Reasoning About Functional Programs in Nuprl

Douglas J. Howe

Department of Computer Science, Cornell University
Ithaca, NY 14853, USA

Abstract. There are two ways of reasoning about functional programs in the constructive type theory of the Nuprl proof development system. Nuprl can be used in a conventional program-verification mode, in which functional programs are written in a familiar style and then proven to be correct. It can also be used in an *extraction* mode, where programs are not written explicitly, but instead are extracted from mathematical proofs. Nuprl is the only constructive type theory to support both of these approaches. These approaches are illustrated by applying Nuprl to Boyer and Moore's "majority" algorithm.

1 Introduction

A type system for a functional programming language can be *syntactic* or *semantic*. In a syntactically typed language, such as SML[1] [25], typing is a property of the syntax of expressions. Only certain combinations of language constructs are designated "well-typed", and only well-typed expressions are given a meaning. Each well-typed expression has a type which can be derived from its syntax, and evaluation of the program results in a value which is "in" the type. It is usually taken to be a requirement of syntactic typing that there be an algorithm deciding whether an expression is well-typed and has a certain type; we adopt this convention here.

In a semantically typed language, such as Lisp, typing is a property of the *meaning* of programs. Evaluation is defined for essentially any combination of constructs. A program has a type exactly if the value of the program (if it exists) is in the type. This kind of typing is usually undecidable. For example, there is no algorithm which decides whether an arbitrary Lisp program produces an integer.

Consider the expression *head([1,2,true])* which takes the head of a list consisting of two integers and a boolean. In SML, list elements must all have the same type, so this would not be a well-typed program. Even though the evaluation semantics of SML make sense for the expression, it need not be accounted for by a compiler or mathematical semantics for the language. On the other hand, the analogous program in Lisp has type integer simply because it evaluates to an integer.

Existing syntactically type functional languages are clearly more restrictive than semantically typed ones. There are useful programs that can be written in Lisp, for example, that have no analog in SML. However, syntactic type systems currently in use are in fact quite flexible and usually do not hinder natural styles of coding.

In a *constructive type theory*, which we here define to be a functional programming language with a type system that is sufficiently rich to serve as a logic for

[1] When referring to a language as functional, we mean the functional portion of the language.

specifying and reasoning about programs, the cost of having a syntactic type system is that strong restrictions are made on the kinds of programs that can be written. Consider the type

$$T \equiv x{:}N \to \{\, y{:}N \mid y^2 \leq x < (y+1)^2 \,\}.$$

This is a *dependent* version of the ordinary function type. T has as members functions which map each natural number x to a member of the type of all natural numbers y such that $y^2 \leq x < (y+1)^2$. We can consider this type to be a specification of a square-root program. A program satisfies this specification if it has T as its type. With semantic typing, in order to show a program e has type T, we need to argue that for every x of type N, evaluating the application $e(x)$ results in a number y such that $y^2 \leq x < (y+1)^2$. With syntactic typing, there must be an algorithm deciding whether e has type T. This is a strong requirement, and essentially means that e must somehow encode its own correctness proof.

Two of the most well-known constructive type theories are Martin-Löf's theory [23], which we will refer to here as CMCP, and the Calculus of Constructions [13]. The Calculus of Constructions is syntactically typed, and CMCP, although semantic in conception, places restrictions on programs similar to those of its syntactically typed descendents (such as in [27]).

Neither of these theories permits the definition of a well-typed function by ordinary recursion. For example, encoding the function

```
exp(x,n) = if n=0 then 1
           else if even(n) then square(exp(x,n/2))
           else x*exp(x,n-1)
```

in CMCP requires the use of *primitive recursion* over the natural numbers, as in

```
exp'(x,n) =
  letrec f(m) = if m=0 then λn.0
                else λn. if n=0 then 1
                         else if even(n) then square(f(m-1)(n/2))
                         else x*(f(m-1)(n-1))
  in f(n+1)(n).
```

Here we use lambda-abstraction to form functions (so $\lambda n.0$ is the function mapping n to 0). We also use some syntactic sugar for CMCP's rather unconventional syntax. The *letrec* form, although it looks like a normal recursive definition, is being used here as notation for CMCP's unfamiliar construct for primitive recursion. It locally defines a primitive recursive function *f(m)* whose recursive calls are only on *m-1*. For each m, *f(m)* returns a *function* which computes *exp(x,n)* for any $n<m$. Thus in $f(n+1)(n)$ the $n+1$ only serves to bound the number of recursive calls. A version of *exp* in the Calculus of Constructions would be quite different from *exp'*, but would be at least as far-removed from the original definition. In addition to having an unconventional form, the programs in the Calculus of Constructions, and in syntactically typed versions of CMCP (in [27] for example), would be annotated with type information.

One of the main attractions of most constructive type theories is that one need not write programs explicitly as above. Instead, one can *extract* a program from a mathematical proof. This is possible because of the *propositions-as-types* correspondence, whereby logical statements are encoded as types. For example, using this encoding, the type given above for square root programs corresponds to

$$\forall\, x{:}N.\, \exists\, y{:}N.\, y^2 \leq x < (y+1)^2.$$

A proof of this statement, using only constructive rules of inference, will implicitly give a program that belongs to the corresponding type. Thus, instead of writing a program and then showing that it satisfies a specification (that is, belongs to some type), one can treat the specification as a logical statement, prove it, and have the program automatically constructed.

This paper has two main purposes. The first is to give an introduction to the Nuprl proof development system from a functional programming perspective. The second is to illustrate an important feature of the type theory of Nuprl that has neither been described in previous publications nor been exploited in past applications of the system. In particular, although it is a descendent of CMCP and supports a similar propositions-as-types principle, it is a "fully" semantic type system and supports explicit programming in a conventional style.

One way of viewing Nuprl's type theory is that it is CMCP with Martin-Löf's informal semantics taken seriously. The informal semantics he gives in [23] starts with a fairly simple untyped programming language, and the meaning of the judgment $e \in T$, for a program e and a type expression T, is that e has a *canonical form*, or, in other words, a value under evaluation, that has the appropriate form. However, this informal semantics is not fully reflected in the inference rules. Removing this discrepancy requires some substantial changes to the semantics and rules, as we will explain.

We will not give a detailed description of Nuprl in this paper. Nuprl is a system supporting formal proofs in its constructive type theory. The features that distinguish Nuprl from other interactive theorem proving systems are its approach to display and editing of mathematical text, its type theory, and its mechanism for automating formal reasoning, which is based on the *tactic* mechanism of the LCF system [15]. The reader is referred to the Nuprl book [12] for a roughly complete account of the implemented system. Modifications made to the system since the book was published are documented in a reference manual distributed with the system. Information on how to obtain the system can be obtained from the author. A number of research papers related to Nuprl have been published. Some extensions to the type theory are described in [1, 2, 3, 6, 10, 19, 20, 24, 26]. Numerous applications of Nuprl have been made; these include [5, 8, 9, 11, 16, 17, 18, 21, 22].

In order to illustrate explicit and implicit programming in Nuprl, and to show how the system works, we focus on a simple case study in which we verify Boyer and Moore's "majority" algorithm. We first take a conventional approach, writing the program and then proving that the output is appropriately related to the input. We then show how the algorithm (or, rather, something close to it) can be extracted from a proof. We explain the features of the type theory that make these two approaches possible, and in particular point out some semantic subtleties on which the first approach depends. Finally, we make some concluding remarks.

2 The Majority Algorithm

A *poll* is an array $p[0..i-1]$ of "candidates" ($i \geq 0$). The *vote count* for a candidate c is the number k of distinct j, $0 \leq j < i$, such that $p[j] = c$. c has a *majority* if $2k > i$. The problem is to find the majority candidate if one exists. Boyer and Moore's solution [7] is a very simple algorithm: step through the poll from start to end, keeping track of a single "leading" candidate c, arbitrary initially, and the size l of his "lead", 0 initially. At each step, if the next candidate a in the poll is c, increase l by one. If $a \neq c$, decrease l by one, unless $l = 0$, in which case take a as the new c and set $l = 1$. Although the algorithm is simply described, it is not so obvious why it works. A key point is that the algorithm is allowed to return an arbitrary candidate in the case where there is no majority candidate.

Nuprl's type theory has built-in types for integers and character strings. We use the latter for the type `Cand` of candidates. We represent a poll as a function mapping integers to candidates.

We define two recursive functions: `m` computes the majority candidate together with its "lead" over the other candidates, and `count` (later displayed as `#`) computes vote counts.

```
letrec m(p,i)=
  if 0<i then
    let <c,l> = m(p,i-1) in
    if p(i-1)=c then <c,l+1>
    else if 0<l then <c,l-1>
    else <p(i-1),1>
  else <"noname",0>

letrec count(p,c,i)=
  if 0<i then
    if p(i-1)=c then 1+count(p,c,i-1)
                else count(p,c,i-1)
  else 0
```

The above programs make use of Nuprl's definition mechanism. Objects such as programs and types are entered into Nuprl using a structure editor. Instead of typing text which is then parsed, a user builds an object by calling up templates with slots to be filled in. This provides a great deal of liberty in defining new notations, since there is no need for them to be parsed. The syntax of Nuprl's base language is quite primitive. The programs above are written directly in the base language, but new notations, such as `let`, `letrec` and `if`, have been introduced for readability. The `letrec` form uses ordinary recursion, and should not be confused with the primitive-recursion form used in the definition of *exp'*.

The first task is to show that these functions return values of the right kind. `m` should return a candidate-integer pair whenever it is applied to a poll and an integer in the correct range. We can formalize this statement in Nuprl directly.

```
∀p:N->Cand. ∀i:N. m(p,i) in Cand#N
```

Here -> and # are Nuprl's type constructors for function space and Cartesian product. The type N of natural numbers is formed from the primitive integer type by using a subtype constructor. The proof of this is given in Figs. 1 and 2 as a pair of "windows" on a tree structured proof.

```
* top
>> ∀p:N->Cand. ∀i:N. m(p,i) in Cand#N

BY (Id ...) THEN
   (On `i` Induction...) THEN
   (Unwind1 ``mjrty`` 0 ...)

1* 1. p: N->Cand
   2. j6: int
   3. 0<j6
   4. m(p,j6-1) in Cand#N
   5. 0<j6
   6. p(j6-1)=fst(m(p,j6-1))
   7. snd(m(p,j6-1))+1 < 0
   >> False
```

Fig. 1. First step of the proof that m is well-typed.

```
* top 1
1. p: N->Cand
2. j6: int
3. 0<j6
4. m(p,j6-1) in Cand#N
5. 0<j6
6. p(j6-1)=fst(m(p,j6-1))
7. snd(m(p,j6-1))+1 < 0
>> False

BY (Properties ['snd(m(p,j6-1))']  ...)
```

Fig. 2. Second step.

Each window shows a part of the tree. What is displayed here is identical to what the system actually shows, except for some white space adjustment and removal of a few redundant parentheses. In the second line of a Nuprl window begins the *goal*

statement. `>>` is Nuprl's turnstile ($\vdash$), which is used to separate hypotheses from conclusions. In Fig. 1, the goal is the statement of the theorem we are proving. In Fig. 2, the goal is to prove `False` (the contradictory proposition, equivalent to $0 = 1$), under the hypotheses `1` to `7` listed above the turnstile. A hypothesis list together with a conclusion is called a *sequent*.

The text following `BY` is entered by the user and is a *tactic*, which is an ML program that reduces the goal to some (hopefully) simpler subgoals. The tactic applied in the first step of the proof (Fig. 1) is

```
(Id ...) THEN
(On `i` Induction...) THEN
(Unwind1 ``mjrty`` 0 ...).
```

It executes the informal reasoning step of "do induction on `i`, then unwind the definition of majority function". By "unwind", we mean replace an application of a recursively defined function by its definition.

The text of the tactic used in Fig. 1 corresponds quite closely to this informal step. `Unwind` unwinds an application of a recursive function. `Unwind1` applies only to the specified hypothesis, with 0 indicating the conclusion of the goal. The argument `mjrty` to `Unwind1` is the full name of the definition of `m`. The "`...`" in the tactic text signals a hidden use (hidden by Nuprl's text editor) of the *autotactic*, a tactic that attempts to clean up nasty formal details. `Id` does nothing, so the tactic `(Id ...)` simply has the effect of invoking the autotactic, which in this case just moves assumptions, such as the assumption that `p` has type `N->Cand`, into the hypothesis list. Another autotactic notation of this kind is "`...+`". This means that a stronger, but slower, version of the autotactic is used. Finally, `THEN` is used to sequentially apply two tactics.

The execution of the tactic in Fig. 1 produces a single subgoal, which is displayed in the window just below the tactic. Note that this subgoal is the goal in the window in Fig. 2. The subgoal is a detail that the autotactic was unable to take care of. It corresponds to the first branch of the inner `if` expression in the definition of `m`, and requires showing the the second component of the returned result in this case is non-negative. This follows easily from the induction hypothesis (`4`): we just need to note that `snd(m(p,j6-1))` is non-negative since it is in `N`. This is accomplished in Fig. 2 by using the tactic `Properties`, which adds as new hypotheses all the properties of the given term that follow directly from its type.

As this example shows, a Nuprl proof is tree-structured and built in a top-down manner. The theorem being proved is at the root of the tree. The proof editor displays a node of the proof tree through a window which shows a sequent, the tactic applied to it, and the subgoals produced by the tactic. The first character of the first line of a window gives the *status* of the subproof rooted at the displayed node. The `*` in Figs. 1 and 2 indicate that subproofs are *complete*, so that the goals at each node have been proven. The remainder of the first line gives an address of the node within the entire proof tree.

The statement that `count` is well-typed,

```
∀p:N->Cand. ∀c:Cand. ∀i:N. #(p,c,i) in Int,
```

is proved in exactly the same way as for `m`, except that only the first step is required.

We prove `m` correct by first proving an "invariant" lemma. The formulation of the lemma, as well as the exact form of the program `m`, is due to N. Shankar, who proved the program correct in the PVS system currently under development at SRI. We prove by induction on `i` that if `m(p,i)` returns a pair `<c,l>`, then an upper bound for twice the vote count in `p[0..i-1]` of each candidate `a` is `i+l` if `a=c` and `i-l` if `¬a=c`. The formalization of this statement is the goal in Fig. 3.

```
* top
>> ∀p:N->Cand. ∀i:N.
        let <c,l> = m(p,i) in
        ∀a:Cand. 2 * ( #(p,a,i) + (if a=c then 0 else l) ) < i+l+1

BY (New ``p j`` (On `i` Induction ...) ...)

1* 1. p: N->Cand
   >> let <c,l> = m(p,0) in
      ∀a:Cand. 2*(#(p,a,0)+(if a=c then 0 else l)) < (0+l)+1

2* 1. p: N->Cand
   2. j: int
   3. 0<j
   4. let <c,l> = m(p,j-1) in
      ∀a:Cand. 2*(#(p,a,j-1)+(if a=c then 0 else l)) < ((j-1)+l)+1
   >> let <c,l> = m(p,j) in
      ∀a:Cand. 2*(#(p,a,j)+(if a=c then 0 else l)) < (j+l)+1
```

Fig. 3. The main lemma.

The first step simply applies induction. The tactic `New` just picks some nice variable names to be used in the subproofs. The first subgoal is knocked off in one step, shown in Fig. 4.

```
* top 1
1. p: N->Cand
>> let <c,l> = m(p,0) in
   ∀a:Cand. 2*(#(p,a,0)+(if a=c then 0 else l)) < (0+l)+1

BY (Unwind ``count mjrty`` ...+)
```

Fig. 4. Base case of induction.

The induction step is shown in Fig. 5. An obvious approach to proving this goal is to unwind `m`, replacing `m(p,j)` with the body of `m`, and then do the case analysis suggested by the `if` expression in the definition of `m`. The tactic `UnwindTerm` accomplishes the first task, and `Cases` the second. The purpose of `ETerm` is as follows. When we unwind `m`, we get an expression containing

```
let <c,l> = m(p,j-1)
```

`ETerm 'm(p,j-1)'` generalizes `m(p,j-1)` to a variable, and then, since the variable has a type which is a product, splits the variable into a pair of variables (so it "eliminates" the term). This enables the simplifier (in the autotactic) to remove the `let` expression.

Each of the three subgoals generated by the case analysis in Fig 5 is proved in one step. Rather than show the entire step, which would include redisplaying the subgoal, we just show the tactics used. The first subgoal is proven by

```
(InstHyp ['a'] 7 ...)
THEN UnwindTerm '#(p,a,j)'
THEN (Cases ['a=c'] THEN Cases ['a=p(j-1)'] ...+)
```

The step here is to use the induction hypothesis on `a` and do the case analysis suggested by the `if` expressions in the instantiated induction hypothesis and in the conclusion of the goal. `InstHyp` instantiates the universally quantified formula in hypothesis `7` on the term `a`.

The proof of the second subgoal of Fig. 5 is identical to that of the first, except that `7` becomes `8`, and the proof of the third is the same as that of the second, except that we need to give a hint to the simplifier by using the tactic

```
(Assert 'l = 0' ...)
```

This simply notes as a new hypothesis the obvious fact that `l=0`; the simplifier will use this to simplify `l` to `0` wherever it occurs.

The final statement of correctness of `m` is the goal in Fig. 6. It says that if `a` is a majority candidate, then `m` computes it. The theorem is proved by instantiating the lemma we just proved, and breaking `m(p,i)` into a pair `<c,l>`. The autotactic is not smart enough to complete the proof. We help it in Fig. 7 by pointing out the case analysis suggested by the `if` expression in hypothesis `7` and then proving the resulting subgoal by doing some low-level reasoning about the monotonicity properties of integer inequalities, using the tactic

```
(Mono 7 '-' 8 THEN OnLast λi. MonoWithR i '-' 'l = l' ...)
THEN (Assert '0≤l' ...)
```

It would be straightforward to encapsulate in a tactic the operation of performing case analyses suggested by `if` expressions, and the low-level inequality reasoning can be dealt with automatically by a procedure such as Shostak's [28], (a Nuprl implementation of which has just been completed).

```
* top 2
1. p: N->Cand
2. j: int
3. 0<j
4. let <c,l> = m(p,j-1) in
   ∀a:Cand. 2*(#(p,a,j-1)+(if a=c then 0 else l)) < ((j-1)+l)+1
>> let <c,l> = m(p,j) in
   ∀a:Cand. 2*(#(p,a,j)+(if a=c then 0 else l)) < (j+l)+1

BY (New ``c l`` (UnwindTerm 'm(p,j)' THEN ETerm 'm(p,j-1)' ...) ...)
   THEN (Cases ['p(j-1)=c'; '0<l'] ...+)
        % suggested by the definition of mjrty %

1* 1. p: N->Cand
   2. j: int
   3. c: Cand
   4. l: N
   5. a: Cand
   6. p(-1+j)=c
   7. ∀a1:atom. 2*#(p,a1,-1+j)+2*(if a1=c then 0 else l) < j+l
   8. 0<j
   >> 2*#(p,a,j)+2*(if a=p(-1+j) then 0 else 1+l) < 2+(j+l)

2* 1. p: N->Cand
   2. j: int
   3. c: Cand
   4. l: N
   5. a: Cand
   6. 0<l
   7. ¬(p(-1+j)=c)
   8. ∀a1:atom. 2*#(p,a1,-1+j)+2*(if a1=c then 0 else l) < j+l
   9. 0<j
   >> 2*#(p,a,j)+2*(if a=c then 0 else -1+l) < j+l

3* 1. p: N->Cand
   2. j: int
   3. c: Cand
   4. l: N
   5. a: Cand
   6. ¬(0<l)
   7. ¬(p(-1+j)=c)
   8. ∀a1:atom. 2*#(p,a1,-1+j)+2*(if a1=c then 0 else l) < j+l
   >> 2*#(p,a,j)+2*(if a=p(-1+j) then 0 else 1) < 2+j
```

Fig. 5. Induction step.

```
* top
>> ∀p:N->Cand. ∀i:N. ∀a:Cand.
    i < 2 * #(p,a,i)  =>  a = fst(m(p,i))

BY (Id ...)
   THEN (InstLemma `invariant` ['p';'i';'a'] ...)
   THEN ((New ``c l`` ((ETerm 'm(p,i)') ...) ...+) ...+)

1* 1. p: N->Cand
   2. i: N
   3. a: Cand
   4. c: Cand
   5. l: N
   6. i < 2*#(p,a,i)
   7. 2*#(p,a,i)+2*(if a=c then 0 else l) < 1+(i+l)
   >> a=c
```

Fig. 6. Correctness of m.

```
* top 1
1. p: N->Cand
2. i: N
3. a: Cand
4. c: Cand
5. l: N
6. i < 2*#(p,a,i)
7. 2*#(p,a,i)+2*(if a=c then 0 else l) < 1+(i+l)
>> a=c

BY (Cases ['a=c'] ...+)

1* 1. p: N->Cand
   2. i: N
   3. a: Cand
   4. c: Cand
   5. l: N
   6. ¬(a=c)
   7. 2*l+2*#(p,a,i) < 1+(i+l)
```

Fig. 7. An obvious case analysis.

3 Extracting a Majority Algorithm

The proof presented in the previous section can easily be modified so that a majority vote algorithm can be extracted from it. Before we explain the features of Nuprl's type theory that make this extraction possible, we will show the modified proof and point out the steps which implicitly introduce parts of the program.

We modify the statement of the correctness theorem by replacing the statement that `m` produces appropriate output with a statement that just says that an appropriate output value exists. So, instead of

```
∀p:N->Cand. ∀i:N.
    ∀a:Cand. i < 2 * #(p,a,i)  =>  a = fst(m(p,i)),
```

we have

```
∀p:N->Cand. ∀i:N. ∃c:Cand.
    ∀a:Cand. i < 2 * #(p,a,i) => a=c.
```

To prove this, we first prove an analogous modification of the invariant lemma. The prove is again by induction. The first step is shown in Fig. 8. Although this step is the same as before, it now has the additional effect of introducing part of a program. In particular, its contribution to the program extracted from the proof of this theorem is a construct for primitive recursion over the integers; this will play the role of the ordinary recursion in the definition of `m`.

```
* top
>> ∀p:N->Cand. ∀i:N. ∃c:Cand. ∃l:N.
      ∀a:Cand. 2 * ( #(p,a,i) + (if a=c then 0 else l) ) < i+l+1

BY (New ``p j`` (On `i` Induction ...) ...)

1* 1. p: N->Cand
   >> ∃c:Cand. ∃l:N. ∀a:Cand.
         2*(#(p,a,0)+(if a=c then else l)) < (0+l)+1
2* 1. p: N->Cand
   2. j: int
   3. 0<j
   4. a19: Cand
   5. a21: N
   6. ∀a:Cand. 2*(#(p,a,j-1)+(if a=a19 then 0 else a21))
               < ((j-1)+a21)+1
   >> ∃c:Cand. ∃l:N. ∀a:Cand.
         2*(#(p,a,j)+(if a=c then 0 else l)) < (j+l)+1
```

Fig. 8. First step in modified lemma.

The first subgoal is proved as before, except that we additionally need to provide witnesses for the existential quantifiers. The goal is proved by the tactic

```
(ITerms ['"noname"';'0'] ...) THEN (Unwind ``count`` ...+)
```

where `ITerms` provides the values to be used for `c` and `l`. This step introduces the "base case" value into the program, which corresponds to last line of the definition of `m` (the `i=0` case).

The proof of the second subgoal proceeds as before, except that we first use a tactic that repairs the unfortunate choice the tactic made of `a19` and `a21` as names for the witnesses in the induction hypothesis. Fig. 9 shows the step following the repair. We do the same case analysis as before, but we additionally provide the existential witnesses for each of the three cases. The bulk of the extracted program is built in this step. The case analysis contributes a conditional expression corresponding to the inner `if` in the definition of `m`, and supplying the witnesses contributes the pairs to be returned in the branches of the conditional. We do not show the subgoals in Fig. 9 because they are almost identical to those in Fig. 5, and are proved exactly the same way.

```
* top 2 1
1. p: N->Cand
2. j: int
3. 0<j
4. c: Cand
5. l: N
6. ∀a:Cand. 2*(#(p,a,j-1)+(if a=c then 0 else l))<((j-1)+l)+1
>> ∃c:Cand. ∃l:N. ∀a:Cand.
      2*(#(p,a,j)+(if a=c then 0 else l))<(j+l)+1

BY (Cases ['p(j-1)=c'; '0<l'] ...) THENL
   [(ITerms ['c';'l+1'] ...+);
    (ITerms ['c';'l-1'] ...+);
    (ITerms ['p(j-1)';'1'] ...+)
   ]
```

Fig. 9. Induction step in modified lemma.

The first step of the proof of the modified correctness theorem is shown in Fig. 10. The step is the same as before, except that we use a tactic to explicitly point out that the candidate satisfying the theorem is the one give by the lemma. The contribution of this step to the final extracted program is to apply the program from the lemma to the arguments `p` and `i`, and to project out the first component (`c`) of the returned result.

Fig. 11 shows the program, syntactically beautified for readability, that was extracted from this theorem. The program refers to the extraction of the lemma (which

```
* top
>> ∀p:N->Cand. ∀i:N.
     ∃c:Cand. ∀a:Cand. i < 2 * #(p,a,i) => a=c

BY (Id ...) THEN
   (New ``c 1`` (InstLemma `invariant_ext` ['p';'i'] ...) ...) THEN
   (ITerm 'c' ...)

1* 1. p: N->Cand
   2. i: N
   3. c: Cand
   4. l: N
   5. ∀a:Cand. 2*(#(p,a,i)+(if a=c then 0 else l))<(i+l)+1
   6. a: Cand
   7. i<2*#(p,a,i)
   >> a=c
```

Fig. 10. Modified correctness theorem: first step.

is named `invariant_ext`); this program is also shown in the figure. Although one can recognize the elements of `m` in these programs, some parts could be simplified, and there is a large amount of what seems to be junk. The `letrec` here is notation for primitive recursion, not general recursion as in the definition of `m`. The `case` statements do a case analysis on the form of their first argument. The value of

```
case e of
inl(x) -> a
inr(x) -> b,
```

if `e` has a value of the form `inl(u)`, is `a`, where `x` takes on the value `u`, and analogously if `e` is `inr(u)`. Note that the `case` statements in the extracted program could be eliminated. The program also contains "junk", most of which involves λ-abstraction or the constant `axiom`. For example, instead of the expected pair `<"noname",0>` in the base case, we have

```
<"noname",<0,λa.axiom>>.
```

There are two sources for this clutter. First of all, the form of an extracted program is very sensitive to the structure of the relevant proofs, and also to the exact way in which relevant lemmas are stated. Developers of tactics and libraries of theorems for Nuprl have focused on the mathematics. Little attention has been paid to whether the design of the tactics or the statements of theorems yields the best programs. A more fundamental reason for the clutter is due to the nature of the program extraction feature. We will return to this point after we discuss the propositions-as-types correspondence in the next section.

An important point to keep in mind is that we can deal with an extracted program abstractly. When writing other programs, we can use it simply by referring to

```
λp λi. let <c,x>=term_of(invariant_ext)(p)(i) in
       let <y,z>=x in
       <c, λu λv. axiom>

λp λi.
  letrec f(n) =
    if n=0 then λx.<"noname",<0,λa.axiom>>;
    else
      λx.
        (let <c,x> = f(n-1)(axiom) in
         let <l,x> = x in
         case (if p(j-1)=c then inl(axiom) else inr(λx.x)) of
         inl(x) -> <c,<l+1,λa.axiom>>
         inr(x) -> case (if 0=l then inl(axiom) else inr(axiom)) of
                   inl(x) -> <c,<l-1,λa.axiom>>
                   inr(x) -> <p(j-1),<1,λa.axiom>>
        )(axiom)
  in f(i)
```

Fig. 11. Extraction from the lemma.

the name of the theorem from which it was extracted. When reasoning about the extracted program, we do not look at its implementation, but rather just use the fact that it satisfies the specification which is the statement of the theorem. Typically, we do not explicitly refer the extracted program at all. Instead, we build further programs by extraction, and it is uses of the rule to invoke previous lemmas that implicitly add previously extracted programs.

4 The Nuprl Type Theory

4.1 Propositions-as-Types

It is the propositions-as-types correspondence that is responsible for the ability to extract programs from proofs. The idea is that for each way of forming a logical statement, there is a corresponding type constructor. Thus logical statements are encoded as types. The members of the types correspond to "computational evidence" for the corresponding propositions. The truth of a logical statement is identified with the corresponding type having a member. "Truth" here is truth in a constructive sense. One way to think of constructive truth is as a refinement of ordinary (i.e. classical) truth: a statement is constructively true if it is true and in addition there is a program, or construction, that "witnesses" the truth. This characterization is philosophically incorrect, but accurate enough for a practical understanding. Table 1 shows the basic correspondence. Each element of the second column uses a built-in type constructor of Nuprl. The third column shows typical elements of the types.

The type $x{:}A \rightarrow B$ is the *dependent function type*, so-called because the type of a value of the function can depend on the argument. Values in this type have the

Table 1. The propositions-as-types correspondence.

$False$	$void$	
$A \Rightarrow B$	$A \rightarrow B$	$\lambda x.\, b$
$A \,\&\, B$	$A \# B$	$\langle a, b \rangle$
$A \vee B$	$A \mid B$	$inl(a),\ inr(b)$
$\forall\, x{:}A.\ B$	$x{:}A \rightarrow B$	$\lambda x.\, b$
$\exists\, x{:}A.\ B$	$x{:}A \,\#\, B$	$\langle a, b \rangle$

form $\lambda x.\, b$ and have the property that for every $a \in A$, $b[a/x] \in B[a/x]$. The type $x{:}A \,\#\, B$ is a dependent version of the usual Cartesian product. Values in it have the form $\langle a, b \rangle$ where $a \in A$ and $b \in B[a/x]$. $A \mid B$ is a disjoint union type, and $void$ is the empty type.

How do we represent the proposition that two objects are equal? In ordinary mathematics, which takes place in set theory, there is one single notion of equality: two objects are equal if and only if they are the same set. The situation is different in type theory because the objects are not sets but programs from an untyped programming language. Consider the programs

$$\begin{aligned} f &\equiv \lambda n.\ 0 \\ f' &\equiv \lambda n.\ \mathit{if}\ n{<}0\ \mathit{then}\ 1\ \mathit{else}\ 0 \end{aligned}$$

Let N be the type of natural numbers, and let Int be the type of all integers. Both f and f' are members of both $N \rightarrow N$ and $Int \rightarrow N$. However, they are equal when considered as members of the first type, since they both output 0 on any $n \in N$, but they are unequal when considered as members of the second type, since they differ on -1. Hence the notion of equality depends on types.

The type theory has a special type constructor to reflect this notion of equality. In particular, for any type A and members a, b of A there is a type denoted

$$a = b \ \in \ A$$

which has a single value $axiom$ as a member when a and b are equal as members of A, and which is empty otherwise. The formulas of the form a `in` A, used in the theorems that `m` and `count` are well-typed, are actually defined in terms of the equality type: a `in` A stands for $a = a \in A$.

For convenience, Nuprl also has a type that reflects integer inequality: if a and b are integers (that is, they are programs that evaluate to integers), then $a < b$ is a type which has $axiom$ as a member if (the value of) a is less than (the value of) b.

The description given above of the dependent function type is incomplete because it does not take equality into account. There is an additional restriction that a member of $x{:}A \rightarrow B$ must respect equality, mapping equal members a, a' of A to equal members of $B[a/x]$. Note that for this to make sense, we must also require that $B[a/x]$ and $B[a'/x]$ have the same members and same equality relation.

To illustrate the propositions-as-types correspondence, consider the statement of the main theorem of the previous section. When we change the logical connectives to the type constructors they stand for, the statement is as follows.

$$p{:}(N \to Cand) \to i{:}N \to c{:}Cand \,\#\, (a{:}Cand \to i < 2*\#(p,a,i) \to a = c \in Cand).$$

A member of this type is a function f that takes p and i as arguments and returns a pair whose first component is a member of $Cand$ and whose second component is a member of the type

$$a{:}Cand \to i < 2*\#(p,a,i) \;\to\; a = c \in Cand$$

The only possible member of this type (up to equality) is $\lambda a.\,\lambda v.\,axiom$, and it is a member if and only if for every $a \in Cand$, if $i < 2*\#(p,a,i)$ then a and c are equal. Thus f applied to p and i returns a pair $< c, u >$ where c is a candidate and u is "evidence" that c is a majority candidate if there is a majority candidate. It is this latter kind of uninteresting evidence which clutters up the program in Fig. 11. It seems to be unavoidable if one sticks to the subset of Nuprl similar to CMCP. For example, it is present in Backhouse's hand-crafted version of the majority algorithm in [4]. One way in which the uninteresting evidence can be eliminated is to use Nuprl's subtype constructor in place of the dependent Cartesian product. See the Nuprl book [12] for more on this approach.

The rules of Nuprl are designed so that whenever a sequent >> T has a proof, then T is a type and we can compute from the proof a member $t \in T$, called the *extraction* of the proof. A sequent with hypotheses also has an extraction, which may mention the variables declared in the hypothesis list. The extraction of a proof can be computed bottom-up since each rule of Nuprl specifies how to compute the extraction for its conclusion given extractions for its premises.

To give an idea of how the rules are designed, we look at two simplified rules for the function type. We show the rules upside-down, with the premises of the rule listed below the conclusion of the rule, in order to parallel the goal/subgoal view taken in the previous sections. We also annotate the rule with a specification of how the extraction is to be computed. The first rule is as follows.

>> $x{:}A \to B$ `ext` $\lambda x.\,b$
 $x : A$ >> B `ext` b.

In terms of the corresponding logical statements, this rule says that to prove $\forall\, x{:}A.\, B$ it suffices to prove B under the assumption that x is in A. The parts following `ext` in the rule say that if b (which may refer to x) is the extraction for a proof of the subgoal, then the extraction for the goal is $\lambda x.\,b$.

The second rule is

>> B `ext` $f(a)$
 >> $A \to B$ `ext` f
 >> A `ext` a.

In logical terms, this is modus ponens: to prove B, it suffices to prove A and $A \Rightarrow B$ for some A. If the extraction for the first subgoal is f (so $f \in A \to B$), and the

extraction for the second subgoal is a (so $a \in A$) then the extraction for the goal is $f(a)$ ($f(a) \in B$).

Each of these two rules introduces a single programming-language construct into an extraction. The first introduces λ-abstraction, and the second introduces application. Every construct in Nuprl's programming language has a corresponding rule that introduces it. As another example, the primitive-recursion form, which appears in the extracted majority program, is introduced by the rule for integer induction.

4.2 Reasoning About Explicit Programs

The ability to reason about explicit programs as we did in the proof that `m` is correct relies on some rather subtle points of Nuprl's type theory. These points are related to the induction rule(s) and the "direct computation" rules. Before discussing these, we need to say more about the relationship of equality to the inference rules.

Consider the following "explicit" version of the first rule above for function types.

$$\begin{array}{l} >> \ \lambda x.b \ \in \ A \ \rightarrow \ B \\ \quad x : A \ >> \ b \ \in \ B. \end{array}$$

This essentially says that $\lambda x.b$ is a member of $A \rightarrow B$ if for every argument it produces a value of the right type. This is a stronger rule than it might at first appear, since a member of $A \rightarrow B$ must also preserve equality. Thus we need to give a semantics to sequents such that if

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

is true, then it follows that if a and a' are equal members of A then $b[a/x]$ and $b[a'/x]$ are equal members of B. The basic idea for such a semantics is due to Martin-Löf [23], and is one of the most important innovations in the design of his type theory. The idea involves incorporating a *functionality* requirement in the semantics of sequents (our sequents correspond to the *hypothetical judgements* of [23]). Part of the meaning of a sequent is that the conclusion is functional in the hypotheses, in the sense that equal members of the hypothesis types give equal conclusions, and also that each hypothesis is functional in previous hypotheses.

To see how Nuprl's rules are used to reason about explicit programs, let us consider a simple example. Define g by

$$letrec \ g(n) = \ if \ n{=}0 \ then \ 1 \ else \ n * g(n-1).$$

To prove

$$n : N \ >> \ g(n) \ \in \ N$$

we use the rule for induction on the natural numbers.[2] The subgoals are

$$>> \ g(0) \ \in \ N$$

and

[2] Actually, Nuprl has a built-in induction rule for the integers, and the restriction to natural numbers is a derived rule.

$n : N,\ n > 0,\ g(n-1)\ \in\ N$ >> $g(n)\ \in\ N$.

Proving both of these subgoals involves using Nuprl's *direct computation* rules, which allow one to perform symbolic computation steps on parts of a sequent. For the first subgoal, we can replace $g(0)$ by the result, 1, of evaluating it. For the second subgoal, by performing some symbolic computation steps, we reduce the sequent to

$n : N,\ n > 0,\ g(n-1)\ \in\ N$ >> *(if* n=0 *then* 1 *else* $n * g(n-1)$*)* $\in\ N$.

This is easily proved using the induction hypothesis.

The direct computation rules are justified because of Nuprl's semantic typing. The soundness proof for these rules is based on results in [19]. The induction rule is surprisingly difficult to justify, and is in fact false if straightforward adaptations of Martin-Löf's definition of functionality are used. A version of functionality under which the rule is valid was discovered by Allen [2]. This version, called *pointwise functionality*, is fairly complicated, but does capture an intuitively appealing notion of functionality.

Direct computation and the induction rule can be used to derive a new induction rule which does not introduce an `ind` form into extractions, but, instead, introduces a scheme for ordinary recursion of the kind used in `m`. To illustrate this, we consider a simple example. Suppose N_k is the type of all integers $0 \le i < k$. N_k is definable in Nuprl as a subtype of the integers. A derived induction rule, in the form of a lemma, is

$$\forall P{:}N \rightarrow U.\ [\forall k{:}N.\ (\forall n{:}N_k.\ P(n)) \Rightarrow P(k)]\ \Rightarrow\ \forall n{:}N.\ P(n).$$

where U is the type of all "small" types (U itself is not "small"). We can prove the lemma in such a way that the extracted program is a scheme for general recursion. To do this, we simply supply the desired scheme in the first step of the proof. The resulting subgoal is to show that the scheme is a member of the above type.

The scheme is

$$\lambda P.\ \lambda F.\ \mu(F),$$

where

$$\mu(F)\ \equiv\ \textit{letrec f(x)=F(x)(f).}$$

We need to prove

1. $P{:}\ N\ \rightarrow\ U$
2. $F{:}\ \forall k{:}N.\ (\forall n{:}N_k.\ P(n))\ \Rightarrow\ P(k)$
3. $n{:}\ N$
>> $\mu(F)(n)\ \in\ P(n)$

The proof of this uses induction on n and is similar to the proof for g above. It uses the fact that $\mu(F)(n)$ can be rewritten to $F(n)(\mu(F))$ with some symbolic computation steps. The induction step uses hypothesis 2, which implies that if $\mu(F)(n) \in P(n)$ for all $n < k$, then $F(k)(\mu(F)) \in P(k)$.

5 Conclusion

It was Stuart Allen who first noticed that Nuprl's induction rule permitted reasoning about programs defined by general recursion. In this paper we have pursued this insight a bit further, and have provided some evidence that there is no practical reason to include cumbersome constructs like primitive recursion in the underlying programming language of Nuprl's type theory. As with Feferman's theories [14], there is a large degree of flexibility to reason about the computational behaviour of programs, and it seems clear that a Nuprl-like type theory could be based on just about any untyped functional programming language. In fact, it should be possible to use the functional portion of Standard ML [25] (discarding ML's types).

References

1. S. F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE Computer Society, 1987.
2. S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
3. S. F. Allen, R. L. Constable, D. J. Howe, and W. B. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic and Computer Science*, pages 95–107. IEEE Computer Society, June 1990.
4. R. Backhouse. Algorithm development in Martin-Löf's type theory. Technical report, University of Essex, 1984.
5. D. A. Basin and P. Delvecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989.
6. D. A. Basin and D. J. Howe. Some normalization properties of Martin-Löf's type theory, and applications. In T. Ito and A. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pages 475–494. Springer-Verlag, 1991.
7. R. Boyer and J. Moore. MJRTY, a fast majority vote algorithm. Technical report, University of Texas at Austin, 1981.
8. J. Chirimar and D. J. Howe. Implementing constructive real analysis: a preliminary report. In *Symposium on Constructivity in Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, 1991. To appear.
9. R. Constable and D. Howe. Nuprl as a general logic. In P. Odifreddi, editor, *Logic in Computer Science*, pages 77–90. Academic Press, 1990.
10. R. Constable and S. Smith. Partial objects in constructive type theory. *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 183–193, March 1987. (Cornell TR 87-822).
11. R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.
12. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
13. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

14. S. Feferman. A language and axioms for explicit mathematics. In Dold, A. and B. Eckmann, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 87–139. Springer-Verlag, 1975.
15. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
16. D. Howe. The computational behaviour of Girard's paradox. *Proc. of Second Symp. on Logic in Comp. Sci., IEEE*, pages 205–214, June 1987.
17. D. J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.
18. D. J. Howe. Computational metatheory in Nuprl. *Ninth Conference on Automated Deduction*, pages 238–257, May 1988.
19. D. J. Howe. Equality in lazy computation systems. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, June 1989.
20. D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science.* IEEE Computer Society, 1991.
21. P. B. Jackson. Developing a toolkit for floating-point hardware in the Nuprl proof development system. In *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies.* Elsevier, 1991.
22. P. B. Jackson. Nuprl and its use in circuit design. In V. Stavridou, T. Melham, and R. Boute, editors, *Theorem Provers in Circuit Design*, pages 311–336. North-Holland, 1992.
23. P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
24. P. Mendler. *Inductive Definition on Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.
25. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
26. C. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 96–107. IEEE Computer society, 1991.
27. B. Nördstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science.* Oxford Science Publications, 1990.
28. R. E. Shostak. On the SUP-INF method for proving Presburger formulas. *J. ACM*, 24(4):529–543, October 1977.

This article was processed using the LaTeX macro package with LLNCS style