Computational Metatheory in Nuprl*

Douglas J. Howe

88-899
March 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported in part by NSF grant CCR-8616552. This paper is to be published in *The Proceedings of the Ninth International Conference on Automated Deduction* (in the *Lecture Notes in Computer Science* series), Springer-Verlag, May 1988.

Computational Metatheory in Nuprl *

Douglas J. Howe
Department of Computer Science
Cornell University

February 29, 1988

Abstract

This paper describes an implementation within Nuprl of mechanisms that support the use of Nuprl's type theory as a language for constructing theorem-proving procedures. The main component of the implementation is a large library of definitions, theorems and proofs. This library may be regarded as the beginning of a book of formal mathematics; it contains the formal development and explanation of a useful subset of Nuprl's metatheory, and of a mechanism for translating results established about this embedded metatheory to the object level. Nuprl's rich type theory, besides permitting the internal development of this partial reflection mechanism, allows us to make abstractions that drastically reduce the burden of establishing the correctness of new theorem-proving procedures. Our library includes a formally verified term-rewriting system.

1 Introduction

Most theorem-proving systems that allow the user to soundly extend the inference mechanism with new theorem-proving procedures use one of two

*This research was supported in part by NSF grant CCR-8616552. This paper is to be published in *The Proceedings of the Ninth International Conference on Automated Deduction* (in the *Lecture Notes in Computer Science* series), Springer-Verlag, May 1988.

approaches. The first approach is to require that new procedures be proven correct in a formalized metatheory [5,2,14,11]. The second is to provide a tactic mechanism [7,4], which permits arbitrary new procedures, but which requires each application of such a procedure to generate a proof in terms of the primitive inference rules. It appears that only the second approach has been used in significant applications. In this paper we show how Nuprl [4], which incorporates a tactic mechanism, can be "boot-strapped" to encompass both approaches. We show how Nuprl's powerful type theory can be used to develop a completely internal account of a portion of its metatheory, and to make abstractions that significantly reduce the burden of formally verifying new theorem proving procedures. This partial reflection mechanism, together with some applications, has been implemented in Nuprl, resulting in a completely formal account of an extension to Nuprl that allows new theorem-proving procedures to be programmed in the type theory. This implementation provides some evidence for the practical potential of the approach.

As a simple example of the limitations of the tactic mechanism, consider equations involving an associative commutative operator "$\cdot$". One way to check that an equation

$$a_1 \cdot a_2 \cdot \ldots \cdot a_m = b_1 \cdot b_2 \cdot \ldots \cdot b_n$$

holds is to sort the sequences $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, with respect to some ordering on terms, and check that the results are identical. A tactic that emulated this informal procedure would have to chain together appropriate instances of lemmas (*e.g.*, for the associativity and commutativity of $\cdot$) and rules (*e.g.*, the substitution rule). This indicates two major problems. First, the tactic writer must be continually concerned with generating Nuprl proofs, and this can increase the intellectual effort involved in constructing theorem-proving procedures. Secondly, there is a major efficiency problem. In the example, even though it is known in advance that the sorting algorithm is sufficient to establish equality, every time the tactic is called it must "re-justify" the algorithm.

The mechanisms we have implemented allow procedures such as the one just described to be used directly. The main part of our implementation consists of a large collection of Nuprl definitions, theorems and proofs. This ***library*** may be regarded as the beginning of a book of formal mathematics;

it contains the formal development and explanation of a useful subset of Nuprl's meta-theory, and of a mechanism for translating results established about this embedded meta-theory to the object level. The most important application of this is to the automation of reasoning: one can write Nuprl programs that directly encode such meta-level procedures as the sorting-based algorithm given above, prove that the program is correct, and then apply it in the construction of Nuprl proofs. Applications of such programs can be done in a manner consistent with Nuprl's proof development paradigms.

The core of our library contains the development of the partial reflection mechanism. Natural representations for a certain subclass of the terms of the type theory (roughly, the quantifier-free terms) and for contexts (*i.e.*, definitions and hypotheses) are constructed. Theorems are then proven in Nuprl that connect the representations to the objects represented. This allows the results of computations involving representations to be used in making judgments about the represented objects. The remainder of the library contains the development of several applications. These include a term rewriting system and a procedure involving the algorithm discussed above.

The basic idea of the reflection mechanism is simple. Using Nuprl's recursive-type constructor, we define a type `Term0`[1] that represents a certain subset of the terms of Nuprl's theory. We then define what it means for a term (*i.e.*, a member of `Term0`) to be well-formed with respect to an environment, which is a Nuprl object which associates atoms with Nuprl objects. Environments are used to represent definitions and hypotheses. Let `Term`(α) be the type of all members of `Term0` that are well-formed with respect to α. We can demonstrate the existence of Nuprl functions `type` and `val` such that for any α,

$$\texttt{type}(\alpha) \quad \texttt{in} \quad \texttt{Term}(\alpha)\texttt{->SET}$$

and

$$\texttt{val}(\alpha) \quad \texttt{in} \quad \texttt{t:Term}(\alpha)\ \texttt{->}\ \texttt{type}(\alpha)\texttt{(t)},$$

where a member of `SET` is a type together with an equality relation on that type.

[1] We will use typewriter typeface for terms (that possibly contain definition instances) of Nuprl's type theory. Italicized identifiers within these terms will be meta-variables.

To see how the above can be put to use, consider a simplified example. Suppose that we have constructed a function f of type

$$\texttt{Term0 -> Term0},$$

and that we can show that it preserves equality in some environment α, *i.e.*, that for any term t in $\texttt{Term}(\alpha)$,

$$\texttt{type}(\alpha)(t) = \texttt{type}(\alpha)(f(t)) \texttt{ in SET}$$

and

$$\texttt{val}(\alpha)(t) = \texttt{val}(\alpha)(f(t)) \texttt{ in type}(\alpha)(t)$$

(*i.e.*, for the second equality, the equivalence relation from $\texttt{type}(\alpha)(t)$ is satisfied). For example, f might be based on the sorting procedure mentioned earlier. Suppose that we are in the course of proving a Nuprl theorem, that the current goal is of the form >> T (where >> is Nuprl's turnstile), and that we want to "apply" f to T. The first step is to *lift* the goal. To do this, we apply a special tactic which takes as an argument the environment α. This tactic first computes a member t of $\texttt{Term}(\alpha)$ such that the application $\texttt{val}(\alpha)(t)$ computes to a term identical to T, and then it generates the subgoal

$$\texttt{>> val}(\alpha)(t)$$

(assuming that the environment α is appropriate). We now apply a tactic which, given f, has the effect of computing $f(t)$ as far as possible, obtaining a value $t' \in \texttt{Term}(\alpha)$, and producing a subgoal

$$\texttt{>> val}(\alpha)(t').$$

At this point, we can proceed by applying other rewriting functions, or by simply computing the conclusion and obtaining an "unlifted" goal >> T' to which we can apply other tactics.

An important aspect to this work is that the partial reflection mechanism involves no extensions to the logic; the connection between the embedded meta-theory and the object theory is established completely within Nuprl. This means, first, that the soundness of the mechanism has been formally verified. Secondly, there is considerable flexibility in the use of the mechanism. All of its components are present in the Nuprl library, so one can use them for new kinds of applications without having to do any metatheoretic justification. Rewriting functions are just an example of what can be done; many other procedures that have proven useful in theorem proving, congruence closure for example, can be soundly added to Nuprl. Thirdly, the mechanism provides a basis for stating new kinds of theorems. For example, one can formalize in a straightforward way the statement of a familiar theorem of analysis: *if $f(x)$ is built only from x, $+$, $\cdot$, ..., then f is continuous.*

Finally, and perhaps most importantly for the practicability of this approach, it is possible to make abstractions that drastically reduce the burden of verifying the correctness of new procedures. Since contexts (environments) are represented, one can prove the correctness of a procedure once for a whole class of applications. For example, a simplification procedure for rings could be proven correct for any environment where the values of certain atoms satisfy the ring axioms, and then be immediately applied to any particular context involving a ring. A general procedure such as congruence closure could be proved once and for all to be correct in any environment whatsoever. Another kind of abstraction is indicated by the rewriting example to be given later. Functions such as `Repeat` are proved to map rewriting functions to rewriting functions; these kinds of combinators can often reduce the proof of correctness for a new rewriting function to a simple typechecking task that can be dealt with automatically.

One of the main motivations of this work is to provide tools that will be of use in formalizing Bishop-style constructive real analysis [1]. The design of the reflection mechanism incorporates some of the notions of Bishop's set theory. However, our work is more generally applicable, since Nuprl's type theory is capable of directly expressing problems from many different areas. It should be of use in other areas of computational mathematics (*e.g.*, in correct-program development). Using the techniques developed in AUTOMATH [6] and LF [8], it can be applied to reasoning in a wide variety of other logics (constructive in character or not). For example, any first-order logic can be embedded within Nuprl; in such a case, the reflection mechanism will encompass the quantifier-free portion of the logic.

In the next section is a brief description of some of the relevant components of Nuprl. This is followed by a concrete example of term-rewriting. The following section gives some details about what has been implemented. Finally, there is a discussion of some related work, and some concluding remarks are made. For a more complete account of the work described in this paper, and for a complete listing of the library that was constructed, see [9].

2 Nuprl

Nuprl [4] is a system that has been developed at Cornell by a team of researchers, and is intended to provide an environment for the solution of formal problems, especially those where computational aspects are important. One of the main problem-solving paradigms that the system supports is that of *proofs-as-programs*, where a program specification takes the form of a mathematical proposition implicitly asserting the existence of the desired programs. Such a proposition can be formally proved, with computer assistance, in a manner resembling conventional mathematical arguments. The system can extract from the proof the implicit computational content, which is a program that is guaranteed to meet its specification. With this paradigm, the construction of correct programs can be viewed as an applied branch of formal constructive mathematics.

The logical basis of Nuprl is a constructive type theory that is a descendent of a type theory of Martin-Löf [12]. The rules of Nuprl deal with *sequents*, which are objects of the form

$$x_1 : H_1,\ x_2 : H_2,\ \ldots,\ x_n : H_n \ >>\ A\,.$$

Sequents, in the context of a proof, are also called *goals*. The terms H_i are referred to as *hypotheses* or *assumptions*, and A is called the *conclusion* of the sequent. Such a sequent is said to be true when, roughly, there is a procedure that, under the assumption that H_i is a type for $1 \leq i \leq n$, takes members x_i of the types H_i, and produces a member of the type A. An important point about the Nuprl rules is that they allow top-down construction of this procedure. They allow one to *refine* a goal, obtaining subgoal sequents such that a procedure for the goal can be computed from procedures for the subgoals.

The Nuprl type theory has a large set of type constructors. They can be roughly described as follows. The type `Int` is the type of all integers. **`Atom`** is the type of all character strings (delimited by double quotes). The disjoint union $A \mid B$ has members $\mathtt{inl}(a)$ and $\mathtt{inr}(b)$ for $a \in A$ and $b \in B$. The dependent product $x{:}A\#B$ (written $A\#B$ when x does not occur free in B) has members $\mathtt{<}a\mathtt{,}b\mathtt{>}$ for $a \in A$ and $b \in B[a/x]$. The dependent function space $x : A\texttt{->}B$ (written $A\texttt{->}B$ when x does not occur free in B) has as members all functions $\lambda x.b$ such that for all $a \in A$, $b[a/x] \in B[a/x]$. The type A `list` has as members all lists whose elements are in A. The type $\{x : A \mid B\}$ is the collection of all members of A such that $B[a/x]$ has a member. The type $a\texttt{=}b$ `in` A (for $a, b \in A$) has the single member **`axiom`** if a and b are equal as members of A, and has no members otherwise. The members of the recursive type $\mathtt{rec(}T\,.\,A\mathtt{)}$ are the members of $A[\mathtt{rec(}T\,.\,A\mathtt{)}/T]$. Finally, there is a cumulative hierarchy of universes `U1`, `U2`, ..., where each $\mathtt{U}i$ is obtained via closing under the other type constructors.

Associated with each type constructor are forms for making use of members of the type; *e.g.*, projection for pairs, an induction form for integers, and application for functions. These forms give rise to a system of computation, and the proceeding discussion of membership, to be accurate, must be amended to reflect the fact that $b \in A$ whenever $a \in A$ and b computes to a. The usual connectives and quantifiers of predicate calculus can be defined in terms of the above type constructors using the *propositions-as-types* correspondence.

Proofs in Nuprl are tree-structured objects where each node has associated with it a sequent and a refinement rule. The children of a node are the subgoals (with their subproofs) which result from the application of the refinement rule of the node to the sequent. A refinement rule can be either a primitive inference rule, or a tactic (written in the ML language [7]). If it is a tactic, then there is a hidden proof tree whose leaves are the children of the refinement.

Nuprl has a large number of inference rules; we only point out one which is of special significance for our work. The *evaluation rule* allows one to invoke Nuprl's (reasonably efficient) evaluator on either a hypothesis or the conclusion of a sequent. The term is given as input to the evaluator, is computed as far as possible, and is replaced in the sequent by the result of the computation. This rule is used to execute theorem-proving procedures that are based on the reflection mechanism.

Interactions with Nuprl are centred on the *library*. A library contains an ordered collection of definitions, theorems, and other objects. New objects are constructed using special-purpose window-oriented editors. The text editor, together with the definition facility, permit very readable notations for objects of Nuprl's type theory. The proof editor is used to construct proofs in a top-down manner, and to view previously constructed proofs. Proofs are retained by the system, and can be later referred to either by the user, or by tactics.

3 An Example

We will now look at an example developed in our library, where a simple rewriting function is applied to an equation involving addition and negation over the rational numbers. The environment used is α_Q, which contains "bindings" for rational arithmetic. For example, it associates to the Nuprl atom "Q" the set (*i.e.*, member of Set) Q of rational numbers, and to the atom "Q_plus" the addition function. The rewriting functions for this environment are collected into the type Rewrite(α_Q). The requirement for being a rewriting function over α_Q is actually stronger than indicated earlier; the criteria given there must hold for any α which extends α_Q.

The member of Rewrite(α_Q) we will apply is

```
Repeat( TopDown( rewrite[x;y]( -(x+y) -> -x+-y ) ) )
```

(this function is named norm_wrt_Q_neg). This does just what one would expect: given a term, it repeatedly, in top-down passes, rewrites subterms of the form -(x+y) to -x+-y. The functions Repeat and TopDown both have type

Rewrite(α) -> Rewrite(α)

for any α. The argument to TopDown is by definition the application of a function of three arguments, which must be: a list of atoms that indicate what constants are to be interpreted as variables for the purpose of matching; and two members of Term0 which form the left and right sides of a rewrite-rule. Nuprl definitions are used to make members of Term0 appear as close as possible to the terms they represent. The term `x+y` that appears in the definition of `norm_wrt_Q_neg` is, when all definitions are expanded,

```
inl( < "Q_plus", <inl(<"x",nil>) . inl(<"y",nil>) . nil > ).
```

The sequent we apply our function to is

```
>> -(w+x+y*z+z) = -w+-x+-(y*z)+-z in Q
```

(the hypotheses declaring the variables to be rationals have been, and will be, omitted). The first step is to apply the tactic

```
(LiftUsing ['α_Q'] ...),
```

(where the "..." indicates the use of a Nuprl definition that applies a general purpose tactic called the *autotactic*) which generates the single subgoal

```
α:Env, (...)
>> ↓( val( α, -(w+x+y*z+z) = -w+-x+-y*z+-z in Q ) )
```

(↓ is an information-hiding operator that will not be discussed here—see [4] for an explanation). The notation "(...)" indicates that the display of a hypothesis has been suppressed by the system. This elided hypothesis contains (among other information) the fact that α is equal to an environment formed by extending α`_Q` by entries for representatives of the variables `w`, `x`, `y` and `z`. The next step is to apply the tactic

```
(RewriteConcl 'norm_wrt_Q_neg' ...),
```

which generates a single subgoal with conclusion

```
↓( val( α,-w+-x+-(y*z)+-z = -w+-x+-(y*z)+-z in Q ) ).
```

The proof is completed using an equality procedure to be described later.

4 The Implementation

In this section is a presentation of the Nuprl library containing the partial reflection mechanism and applications. The library contains more than

1300 objects, and is by far the largest development undertaken so far using Nuprl. An ideal description of the library would involve the use of the Nuprl system itself; the system is designed not only for the construction of formal arguments, but also for their presentation. The best that can be done here is to explain some of the basic definitions and theorems. The next three subsections are devoted to the library objects and tactics that establish the reflection mechanism. The last two subsections describe some implemented examples of the use of the mechanism.

Of the 1300 objects in the library, about 800 are theorems, 400 are definitions, and 100 are ML objects. The library can be roughly divided according to topic (in order) as follows.

- Basics.
- Lists.
- Sets.
- The representation of Nuprl terms.
- Association lists and type environments.
- Function environments and combined environments.
- Booleans and "partial" booleans.
- Well-formedness and evaluation of terms.
- The monotonicity of certain functions with respect to environment extension, and operations on environments.
- Term rewriting.
- Other applications.

The library divisions corresponding to the first three topics are of somewhat lesser interest, and will be discussed very briefly in this section.

Before proceeding, we need to establish a few notational conventions. Typewriter typeface will be used exclusively to denote Nuprl terms or names of Nuprl objects. In the context of a Nuprl term, a variable which is in italics will denote a meta-variable. Thus x+y denotes a Nuprl term where x and y are *Nuprl* variables, whereas in x+y, x and y range over Nuprl terms. We will be somewhat sloppy in the use of meta-variables, using them, for example, to denote parameters of Nuprl definitions, but the context should make clear what the intention is.

The presentation below is somewhat in the style of conventional mathematics. Usually, explicit references to the existence of Nuprl objects will not be made. Unless stated otherwise, when a definition is made, there is a corresponding definition in the library; and when it is asserted that a certain statement, written in Nuprl syntax, is true, or can be proved, then there is a corresponding theorem in the library.

The "basics" section of the library contains mostly simple definitions, such as for logical concepts and definitions pertaining to the integers. There are also definitions that provide alternate notations for simple combinations of base Nuprl terms. For example, we define a "let" construct in the usual way:

$$\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ t' \equiv (\lambda\ x\,.\,t')(t),$$

and use it (together with the definitions for projection from pairs) in the definition let2:

$$\texttt{let}\ x\texttt{,}y\ \texttt{=}\ p\ \texttt{in}\ t \equiv \texttt{let}\ x\ \texttt{=}\ p\texttt{.1}\ \texttt{in}\ \texttt{let}\ y\ \texttt{=}\ p\texttt{.2}\ \texttt{in}\ t\,.$$

The "lists" section of the library contains a fairly substantial development of list theory.

In order to account for equivalence relations that are not built into the type theory[2] (*e.g.*, equality of real numbers), we define a *set* to be a pair consisting of a type together with an equivalence relation on the type. We define a hierarchy of collections of sets that parallels the universe hierarchy: a member of Set(i) is a pair consisting of a type from Ui together with an equivalence relation on that type. For S a set, define $|S|$ to be the type component of S, and for a and b members of $|S|$, define a=b in S to be the application of the equivalence relation of S to a and b. We also define some particular sets that will be used later in the library: Set and SET are Set(6) and Set(7), respectively, and

$$\texttt{Prop} \equiv \texttt{<U6, }\lambda\texttt{ P,Q. P <=> Q>.}$$

[2]Nuprl has a quotient-type constructor, but it cannot be used for our purposes (see [4]).

The choices of 6 and 7 are somewhat arbitrary; we only need to guarantee that the levels are high enough to accommodate anticipated applications.

It is a simple matter to define some basic set constructors. We will overload our notations and use S_1->S_2 to denote the *set* of functions from S_1 to S_2, and S_1#S_2 for the product of S_1 and S_2. Also, #(L) will denote the product of the sets in the list L. We will not discuss here our treatment of Bishop's notion of *subset* [1].

4.1 Representation

In this section we discuss first the representation in Nuprl of a simple class of Nuprl terms, and then the representation of contexts (*i.e.*, the library and hypotheses).

The type `Term0` of "raw" terms, which do not necessarily represent an object of the Nuprl theory, is defined via the recursive type

```
rec(T. (Atom#T list) | (T#T#Atom) | (Atom#T#T) | Int ).
```

Members of this type, which will be referred to as *meta-terms*, or just *terms* when no confusion can result, can be thought of as trees with four kinds of nodes. The meaning of the different kinds of nodes is suggested by the notations for the injections:

$$
\begin{array}{l}
f\texttt{(}l\texttt{)} \equiv \texttt{inl(<}f\texttt{,}l\texttt{>)} \\
x \texttt{ = } y \texttt{ in } A \equiv \texttt{inr(inl(<}x\texttt{,}y\texttt{,}A\texttt{>))} \\
y\{i\ x\} \equiv \texttt{inr(inr(inl(<}i\texttt{,}x\texttt{,}y\texttt{>)))} \\
n \equiv \texttt{inr(inr(inr(}n\texttt{)))}.
\end{array}
$$

The above are all members of `Term0` whenever f, A, i are in `Atom`, x and y are in `Term0`, l is a list of members of `Term0`, and n is an integer. These kinds of nodes will be referred to, respectively, as function-application nodes, equality-nodes, i-nodes, and integer nodes. The first will represent terms which are function applications; the second, terms which are equalities (in the set sense); and the last, terms which are integers. The inclusion of i-nodes is motivated by Bishop's notion of *subset* [1]; i-nodes will be ignored in the rest of this paper.

Contexts are represented with association lists (or "a-lists"), of type

(Atom # A) list

for A a type. Such lists are used to associate Nuprl objects with the atoms that appear in meta-terms. In dealing with a-lists, and in the term-rewriting system described later, extensive use is made of *failure*. For A a type, define `?`A to be the type A`|True` (where `True` is a single-element type). A value `inl(`a`)` of `?`A is denoted by `s(`a`)` (`s` for "success"), and the unique value of the form `inr(`a`)` is denoted by `fail`. In later sections, the distinction between a and `s(`a`)` will often be glossed. The library has many objects related to failure and a-lists.

A basic definition for a-lists is of a-list application, written l`{`A`}(`a`)`. If A is a type, if l is in `Atom#`A `list`, and if a is in `Atom`, then l`{`A`}(`a`)` is in `?`A. Evaluation of l`{`A`}(`a`)` either results in failure, or returns some b such that `<`a`,`b`>` is a member of l.

Contexts are represented as pairs of a-lists, the first component of which is a type environment, and the second of which is a function environment. Type environments associate atoms with members of `Set` and are members of the type:

$$\texttt{TEnv} \equiv \texttt{(Atom \# S:Set \# ?triv\_eq(S)) list.}$$

The predicate `type_atom` is defined so that `type_atom(`γ`,`a`)` is true if and only if either a is the atom `"Prop"` or a is bound in the type environment γ. Also, `type_atom(`γ`,`a`)` has the property that if, during the course of a proof, γ and a are sufficiently concrete (*e.g.*, if they do not contain any free variables), it can be evaluated to either `True` or `False`. Define

$$\texttt{AtomicMType}(\gamma) \equiv \{\ \texttt{a: Atom | type\_atom}(\gamma\texttt{,a})\ \}$$

(`M` is for "meta"). For members a of this type, define γ`(`a`)` to be `<Prop, fail>` if a is `"Prop"`, otherwise the value obtained from looking up a in γ. We define

$$\texttt{MType}(\gamma) \equiv \{\ \texttt{t: Atom list \# Atom | all\_type\_atoms}(\gamma\texttt{,t})\ \}.$$

This is the type of "meta-types" for functions; the first component of such a meta-type is a list of atoms that represent a function's argument types, and the second component represents the result type.

We can now define evaluation for members of `MType(`γ`)`. If mt is such a member, and mt`.1` (*i.e.*, the first component of the pair mt) is non-empty, then we can evaluate *mt*'s "domain type":

```
dom_val(γ,mt)  ≡
   let l,b = mt in
   #((map λa. val(γ,a) on l to SET list)).
```

Evaluation of function meta-types is defined as:

```
val(γ,mt)  ≡
let l,b = mt in
if null(l) then val(γ,b) else dom_val(γ,mt) -> val(γ,b).
```

A value val(α,x) will often be referred to as the *meaning* or *value* of x in α.

Function environments associate an atom, called a *meta-function*, with a meta-type, with a value that is a member of the value of the meta-type, and with some additional information about the value. More specifically, for γ a type environment, define

```
FEnv(γ)  ≡
Atom # mt:MType(γ) # f:|val(γ,mt)| # val_kind(γ,mt,f)  list.
```

The type val_kind(γ,mt,f) can be ignored here. Definitions analogous to those made for type environments are made for function environments. Thus define function-environment application, the predicate fun_atom, and the type MFun(α) of meta-functions. For f in MFun(α), mtype(α,f) is the meta-type assigned to f by α.2, and val(α,f) is the value assigned to f.

We can now define the type whose members represent contexts:

```
Env  ≡  γ:TEnv # FEnv(γ).
```

Members of Env will simply be called *environments*, and the variable α will always denote an environment. For most of the definitions presented so far that take a type environment as an argument, there is a new version which takes an environment as an argument. In what follows, only these new versions will be referred to, so the same notation will be used.

4.2 Well-Formedness and Evaluation

This section presents the core of the partial reflection mechanism. We formalize a notion of well-formedness for members of `Term0`, and construct an evaluation function for well-formed terms.

Roughly, a meta-term t is well-formed in an environment α if "the meta-types match up". For example, if t is a function-application $f(l)$, then we require that the domain component of the meta-type associated with f in α "matches" the list of terms l; *i.e.*, that they have the same length as lists, and that the atomic meta-types of f's domain "match" the atomic meta-types computed for the terms in l. A simple definition of "match" for atomic meta-types would be equality as atoms. However, since each f can have only one meta-type associated with it, this definition would preclude us from representing such simple terms as **x+y**, where **x** and **y** are Nuprl variables declared to be in a sub-type of the integers, and where **+** is addition over the integers. Also, it would preclude us from dealing with simple partial functions, *i.e.*, from representing terms like $g(x)$ where the domain type of g is a subtype of the type of x and x satisfies the predicate of the subtype. This leads us to consider a notion of matching of atomic meta-types that extends the simple version by allowing the associated values to stand in a subtype relationship. Consider the case of a meta-term $f(t)$, where f has meta-type domain the singleton list $[A]$, t has the meta-type B, and the value in α of A is a subtype of the value of B. Then for $f(t)$ to be well-formed we require that the *value* of t satisfy the subtype predicate associated with A.

Thus well-formedness depends on evaluation, which in turn is only defined on well-formed terms, and so we need to deal with these notions in a mutually recursive fashion. In particular, we first define the function which computes meta-types of terms, the evaluation function, and the well-formedness predicate, and then prove that they are well-defined (*i.e.*, have the appropriate types) simultaneously by induction.

Several functions below are defined by recursion. We will give the recursive definitions informally, although the formal versions are obtained directly.

The meta-type of a well-formed term is simple to compute; define $\mathtt{mtype}(\alpha,t)$ to be: if t is $f(l)$, then $\mathtt{mtype}(\alpha,f)\mathtt{.2}$; if t is $u\mathtt{=}v$ $\mathtt{in}$ A, then "Prop"; if t is n then "Int". Also, define type(α,t) to be the value in α of t's metatype. Evaluation of terms is defined using an auxiliary function that applies a function g to a list l and produces a tuple (denoted by $g\{\alpha\}(l)$) of the results. Thus we define val(α,t) to be: if t is f(l), then

```
if null(mtype(α,f).1) then val(α,f)
else val(α,f) (g{α}(l))
```

where $g(t)$ is val(α,t); if t is u=v in A, then

```
val(α,u)=val(α,v) in val(α,A);
```

and if t is n, then n.

Due to space limitations, we will not give a detailed definition of the well-formedness predicate. Instead, we will just informally describe the criteria for a function-application f(l) to be well-formed in an environment α. First, f must be bound as a function in α, and the domain part of its meta-type must be a list r with the same length as l. Secondly, the terms in l must be well-formed in α. Finally, for each t in l and corresponding a in r, a and mtype(α,t) must "match" in the sense outlined earlier in this section. Note that this last criterion involves val(α,t).

The assertion that a term t is well-formed in an environment α is written **wf@**(α,t). The central theorem of the partial reflection mechanism is the following:

```
∀α:Env. ∀t:Term0. wf@(α,t) in U &
↓(wf@(α,t)) => (mtype(α,t) in AtomicMType(α)
                & val(α,t) in |type(α,t)|)
```

(where U is a particular universe—we use membership in a universe to assert that a term is a well-formed type). Finally, define Term(α) to be the set of all meta-terms that are well-formed in α.

For the reflection method to be useable, it is necessary that the lifting procedure outlined in the first section of this chapter be reasonably fast. Since each lifting of a sequent will require establishing that some meta-terms are well-formed, the well-formedness predicate **wf@** is not suitable, imposing excessive proof obligations. Instead, we use a characterization **wf** of the predicate that embodies an algorithm for the "decidable part" of well-formedness. To prove `wf(`α`,`t`)` for concrete α and t, we apply Nuprl's evaluation rule, which simplifies the formula to a conjunction of propositions that assert that some terms satisfy some subtype predicates. In many cases arising in practice, the simplified formula will be `True` (or `False`).

The characterization `wf` is accomplished using the "partial booleans". Define

$$\texttt{Bool} \equiv \texttt{True}\vee\texttt{True} \qquad \text{and} \qquad \texttt{PBool} \equiv \texttt{Bool}\vee\texttt{U},$$

so a partial boolean is either a boolean or a proposition. We define `PBool` analogues of the propositional connectives, and of some of the other basic predicates. The characterization is built essentially by replacing components of `wf@` by their `PBool` analogues.

We will usually want our assertions about the correctness of theorem-proving procedures to respect environment extension. For example, we may have constructed a function f that normalizes certain expressions and that is correct in an environment α_Q that associates certain atoms with the operations of rational arithmetic. Clearly we will want to do more than just apply f to terms that are well-formed in α_Q; at the very least, we will want to apply f to a term like `x+y`, where `x` and `y` are variables declared in some sequent. This necessitates a notion of *sub-environment*. The environment α_1 is a sub-environment of α_2, written $\alpha_1 \subset \alpha_2$, if α_1`.1` and α_1`.2` are sub-a-lists of α_2`.1` and α_2`.2`, respectively. Most of the functions that have been defined so far that take an environment as an argument are *monotonic* with respect to environment extension. For example, if $\alpha_1 \subset \alpha_2$, and if t is well-formed in α_1, then t is well-formed in α_2 and its values in the two environments are equal. A substantial portion of the library is devoted to monotonicity theorems.

4.3 Lifting

Suppose that we wish to use some procedure that operates on meta-terms to prove a sequent

$$A_1,\ A_2,\ \ldots,\ A_n \ \texttt{>>}\ A_{n+1}.$$

We first *lift* the sequent using the tactic invocation `LiftUsing` *envs*, where *envs* is an ML list of Nuprl terms that are environments, call them α_1, α_2,

$\ldots$, α_m. All but one of the subgoals generated by this tactic will in general be proved by the autotactic (a general purpose tactic that runs after all other top-level tactic invocations). The remaining subgoal will have the form

$$\alpha\texttt{:Env, (...), val(}\alpha\texttt{,}A'_1\texttt{), val(}\alpha\texttt{,}A'_2\texttt{), ..., val(}\alpha\texttt{,}A'_n\texttt{)}$$
$$\texttt{>> val(}\alpha\texttt{,}A'_{n+1}\texttt{),}$$

where `(...)` is an elided hypothesis, and each A'_i is a member of `Term0` that represents A_i. By the use of definitions, A'_i will usually appear to the user to be exactly the same as A_i. The preceding description of lifting contains two simplifications. First, the hypotheses shown may be interspersed with hypotheses that have declared variables; the lifting procedure ignores such hypotheses. Secondly, some of the A_i may remain in the hypothesis list, not being replaced by val(α, A'_i). This happens when the representation computed for A_i is trivial.

The first step in lifting a sequent is to apply a procedure that uses the α_j's to compute the representations A'_i in `Term0` for the terms A_i, $1 \leq i \leq n+1$. This procedure also produces a list of environment additions for those subterms that are unanalyzable with respect to the α_j's; each such subterm s will be represented by a new metafunction constant whose associated value will be s. The new variable α is added to the sequent to be lifted, as well as the new hypothesis `(...)`. This elided hypothesis contains the following assertions: that the α_j are consistent (*i.e.*, that they agree on the intersection of their domains); that α is equal to the combination (call it $\overline{\alpha}$) of the α_j and the environment additions; that $\alpha_j \subset \alpha$ for each j; and that each A'_i is a well-formed meta-term that has meta-type `"Prop"`. Of course, to justify the addition of this hypothesis, the lifting tactic must prove each of the assertions. This is accomplished by the use of simplification (using the evaluation rules) and of lemmas (to handle some details concerning the fact that some substitutions of $\overline{\alpha}$ for the variable α must be done before simplification). Most of these assertions will be proved automatically. The final stage of the lifting procedure is to replace each A_i by val(α, A'_i); the tactic only needs to use evaluation in order to justify this.

The elided hypothesis contains most of the information necessary to prove the applicability of meta-procedures. For example, if f is in

$$\texttt{Rewrite(}\alpha'\texttt{)}$$

then to prove that f is applicable all that needs to be shown is that $\alpha' \subset \alpha$. An important point about this new hypothesis is that it is easy to maintain. For example, rewriting functions preserve meta-types and values, so when a rewrite is applied to a sequent, we can quickly update the hypothesis to reflect the changes in the sequent. Thus, the work of applying a rewrite to a lifted sequent is dominated by the work of doing the actual rewriting.

4.4 Term Rewriting

The most important application of the reflection mechanism is to equality reasoning. In this section is a description of the implementation in Nuprl of a verified term-rewriting system. This implementation consists of a collection of definitions, theorems, and tactics that aid the construction of term-rewriting (*i.e.*, equality-preserving) functions. Using this collection, one can easily turn equational lemmas into rewrite rules, and concisely express a wide variety of rewriting strategies. Proving the correctness of rewriting strategies generally reduces to simple typechecking.

4.4.1 The Type of Rewriting Functions

Informally, a function f of type `Term0->?Term0` is a rewrite with respect to an environment α if for any term t that is well-formed in some extension α' of α, $f(t)$ either fails or computes to a term t' such that: (1) t' is well-formed in α'; (2) t' and t have the same meta-type, call it A; and (3) the meanings of t and t' are equal in the set which is the meaning of A. Define `Rewrite(`α`)` to be the collection of all such f. Note that monotonicity (with respect to environment extension) is built into the definition of `Rewrite`, so that if f is in `Rewrite(`α`)`, and $\alpha \subset \alpha'$, then f is in `Rewrite(`α'`)`.

4.4.2 Basic Rewriting Functions

The basic units of conventional term rewriting are *rewrite rules*. These are ordered pairs $\langle t, t' \rangle$ of terms containing variables, where the variables of t' all occur in t. A term u rewrites to a term u' via the rule $\langle t, t' \rangle$ if there is a substitution σ for the free variables of t such that u' is obtained by replacing an occurrence of $\sigma(t)$ in u by $\sigma(t')$. In our setting, we will view a rewrite rule as a member of `Rewrite(`α`)`; corresponding to $\langle t, t' \rangle$ will be the function which, given u, if there is a substitution σ such that $\sigma(t) = u$, returns $\sigma(t')$, or else fails. These rules will usually be obtained by "lifting" a theorem that is a universally quantified equation.

We first need to develop a theory of substitution for meta-terms. The *variables* of a meta-term t are defined with respect to a list l of identifiers, and are the subterms of t that are applications of a member of l to no arguments. A *substitution* is a member of the type `(Atom # Term0) list`. The application of a substitution *subst* to a meta-term t is written *subst*(t). A matching function is extracted from the proof of the specification

```
∀vars:Atom list. ∀t1,t2:Term0.
?(∃s:Atom#Term0 list where ... & s(t1)=t2 in Term0)
```

(where we have elided an uninteresting conjunct).

The rewrite associated with a "lifted" equation is simply defined in terms of the matching function (call it `match`). For *vars* a list of atoms, and for u, v terms, the function is

`rewrite{`*vars*`}(`u`->`v`)` $\equiv$
λ`t. let s(subst) = match(`u`,t,vars) in s(subst(`v`)): ?Term0.`

(recall that `s(`a`)` denotes a successful computation). To prove that

`rewrite{`*vars*`}(`u`->`v`) in Rewrite(`α`)`

for particular u and v, one uses a special tactic that makes use of a collection of related lemmas. This tactic often completes the proof without user assistance.

There are two simple basic rewrites that are used frequently. The first is the trivial rewrite,

`Id` $\equiv$ λ`t. s(t).`

The second is `true_eq`, which rewrites an equality meta-term a`=`b `in` A to `True` when a and b are "syntactically" equal.

4.4.3 Building Other Rewriting Functions

The approach to term rewriting of Paulson [13] is easily adapted to our setting. His approach involves using analogues of the LCF *tacticals* [7] as combinators for building rewrites, where his rewrites produce LCF proofs of equalities. Using these combinators, we can concisely express a variety of rewriting strategies (Paulson used his rewriting package to prove the correctness of a unification algorithm [13]). Furthermore, since we can prove that each combinator is correct (*i.e.*, it combines members `Rewrite`(α) into a member of `Rewrite`(α)), proving that a complex rewriting function is correct often involves only simple typechecking.

If f and g are in `Rewrite`(α), then so are the following.

- f `THEN` g. This rewriting function, when applied to a term t, fails if $f(t)$ fails, otherwise its value is $g(f(t))$.
- f `ORELSE` g. When applied to t, this has value $f(t)$ if $f(t)$ succeeds, otherwise it has value $g(t)$.
- `Progress` f. This fails if $f(t)$ succeeds and is equal (as a member of `Term0`) to t, otherwise it has value $f(t)$.
- `Try` f. If $f(t)$ fails then the value is t, otherwise it is $f(t)$.
- `Sub` f. This applies f to the immediate subterms of t, failing if f failed on any of the subterms.
- `Repeat` f. This is defined below.

In Nuprl, all well-typed functions are total. However, many of the procedures, rewrites in particular, that we will want to write will naturally involve unrestricted recursion, and in such cases we will not be interested in proving termination. For example, many theorem-proving procedures perform some kind of search, and searching continues as long as some criteria for progress are satisfied. In such a case, it will often be difficult or impossible to find a tractable condition on the inputs to the procedure that will guarantee termination. It appears that the effort to incorporate partial functions into the Nuprl type theory has been successful [3], but a final set of rules has not yet been formulated and implemented.

For our purposes, there is a simple-minded solution to this problem which should work in all cases of practical interest. We simply use integer recursion with a (very) large integer. Thus, for A a type, a a member of

A, and f a function of type A->A, we can define fix{A}(a,f) to be an approximation to the fixed-point of f. One difference between this and a true fixed-point operator is that we must take into account the effectively bogus base case a. Using this definition it is easy to define **Repeat**:

```
Repeat(f)  ≡
fix{Term0->?Term0}(Id, λg. (f THEN g) ORELSE Id).
```

We can also directly define, using Id as the "base case", a fixed-point operator **rewrite_letrec** for rewriting functions that satisfies

```
∀α:Env. ∀F:Rewrite(α)->Rewrite(α).
        rewrite_letrec(F) in Rewrite(α).
```

As an example of the use of our combinators, we define a rewrite that does a bottom-up rewriting using f:

```
BotUp(f)  ≡  letrec g = (Sub(g) THEN Try(f)).
```

Normalization with respect to f could be accomplished with

```
Repeat (Progress (BotUp(f))).
```

A rewriting function is applied using a special tactic. The subgoals generated by the tactic are the rewritten sequent, and a subgoal to prove that the rewriting function is well-typed. This latter subgoal often requires only simple typechecking that can be handled automatically.

4.5 Other Applications

There are also two smaller implemented applications in our library. First, as an example of a procedure that works on an entire lifted sequent, there is an equality procedure that uses equalities from the lifted hypotheses, and decides whether the equality in the conclusion follows via symmetry, transitivity, and reflexivity. Secondly, there is an implementation of a version of the sorting algorithm described earlier. This algorithm normalizes expressions over a commutative monoid (a set together with a commutative associative binary operation over that set and an identity element). Given a term of the form $a_1 \cdot a_2 \cdot \ldots a_n$, where $\cdot$ is the binary operation of the monoid, and where no a_i is of the form $b \cdot c$, the algorithm "explodes" the term into the list $[a_1; \ldots; a_n]$, removes any a_i which is the identity element of the monoid, sorts the list, according to a lexicographic ordering on terms induced by an ordering on atoms taken from the current environment, and finally "implodes" the list into a right-associated term.

5 Comparison with Other Work

The idea of using a formalized metatheory to provide for user-defined extensions of an inference system has been around for some time. Some of the early work was done by Davis and Schwartz [5] and by Weyrauch [14].

The work that is closest to our own is that of Boyer and Moore on *meta-functions* [2]. Their metafunctions are similar to our rewriting functions. One of the points which most sharply distinguishes our work is that in our case the entire mechanism is constructed within the theory. An important implication is that we can abstract over environments, drastically reducing the burden of proving the correctness of theorem proving procedures. Also, we do not have, as Boyer and Moore do, one fixed way of applying proven facts about the embedded meta-theory. The user is free to extend the mechanisms; for example, to provide for the construction and application of conditional rewrites. Another difference is that the higher-order nature of the Nuprl type theory allows us to construct useful functions such as the rewrite combinators. Finally, the represented class of objects is much richer than what can be represented in the quantifier-free logic of Boyer and Moore.

Constable and Knoblock [11,10] have undertaken an independent program to unify the meta-language and object theory of Nuprl. They have shown how to represent the proof structure and tactic mechanism of Nuprl in Nuprl's type theory. This is largely complementary to the work described here. Given an implementation of their approach, in order to obtain the same functionality provided by our library, virtually all of our work would have to be duplicated, although in a more convenient setting. They do not deal with representing contexts, and do not impose the kind of structure on represented terms that allows direct construction of such procedures as rewriting functions.

6 Conclusion

There are some questions concerning the practicability of this whole reflection-based approach that cannot yet be fully answered. Some small examples have been developed, but conclusive answers will not be possible until the approach is tested in a major application. It is hoped that it will be a useful tool for the implementation of a significant portion of Bishop's book on constructive analysis [1].

One of these questions is whether encodings in Nuprl of reasoning procedures are adequately efficient. The data types these procedures operate over are not significantly different from those used in the implementation of Nuprl, so this question reduces to the open question of whether Nuprl programs in general can be made efficient. In the example given earlier involving rational arithmetic, the actual rewriting (the normalization of the term that is the application of the rewriting function to its argument) took about ten seconds. The normalization was done using Nuprl's evaluator, which employs a naive call-by-need algorithm, and with no optimization of extracted programs.

Another question is whether lifting a sequent and maintaining a lifted sequent are sufficiently easy. Currently, these operations are somewhat slow. For instance, in the example, the lifting step and the rewriting step each took a total of about two to three minutes. The main reason for this problem has to do with certain gross inefficiencies associated with Nuprl's term structure and definition mechanism. Solutions are known and awaiting implementation.

The most difficult and important question is whether it is feasible to formally verify in Nuprl significant procedures for automating reasoning. The term rewriting system gives some positive evidence, as do some of the other examples outlined above that point out the utility of the means for abstraction available in Nuprl.

Acknowledgments

Stuart Allen, David Basin, Bob Constable, and the CADE-9 referee made helpful comments on the presentation of this work. The author is especially grateful to Bob Constable for his support and encouragement.

References

[1] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, New York, 1967.

[2] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, chapter 3, Academic Press, 1981.

[3] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, IEEE, 1987.

[4] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[6] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration, Lecture Notes in Mathematics vol. 125*, pages 29–61, Springer-Verlag, New York, 1970.

[7] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*, IEEE, 1987.

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

[10] Todd B. Knoblock. *Metamathematical Extensibility in Type Theory*. PhD thesis, Cornell University, 1987.

[11] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, IEEE, 1986.

[12] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North Holland, Amsterdam, 1982.

[13] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[14] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.