An Environment For Automated Reasoning About Partial Functions*

David A. Basin

87-884
November 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This research was supported in part by NSF grant DCR83-03327.

An Environment For Automated Reasoning About Partial Functions*

David A. Basin
Department of Computer Science,
Cornell University, Ithaca, NY 14853

November 24, 1987

Abstract

We report on a new environment developed and implemented inside the Nuprl type theory that facilitates proving theorems about partial functions. It is the first such automated type-theoretic account of partiality. We demonstrate that such an environment can be used effectively for proving theorems about computability and for developing partial programs with correctness proofs. This extends the well-known proofs as programs paradigm to partial functions.

1 Introduction

Over the past 20 years, research by Martin-Löf[20], Constable[7,8], Huet and Coquand[12], and others has demonstrated that constructive type theory provides a useful foundation for theorem proving and program development. The Nuprl proof system, developed at Cornell[9], has been used to demonstrate that type theory does indeed provide a rich framework for theorem proving. Howe has used Nuprl to prove the fundamental theorem of arithmetic[18] and Girard's paradox[17]. Cleaveland developed the

*This research was supported in part by NSF grant DCR83-03327

synchronization-tree model of CCS[6], and Kreitz proved theorems in constructive automata theory[19]. However, current type theories are inadequate for reasoning about partial functions. The theories of Martin-Löf and Huet/Coquand cannot represent partial functions. In Nuprl, one can approximate partial functions by considering them as total functions on subsets of their domain. However, the exact domain of convergence cannot be represented and only head normalizing functions may be extracted from Nuprl proofs. As a result, it is difficult to use these theories to prove theorems about computation and impossible to use them to development partial programs.

Recent work by Constable and Smith has established a theoretical foundation for reasoning about partial objects in type theory. In [11], they present a method of extending the logic of Nuprl to reason about nontermination. For each type T in the underlying type theory, they add a new type $\overline{T}$ consisting of terms that represent computations of elements in T. An inhabitant of $\overline{T}$ may diverge, but if it terminates, it converges to an inhabitant of T. For example, the type $\mathtt{int} \rightarrow \overline{\mathtt{int}}$ corresponds to the standard notion of a partial function space; it is inhabited by functions that take an integer argument and, if they converge on their input, return an integer.

We demonstrate that this partial type theory, along with proof assisting *tactics*, can be easily implemented within Nuprl. Once implemented, the resultant *partial type environment* is surprisingly powerful. We simultaneously gain the ability to give concise proofs of familiar theorems about computation, and we can extend the "proofs as programs" paradigm to partial program development via the type-theoretic equivalent of partial correctness reasoning. Hence, we dramatically increase the power of Nuprl, both as a theorem proving system and as a program development system.

In the next section, we show how a partial type environment can be implemented within Nuprl, given the specification of a partial type theory. In the third section, we demonstrate how we can give concise proofs of interesting theorems in this environment. In section 4, we present a new paradigm for partial program development and provide an example of its use. In the final section, we draw conclusions from our work.

2 The Environment

2.1 The Partial Type Theory

A type theory may be specified by defining the terms of the theory, an evaluation relation "$\rightarrow$" defined on closed terms, a definition of types and type equality, and a type membership relation. Constable and Smith[24] extend the Nuprl type theory to a partial type theory as follows: The terms of the partial type theory are the terms of the underlying theory (Nuprl) plus a new term $\texttt{fix}(f,x.b)$. The evaluation relationship is the old evaluation relationship augmented with the reduction shown below.

$$\texttt{fix}(f,x.b)(a) \rightarrow b[\texttt{fix}(f,x.b),a/f,x]$$

The rules for defining types and determining type membership are augmented with rules for reasoning about the partial types. They allow us, for example, to prove that a partial, or "barred", type $\overline{T}$ is a type whenever T is a type, that if an element of $\overline{T}$ converges, it converges to an inhabitant of T, and that two computations t and t' are equal in a partial type $\overline{T}$ when if either converge, they both converge to equal terms in T.

Recall[1] that proofs in Nuprl consist of trees where a goal and a *refinement rule* are associated with each node. Refinement rules are either primitive inference rules or ML programs called *refinement tactics* and they are are applied in a top down fashion to construct members of types. In the presentation of a rule, the first line contains the goal to which the rule is applied. The "$>>$" symbol is the printable equivalent of the logical turnstile, and H represents a (possibly empty) list of hypotheses. After the goal, indented lines contain the subgoals that result from the application of the refinement rule. These subgoals correspond to the children of the current node. If there are no subgoals, then that branch of the proof tree is complete.

A complete list of rules for the partial type theory may be found in [24]. Some representative ones, which we present top down, or *refinement* style, are given in figure 1. `BarIntro` is a formation rule used for proving typehood. If a subgoal is to show that $\overline{T}$ is an element of $\texttt{U}i$, then the

[1]The reader is referred to [9] for a complete description of Nuprl and refinement style theorem proving.

1. H >> $\overline{T}$ in Ui by BarIntro
   H >> T in $\overline{\text{Ui}}$
2. H >> t in $\overline{T}$ by BarCTotality
   H >> t in T
3. H >> t in! T by BarInIntro
   H >> t in T
4. H >> fix(f,x.b) in x:A->$\overline{B}$ by BarFix Ui
   H,f:(x:A->$\overline{B}$),x:A >> b in $\overline{B}$
   H>> A in Ui
   H,x:A >> B in $\overline{\text{Ul}}$

Figure 1: Sample Partial Type Rules

refinement rule BarIntro reduces the proof obligation to showing that T is an element of $\overline{\text{Ui}}$. BarCTotality provides one way of demonstrating that t is in some type $\overline{T}$, namely by showing that t is total, i.e., t in T. Constable and Smith's partial type theory comes with a termination predicate in! that allows for abstract reasoning about termination. If t converges in the type $\overline{T}$ then t in! T, meaning t is in T. By rule 3, when we can prove that t inhabits T we can can then prove t in! T. Perhaps the most interesting rule is BarFix which allows us to type partial functions. Given a functional $\lambda f.\lambda x.b$, where if f inhabits $A \rightarrow \overline{B}$ and x inhabits A, then if we can demonstrate that the body b inhabits $\overline{B}$ and that A and $\overline{B}$ are types, then, by BarFix, the fixedpoint of $\lambda f.\lambda x.b$ inhabits $A \rightarrow \overline{B}$. We will see later that this rule is also useful for proving goals via computational induction.

2.2 The Implementation

Given our partial type specification, we must add its new terms and refinement rules to the underlying type theory. One approach would be to make extensive modifications to the Nuprl source. Alternatively, we chose to implement the theory by creating an appropriately endowed Nuprl *library*, a collection of definitions and theorems, which we can then use for developing theorems in the extended theory. Our partial type library is developed in

three phases:

1. New primitive objects are defined to reflect new partial type theory syntax.

2. Library objects are created that implement the new refinement rules.

3. Support tactics are written that assist proving partial type goals.

The first phase consists of using the Nuprl *def* mechanism to define three objects at the top of our library. The operator *bar*, represented as a horizontal line over a type T, is simply defined as the string "bar" and similarly the predicate `in!` is defined as the string "in!". Less straightforward is the definition of the fixedpoint operator `fix(`$f,x.b$`)` whose defining object must have the reduction characteristic stated in the previous section. While not all terms in the λ calculus are typeable, any of them may serve as a Nuprl term. Thus, we define `fix(`$f,x.b$`)` as $Y(\lambda f.\lambda x.b)$, where Y is the fixedpoint combinator $\lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$.

For each new refinement rule in the partial type theory, in the second phase we create a theorem and an ML object. The theorem object states that the subgoals of a refinement rule imply that rule's goal. The ML object, named after the refinement rule it implements, applies the theorem object in proofs and contains checks to insure that the refinement rule is properly applied. These objects are constructed so that when they are applied to a goal, they give the illusion that a primitive inference rule has been invoked.[2]

With the refinement rules implemented, the environment is completed with the addition of a collection of proof assisting tactics. These tactics partially automate the theorem proving process and thereby relieve the user from filling-in many tedious proof details. The most important tactic implemented is a partial type `Autotactic` which is automatically applied to any unproven subgoals that arise after each refinement step. This tactic contains modules that perform type checking, propositional and arithmetic reasoning, and backchaining to prove or simplify subgoals. We extend this

[2]We do not describe these objects in detail as future versions of Nuprl will contain a context mechanism that will provide the user with a facility for implementing new rules.

tactic to automatically prove the well-formedness of most partial type expressions and perform simple kinds of reasoning about partial types. Another important tactic is `FixInd` which helps prove goals via fixedpoint induction. We shall discuss this tactic in section 4.

3 Partial Object Proofs

Our initial motivation in implementing the partial type theory was to design an environment in which we can prove abstract versions of recursion-theoretic theorems. It was our hope that, by making the Nuprl type theory more expressive, we could then prove interesting facts about recursive and recursively enumerable sets, complete sets, reducibilities, and unsolvable problems. Moreover, we hoped that the resulting proofs would be comprehensible so that those who wished to learn about recursion theory could use the environment and the proofs as an interactive textbook on the subject.

Our experience to date with the environment has been quite positive. We have, for example, used it to prove the undecidability of the halting problem, Rice's theorem, and have shown that not all recursively enumerable sets are recursive[10]. Some of these proofs involve subtle reasoning and reductions of previously proved problems. We have also begun using the environment to explore a theory of abstract fixedpoint algebras in which we prove theorems of recursion theory[1]. In this abstract setting, it is easy to err, and the environment has proved valuable as a proof checker.

In this section, we present one of the proofs developed in the environment, a proof of the undecidability of the halting problem. This proof is interesting for a number of reasons. First, it demonstrates that the partial type theory is expressive enough to construct partial functions and reason about their termination properties. Second, it demonstrates the power of partial type tactics in theorem proving. Tactics automatically type the diagonalizing combinator, prove all well-formedness subgoals, and manipulate fixedpoint terms. Finally, the proof is direct. With the exception of some equality reasoning, the refinement steps are concise and the amount of reasoning compares favorably with that found in many textbook proofs.

The undecidability of the halting problem may be stated as follows. There does not exist any total procedure h such that, when given an integer

```
* DEF bot
⊥ == fix(f,x.f(x))(0)
* DEF d
d == fix(f,x.int_eq(h(f(x));1;⊥;1))
```

Figure 2: Nuprl Definitions For Halting Problem.

computation x, h decides if x halts. Using the partial type $\overline{\texttt{int}}$ and the convergence predicate `in!` we express the halting problem in our theory as follows.

```
>> ¬(∃h:int̄→int.∀x:int̄. x in! int ⇔ h(x) = 1 in int)
```

Our proof of this statement is similar to traditional proofs found in such textbooks as [22]. It does not, however, rely on the enumerability of the partial recursive functions. Informally, our proof is as follows: Assume that a decision procedure h exists. To derive a contradiction, define a diagonal function d defined as follows.

$$d(x) = \begin{cases} \bot & \textit{if } h(d(x)) = 1 \\ 1 & \textit{otherwise} \end{cases}$$

Our Nuprl definition for d is given in figure 2. Now consider the value of $h(d(1))$. There are two cases, and as h is total, we can decide which holds. The first case is when $h(d(1)) = 1$. In this case, by the definition of d, $d(1) = \texttt{int\_eq}(h(d(1)); 1; \bot; 1)$ which evaulates to $\bot$ when $h(d(1)) = 1$ and 1 otherwise. Thus, by direct computation $d(1) = \bot$ and, by the definition of h, $h(d(1)) \neq 1$ as $\bot$ diverges. Hence, $h(d(1)) = 1 \Rightarrow h(d(1)) \neq 1$, yielding a contradiction. The other case $h(d(1)) \neq 1$ is argued similarly.

A linearized proof tree of the actual theorem is given in figure 3. Here, an expression's level of indentation indicates its position in the tree. Goals are preceded by $>>$ and their hypotheses are the numbered lines found above at lesser indentation levels. Goals are followed by refinement rules and subgoals, if any, occur below at one deeper level of indentation.

Through the use of powerful general purpose tactics, the logical structure of the proof closely follows the our informal explanation. Initially, we

```
>> ¬(∃h:int->int. ∀x:int. x in! int <=> h(x) = 1 in int)
|  BY (Intro ...)
|  1. ∃h:int->int. ∀x:int. x in! int <=> h(x) = 1 in int
|- >> void
|  |  BY elim 1 new h
|  |  2. h:int->int
|  |  3. ∀x:int. x in! int <=> h(x) = 1 in int
|  |- >> void
|  |  |  BY (Seq ['d in int->int'; 'd(1)=int_eq(h(d(1));1;⊥;1) in int'] ...)
|  |  |  4. d in int->int
|  |  |- >> d(1)=int_eq(h(d(1));1;⊥;1) in int
|  |  |  |  BY (ReduceFixConcl 'z' 'z(1)=int_eq(h(d(1));1;⊥;1) in int' ...)
|  |  |  4. d in int->int
|  |  |  5. d(1)=int_eq(h(d(1));1;⊥;1) in int
|  |  |- >> void
|  |  |  |  BY (Decide 'h(d(1))=1 in int' ...)
|  |  |  |  6. h(d(1))=1 in int
|  |  |  |- >> void
|  |  |  |  |  BY (Seq ['int_eq(h(d(1));1;⊥;1)=⊥ in int';'d(1)=⊥ in int'] ...) THEN Try
(ReduceDecisionTerm 1 true ...)
|  |  |  |  |  7. int_eq(h(d(1));1;⊥;1)=⊥ in int
|  |  |  |  |  8. d(1)=⊥ in int
|  |  |  |  |- >> void
|  |  |  |  |  |  BY (EOn '⊥' 3 ...) THEN (OnLastHyp Elim ...)
|  |  |  |  |  |  9. ⊥ in! int->h(⊥)=1 in int
|  |  |  |  |  |  10. h(⊥)=1 in int->⊥ in! int
|  |  |  |  |  |- >> h(⊥)=1 in int
|  |  |  |  |  |  |  BY (SubstForInHyp 'd(1)=⊥ in int' 6 ...)
|  |  |  |  |  |  9. ⊥ in! int->h(⊥)=1 in int
|  |  |  |  |  |  10. h(⊥)=1 in int->⊥ in! int
|  |  |  |  |  |  11. ⊥ in! int
|  |  |  |  |  |- >> void
|  |  |  |  |  |  |  BY (NoBotConv [11] ...)
|  |  |  |  6. ¬(h(d(1))=1 in int)
|  |  |  |- >> void
|  |  |  |  |  BY (Seq ['int_eq(h(d(1));1;⊥;1) = 1 in int'; 'd(1) = 1 in int'] ...) THEN
Try (ReduceDecisionTerm 1 false ...)
|  |  |  |  |  7. int_eq(h(d(1));1;⊥;1) = 1 in int
|  |  |  |  |  8. d(1) = 1 in int
|  |  |  |  |- >> void
|  |  |  |  |  |  BY (EOn '1' 3 ...) THEN OnNthLastHyp 2 Elim THEN Try (BarInIntro ...)
|  |  |  |  |  |  9. 1 in! int->h(1)=1 in int
|  |  |  |  |  |  10. h(1)=1 in int->1 in! int)
|  |  |  |  |  |  11. h(1)=1 in int
|  |  |  |  |  |- >> void
|  |  |  |  |  |  |  BY (SubstForInHyp 'd(1) = 1 in int' 6 ...)
```

Figure 3: Proof Of The Undecidability Of The Halting Problem.

apply `Intro` and `elim` refinement rules, which set up a proof by contradiction by hypothesizing the existence of the decision procedure h. Then, using a tactic `Seq`[3], we cut d into the hypotheses list and use a partial type tactic `ReduceFixConcl` to unwind the fixedpoint combinator and perform the β-reductions necessary for equality reasoning. The `Decide` tactic sets up the case analysis. In the case $h(d(1)) = 1$, we cut in and prove that $d(1) = \bot$. Then, we introduce $\bot$ into hypothesis 3, which yields that $(h(\bot) = 1) \Rightarrow (\bot$ `in! int`$)$. This allows us to substitute $d(1) = \bot$ into the hypothesis $h(d(1)) = 1$ and conclude that $\bot$ `in! int`. But as $\bot$ diverges, the tactic `NoBotconv` yields a contradiction, i.e., a proof that the empty type `void` is inhabited. Again, the other case is proven similarly.

It is interesting to compare our proof with the Boyer and Moore proof[5], the first machine verified proof of the unsolvability of the halting problem. They took 4 days to create their proof outline, which spans five pages of text, and contains 29 definitions and theorems, the final theorem being the unsolvability of the halting problem. Their proof took 75 minutes to verify on a DEC 2060. On the other hand, our type theory comes equipped with much of the computational machinery they needed to build. As a result, the development of our proof took under an hour. It consists of only two definitions, d and $\bot$, and no preliminary lemmas. Twelve refinement steps were required, each using primitive refinement rules or general purpose tactics, and each reflecting a step in our informal proof. The proof was verified by a Symbolics 3670 Lisp Machine in under 20 seconds.

4 Partial Program Development

As our partial type theory is constructive, a proof of a statement contains information on how to build a witness for the statement. For example, the constructive content of a proof of

$$>> \forall x{:}T.\exists y{:}T'.R(x,y) \qquad (*)$$

[3]`Seq` is short for sequence and is used like cut in logic to add new facts to the hypothesis list. Any sequenced fact must first be proved before it can be used. However, this is often automatically done by `Autotactic`. For a complete description of all tactics found here, see [16].

is a function that for each x in type T yields a pair: a y in T' and a proof of $R(x, y)$. This program can be automatically *extracted* from a proof with the Nuprl `term_of` operator and executed with the Nuprl *evaluator*. Thus, the type theory provides a natural way of interpreting proofs as programs[2].

In the base type theory, all extracted terms head normalize to canonical terms, which inhabit the goal of the proof from which they were extracted. This can be viewed as a type-theoretic analog of total correctness. Recall that total correctness may be formalized in terms of triples

$$\{P\}\ S\ \{Q\}$$

where S is a program and P and Q first order predicates about the states of program variables. The triple $\{P\}\ S\ \{Q\}$ is true if whenever P holds for the initial values of program variables, execution of S terminates with Q satisfying the final values of the program variables[13]. Analogously, in Nuprl, given a proof P with goal G, we can form the pair

$$\texttt{term\_of}(P)\ \{G\}$$

and the program `term_of(`P`)` is said to be *type-theoretically totally correct* when `term_of(`P`)` inhabits G. As a proof P in Nuprl is a demonstration of type inhabitation where `term_of(`P`)` is the inhabiting object, all programs developed are guaranteed to be correct in the above sense.

This correctness guarantee can be used for program development when the specification is given by $(*)$. Here the type specifies a functional relationship of y on x given by the predicate R. As our type theory embodies the *propositions-as-types* principle[15], we can reflect the logical structure of any desired specification directly into the goal. This technique can be used to express program specifications almost identical to those given by standard Hoare postconditions. For example, if one wishes to develop a sorting program, the specification G could be defined as

```
>> ∀x:int list.∃y:int list.Perm(x,y) & Ordered(y)
```

where `Perm` and `Ordered` are definitions stating that y is a permutation of x and y is ordered smallest to largest. This goal would be proved in Nuprl by showing that given an unordered list x, we can find an ordered list y that is a

permutation of x. The extracted function would be a sorting program. This approach to program development has been used by Howe[16] and others to develop saddleback search, quicksort, and integer factoring programs.

Total correctness is a strong requirement. As the termination of some programs cannot be proved, the weaker requirement of partial correctness [14] is often needed to prove that a program meets its specification. That is, given a precondition P, a program S, and a postcondition Q, then a triple

$$P\ \{S\}\ Q$$

is partially correct if when S is started with its program variables satisfying P and it either fails to terminate, or it terminates with its variables satisfying Q. Analogously, given a proof P with goal G, we call a pair

$$\{\texttt{term\_of}(P)\}\ G$$

type-theoretically partially correct when $\texttt{term\_of}(P)$ either diverges or converges to a value inhabiting G.

By extending the Nuprl type theory to a partial type theory, we extend our ability to automatically synthesize programs meeting type-theoretic total correctness specifications to those meeting type-theoretic partial correctness specifications. This follows from the semantics given to the partial types. They guarantee that if an object in $\overline{T}$ converges, then it inhabits the underlying type T. Hence, by placing a bar over all or part of the goal, we attain a partial correctness specification. For example, if we are trying to extract a partial function in $T \rightarrow \overline{T'}$, then instead of stating a specification in the form of $(*)$, our goal becomes

$$>>\ \forall \texttt{x:T}.\ \overline{\exists \texttt{y:T'}.\ \texttt{R(x,y)}}.$$

The extracted object from a proof P of the above statement will be a function that takes an x in T, and, if it converges, produces a witness y in T' such that `R(x,y)`. Thus, just as in the case of total correctness, the extracted program is guaranteed to meet its type-theoretic partial correctness specification.

As an example, consider a simple program that returns the integer square root y of an integer x. Our partial correctness specification for

this program is the following.

$$\texttt{>> } \forall \texttt{x:int}.\overline{\exists \texttt{y:int where y*y = x}} \qquad (**)$$

Such a program could be developed in a total type theory only by altering the specification; for example, finding the largest y such that $y * y \leq x$. However, in our extended theory, we can prove the partial specification and extract a proof from it that computes the square root of integers that are perfect squares.

In figure 4, we give our proof of $(**)$. The highlight of this proof is the way in which fixedpoint induction is used to prove partial type inhabitation. Here, the **BarFix** refinement rule provides a not necessarily well-founded induction principle which can be used to prove the existence of a partial function meeting our correctness specification. A tactic **FixInd** is used that applies **BarFix** to the goal and manipulates the resulting subgoals to set up the inductive proof.

The proof begins with the tactic **Intro** which moves x, the integer whose square root we seek, into the hypothesis list. The goal is now of the form $\overline{P}$ where P is

$$\exists \texttt{y:int where y*y = x}.$$

To make an induction argument go through, we cut in something stronger than $\overline{P}$, namely $\forall \texttt{z:int}.\,\overline{P}$. Under the propositions as types interpretation, this new proposition is the function type $\texttt{z:int} \rightarrow \overline{P}$ and by **BarFix** this type is inhabited by a fixedpoint term whenever $(\texttt{z:int} \rightarrow \overline{P}) \rightarrow (\texttt{z:int} \rightarrow \overline{P})$ is inhabited. But the inhabitation of the latter type is equivalent (by two applications of **Intro**) to demonstrating that $\overline{P}$ is inhabited given the hypotheses $\texttt{z:int} \rightarrow \overline{P}$ and **z:int**. Thus, the **FixInd** tactic sets up a proof that $\overline{P}$ is inhabited, giving us a function $\texttt{z:int} \rightarrow \overline{P}$ and a **z in int**. This construction is subtle and if we are not careful **Autotactic** will use these two new hypotheses to prove $\overline{P}$ by introducing $\perp$, a diverging term that trivially satisfies our specification. Instead, we prove $\overline{P}$ by case analysis on $z * z = x$. When $z * z = x$ we explicitly introduce z as our witness for $\overline{P}$. Using **BarCTotality** (rule 2, figure 1) **Autotactic** proves that z inhabits T by hypothesis 4. In the case $z * z \neq x$, we use the function $\texttt{z:int} \rightarrow \overline{P}$, which we are inductively defining, on $z + 1$. The proof is completed by applying 0 to the function we constructed, proving $\overline{P}$ and providing a starting point for our square root search.

```
>> ∀x:int. ∃y:int where y*y = x
|  BY (Intro ...)
|  1. x:int
|- >> ∃y:int where y*y = x
|  |  BY Seq ['∀z:int. ∃y:int where y*y = x']
|  |- >> ∀z:int. ∃y:int where y*y = x
|  |  |  BY FixInd
|  |  |  2. ∀z:int. ∃y:int where y*y = x
|  |  |  3. z:int
|  |  |- >> ∃y:int where y*y = x
|  |  |  |  BY (Decide 'z*z = x' ...)
|  |  |  |  4. z*z = x
|  |  |  |- >> ∃y:int where y*y = x
|  |  |  |  |  BY (ExplicitI 'z' + BarCTotality)
|  |  |  |  4. ¬(z*z = x)
|  |  |  |- >> ∃y:int where y*y = x
|  |  |  |  |  BY (EOn 'z+1' 2 ...)
|  |  2. ∀z:int. ∃y:int where y*y = x
|  |- >> ∃y:int where y*y = x
|  |  |  BY (EOn '0' 2 ...)
```

Figure 4: Square Root Proof.

The extracted program from our proof is given below.[4]

$$\lambda x.(\mathbf{fix}(\lambda f.\lambda z.\mathtt{int\_eq}(z * z; x; z; f(z+1)))(0))$$

This is a function that when given an integer x, begins at $z = 0$ and iterates through the natural numbers until it finds a z such that $z * z = x$. For perfect squares, it terminates with their square root, otherwise it diverges.

5 Conclusion

We have used the environment to develop proofs of interesting results in recursion theory and as a tool to explore new theories of computation. In these settings, the environment has proved valuable in preventing faulty reasoning and in leading to the development of readable and natural proofs. We have also used it to develop partial programs which cannot be developed in any total type theory. The proofs generating these programs are concise, as the user provides the main algorithmic ideas leaving `Autotactic` to fill in

[4]Several inner redices are β-reduced to clarify program structure.

the details. These results demonstrate that partial type environments serve as important tools for both abstract theorem proving and partial program development.

Acknowledgements

I would like to thank the following people: Robert Constable and Scott Smith for laying the foundations of the Nuprl partial type theory. Doug Howe for his over-abundant assistance in using the Nuprl system. And Stuart Allen and Nax Mendler for helpful discussions.

References

[1] David Basin. *Using Partial Types In Nuprl.* Technical Report, Cornell Universty, 1987. In preparation.

[2] Joseph L. Bates and R.L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, January 1985.

[3] Nordstrom Bengt. Programming in constructive set theory: some examples. In *Conference on Functional Programming Languages and Computer Architecture*, 1981.

[4] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[5] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, July 1984.

[6] W.R. Cleaveland II. *Type-Theoretic Models of Concurrency.* PhD thesis, Cornell, 1987.

[7] R.L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of IFIP Congress*, Ljubljana, 1971.

[8] R.L. Constable. On the theory of programming logics. In *proceedings of the 9th Annual ACM Symposium on Theory of Computing*, Boulder, Colorado, 1971.

[9] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Developement System.* Prentice Hall, 1986.

[10] R.L. Constable and S.F Smith. Computational foundations of basic recursive function theory. Pre-print.

[11] R.L. Constable and S.F. Smith. Partial objects in constructive type theory. In *Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, 1987.

[12] Thierry Coquand and Gérard Huet. A theory of constructions. 1984. Unpublished manuscript.

[13] David Gries. *The Science Of Programming.* Springer-Verlag, 1981.

[14] C.A.R Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, October 1969.

[15] W. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, Academic Press, 1980.

[16] Douglas J. Howe. *Automating Reasoning in an Implimentation of Constructive Type Theory.* PhD thesis, Cornell, 1987.

[17] Douglas J. Howe. *The Computational Behaviour of Girard's Paradox.* Technical Report 87-820, Cornell Universty, 1987.

[18] Douglas J. Howe. Implementing number theory: an experiment with Nuprl. In *8th International Conference On Automated Deduction*, 1986.

[19] Christoph Kreitz. *Constructive Automata Theory Implemented with the Nuprl Proof Development System.* Technical Report 86-779, Cornell Universty, 1986.

[20] 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.

[21] Lawrence Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Comp. J.*, 28(5), 1985.

[22] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[23] N. Shankar. *Towards Mechanical Metamathematics.* Technical Report 43, University of Texas at Austin, 1984.

[24] S.F. Smith. *The Structure Of Computation in Type Theory.* PhD thesis, Cornell, 1988. In preparation.

[25] E.G Wagner. Uniformly reflexive structures: on the nature of Gödelizations and relative computability. In *Studies In Logic and The Foundations Of Mathematics — Logic Colloquim '69*, North-Holland, 1969.