Moving Proofs-As-Programs Into Practice

James L. Caldwell
NASA Ames Research Center
Computational Sciences Division
Mail Stop 269-1
Moffett Field, CA, 94035-1000
caldwell@cs.cornell.edu*

Abstract

Proofs in the Nuprl system, an implementation of a constructive type theory, yield "correct-by-construction" programs. In this paper a new methodology is presented for extracting efficient and readable programs from inductive proofs. The resulting extracted programs are in a form suitable for use in hierarchical verifications in that they are amenable to clean partial evaluation via extensions to the Nuprl rewrite system. The method is based on two elements: specifications written with careful use of the Nuprl set-type to restrict the extracts to strictly computational content; and on proofs that use induction tactics that generate extracts using familiar fixed-point combinators of the untyped lambda calculus. In this paper the methodology is described and its application is illustrated by example.

1. Introduction

The ability to extract "correct-by-construction" programs directly from proofs is the most notable feature of constructive type theories as implemented in Nuprl [1] and related systems [8, 2, 10]. However, there still is no established software engineering practice based on the extraction of programs from constructive proofs in any of the constructive systems. A key obstacle to the transition of program extraction methods into practice is the lack of a methodology for extracting clear and readable programs from formal proofs. Often, the extracted programs contain unnecessary structure, artifacts of the constructive proof, that do not contribute to the computational content of the extracted term. Researchers have addressed this problem in various constructive systems in an attempt to improve the efficiency, readability, and understanding of extracted programs [11, 5, 12].

This paper presents a methodology for specification and proof in the Nuprl system that yields clean recursive programs as extracts. The methodology for directly defining recursive functions and proving properties about them is well established in Nuprl practice. Indeed many of the pieces for the new methodology described here are already present in [5]. The approach described here makes it possible to extract programs from proofs that can be uniformly manipulated by the Nuprl system using current methodology. Although it has always been possible in Nuprl to approach program development from either side, *i.e.* via verification or extraction, the methodology reported on here allows seamless integration of the two. In some cases the explicit program is already known and then verification is the best approach. In other cases, the complexity of detail in the program makes extraction from a proof the more convenient approach.

A cornerstone of existing Nuprl verification methodology is the happy fact that recursive programs can be defined in the untyped computation system of Nuprl using standard fixed-point combinators[1]. Once a definition is defined by a general recursive program (a lambda term of Nuprl's untyped programming language) the Nuprl rewrite system is extended to include selective unfolding of the definition to allow for partial evaluation of the function when it is applied to certain well-behaved non-ground terms. For each such abstraction a well-formedness lemma is proved which asserts the abstraction inhabits the appropriate type thus showing termination. Typically, to be of most utility, any number of lemmas characterizing the abstraction with respect to other functions and operators are required. For large verifications the main theorems of a verification are proved using many lemmas, functions and operators defined and verified as just

*This work was performed by the author while a member of the formal methods group at NASA Langley Research Center. The author is currently visiting Cornell University, and can be contacted at 4116 Upson Hall, Cornell University, Ithaca, NY, 14850.

[1] The possibility of such definitions was first noticed by Stuart Allen and was subsequently put into practice by Doug Howe and Paul Jackson.

described.

Current Nuprl methodology is one of program verification in type theory; program extraction mechanisms are only applied at the topmost levels of a verification if at all. Although Nuprl offers a powerful environment for program synthesis, its application as a program synthesis tool has been relatively unexplored If the proofs-as-programs interpretation is to be accepted into practice the method must be uniformly applicable at all levels of proof development.

In following sections of the paper the Nuprl type system is introduced, the current methodology for defining functions via general recursion is outlined, and a new methodology is presented that builds on existing methods to allow the proofs-as-programs interpretation to be applied all levels of verification.

2. An Overview of the Nuprl System

The Nuprl type theory is a sequent presentation of a constructive type theory via type assignment rules. The underlying programming language is untyped and the objective of a proof is to either prove a type is inhabited, *i.e.* to show that some term (program) is a member of the type, or to show that a term inhabits a particular type. A complete presentation of the type theory can be found in the Nuprl book [1].

The Nuprl system, as distinguished from the type theory, implements a rich environment to support reasoning about and computing with the Nuprl type theory. The system implementing the type theory has evolved since publication of the book but (with a few extensions) the type theory presented there is faithfully implemented by the Nuprl system. Complete documentation is included in the Nuprl V4.2 distribution. [2]

2.1. The computation system

Nuprl *terms* include the constructs of its untyped functional programming language with additional constructs for denoting types and propositions. Terms are printed here in `typewriter` font. The Nuprl computation system provides reduction rules for a left-most outermost (lazy) evaluation strategy. The rules are implemented by the evaluator.

For terms `t` and `t'` we will write `t` $\rhd$ `t'` to indicate that `t` evaluates to `t'` under the reduction rules. In later sections we apply an extended version of the basic computation system via the rewrite facility. For terms `t` and `t'` we will write `t` $\rhd_R$ `t'` to indicate that `t` reduces to `t'` in the extended system.

[2] The Nuprl system is available from Cornell at http://www.cs.cornell.edu/Info/Projects/Nuprl/nuprl.html or by anonymous ftp from ftp.cs.cornell.edu.

As usual, the notation `t[t'/x]` denotes the term resulting from the substitution of `t'` for free occurrences of `x` in `t`. Similarly, $\mathtt{t[t_1,\cdots,t_n/x_1,\cdots,x_n]}$ denotes the simultaneous substitution of each $\mathtt{t}_i$ for each $\mathtt{x}_i$ in `t`. We will sometimes write $\bar{t}$ to denote a vector of terms or variables.

2.2. The type theory

A Nuprl type is a term `T` of the computation system with an associated transitive and symmetric relation denoted by the term $\mathtt{x=y\in T}$. This relation is known as *type membership equality* and it respects evaluation in terms `x` and `y` (it is an equivalence relation when restricted to members of `T`). A point of confusion to the tyro is that, unlike set theory, type membership equality is well-formed (is a proposition) only when `T` is a type and `x` and `y` are both elements of type `T`; if `T` is not a type or either of `x` or `y` (or both) are not elements of `T` then the term $\mathtt{x=y\in T}$ denotes nothing, it is nonsense. The term $\mathtt{x\in T}$ is an encoding of $\mathtt{x=x\in T}$ Interpreting the type membership equality relation and type membership as types is made sensible via the propositions-as-types interpretation [1, pg.29–31].

In addition to the type membership equality provided with each type, there is an equality on types. Equality of types is intensional *i.e.* type equality in Nuprl is a structural equality modulo the direct computation rules. This means that, unlike sets which enjoy extensional equality, two types may contain the same elements and share an equality relation but not be equal types. For example, although `T` and $\{\mathtt{x:T} \mid \mathtt{True}\}$ have the same members and equality relations, they are not equal types in Nuprl.

Like the related type theory of Martin-Löf [8] or the type theory of Whitehead and Russell's *Principia Mathematica*, Nuprl's type theory is predicative, supporting an unbounded cumulative hierarchy of type universes. Every universe is itself a type and every type is an element of some universe.

$\mathbb{U}\{i\}$ denotes the type *universe* where i is a polymorphic specification of universe level.The property of being a type is formally written $\mathtt{T}\in\mathbb{U}$.

$\mathbb{P}\{\mathtt{i}\}$ is a synonym for $\mathbb{U}\{\mathtt{i}\}$ and is sometimes used to emphasize the propositional side of the propositions-as-types interpretation.

An incomplete list of other Nuprl types and their members include the following:

`Void` is the *empty* type.

$\mathbb{Z}$ is the *integer* type.

`Atom` is the type whose elements are *strings*.

`T list` is the type of *lists* over type `T`.

$\mathtt{y:A\longrightarrow B[y]}$ is the *dependent function* type containing functions with domain of type `A` and range type `B[y]` where `y` is a variable possibly occurring free in `B`.

$\mathtt{x:A\times B[x]}$ is the *dependent product* type consisting of pairs `<a,b>` where $\mathtt{a\in A}$ and $\mathtt{b\in B[a/x]}$.

$A \mid B$ denotes the *disjoint union* of A and B.

rec(x.T) is the Nuprl *inductive type* constructor where x is a variable bound in term T, free occurrences of x in T denote inductively smaller elements of the type.

$\{y{:}T \mid P[y]\}$ denotes a *set type* when T is a type and P[y] is a proposition possibly containing free occurrences of y.

2.3. Logic via propositions-as-types

A constructive logic is encoded within the Nuprl type theory. The following definitions in the Nuprl V4 core_1 system library encode the logic.

$$
\begin{array}{l}
\texttt{True} == 0 \in \mathbb{Z} \\
\texttt{False} == \texttt{Void} \\
P \wedge Q == P \times Q \\
P \vee Q == P \mid Q \\
P \Rightarrow Q == P \rightarrow Q \\
\neg A == A \Rightarrow \texttt{False} \\
\exists x{:}A.\ B[x] == x{:}A \times B[x] \\
\forall x{:}A.\ B[x] == x{:}A \rightarrow B[x]
\end{array}
$$

The Nuprl tactics have been built to manipulate both the propositions and types formulation uniformly.

2.4. Judgements

Nuprl judgements are the assertions one proves in the system. They are of the following form. Nuprl judgements take one of two forms[3]:

$$x_1{:}T_1, \cdots, x_n{:}T_n \vdash S \text{ [ext s]}$$
or
$$x_1{:}T_1, \cdots, x_n{:}T_n \vdash s \in S \text{ [ext } \mathbf{Ax}]$$

where $x_1, \cdots, x_n$ are distinct variables and $T_1, \cdots, T_n$, S, and s are terms (n may be 0), every free variable of T_i is one of $x_1, \cdots, x_{i-1}$ and every free variable of S or of s is one of $x_1, \cdots, x_n$. The list $x_1{:}T_1, \cdots, x_n{:}T_n$ is called the *hypothesis list*, each $x_i{:}T_i$ a declaration (of x_i), each T_i is a *hypothesis*, S ($s \in S$) is the *consequent* or *conclusion*, the term following the keyword ext is the *extract*, and the entire form is a Nuprl *sequent*. Judgements of the second form are also called *well-formedness goals*. The computational content of well-formedness goals is trivial.

Stating the conditions under which a Nuprl sequent is deemed true are technically complicated by functionality constraints; the reader is referred to the Nuprl book [1, pg.141] for a fuller account. Somewhat informally, a judgement asserts that, assuming the hypotheses are well-formed types, and the hypotheses, conclusion and extract terms are functional in those types, then the term S is an inhabited type and the extract s is an inhabitant. The fact that the extract term s inhabits S is an artifact of the proof that S is inhabited. If S is inhabited there may be more than one inhabitant and different proofs may yield different inhabitants.

A Nuprl goal is a judgement having no hypotheses.

A Nuprl proof is a decorated tree structure having a goal as its root and where the children of each node are instances of sequents justified by the rules of the type theory. A proof of a sequent shows that the goal is both well-formed and inhabited. Given terms inhabiting the hypotheses of a rule, a proof specifies how to construct a term inhabiting the type in the conclusion of the rule; thus, proofs contain instructions for the construction of witness terms. *Extraction* is the process of constructing a witness term as specified by a proof. The extract of a completed proof of a sequent is a closed term; the extract of an incomplete proof is a term possibly containing free variables.

[3]The first form subsumes the second; there is really only one form but it is useful to make the distinction as if there were two so we do.

2.5. The Nuprl system

The Nuprl system supports construction of top-down proofs by refinement. The prover is implemented as a tactic based prover in the style of LCF [3] and is built on a base of ML. In Nuprl the proposition-as-types interpretation allows for presentations to be cloaked in either logical or more purely type-theoretic terms.

The system supports a library mechanism which provides for grouping of Nuprl objects. The status and class of an object is indicated in the library by a two character sequence preceding the name of the entry in the library. For the purposes of this paper we are concerned with display form objects, definition objects (or abstractions), theorem objects, and ML objects.

3. Recursive Function Definitions In Nuprl

In current Nuprl methodology recursive functions are directly defined by applying Curry's Y combinator [5, 7]. The Y combinator is defined in the Nuprl system library core_2 as follows:

```
*A ycomb   Y == λf.(λx.f(x x))(λx.f(x x))
```

The methodology for effective use of definitions incorporating Y depend on the rewriting system. The rewrite conversion encapsulates the fixed-point property of the Y combinator:

$$Y\ F \quad \triangleright_R \quad F\ (Y\ F)$$

where F is any term. While the Y itself can not be assigned a type in Nuprl (*i.e.* it is not well-formed); well-formedness theorems for functions defined using Y show they inhabit the appropriate type. This approach is possible because the fixed-point behavior is justified via Nuprl's direct computation rules which implicitly preserve typing [1]. Because the direct computation rules preserve typing, well-formedness goals are not generated when the conversion to unfold `Y` is applied and one is never required to assign a type to `Y` itself.

Tactic support for the methodology is described in [6, 7]

In this paper, and in Nuprl libraries developed by the author, the `Y` combinator is hidden by the more familiar `letrec` form. It is defined as follows:

$$\texttt{(letrec f}\bar{x}\texttt{ = }b\texttt{[f;}\bar{x}\texttt{])} == \texttt{Y(}\boldsymbol{\lambda}\texttt{f.(}\boldsymbol{\lambda}\bar{x}.b\texttt{[f;}\bar{x}\texttt{]))}$$

The computational behavior of an application of a `letrec` term is as follows:

$$\texttt{(letrec f }\bar{x}\texttt{ = }b\texttt{[f;}\bar{x}\texttt{])}\bar{t} \;\triangleright_R\; b\texttt{[(letrec f }\bar{x}\texttt{ = }b\texttt{[f;}\bar{x}\texttt{]),}\bar{t}\texttt{/f,}\bar{x}\texttt{]}$$

i.e. the recursive call is substituted for `f` in the term `b[f;t]` and the arguments $\bar{t}$ are substituted for the variables $\bar{x}$.

3.1. An example recursive definition

In this section a list quantification operator is developed illustrating the established methodology.

A display form object in the library provides a template for display of instances of the new operator as shown on the left side of the definition. A second library entry contains the definition or abstraction.

```
*A list_all
 ∀x∈L.P[x] ==
  (letrec list_all (L) =
     if null(L) then True
     else h::t = L in P[h] ∧ list_all(t)
     fi)(L)
```

An ML object defines the rewrite conversions used to selectively unroll occurrences of the `list_all` operator. The conversion extends the reduction system to include the following behavior:

$$\forall\texttt{x}\in\texttt{[].P[x]} \;\triangleright_R\; \texttt{True}$$
$$\forall\texttt{x}\in\texttt{h::t.P[x]} \;\triangleright_R\; \texttt{P[h]} \wedge \forall\texttt{x}\in\texttt{t.P[x]}$$

When `L` is a non-canonical list form, occurrences of `∀x∈L.P[x]` are not evaluated. The code implementing the conversion is uniform (modulo function names and term instances) for all recursive functions defined using letrec. The uniformity of the mechanism for clean unfolding terms is perhaps the main advantage of using ordinary recursive definitions.

The third library entry related to this operator definition is a well-formedness theorem characterizing its type.

```
*T list_all_wf
∀T:𝕌.∀P:T→ℙ.∀L:T List.∀x∈L. P[x]∈ℙ
```

The theorem is named following the convention used by the well-formedness tactics which search for it by name and automatically apply it when well-formedness goals are induced during proofs. It says that for appropriately typed arguments, the `list_all` operator denotes a proposition. The theorem is proved by induction on `L` and then reduction under the `Reduce` tactic extended by the conversion for unfolding occurrences of `list_all`.

The following lemma characterizes the `list_all` operator in terms of those elements of `T` that are members of the list `L`.

```
*T list_all_all_lemma
∀T:𝕌.∀P:T→ℙ.∀L:T List.∀eq:{T=₂}.∀x∈L.
  P[x] ⇔ (∀x:{x:T|↑(x(∈eq) L)}.P[x])}
```

The type of boolean equalities respecting the built-in type equality for `T` is denoted $\{\texttt{T=}_2\}$. Types having such equalities are said to be discrete. The list membership operator used here is a boolean function parameterized by a boolean equality. The `assert` operator, denoted $\uparrow(b)$, lifts a boolean to a proposition.

In this lemma, the set type is being used as a subtyping mechanism to restrict attention to elements of `T` that happen to occur in the list `L`. We shall see a distinctly different application for the set type in the next section.

4. Extraction of general recursive content

The definition and lemma just given for `list_all` does not apply the proofs-as-types interpretation. It is essentially classical program verification performed in the Nuprl type theory. In this section we specify and prove a theorem whose extract is the function used above to define the `list_all` operator.

4.1. A first specification

First, we state a theorem whose inhabitants are of the correct type. The type we are interested in is given by the well-formedness theorem for the `list_all` operator, *i.e.*

$$\mathbb{U} \rightarrow \texttt{P:(T} \rightarrow \mathbb{P}\texttt{)} \rightarrow \texttt{L:T List} \rightarrow \mathbb{P}$$

We use the characterization given by the lemma `list_all_all` as a basis for the specification of the behavior of the function.

```
 *T  list_all_exists_lemma
∀T:𝕌.∀P:T→ℙ.∀L:T List.
 ∃p:ℙ.∀eq:{T=₂}.
  p⇔(∀x:{x:T|↑(x(∈eq) L)}.P[x])
```

Under the propositions-as-types interpretation we can understand the theorem as a specification for functions of type

$$\texttt{T:}\mathbb{U} \rightarrow \texttt{P:(T} \rightarrow \mathbb{P}\texttt{)} \rightarrow \texttt{L:T List} \rightarrow \texttt{p:}\mathbb{P} \times \mathcal{T}\texttt{[p]}$$

where $\mathcal{T}\texttt{[p]}$ is the proposition

```
∀eq:{T=_2}.p ⇔ (∀x:{x:T|↑(x(∈eq)L)}.P[x])
```

The elements of the type are the terms inhabiting (proving) the proposition. In this specification, p is a proposition (an element of type $\mathbb{P}$) that is true whenever $\mathcal{T}$ [p] is inhabited.

Using the extract of this theorem we can easily define a function that is extensionally equivalent to the one we are after by taking the first projection of the result of applying it to the appropriate arguments. Thus if f is the extract of the theorem, we can easily prove

```
∀T:𝕌.∀P:T→ℙ.∀L:T List.
    (f(T)(P)(L)).1 ∈ ℙ
```

Where for any pair ⟨x,y⟩, ⟨x,y⟩.1 = x.

This is precisely the approach described in the Nuprl book [1, section 4.4] and elsewhere [10, section 21.1]. But the approach fails if we are interested in using the extract in proofs where we need efficient selective unfolding of terms and partial evaluations. The following term, shown after one step of reduction, was extracted from a natural proof of the list_all_exists_lemma.

```
λT,P,L.
(letrec f (L) =
 if null(L)
  then <True,λeq.<λ%,x.any Ax,λ%.Ax>>
  else h::t=L in
   let <p,%1>=(f(t)) in
   <P[h] ∧ p,
    λeq.<λ%1@0.let <%2,%3>=%1@0 in
     λx.let <%4,%5>=(%1(eq))  in
      let <%9,%10>=
      (ext{discrete_eq_props}(T)(eq)) in
       case
        ext{decidable_assert}(eq(x)(h))
       of inl(%14)⇒
           let <%18,%19>=(%9(x)(h))in %2
         | inr(%15)⇒%4(%3)(x),
            λ%1@0.<%1@0(h),let <%3,%4>=
           (%1(eq)) in %4(λx.%1@0(x))>>>
  fi)(L)
```

There is no obvious way to evaluate the application of this term to a non-ground list without blowing up the term size with every unfolding. This is a serious problem in practice.

One approach is to minimize the logical content by hiding most of it in a set type.

```
 *T  list_all_exists_lemma_1
∀T:𝕌.∀P:T→ℙ.∀L:T List.
 ∃p:{p:ℙ|∀eq:{T=_2}.
    p ⇔ (∀x:{x:T|↑(x(∈eq)L)}.P[x])}.True
```

Proofs of this theorem still compute pairs but the right element of the pair is the term Ax.

4.2. A Refined Specification

The Nuprl set type was used above to define a subtype, now we use it to discard the unwanted computational content carried by proofs of specifications based on the existential quantifier. The following theorem justifies the replacement of the existential quantifier with a set type.

```
 *T  exists_iff_set
∀T:𝕌.∀P:T→ℙ.
 ∃{x:T|P[x]}. True ⇔ {x:T|P[x]}
```

This leads to the following specification.

```
 *T  list_all_ext
∀T:𝕌.∀P:T→ℙ.∀L:T List.
  {p:ℙ|∀eq:{T=_2}.
       p ⇔ (∀x:{x:T|↑(x(∈eq)L)}.P[x])}
```

The proof rules for the set type are noticeably similar to those for dependent product, but the extract of the set type does not include the proof that $\mathcal{T}$ [p] holds. The type of this specification is the one we want.

```
T:𝕌 → P:(T → ℙ) → L:T List → ℙ
```

Having a statement of the theorem with the correct type and with the intended meaning we must prove it in a way that generated the extract we are most interested in. Specifically, we want to prove the theorem so that extract is a recursive function defined by letrec. We will return to the proof of the theorem list_all_ext after developing the necessary mechanism.

4.3. List Induction Extracting letrec

Induction on lists is defined by the Nuprl inference rule listElimination. The application of the rule generates the extract using the list_ind term whose computational behavior is as follows:

```
list_ind([];b;x,y,z.u)  ▷ b
list_ind(h::t;b;x,y,z.u) ▷
      u[h,t,list_ind(t;b;x,y,z.u)/x,y,z]
```

The ListInd tactic applies the rule. The goal in this section is to develop a new list induction tactic whose behavior mimics the ListInd tactic but having a recursion combinator defined using letrec as its extract.

The following theorem captures the familiar list induction principle.

```
 *T  list_ind_with_y
∀T:𝕌.∀P:T List→ℙ'.
  P[[]]⇒(∀u:T.∀v:T List.P[v]⇒P[u::v])
    ⇒ (∀M:T List. P[M])
```

Since our goal is a specific extract, to prove the induction principle we explicitly provide the witness term we are interested in.

```
λT,P,b,g.
  letrec f (L) =
    if null(L) then b
    else  h::t = L in g(h)(t)(f(t))
    fi
```

Given the witness, the remainder of the proof is a verification that the witness term does indeed inhabit the type specified by the theorem. The proof is surprisingly intricate although it is modeled on a similar induction principle developed by Howe [5] for natural numbers and having a recursion combinator defined using `Y` as its extract.

A new tactic, `ListIndY`, facilitates the application of the induction principle. `ListIndY` duplicates the behavior of the ordinary `ListInd` tactic in most contexts. Taking as argument the hypothesis number of the induction variable, the tactic constructs the induction proposition (the function `P` of type `T` $\rightarrow$ $\mathbb{P}$) and then instantiates the `list_ind_with_y` lemma. The instantiation of the lemma generates a number of well-formedness goals which are, in most contexts, easily discharged by the `Auto` tactic. Of the three remaining goals, one corresponds to the base case, the other to the induction step, and the third to the original sequent with the induction principle fully instantiated as a hypothesis. This third subgoal is discharged by an application of `HypBackchain THEN Auto` leaving only two subgoals which match those produced by the `ListInd` tactic.

Extracts of theorems proved with the `ListIndY` tactic refer indirectly to the computational content of this theorem by including the term ⌈`ext{list_ind_with_y}{i:l}`⌉. An ML object extends the reduction system to automatically unfold the extract when it is encountered by `Reduce`.

The context in which `ListIndY` does not behave as its counterpart `ListInd` is when proving well-formedness goals. The `ListIndY` tactic can not be used to show well-formedness. This is because the instantiation of the `list_ind_with_y` lemma generates well-formedness subgoals for the induction proposition, these will essentially be identical to the original well-formedness lemma. However, this is not a limitation to the methodology since well-formedness goals for extract terms are easily discharged by appeal to the proof the term is extracted from.

Application of the tactic is shown in the next section.

4.4. A proof and extract

In this section we step through the proof of `list_all_ext` until we've completed as much as is required to generate the desired extract.

Recall the statement of the theorem (displayed here as a sequent with no hypotheses.)

```
⊢ ∀T:𝕌.∀P:T→ℙ.∀L:T List.
     {p:ℙ|∀eq:{T=₂}.
          p ⇔ (∀x:{x:T|↑(x(∈eq)L)}.P[x])}
```

Stripping off the quantified variables results in the following sequent.

```
1. T: 𝕌
2. P: T→ℙ
3. L: T List
⊢ {p:ℙ|∀eq:{T=₂}.
     p ⇔ (∀x:{x:T|↑(x(∈eq)L)}.P[x])}
```

The proof is by induction on the list `L` so we apply the tactic `ListIndY (-1)` which results in two subgoals.

The first is the base case where `L` has been replaced by the empty list `[]` in the conclusion.

```
⊢ {p:ℙ|∀eq:{T=₂}.
     p ⇔ (∀x:{x:T|↑(x(∈eq)[])}.P[x])}
```

To complete the proof we must choose a witness for `p`. Noticing that `↑x(∈eq)[]` is false, we see that the right side of the iff is vacuously true and so we supply `True` as the witness for `p`. At this point the computational content on this branch of the proof is complete. The resulting subgoal is to verify the logical property that the proposition defining the set is indeed true when `True` substituted for `p`.

The subgoal for the inductive case is the following.

```
3. u: T
4. v: T List
5. {p:ℙ|∀eq:{T=₂}.
       p ⇔ (∀x:{x:T|↑(x(∈eq) v)}.P[x])}
⊢ {p:ℙ|∀eq:{T=₂}.
    p ⇔ (∀x:{x:T|↑(x(∈eq) (u::v))}.P[x])}
```

Decomposing the induction hypothesis results in the following.

```
5. p: ℙ
[6]. ∀eq:{T=₂}.
      p ⇔ (∀x:{x:T|↑(x(∈eq) v)}.P[x])
⊢ {p:ℙ|∀eq:{T=₂}.
    p ⇔ (∀x:{x:T|↑(x(∈eq)(u::v))}.P[x])}
```

Hypothesis `6` is a hidden hypothesis (denoted by the brackets) that can not be used to build computational content. But we construct the witness for the set type in the conclusion without relying on `6`. The variable `p` of hypothesis `5` corresponds the proposition that is true iff the specification holds for the list `v`. Thus, the proposition `P[u]` $\wedge$ `p` is the witness for the set type in the conclusion.

Once the witness is provided, the computational content is completed and the hidden hypotheses can be used in the verification of the resulting logical property. The soundness of the dependent_setFormation proof rule is the justification for unhiding hidden hypotheses when the witness is provided. A key to making this and other proofs like it succeed is the decomposition of hypotheses declaring set types whose logical content will be needed in the verification of the logical part of the conclusion.

Applying the Reduce tactic (in the system extended to unfold occurrences of list_ind_with_y as defined above) to the extract of the proof just given results in the following term.

```
λT,P,L.
 (letrec f (L) =
    if null(L) then True
    else h::t = L in P[h] ∧ f(t)
    fi )(L)
```

To use this term in the same way the original list_all operator is used, a display form, a definition (having the extracted term as the right hand side of the definition), and a well-formedness theorem are defined. The ML function add_extract_abs, accepts a display-form template and the name of a theorem and constructs a display form object, a definition, and a well-formedness goal, which it tries to prove. All arguments outside the scope of the application of letrec that occur within the scope of the body of the letrec are made parameters of the abstraction. In practice this approach seems to work. Note that the first argument T in the extract above does not occur in the body of the letrec and so is not included as an argument of the generated abstraction.

4.5. Some list examples

The following two list theorems have interesting extracts. The first theorem declares the existence of a list containing all sublists of its list argument.

```
*T  all_sublists_ext
∀T:𝕌.∀L:T List.
 {M:T List List|
  ∀eq:{T=₂}.∀eq1:{(T List)=₂}.∀N:T List.
    N(⊆eq)L⇒(∃N':{N':T List|N(∼eq)N'}.
      ↑N'(∈eq1)M)}
```

The following two definitions make the statement comprehensible.

```
  N(⊆eq)L == ∀x∈N.↑(x(∈eq) L)
  N(∼eq)L == N(⊆eq)L ∧ L(⊆eq)N
```

The term extracted from a straight forward induction proof provides an obvious and elegant solution.

```
λL. (letrec f L =
    if null(L) then []::[]
    else h::t = L in
     append(map(λz.(h::z);f(t)),f(t))
    fi)(L)
```

The second theorem asserts that for every two lists L and L', there is a list of pairs containing all pairings of elements from L and L'.

```
*T  all_pairs_ext
∀T:𝕌.∀L,L':T List.
 {M:(T × T) List|
  ∀eq:{T=₂}.∀eq2:{(T × T)=₂}.
   ∀x:{p:T×T|↑(p.1(∈eq)L)∧
              ↑(p.2(∈eq)L')}. ↑x(∈eq2)M}
```

The following term, extracted from the inductive proof is an elegant, if somewhat intricate, recursive solution.

```
λL,L'.
   (letrec f L =
       if null(L) then []
       else h::t = L in
         if null(L') then f(t)
         else
          append(map(λz.<h, z>;L'),f(t))
         fi
       fi)(L)
```

5. Related Work

The methodology in this paper owes much to [5]. In that paper, Howe described verification and extraction methodologies applied to Boyer-Moore's fast majority algorithm in Nuprl 3. He developed a natural number induction theorem having as its extract the recursion combinator defined by Y. Surprisingly, although Howe mentions the possibility of using the set type to clean-up the extracts, he did not do it there.

Certainly it is well known in the Nuprl community how the set type can be used to hide unwanted computational content. However, the approach is rarely applied in practice. Indeed, even in examples where the goal of the exercise is to extract computational content [1, pg.86–93] they prefer the existential quantifier to the set type and choose to project the first element of the pair.

The problem of extracting clear programs from proofs in the Calculus of Constructions has been addressed by Paulin-Mohring [9, 11]. The approach separates "computationally informative" and "non-informative" propositions by syntactic means. This provides a means of eliminating the parts of the program corresponding to the logical specification. A similar idea of separating non-computational content from computationally interesting content is implemented in the system PX [4]. But neither system can define functions by ordinary recursion and neither provides a means for proving new induction principles as we have above.

6. Conclusions and Future Work

The work reported on here was motivated by need. In large proofs, it often happens that the form of a program is well known. In that case, the existing verification methodology works well. However, when development is driven from the proof side, often, the logical specification is known but the implementation is not. In the methodology reported on here, extracted programs obtain the same status as verified programs with respect to later use in other contexts of definition and proof.

We have shown by example how to massage the statement of the theorem `list_all_exists_lemma` into a form of the correct type. The methodology has been applied to define a number of abstractions being applied in a proof of propositional intuitionistic decidability. We have developed tactics which generate proofs having `letrec` forms as their extracts.

Perhaps most surprisingly, the proof of the example theorem `list_all_ext` is *identical* to the proof for the existential version `list_all_exis<ts_lemma`. This seems to be generally true, the natural proof of the existential form is identical to the natural proof of the reformulated theorem with the set type replacing the existential quantifier. The identity of the proofs suggests that in many contexts, the existential quantifier, although it is the natural form, is the wrong one.

A number of other induction tactics have been defined which generate recursive functions as extracts. This includes an ordinary induction principle for the natural numbers, Complete Induction on the naturals, and induction principles over types defined using the Nuprl rec-type. The proofs of the induction principles are schematic to the one developed for list induction here. It is not inconceivable that these proofs could be automated to allow automatic generation of induction principles for new types defined using the Nuprl rec-type.

References

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

[2] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[3] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[4] S. Hayashi and H. Nakano. *PX: A Computational Logic*. Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[5] D. J. Howe. Reasoning about functional programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, Berlin, 1993. Springer Verlag.

[6] P. Jackson. The Nuprl proof developemnt system, version 4.2 reference manual and user's guide. Computer Science Department, Cornell University, Ithaca, N.Y. Manuscript available at
http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html, July 1995.

[7] P. B. Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra*. PhD thesis, Cornell University, 1995.

[8] P. Martin–Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. Amsterdam:North Holland, 1982.

[9] C. Mohring. Algorithm development in the Calculus of Constructions. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 84–91. IEEE, 1986.

[10] B. Nordstrom, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.

[11] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 89–104. ACM, 1989.

[12] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.