H' \vdash A \\ H, f : A \Rightarrow B, H', B \vdash G \end{array} $

The more conventional form is

$$\frac{H \vdash A \qquad H, A \Rightarrow B, B \vdash G}{H, A \Rightarrow B \vdash G}$$

4.6 Tactic-tree Proofs

Tactic-tree proofs are just proofs which allow tactic applications as justifications. Since they can be reduced to primitive proofs by executing the tactic, one might think that it is not necessary to give a reflected account of them. But there are reasons for reflecting.

The first reason is that we want to provide a theoretical basis for the idea that if we prove that the tactic works, then we can treat it as a new proof rule. This will both raise the level of abstraction of proofs and save the resources involved with executing the tactic call. Doing this requires that we have a formal definition of a tactic-tree proof since it is exactly objects in this class which will serve as the more abstract proofs.

A second reason to formalize the concept is that we wish to provide a rigorous account of what Nuprl actually does. This account will be a *formal explanation* of the system. This second reason influences the way we formalize the concept.

A very interesting problem arises as soon as we try to formalize tactic trees because we are now entering the realm of self-reference. A tactic is a Nuprl function, so it must operate on some internal data type, something we will eventually call **Proof**. But we use the tactic in explaining the top level notion of proof, the concept we have been denoting as *proof*. This concept does not even make sense until we have said how **Proof** represents *proof*. This involves some level of reflection already. *

If we are at a node with sequent s_0 whose subgoals are generated by t as justification, this means that $t(s_0)$ is **Proof** and the frontier of $t(s_0)$, say $s_1 \dots, s_n$ represent the sequents which are subgoals of s_0 . This would suggest that justifications are

$$1 + \text{primitive_rules} + \text{Proof}$$

In actual practice, we do not write a function application as a justification. We care about the exact computational behavior of the algorithm, so we really write a term t which denotes a function **(Sequent \rightarrow Proof)**. Since we want to analyze the term, it is more useful, if somewhat more complex, to use an element from **Term**.[†]

We spell this out precisely using these abbreviations. Let the *premises* of a proof be the unproven sequents at the leaves, the frontier of the proof tree. Given this frontier of sequents, let $\bar{p}_{1\dots n}$ denote the sequence of them. Given a proof p , let $\text{root}(p)$ be the root of the tree, which is the main goal or the theorem being proved. If j is a proof rule, then let $\text{instance}(j, s, \text{roots}(\bar{p}_{1\dots n}))$ be a predicate that checks that the roots of the subtrees $\bar{p}_{1\dots n}$ are the correct subgoals generated from s by j . Finally we use the relation $j \text{ reps } x$ to mean that the term j represents the pre-proof x .

When we write these concepts in bold letters,

$$\text{say } \mathbf{premises}(x), \mathbf{root}(p), \mathbf{instance}(j, s, \mathbf{roots}(x)),$$

they apply to the internal notion of proof, **Proof**.

Now we have these definitions.

Definition

$$\begin{aligned} \text{pre-proof} &= \text{sequent} \times \text{justification} \times \text{pre-proof list} \\ \text{justification} &= 1 + \text{rule} + \text{Term} \\ \text{proof} &= \{p : \text{pre-proof} \mid \text{is_a_proof}(p)\} \end{aligned}$$

*Let us assume that the type **Proof** exists and it represents *proof*. Then a justification could be a function $t \in \mathbf{Sequent} \rightarrow \mathbf{Proof}$.

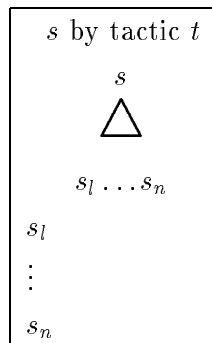
[†]In a system like HOL based on Standard ML, say SML/HOL, these issues don't arise. In this system, the tactics are not part of the logic.

$is_a_proof(< s, j, \bar{p}_1 \dots n >) =$
 case j of
 premise $\rightarrow n = 0$
 rule $\rightarrow instance(j, s, roots(\bar{p}_{1\dots n}))$
 tactic $\rightarrow \exists x : pre\text{-}proof.$
 $root(x) = s \ \&$
 $premises(x) = roots(\bar{p}_{1\dots n})$
 $\& j \text{ reps } x \ \& is_a_proof(x)$

where $is_a_proof(p_i)$.

Tactic-tree Diagram

$< s, j, [P_1, \dots, P_n] >$
 $j = t$ a Term, tactic text
 $\exists x : pre\text{-}proof. \underline{\hspace{2cm}}$



Tactic-trees Internally

Definition

PreProof == rec(PP. Sequent × Justification × PP list)

Justification == 1 + Rule + Term

Proof == { $p : PreProof \mid IsAProof(p)$ } where

$$\text{IsAProof}(\langle S, J, [P_1, \dots, P_n] \rangle) =$$
 case J of

- premise** $\rightarrow n = 0$
- rule** $\rightarrow \text{instance}(J, S, \text{roots}(P_1, \dots, P_n))$
- tactic** $\rightarrow \exists x : \text{PreProof}.$
 - $\text{root}(x) = S$ &
 - $\text{premises}(x) = \text{roots}(P_1, \dots, P_n)$
 - & $J \text{ Repr } x$ & $\text{IsAProof}(x)$

where $\text{IsAProof}(p_i)$.

Note the internal Repr map.

4.7 Reflection Rule

The material in the section is from the article *The Semantics of Reflected Proof* [6]. We want to take full advantage of the fact that we can reason about tactics. In particular, if we demonstrate that a tactic t generates a proof of some sequent s , we would like to be able to cite this demonstration rather than run the tactic.

Our general plan for doing this is to add a rule that essentially asserts that if

$$\exists p : \text{Proof}. \text{root}(p) = \text{rep}(H \vdash G)$$

then $H \vdash G$ is true, where $\text{rep}(s)$ is the representation of a sequent s . The rule might be

$$\text{Refl} \quad H \vdash G \text{ by Reflection} \\ \vdash \exists p : \text{Proof}. \text{root}(p) = \text{rep}(H \vdash G).$$

But the rule is not sound. From Löb's theorem on proof predicates [7], we suspect that this is not valid. Aitken discusses the situation in his thesis [3]. The idea is this. Given

$$\boxed{\exists p : \text{Proof}. \text{root}(p) = \text{rep}(H \vdash G) \mid R}$$

to conclude $H \vdash G$ we need a new rule, say *Refl*, but this is not part of *proof*. If we add it, we get a new notion of *proof*₀, say *proof*₁. If we allow reflection over these, we get *proof*₂, etc. Can we close this off? Almost! We can close it syntactically, but what does it mean? Notice that the extractor which is needed to build a canonical proof from \boxed{R} depends on this meaning.

Reflected Pre-Proofs

To get *syntactic closure* under reflection, we want a rule like *Refl* above. Where **Proof** reps proof with **Proof** in the rule itself. To accomplish this, we first parameterize proof to *proof(Q)* where *Q* is the term used in the rule. This forces us to consider **Proof(Q)**.

Syntactic Closure of Proof Definition

Define *proof(Q)* to allow the rule

$$\boxed{\begin{array}{c} H \vdash G \text{ by Refl, } i, t \\ \vdash \exists p : Q . \text{root}(p) = \text{rep}(H \vdash G) \\ \uparrow \end{array}}$$

In the [6] we show how to take a *syntactic fixed point* through *Q*.

Here is another approach.

Add to the library a definition

$$\text{PF} == \text{Term representing} \\ \text{proof(PF)}$$

This is straightforward from **PreProof** since PF is just a term.

The actual rule uses an indexing mechanism to keep track of how deeply the use of reflection is nested. The function *level(p)* gives the maximum of the level numbers occurring in reflection rules of *p*. Using this we incorporate the rule in the definition of *proof* as follows.

$$\begin{aligned} \text{proof} &= \{p : \text{pre-proof} \mid \text{is_a_proof}(p)\} \\ \text{is_a_proof} &(\langle s, j, [p_1, \dots, p_n] \rangle) \\ &\text{case } j \text{ of} \\ &\quad \mathbf{premise} \rightarrow n = 0 \\ &\quad \mathbf{rule} \rightarrow \dots \\ &\quad \mathbf{tactic} \rightarrow \dots \\ &\quad \mathbf{refl} \rightarrow p_l \vdash \exists p : PF. \text{root}(p) = \text{rep}(s) \quad (\text{where } j = \langle i, t \rangle) \\ &\quad \quad \& \text{level}(p) < i \\ &\quad \quad \& t \text{ reps roots}(\bar{p}_{2..n}) \& \text{is_a_proof}(p) \\ &\text{where } \text{is_a_proof}(p_i) \text{ for } i = 2, \dots, n. \end{aligned}$$

4.8 Validity Theorem

Definition A proof is valid if the goal is true when the premises are.

Theorem: Proofs are valid.

Pf: By induction on level, n , and for each n by induction on the subproof relation.

Let $\Delta = (s, r, \Delta_1, \dots, \Delta_k)$ be a proof whose premises are true. Let $\bar{\Delta} = (\Delta_1, \dots, \Delta_k)$.

Consider the cases for rule r .

1. $r = \bullet$, s is a premise, hence true.
2. r is a primitive rule, then s is true by validity of the primitive rules.
3. r is a tactic with term t . Then t represents a subproof Δ' which is valid by induction. The premises of Δ' are roots of $\bar{\Delta}$, they are true, so the goal of Δ' is, which is s .
4. r is Refl i, t . So the goal of Δ_1 is true by induction. This means that there is a proof Δ' of level less than i ; thus Δ' is valid. The premises of Δ' are the roots of $\Delta_2, \dots, \Delta_k$, hence true, so s is true as well. QED.

5 Applications

We will demonstrate an application of reflection to the task of adding sound decision procedures to tactic-oriented provers. For the sake of illustration, we take a very simple and familiar example — the decision problem for classical tautologies. A more realistic example would be decision procedures commonly used such as *arith* or *sup-inf*, but those are too complex to treat in these lectures, and the tautology procedure is used in some provers, see [18].

5.1 Sample Decision Procedure

The tableau decision procedure for classical propositional formulas is especially well understood (and still current, see the account in [40]). The idea is that given a formula, we try to systematically falsify it by building up a truth assignment to the subformulas, eventually including the atomic ones (the variables). If we succeed, then the formula is not a tautology as the falsifying assignment attests. If we fail in this systematic attempt, then we can show that the resulting formula is valid and has a propositional proof, that is, one using only rules of the propositional calculus. To illustrate, consider the following formula.

Decision Procedure

Show that

$$((A \Rightarrow B_1) \& (B_1 \Rightarrow B_2) \& (B_2 \Rightarrow B_3)) \Rightarrow (A \Rightarrow B_3) \text{ is a tautology.}$$

Try to assign it the truth-value **F**

$$\mathbf{T}(A \Rightarrow B_1 \ \& \ B_1 \Rightarrow B_2 \ \& \ B_2 \Rightarrow B_3) \quad \mathbf{F}(A \Rightarrow B_3)$$

Decomposing $(A \Rightarrow B_3)$ in the same way we get: $\mathbf{T}A, \mathbf{F}B_3$

Now force the assignment to the other atoms in the natural way:

$$\mathbf{T}(A \Rightarrow B_1)$$

$$\mathbf{T}(B_1 \Rightarrow B_2)$$

$$\mathbf{T}(B_2 \Rightarrow B_3), \mathbf{T}A, \mathbf{F}B_3$$

Now consider these observations.

$\mathbf{F}A$ is not possible, so $\mathbf{T}B_1$

$\mathbf{F}B_1$ is not possible, so $\mathbf{T}B_2$

$\mathbf{F}B_2$ is not possible, so $\mathbf{T}B_3$ which contradicts $\mathbf{F}B_3$

5.2 Formalizing the Decision Procedure

There are many ways to present the tableau decision procedure formally. In Nuprl we have looked at two of them [9, 14]. Caldwell's account [9] is based on the approach in [14].

The methods referred to above can also be extended to the full set of connectives and to a notion of propositional proof derived from the internal definition of **Proof**. The details of how to do this are in [5].

Define **PropSeq** as a subtype of **Term** and **PropProof** as a subtype of **Proof**. Informally these are the definitions.

Definition: $\mathbf{PropSeq} = \{s : \mathbf{Sequent} \mid \text{hypotheses of } s \text{ are variable declarations, the goal of } s \text{ is a proposition built from } \&, \vee, \Rightarrow, \mathbf{False} \text{ using the declared variables.}\}$

$\mathbf{PropProof} = \{p : \mathbf{Proof} \mid \text{the goal of } p \text{ is } \mathbf{PropSeq} \text{ and all rules are primitive and are rules of the classical propositional calculus.}\}$

As an abbreviation let P or not mean $P \vee \neg P$, then based on the proof method in [5, 14] we can constructively prove:

Theorem (dp):

$$\forall s : \mathbf{PropSeq}. (\exists p : \mathbf{PropProof}. \mathbf{root}(p) = s) \text{ or not}$$

Given a sequent $H \vdash G$ which is propositional, we have that the computable function extracted from dp , called here $extract(dp)$, decides whether or not there is a propositional proof.

5.3 Applying the Formal Decision Procedure

Let $d = \text{extract}(dp) (\text{rep}(H \vdash G))$. The value of d is either $\text{inl}(p)$ for some proof p or $\text{inr}(f)$ for some function f . In the case of $\text{inl}(p)$, we have $p \in \exists p : \mathbf{PropProof}.\text{root}(p) = \text{rep}(H \vdash G)$. Notice that $P = \text{decide}(d; u.u; v.v)$. Let \hat{d} be the term $\text{decide}(p; u.u; v.v)$.

We can now prove $H \vdash G$ by reflection if it is true.

$$\begin{aligned} H \vdash G & \text{ by Refl 1, nil} \\ & \vdash \exists p : \mathbf{Proof}.\text{root}(p) = \text{rep}(H \vdash G) \text{ by } D \hat{d} \end{aligned}$$

We can write a tactic

$$\text{Decide}(dp)$$

which checks d for $\text{inl}(p)$, writes \hat{d} and uses it as the witness, or fails.

$$H \vdash G \text{ by } \text{Decide}(dp).$$

This simple example illustrates the essential ideas behind the method of using reflection to justify decision procedures and for type checking in programming languages [10]. I think that this will be an important technique for modern provers that employ decision procedures. There are many other uses of reflection, and I recommend ‘looking at articles by Boyer and Moore [8], Weyrauch [50], Howe [29], as well as [3, 6, 16, 37, 47, 48].

Acknowledgements

I want to thank Kate Ricks for preparing this manuscript in \LaTeX and for cheerfully tolerating such complexity in her first such project. I appreciate Stuart Allen’s comments on an earlier draft.

References

- [1] S. Abramsky. The lazy λ -calculus. In D. A. Turner, editor, *Logical Foundations of Functional Programming*, pages 65–116. Addison-Wesley, 1990.
- [2] W. Aitken. A formal introduction to the lambda calculus. Computer Science Dept., Cornell University, Ithaca, NY, 1993.
- [3] W. Aitken. *Metaprogramming in Nuprl Using Reflection*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 1994.
- [4] W. Aitken and R. L. Constable. Reflecting on Nuprl Lessons 1-4. Technical report, Cornell University, Computer Science Dept., Ithaca, NY, 1992. To appear.
- [5] W. Aitken, R. L. Constable, and J. Underwood. Using reflected decision procedures. Technical report, Computer Science Dept., Cornell University, Ithaca, NY, 1993.
- [6] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The Semantics of Reflected Proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.
- [7] G. S. Boolos and R. C. Jeffrey. *Computability and Logic, Second Edition*. Cambridge University Press, 1980.
- [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] J. Caldwell. A constructive proof of propositional completeness in Nuprl. implementation notes, June 1993.
- [10] L. Cardelli and P. Longo. A semantic basis for Quest. In *Journal of Functional Programming*, pages 1:417–458, 1991.
- [11] T. Chan. An algorithm for checking PL/CV arithmetic inferences. In G. Goos and J. Hartmanis, editors, *An Introduction to the PL/CV Programming Logic*, Lecture Notes in Computer Science, Vol. 135, pages 227–264. Springer-Verlag, 1982.
- [12] R. L. Constable. Assigning meaning to proofs: a semantic basis for problem solving environments. *Constructive Methods of Computing Science*, NATO ASI Series, Vol. F55, pages 63–91, 1989.
- [13] R. L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

- [14] R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.
- [15] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
- [16] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. with Applications*, 5:217–230, 1979.
- [17] N. deBruijn. A survey of the project Automath. In *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.
- [18] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, pages Vol.57, Number 3, September 1992.
- [19] S. Feferman. Polymorphic typed lambda-calculi in a type free axiomatic framework. *Contemporary Mathematics*, 106:101–135, 1990.
- [20] S. Feferman. Logics for termination and correctness of functional programs. Technical report, Stanford University, Dept. of Mathematics, 1993.
- [21] J.-Y. Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.
- [22] J.-Y. Girard. *Proof Theory and Logical Complexity, Volume 1*. Bibliopolis, Napoli, 1987.
- [23] J. Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.
- [24] M. Gordon and T. Melham. *Introduction to HOL*. University Press, Cambridge, 1993.
- [25] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, NY, 1979.
- [26] S. Hayashi and H. Nakano. *PX: A Computational Logic*. MIT Press, Cambridge, MA, 1988.
- [27] C. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.
- [28] D. J. Howe. The computational behaviour of Girard’s paradox. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 205–214. IEEE, June 1987.

- [29] D. J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, pages 238–257. Springer-Verlag, New York, 1988.
- [30] S. Igarashi, R. London, and D. Luckham. Automatic program verification I: a logical basis and its implementation. *Acta Informatica*, 4:145–82, 1975.
- [31] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Symp. on Logic in Comp. Sci.*, pages 237–248. IEEE, 1986.
- [32] D. Kozen. *Complexity of Finitely Presented Algebras*. PhD thesis, Computer Science Department, Cornell University, Ithaca, New York, 1977.
- [33] D. Kozen, M. Ben-Or, and J. Reif. The complexity of elementary algebra and geometry. In *Proc. 16th ACM Symp. Theory of Comput.*, pages 457–464, 1984. Invited special issue *J. Comput. Syst. Sci.* 32(2):251-264, 1985.
- [34] P. Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.
- [35] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.
- [36] D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, Cambridge, Mass., 1989.
- [37] J. McCarthy. A basis for a mathematical theory of computation. *Comput. Program. and Formal Syst.*, pages 33–70, 1963. Amsterdam:North-Holland.
- [38] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. on Logic Programming*. ACM, 1987.
- [39] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.
- [40] A. Nerode and R. Shore. *Logic for Applications*. Springer-Verlag, New York, 1994.
- [41] S. Owre, J. Rushby, and N. Shankar. PVS: An integrated approach to specification and verification. Preprint, January, 1992.
- [42] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 313–321, 1989.

- [43] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Dept., Denmark, 1981.
- [44] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report CSL-91-2, Computer Science Laboratory, SRI International, February, 1991.
- [45] B. Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.
- [46] R. Shostak. A practical decision procedure for arithmetic with function symbols. *J. Assoc. Comput. Mach.*, 26:351–360, 1979.
- [47] B. Smith. Reflection and semantics in lisp. *Principles of Programming Languages*, pages 23–35, 1984.
- [48] C. Smorynski. *Self-Reference and Modal Logic*. Springer-Verlag, NY, 1985.
- [49] A. Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, Lecture Notes in Mathematics, Vol. 344. Springer-Verlag, 1973.
- [50] R. Weyrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.