Implementing Number Theory: An Experiment with Nuprl

Douglas J. Howe

TR 86-752
May 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

This paper is to appear in *The Proceedings of the 8th International Conference on Automated Deduction, Lecture Notes in Computer Science*, Springer-Verlag, 1986.

Implementing Number Theory: An Experiment with Nuprl *

Douglas J. Howe
Department of Computer Science, Upson Hall
Cornell University, Ithaca, NY 14853

Abstract

We describe the results of an experiment in which the Nuprl proof development system was used in conjunction with a collection of simple proof-assisting programs to constructively prove a substantial theorem of number theory. We believe that these results indicate the promise of an approach to reasoning about computationally meaningful mathematics by which both proof construction and the results of formal reasoning are mathematically comprehensible.

1 Introduction

In this paper we describe a step toward a high-level environment for formally proving theorems of constructive mathematics. In particular, we describe a formalized proof of the fundamental theorem of arithmetic. The aspects we will emphasize are not restricted in relevance to number theory but instead, we believe, indicate the promise of our approach for other branches of constructive mathematics. The basis for our work is the Nuprl proof development system [4], a system which combines a sequent calculus formulation of a higher-order constructive logic, with a proof system supported by a proof-editor, a definition facility, and a metalanguage in which one can write proof-assisting programs. We have constructed a set of such

*This work was supported, in part, by NSF grant no. MCS-81-04018 programs and used them to prove the well-known theorem that every positive integer has a unique factorization into a product of powers of primes. The resulting collection of definitions, theorems and proofs is mathematically understandable and very readable, and also implicitly defines correct programs which realize the constructive, or computational, content of the theorems proven.

We describe here the results of one of the first major experiments with the Nuprl system. In the process we will discuss some of the features of the system and refer to aspects of the Nuprl logic, but our main concern is to show the kind of formal reasoning that Nuprl makes possible. Central to the discussion in this paper is the structure of reasoning in Nuprl and the role played in it by *proof tactics*. The basic unit of inference in Nuprl is called a *refinement*; one constructs, via a proof editor, a tree-structured proof in a top-down style by successively refining a goal to produce subgoals. The "size" of the refinement steps is under the control of the user; one can write programs which can be used as new inference rules. If such a program can provide justification in terms of the primitive inference rules that the subgoals it generates from a given goal entail the goal, then the name of the program can appear in the proof as the rule used in the refinement step. It thus becomes possible to completely suppress much of the meticulous detail involved in a formal proof, and to construct a collection of programs which approximate the higher level steps of informal mathematics. Although the collection of tactics we used is a long way from meeting this goal, it is an indication of what can be done, and using it we were able to completely prove from scratch in a reasonable amount of time a substantial theorem of number theory.

There are many other systems for aiding in the construction of formal proofs; three such systems, AUTOMATH [3], LCF [8], and the system of Boyer and Moore [2], have been extensively used. None, however, combine the features which we believe make Nuprl a promising tool for developing formalized mathematics, although the project of Coquand and Huet [7] appears to be heading in a similar direction. One of the most important distinguishing features is the mechanism for constructing and manipulating proofs, i.e., the coupling of a high-level programming language serving as the formal system's metalanguage with a highly visual proof editor and definition mechanism. Another is the expressive power of the logic, which permits natural representations of concepts from computationally meaningful higher mathematics.

In the next section we give a short overview of some important aspects of Nuprl. That section describes work done by others at Cornell; the work described in the subsequent sections was done by the author. Section 3 contains a discussion of how the necessary concepts of number theory were formalized, and in section 4 we discuss some of the proof tactics employed and illustrate how they were used in the actual proofs. In the last section we draw conclusions from the experiment and discuss some future directions for our work.

2 Nuprl

Nuprl [4] is a proof development system developed at Cornell under the direction of Joseph Bates and Robert Constable. Its formal basis is similar to the constructive type theory of Martin-Löf [14] and is intended to be suited to the formalization of constructive mathematics. Both constructive logic and objects of constructive mathematics are represented naturally in the Nuprl theory. The implemented system provides a general definition facility so that mathematical formulas have a compact display form which approximates that of mathematics textbooks.

The basic objects of reasoning in the Nuprl theory are types and members of types. The rules of Nuprl deal with *sequents*, i.e., 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*). To assert the truth of the sequent essentially means to assert that given members x_i of the types H_i, a member of the type A can be constructed. An important point about the Nuprl rules is that they allow one to construct a member in a top-down fashion. They allow one to *refine* a goal, obtaining subgoal sequents such that a construction for the goal can be computed from constructions for the subgoals.

Space limitations prevent us from giving details concerning the Nuprl type theory; however, a few general remarks should suffice for the purpose of this paper. Nuprl has a rich set of type-building operations. In addition to such conventional type constructors as cartesian product, there are constructors whose purpose is to represent fundamental notions of constructive logic, via the ***propositions as types*** correspondence. This correspondence gives a direct translation for the usual quantifiers and propositional connectives of logic. The basic idea behind this correspondence is that we can associate a type with each formula such that the formula is (constructively) true if and only if the type associated with it has a member. To prove a statement P of constructive mathematics, then, one first translates it into the Nuprl type theory, obtaining a type T, and then attempts to prove $>>$ T by applying refinement rules until no more unproved subgoals exist. The system then can compute, or *extract*, a term t which is a member of the type T and which embodies the computational content of the theorem P. For example, if we prove in this way the formula

$$\forall x : int \;\; \exists y : int \quad where \;\; x + y = 0$$

the term extracted from the proof will be a function which takes an integer x and produces an integer y such that $x + y = 0$. The existence of such a function is the meaning of constructive truth for the formula. An important point about this translation is that it is largely transparent to the Nuprl user; we will return to this point later. We have phrased the preceding discussion in terms of constructive mathematics. However, one can also view Nuprl as a system for program synthesis in the spirit of [13]; one proves theorems which are program specifications, and from the proofs the system can extract proven correct programs. Applications of the Nuprl methodology to program synthesis are discussed in [4]. The Nuprl system also provides a mechanism for evaluating the programs extracted from proofs.

Proofs in Nuprl are trees where each node has associated with it a sequent and a refinement rule. The children of a node are the subgoals which result from the application of the refinement rule of the node to the sequent. The refinement rule is either a primitive inference rule, or a program written in ML [8]. Such a program is called a ***refinement tactic*** (being similar to an LCF tactic [8]), and when given a sequent as input it applies primitive inference rules and other tactics to build a proof tree with the sequent as the root. This resulting proof tree is hidden except for its unproved leaves; these become the children of the input sequent, and the name of the tactic becomes its associated rule. Tactics, then, act as derived inference rules: the Nuprl display of a node of a proof tree can show as an individual step a tactic invocation; such a higher level step is correct because of the way the type structure of ML is used. For more on the Nuprl tactic mechanism, see [6]. There is one substantial decision procedure which is not a tactic, i.e., which is part of the Nuprl system and is invoked as a primitive inference rule. This procedure is called **arith**, and it proves subgoals which follow by certain simple kinds of reasoning about the primitive relations (equality and less-than) over the integers.

The basic component of a Nuprl session is the *library*, which contains a linearly ordered collection of definitions and theorems. Proofs are stored with the theorems. One interacts with Nuprl by creating, deleting, and manipulating objects using special purpose editors.

Some of the details regarding the components of Nuprl just discussed will be given as necessary in what follows. For a complete account of the system, we refer the reader to [4]. For more on using Nuprl to develop formal mathematics, see the forthcoming report of Kreitz [12] on constructive automata theory.

3 Representation

In this section we discuss how we represent some basic concepts of elementary number theory, and then we present the statements of the main theorems and of some of the important lemmas. An important point to note here is the conciseness and readability of the Nuprl mathematical definitions and statements we have written. In what follows, when we exhibit Nuprl objects what is presented is in exactly the same form (except for small differences in white space) as would appear on the screen during a Nuprl session.

The complete self-contained library constructed for the fundamental theorem of arithmetic contains 59 definitions. Of these, 36 are for generic objects, such as the logical connectives, which would be of use in any mathematical theory built in Nuprl. There are 15 general definitions dealing with basic list and integer relations and types, and only 8 which are in any way particular to the development of this theorem.

As indicated earlier, the logical connectives of predicate calculus have direct encodings in the Nuprl type theory. An important point is that one need not be aware of these representations. The definition mechanism can be used to suppress their display, and the Nuprl rules which apply to the representations are just what one might expect for the corresponding logical notions (interpreted constructively).

The definition mechanism of Nuprl is basically a macro facility. For the definitions for logic, we use this mechanism directly. For mathematical definitions, however, we employ a level of indirection in order to achieve a kind of abstraction. As a simple example we consider the definition of **N**, the natural numbers. This can be defined in terms of the (primitive) type of integers, **Int**, using the subtype constructor. To do this, we first prove a theorem (named N_) whose statement and first refinement rule is:

```
>> U1
BY explicit intro { n:Int | 0≤n }
```

U1 is the type of all ("small") types, and we prove that it has a member by explicitly introducing the type we wish to define. The Nuprl definition for N, then, will reference this theorem instead of the actual term; this reference is made using the Nuprl term **term_of(*theorem-name*)**, which denotes the term extracted from named theorem. Thus N appearing in Nuprl text is just a display form for **term_of(N_)**, which in turn denotes {**n:Int|0≤n**}. More generally, we will use lambda abstraction; e.g., if we were to redefine integer addition in this way, we would prove a theorem **>> Int->Int->Int** by introducing λx.λy.x+y. This technique has several advantages, the most important of which is that it associates a type with each defined object. This makes it possible to construct an effective membership tactic (to be described in the next section) without which our proofs would have been unbearably tedious.

In figure 1 we show the 8 definitions particular to this theory. The name of each definition is shown followed by the statement of the theorem it is extracted from (which is the type of the object), followed by the actual object being defined. The type **Fact** has a definition which uses two previously defined types and the primitive type contructors **list** and **#** (the cartesian product constructor). Thus **Fact** is the type of all lists of pairs of integers where the first integer is at least 2 and the second is

```
Fact:
U1
({2..} # N+) list

ordered:
Fact -> U1
λl. ∀ tails h·t of l. if hh = hd(t) then h.1 < hh.1

divides:
Int -> Int -> U1
λ i n. ∃ k:Int where i*k = n

prime:
Int -> U1
λ n. 1<n & ∀ i:{2..n-1}. ¬ i|n

all_prime:
Fact -> U1
λ l. ∀ tails h·t of l. h.1 prime

exp:
Int -> N -> Int
λ m n. ind(n; n,y.0; 1; n,y.m*y)

eval:
Fact -> Int
λ l. list_ind(l; 1; h,t,v. v * h.1↑h.2 )

PrimeFact:
U1
{l:Fact | l ordered & all factors in l prime }
```

Figure 1: The main definitions.

positive; these lists will represent the factorizations of integers into products of powers. The function **eval** is used to multiply out such factorizations; its definition uses the primitive Nuprl form **list_ind** for recursion over lists and also references the definitions for integer exponentiation **exp** (whose display form uses ↑), and for projection from pairs (denoted .1 and .2). Informally, this definition of a recursive function can be read as follows. Given a list l, if l is **nil** then the value is 1; otherwise, l is **h.t** (the dot serves as the Nuprl notation for *cons*) and so first compute the value v of the function on t, then multiply it by the result of applying **exp** to the two integers comprising the pair **h**. For example, the result of computing the expression **eval(<3,2>.<4,1>.nil)** would be 36. **prime** is defined in terms of **divides** (the display form for the divides relation uses "|", and a dot is used to separate the parts of the quantified formula). The definitions of **ordered** and **all_prime** use definitions which are based on **list_ind**. The definition

∀ tails h·t of l . *P*

is a recursively defined predicate which is true for a list l if *P* is true whenever the cons of **h** and **t** is a tail (or suffix) of the list l. **if h = hd(l) then** *P* is true if l is **nil** or if l is not **nil** and *P* is true for **h** the head of l. This kind of predicate is noteworthy, since it illustrates the expressive power of Nuprl's higher order logic; types are first-class citizens of the theory, and so the same form which is used to define recursive integer-valued functions over lists can also be used to recursively define type-valued functions, and hence predicates (since we represent propositions as types), over lists. Finally, the type **PrimeFact** of all prime factorizations can be defined as a subtype of **Fact** using previous definitions (**all factors in l prime** is the display form of the definition **all_prime**, etc.). For example, the prime factorization of 12 is **<2,2>.<3,1>.nil**.

Most of the lemmas proved in this theory concerned elementary properties of the defined objects, such as the fact that the value of **eval** was always at least 1. Many of these were discovered during the course of proving the major lemmas. These major lemmas, however, were straightforward expressions of lemmas used in the informal proof which the formal proof was based on.

Three of the major lemmas express familiar properties of the integers:

```
>> ∀ i,j,p:N+ where p prime & p|i*j. p|i ∨ p|j

>> ∀ i,n:Int. i|n ∨ ¬ i|n

>> ∀ a,b:N+ where ( ∀ d:{2..}. ¬(d|a & d|b) ).
        ∃ m,n:Int. m*a + n*b = 1
```

All three of these have interesting computational content. For example, the system can extract from the first theorem a (proven correct) program which takes three positive integers i, j and p, where p is a prime dividing the product ij, and returns a value which indicates whether the prime divides i or j, along with the appropriate factor. The program extracted from the third lemma takes two relatively prime positive integers as input and returns a pair of numbers which are the coefficients of a linear combination of the inputs which equals 1.

The existence part of the main theorem is

```
>> ∀ n:{2..}. ∃ l:PrimeFact where eval(l) = n
```

and its proof is just an application of the lemma

```
>> ∀ k:N. ∀ n,i:{2..} where i≤n & n-i≤k
                          & ( ∀ d:{2..(i-1)}. ¬(d|n) ).
        ∃ l:PrimeFact where eval(l) = n
```

This lemma is a recasting of the existence part of the main theorem in a form which allows us to carry more information through the induction (the main step of the proof is to do induction on k). The **where** clause of this lemma can be viewed as a loop invariant.

Finally, we have the uniqueness half of the fundamental theorem preceded by two supporting lemmas.

```
>> ∀p1,p2,i:Int.
     p1 prime => p2 prime => p1<p2 => 0<i => ¬(p1|(p2↑i))

>> ∀l:PrimeFact. ∀ p:Int.
     p prime => (if h = hd(l) then p<h.1) => ¬ p|eval(l)

>> ∀ l1,l2:PrimeFact.
     eval(l1) = eval(l2) => l1 = l2 in Fact
```

In the immediately preceding theorem, **in Fact** is required in order to indicate that the equality relation is over the type **Fact**.

4 Proofs and Tactics

It is the combination of the proof editor, high level metalanguage, and definition mechanism which gives proofs in Nuprl their distinctive character. It provides a basis for the construction of understandable proofs where the component formulas have a form which makes their meaning apparent, and where the inference steps follow an understandable course. At the present, the construction of powerful tactics for use in formalizing mathematics is at a beginning stage, and so the degree of automation of the proving process is rather small in comparison to, say, the system of Boyer and Moore. In this section we attempt to convey the flavour of proofs in Nuprl by discussing some of the tactics used and by discussing the proof of an important lemma. First, however, we give some general information about the experiment.

All of the proofs constructed in this effort were done using only general purpose tactics. These tactics were designed beforehand, without number theory as a target, to be of use in most theorem proving efforts. The total time required to complete the library was under forty hours. This time includes all work relevant to the effort; in particular, it includes the time spent on entering definitions, on informal planning, on lemma discovery and aborted proof attempts, and on proving all the necessary results dealing with the basic arithmetic operators and relations. We estimate that at least half of this time is due to certain gross inefficiencies of the current implementation (having in part to do with the maintenance of display forms for definitions) that we believe are simple to correct and that should not be present in the next version of the system. It is interesting to constrast this figure with the approximately 8 weeks of effort required to prove the same theorem in the PLCV system [5], a natural deduction system for reasoning about PL/C programs which had powerful built-in support for arithmetical and propositional reasoning but which had no tactic mechanism or proof editor. Boyer and Moore also conducted a proof of the fundamental theorem of arithmetic, but they do not say how long the effort took (although they do say that it took only ninety seconds for the final sequence of definitions and lemmas to be checked). Thirty-four theorems were proved in our effort, most of which were simple properties of the defined objects. Much of the work went into the uniqueness theorem and a major lemma for the existence theorem. The final collection of proofs contains 879 refinement steps, most of which were entered by the author (some were automatically applied by a very primitive analogy tactic). This number might seem somewhat large, but we believe that current tactic construction efforts will quickly add large increases in power to the system.

An important point concerns the relationship between the informal proof sketch and the formal Nuprl proof. The informal proof was two pages long and fairly detailed, but no consideration was given to the type theoretic encodings or to proof obligations arising specifically because of Nuprl. However, this sketch was able to serve as a guide; the progression of refinement for the main lemmas of the library for the most part followed the informal argument. Most of the more tedious steps due to formalization occurred near the leaves of the proof trees, and these lower level tactic-generated proof obligations often suggested useful lemmas.

The most important tactic was the *autotactic*. This tactic is generally automatically applied to any unproved subgoals that result when the user invokes a refinement tactic. Only the unproved subgoals generated by the autotactic appear on the screen as children of the refinement, and so the user need never be aware of the many details handled by it. This tactic will prove subgoals which follow by certain kinds of equality and arithmetic reasoning, and also has a component called **Member** which attempts to prove *membership* subgoals, i.e., subgoals requiring proof that some term is a member of a certain type. These subgoals arise because of the nature of Nuprl's type theory; for example, one must prove the well-formedness of all formulas introduced into the proof, and these well-formedness obligations have the form of membership subgoals. Because of the number of membership subgoals which arise, and because of the uninteresting nature of the vast majority of them, Nuprl would be unusable if it were not possible to handle automatically most of the work of proving them. There is no algorithm to prove all true subgoals of this form, since, for example, this is the form of a statement that a program meets its specification. Therefore, **Member** was designed with user participation as a primary concern. Such a concern is to some extent in conflict with the other main purpose of such a tactic, which is to automate as much of the proving process as possible. However, attempting to reduce a membership goal to the simplest possible subgoals will often result in some of those subgoals being false. At present, **Member** is rather conservative, stopping whenever the next step might create a false subgoal. For example, it makes no attempt to prove the validity of an application of a partial function (i.e., a function whose domain is a subtype), since this can involve proving an arbitrary proposition. Even so, **Member** was able to prove almost all of the membership goals which arose. Usually in the cases where it failed to complete a proof, it succeeded after one user-provided step. A crucial factor in the success of this tactic was the "**term_of**" style of definition, since it provided types for defined objects.

Most of the other tactics used are of a more familiar nature. We will briefly describe a few tactics which are somewhat representative of the collection used. Particularly important were the simplification tactics; e.g., **Normalize** was used to put sequents into a kind of normal form. Also important were tactics based on pattern matching. For example, the tactic **Lemma** takes as an argument the name of a lemma and attempts to find an instance of the lemma to apply to the goal, generating as subgoals the non-trivial hypotheses of the lemma instance. There is also **Backchain**, which applies simple backchaining from the conclusion of the goal via the hypotheses of the goal. Nuprl has only one integer induction rule, so several tactics were written to emulate several other common forms of integer induction. Recursive definitions were extensively used in our library, so fairly frequent use was made of tactics which performed unfolding of recursive definitions.

Also used were forms of the LCF tactic-combining functions, such as **THENW** which applies its first argument and then applies its second argument to any remaining subgoals which are not "well-formedness" ones (i.e., subgoals requiring one to show that some term is a well-formed type — these were all handled by the autotactic). An interesting kind of tactic is exemplified by the pair **SquashElim** and **SquashIntro**. They apply to a defined construct we call a "squash", which we denote $\downarrow P$ for P a proposition. The purpose of this operator has to do with information hiding, but the interesting point is that one can use it, reasoning about it via the pair of tactics, without knowing the details of its definition. We are currently working on a general framework for this kind of abstraction mechanism.

Space limitations prevent us from giving a thorough account of the proofs contructed in this effort. Instead we focus on the proof of one of the more important lemmas. This lemma, whose statement appears in the preceding section, is the one which states that any two relatively prime positive integers have a linear combination equal to one, and its proof is somewhat representative of the other proofs in the library. The first step in the proof is to assert a form of the lemma which will give a stronger induction hypothesis. This step gives two subgoals; the nontrivial one is to prove that the new form of the lemma is true. This subgoal, along with the rule applied to it, is shown in figure 2. What is shown is part of a snapshot of the screen of a Nuprl session; window borders have been removed, but otherwise the contents of the figure are what a Nuprl user would see. The first line contains the status of the proof ("*" means that the proof is complete) and an address of the current node within the proof tree. The four major components below the first line are, from top to bottom, the goal of the node, the rule applied to it, and two subgoals generated by the rule application. The numbered vertical lists of formulas in the subgoals are the hypotheses lists. The rule used here was actually a refinement tactic corresponding to the informal step "do induction on k". `THENW` was used to chain together a tactic which stripped off one universal quantifier and made a corresponding new hypothesis, and a tactic which performed induction with a specified base case (the arguments "`1 1 'j'`" are, respectively, the hypothesis number of the variable (k here) induction is being done on, the base of the induction, and a new identifier). This simple looking refinement step actually hides 58 primitive refinements. An interesting point that this snapshot illustrates is that although we are proving that there is a program that performs a certain task, we are not (explicitly) reasoning about computational objects but instead are dealing with something more like conventional mathematics.

To convey what the rest of the proof of the relative primes lemma is like, we give an informal description of some of the other proof steps. The entire proof of the lemma required 44 refinement steps. The main step is the induction step just described. The first step in the proof of its second subgoal (the inductive case) corresponds to the informal step "suppose `a` and `b` are such that ...", and generates one subgoal, with conclusion (the part of the sequent after the >>) the existential statement from the snapshot. The next step, resulting in three subgoals, is to do a case analysis on whether **a<b**, **a=b**, or **b<a**. The first step in the proof of the first case is to apply the inductive hypothesis (numbered 4 in the snapshot) to **a** and **b-a**; this step generates three subgoals. The first of the three is to show that **b-a** is positive; this is done in one further step. The second is to show that **a** and **b-a** satisfied the **where** clause of the induction hypothesis; this is done in seven additional steps. The third is to show that supposing there is a linear combination of **a** and **b-a** equal to 1, one for **a** and **b** can be found. This requires two extra steps. The rest of the proof of the lemma is at about the same level, except for several trivial steps where lemmas about the monotonicity of less-than with respect to addition had to be explicitly applied.

We end this section with a word about the extracted program. As we mentioned earlier, each Nuprl proof implicitly defines a correct program whose specification is the statement of the theorem. The program extracted from the existence part of the fundamental theorem of arithmetic takes as input a number greater than one, and returns its factorization into primes. The program is not hopelessly inefficient; using a fairly naive interpreter and no preprocessing, it took about ten seconds (on a Symbolics 3670) to factor 100!, (a number with 158 digits).

5 Conclusions, Directions For Further Work

We have written a collection of tactics and used it within the framework of the Nuprl proof development system to construct a highly readable and mathematically comprehensible formalization of a substantial theorem of number theory. The average size of a refinement step was rather low in the resulting proofs, however. More powerful tactics will be required (and are under construction) for the more ambitious projects in formalized mathematics that we believe are possible using Nuprl. One such project is the current work of the author, to formally prove a major theorem of constructive analysis.

A significant problem we have encountered concerns the speed of refinement. There are certain gross inefficiencies present in the current system which seriously hamper theorem-proving activity. These inefficiencies can,

```
* top 1
>> ∀ k:{2..}. ∀ a,b: N+
       where a+b<k & ∀ d:{2..}. ¬(d|a&d|b).
     ∃ m,n:Int. m*a + n*b = 1

BY -- Intro THENW NonNegInductionUsing 1 1 'j'

1* 1. k:{2..}
   2. j:int
   3. j=1 in int
   >> ∀ a,b: N+
          where a+b<j & ∀ d:{2..}. ¬(d|a&d|b).
        ∃ m,n:Int. m*a + n*b = 1

2* 1. k:{2..}
   2. j:int
   3. 1<j
   4. ∀ a,b: N+
          where a+b<j-1 & ∀ d:{2..}. ¬(d|a&d|b).
        ∃ m,n:Int. m*a + n*b = 1
   >> ∀ a,b: N+
          where a+b<j & ∀ d:{2..}. ¬(d|a&d|b).
        ∃ m,n:Int. m*a + n*b = 1
```

Figure 2: A snapshot of the main induction step of the relative primes lemma.

for the most part, be corrected, but there is a more fundamental problem. As in LCF, tactics work by applying primitive inference rules, and so derived rules of inference that are encapsulated as tactics must be rejustified at each application. This is a significant obstacle to increasing tactic power; for example, term rewriting tactics are prohibitively slow if required to use the substitution rule for each individual application of a rewrite rule. A solution for this problem is to use a reflection technique, using the data types of the theory to represent classes of Nuprl terms, writing in Nuprl the desired term-rewriting programs, and providing a tactic which applies the results of these programs to the Nuprl terms being reasoned about. The approach to rewriting of Paulson [15] will be useful in this context. Also, the higher-order nature of the Nuprl logic allows kinds of abstraction which will greatly aid this approach. See [10] for more on this scheme. Also relevant here is [11], in which is described a technique for using Nuprl to reason about tactics in order to avoid, in many common cases, having to run them. Another problem we are currently addressing concerns the structure and use of developed theories. At the present, the collection of facts in a Nuprl library has little structure, and the user must invoke explicitly many of the lemmas that are required in a proof.

Acknowledgements

We wish to thank Rance Cleaveland, Bob Constable, Todd Knoblock, and Prakash Panangaden for their helpful suggestions concerning the presentation of this material.

References

[1] E. Bishop. *Foundations of Constructive Analysis.* McGraw–Hill, 1967.

[2] R. Boyer and J S. Moore. *A Computational Logic.* Academic Press, NY, 1979.

[3] N. G. deBruijn. A Survey of the Project AUTOMATH. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds., pages 589–606. Academic Press, 1980.

[4] R. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

[5] R. Constable, S. Johnson, and C. Eichenlaub. *Introduction to the PL/CV2 Programming Logic, Lecture Notes in Computer Science, vol. 135*, Springer–Verlag, Berlin, 1982.

[6] R. Constable, T. Knoblock, and J. Bates. Writing Programs that Construct Proofs. *Journal of Automated Reasoning* v.1 n.3, (1985), pages 285–326.

[7] T. Coquand & G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics.* EUROCAL 85, Linz, Austria, April 1985.

[8] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation. Lecture Notes in Computer Science, vol. 78*, Springer-Verlag, Berlin, 1979.

[9] D. Howe. *Implementing Analysis.* PhD Dissertation, Department of Computer Science, Cornell University, 1986 (expected).

[10] D. Howe *Reflected Term Rewriting In Type Theory.* Technical Report, Department of Computer Science, Cornell University, 1986 (*to appear*).

[11] T. Knoblock, R. Constable. *Formalized Metareasoning in Type Theory.* Technical Report TR 86-742, Department of Computer Science, Cornell University, 1986.

[12] C. Kreitz. *Implementing Automata Theory.* Technical Report, Department of Computer Science, Cornell University. (*in preparation*).

[13] Z. Manna, R. Waldinger. *A Deductive Approach to Program Synthesis. ACM Trans. on Prog. Lang. and Sys.* v.2 n.1 (January 1980), pp 90-121.

[14] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North Holland, Amsterdam, 1982, pages 153–175.

[15] L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming* n. 3, 1983, pp 119-149.