Extracting Recursion Operators in Nuprl's Type Theory

James L. Caldwell[*]

Department of Computer Science, University of Wyoming, Laramie Wyoming
`jlc@cs.uwyo.edu`

Abstract. In this paper we describe the extraction of "efficient" recursion schemes from proofs of well-founded induction principles. This is part of a larger methodology; when these well-founded induction principles are used in proofs, the structure of the program extracted from the proof is determined by the recursion scheme inhabiting the induction principle. Our development is based on Paulson's paper *Constructing recursion operators in intuitionistic type theory*, but we specifically address two possibilities raised in the conclusion of his paper: the elimination of non-computational content from the recursion schemes themselves and, the use of the `Y` combinator to allow the recursion schemes to be extracted directly from the proofs of well-founded relations.

1 Introduction

In this paper we describe the formal derivation, in Nuprl, of a number of induction principles having efficient general recursion schemes as their computational content. The development described here supports the larger Nuprl project goals of developing practical proof-based programming methodologies.

In a widely cited paper [11], Paulson derived recursion schemes from a theory of well-founded relations. In the final section of that paper, Paulson suggests some alternative approaches which would accomplish the following goals.

i.) To eliminate non-computational content from the programs extracted from proofs that use the well-founded induction principles. This makes for more natural and readable extracted programs.
ii.) To use the fixed point combinator Y as the extract of the proofs of the induction principles directly thereby avoiding the introduction of proof rules for the **wfrec** operator and obviating the need to extend the evaluator.

We apply both these techniques in the development described here.

To accomplish (i.) we use Nuprl's set and intersection types to specify the type of well-founded relations. Using Allen's [1] terminology, these are *non-propositional types*, *i.e.* their inhabitants do not contain all the computational content required for proofs of their own well-formedness. A number of authors

[*] This research was supported by NSF CCR-9985239

have rejected non-propositional types [12, 14], especially the set type, and have generally concluded that they are either difficult to use or not useful. Thompson in particular has argued that, under lazy evaluation semantics, the fact that programs extracted from constructive proofs that do not use the set type typically have large parts that are computationally insignificant does not matter. Under a lazy computation system, the computationally insignificant parts of the program will never be evaluated. We disagree completely. If program extraction is to work in practice, the extracted programs must be comprehensible. In programming practice, we care about more than just correctness, we are vitally interested in intensional properties of programs. Do we have an efficient algorithm? Complexity analysis, and program transformations are virtually impossible to apply to programs where vast portions of the text are computationally insignificant. If we cannot read and analyse the extracted program how can we tell if we have the algorithm we intended? In [4], a type theoretic development of the conflict directed back-jump search algorithm is described. The first program extracted from the specification was (by virtue of the fact that it was extracted from a proof of the specification) a "correct" search procedure, but the extracted program was *not* conflict directed back-jump. This was only discovered by analysing the extracted program. Modifications to the proof yielded the desired algorithm.

Addressing point (ii.). Nuprl admits definitions of general recursive functions via the fixed point combinator Y. This is possible because Nuprl's underlying computation system is an untyped λ calculus. This possibility for the Nuprl type system was first noted by Allen in 1984 who realized that applications of Y could be assigned a type. Based on Allen's proof, Howe [6, 7] developed the current methodology for defining and using general recursive functions in Nuprl. Jackson [8, 9] incorporated Howe's methodology into his tactics for Nuprl 4.

We will only present details of Nuprl's type theory here as necessary. A hypertext account of the type theory is available online [2] and the reader is urged to examine this document. Most of the notation used here will be familiar. With only very minor differences, the definitions and theorems presented in this paper appear as they do on the screen in the Nuprl system. It should be noted that the display form of any construct in the Nuprl system is under user control and so may differ from user to user as notational tastes often do.

2 Efficient Induction Schemes

Nuprl provides support (in the form of proof rules) for inducting over built-in types of numbers and lists. The extracts of these induction rules are primitive recursion schemes. Induction on recursive types defined via the `rec` constructor is also supported by proof rules. Induction on `rec` types extracts a `recind` form. Explicit definition of functions by `recind` has been superseded in Nuprl practice by definitions using Y.

The available forms of induction are not fixed, new forms can be added to the system by proving the new principle. The extract of the proof is a recursion scheme. Of course, any proof will have as its extract a correspondingly correct recursion scheme, but we have gone to some lengths to insure our recursion schemes are free of non-computational content. This approach differs somewhat from Paulson's where a rule for well-founded induction is added to the proof system. Here, we *prove* a relation satisfies the definition of well-foundedness and extract the recursion principle from the proof. However, there are limits on the approach which will be discussed in the final section of this paper.

2.1 Well-founded relations and recursion schemes

A type of well-founded binary relations `R` over a type `A` is defined in the Nuprl standard library as follows.

```
WellFnd{i}(A;x,y.R[x;y]) ≝
   ∀P:A⟶ℙ{i}.(∀j:A.(∀k:A.R[k;j] ⇒ P[k]) ⇒ P[j]) ⇒ ∀n:A.P[n]
```

Before describing this definition we introduce Nuprl term structure. Nuprl terms are of the form `<opid>{<parms>}(<bterms>)` where `<opid>` is the name of the operator, `<parms>` is a possibly empty, comma separated list of parameters, and `<bterms>` is a, possibly empty, semi-colon separated list of bound terms. A bound term is of the form `<bvars>.<term>` where `<bvars>` is a, possibly empty, comma separated list of variable names and `<term>` is a term. The variable names in the list `<bvars>` are bound in `<term>`. In this term language, `λx.M` is a display form for the Nuprl term `lambda(x.M)` and `∀x:T.M` is a display for the term `all(T;x.M)`.

The display form `WellFnd{i}(A;x,y.R[x; y])` indicates that the definition is parameterized by the polymorphic universe level `i` and that it takes two arguments, the first is a type and the second is the term `x,y.R[x;y]` where `x` and `y` are bound in the application `R[x;y]`. The definition specifies that a relation `R` over a type `A` is well-founded if, for every property `P` of `A` satisfying the induction hypothesis `∀j:A.(∀k:A. R[k;j] ⇒ P[k]) ⇒ P[j]`, we can show `P[n]` is true of every element `n∈A`.

The following lemma establishes that the natural numbers are well-founded over the ordinary less than relation. The extract of the proof is a recursion operator for the natural numbers [1]

```
 ⊢ WellFnd{i}(ℕ;x,y.x < y)
 Extraction:  λP,g.(letrec f(i) = g(i)(λj,p.f(j)))
```

To better understand the type of the extract, consider the goal after unfolding the definition of well-founded.

```
⊢ ∀P:ℕ⟶ℙ{i}.(∀i:ℕ.(∀j:ℕ.j < i ⇒ P[j]) ⇒ P[i]) ⇒ ∀i:ℕ.P[i]
```

In the extracted term `λP,g.(letrec f(n) = g(i)(λj,p.f(j)))`, `g` corresponds to the computational content of the induction hypothesis. In this scheme, the function `g` takes two arguments, the first being the principal argument on which

[1] For readability we display the application `Y(λf,x.M)` as `letrec f(x) = M`.

the recursion is formed, the second argument being a function inhabiting the proposition (∀j:ℕ. j < i ⇒ P[j]) *i.e.* a function which accepts some element j of type ℕ along with evidence for the proposition j < i and which produces evidence for P[j]. In this scheme, the evidence for j < i is contained in the argument p, the innermost **λ**-binding. The variable p occurs nowhere else in the term and so, clearly, does not contribute to the actual computation of P[j]; instead it is a vestige of the typing for well-founded relations. The argument p is not part of what one would ordinarily consider part of the computation, certainly no programmer would clutter his programs with an unused argument.

Before we introduce an alternative typing of well-founded relations we briefly introduce Nuprl's comprehension subtypes (set types) and intersection types and discuss decidability, stability and squash stability.

2.2 Comprehension Subtypes and Intersection Types in Nuprl

Under the propositions-as-types interpretation, the implication $\phi \Rightarrow \psi$ is just encoded as the independent function type $\phi \rightarrow \psi$. The implication is true if there is some term **λ**x.M inhabiting the type $\phi \rightarrow \psi$. The conjunction $\phi \wedge \psi$ is encoded as the Cartesian product $\psi \times \psi$. The conjunction is true if there is an pair $\langle a, b\rangle$ inhabiting $\phi \times \psi$.

Methods of generating efficient and readable extracts by the use of comprehension subtypes (as opposed to the existential) were presented in [3]. We reiterate the main points here.

x:A×B[x] is the *dependent product type* (or Sigma type ***Σ***x:A.B[x]) consisting of pairs <a,b> where a∈A and b∈B[a/x]. Two pairs <a,b> and <a',b'> are equal in x:A×B[x] when a=a'∈A and b=b'∈B[a/x]. Under the propositions as types interpretation, this is existential quantification and so is sometimes displayed as ∃x:A.B[x].

{y:T|P[y]} denotes a *comprehension subtype* (or set type) when T is a type and P[y] is a proposition possibly containing free occurrences of the variable y. Elements x of this type are elements of T such that P[x/y] is true. Equality for set types is just the equality of T restricted to those elements in the set type.

In the case of an existential, <a,p>∈∃x:T.P[x], the second element of the pair is the evidence for P[a]. In the context of program extraction from a ∀∃ theorem, p is the computational content of the verification that P[a] in fact holds and is often computationally insignificant. Notice that the inhabitants of {x:T|P[x]} are simply the projections of the first elements of the inhabitants of ∃x:T.P[x].

y:A⟶B[y] is the *dependent function type* (or Pi type ***Π***x:A.B) containing functions with domain of type A and where B[y] is a term and y is a variable possibly occurring free in B. **λ**x.M ∈ y:A⟶B[y] when a∈A, B[a/y] is a type, and M[a/x]∈B[a/y]. These are the functions whose range may depend on the element of the domain applied to. Under the propositions as types interpretation this is universal quantification and so is sometimes displayed as `∀x:A.B[x]`.

`⋂x:T.P[x]` denotes the *intersection* type. An element `a` is a member of `⋂x:T.P[x]` if for every `z∈T`, `a∈P[z/x]` (hence the name intersection type.) Informally one may think of it as a function type where the domain has been discarded.

Thus, for our purposes, intersection types can be used to type polymorphic functions *e.g.* for list functions such as `append` that do not depend on the type of elements in the list, `append∈⋂T:𝕌{i}.(T List) →(T List)`.

2.3 Decidability, the Squash Type, and Squash Stability

Decidability for an arbitrary proposition P is not constructively valid. However, for many P it is uniformly decidable (*i.e.* there is an algorithm to decide) which of P or $\neg P$ holds.

`Dec{P}` $\overset{\text{def}}{=}$ `P ∨¬P`

A *squashed type* (or proposition) is one whose computational content has been discarded. The squash operator is defined in Nuprl by a set type as follows:

`↓(T)` $\overset{\text{def}}{=}$ `{True| T}`

Thus for any type (proposition) `T`, `↓(T)` is inhabited if and only if `T` is, and furthermore, has as its only inhabitant the term `Axiom` (the sole inhabitant of the proposition `True`.) The operator is called `squash` because it identifies all inhabitants of `T` with the constant term `Axiom`.

If we can reconstruct an inhabitant of a proposition `P` simply from knowing `P` is true (*i.e.* `↓(P)` is inhabited) we say `P` is *squash stable*.

`SqStable{P}` $\overset{\text{def}}{=}$ `↓(P) ⇒ P`

Squash stability is weaker even that stability (`¬¬P⇒P`) and is related to stability in that they are equivalent for decidable propositions. Squash stability is precisely the condition that Paulson discusses in the final section of his paper.

Many propositions are squash stable. Any stable or decidable proposition is squash stable. Equality on integers and the order relations on the integers are decidable and thus are also squash stable. Type membership is squash stable as is type equality (to see this recall that membership and equality goals are only inhabited by the constant term `Axiom`, the only inhabitant of `True`.)

2.4 Squash stable well-founded relations and recursion operators

As an alternative to the first definition of well-founded relations, which contained unwanted non-computational content in the extract, we have defined a notion of well-foundedness that hides the ordering in a set type.

```
WF₀{i}(A;x,y.R[x; y]) ≝
    ∀P:A⟶ℙ{i}.(∀j:A.(∀k:{k:A| R[k;j]}.P[k]) ⇒ P[j]) ⇒ ∀n:A.P[n]
```

This type can only usefully be applied in proofs when R is squash stable, thus we call it *squash stable well-foundedness*. However, it should also be noted that for program development, the constraint that R be squash stable seems not to matter.

Since the ordering relation is hidden in the right side of a set type it can not contribute to the computational content. The following lemma that the natural numbers are squash stable well-founded has the expected extract term.

```
⊢ WF₀{i}(ℕ;x,y.x < y)
Extraction: λP,g.(letrec f(i) = (g i (λj.f j)))
```

By our own criteria for naturalness (if an argument does not appear in the body of the extracted term it should not be an argument), P should not appear in the extract. We further modify the type of squash stable well-foundedness to eliminate it using Nuprl's intersection type.

```
WF{i}(A;x,y.R[x; y]) ≝
    ∩P:A⟶ℙ{i}.(∀j:A.(∀k:{k:A| R[k;j]}.P[k]) ⇒ P[j]) ⇒ ∀n:A.P[n]
```

Now the well-foundedness of the natural numbers appears as follows.

```
⊢ WF{i}(ℕ;x,y.x < y)
Extraction: λg.(letrec f(i) = (g(i)(λj.f j)))
```

The following two theorems characterize the relationship between the definition of well-foundedness and our definition of squash stable well-foundedness. The first says that squash stable well-foundedness implies well-foundedness.

```
⊢ ∀T:𝕌. ∀R:T ⟶ T ⟶ ℙ.
     WF{i}(T;x,y.R[x;y]) ⇒ WellFnd{i}(T;x,y.R[x;y])
```

The second says that if the relation is squash stable, well-foundedness implies squash stable well-foundedness.

```
⊢ ∀T:𝕌. ∀R:T ⟶ T ⟶ ℙ.
    (∀x,y:T.  SqStable(R[x;y])) ⇒
       WellFnd{i}(T;x,y.R[x;y]) ⇒ WF{i}(T;x,y.R[x;y])
```

2.5 Measure Induction and Lexicographic Induction Schemes

Using squash stable well-foundedness we prove a measure induction principle.

```
⊢ ∩T:𝕌.∩ρ:T ⟶ ℕ.WF{i}(T;x,y.ρ(x) < ρ(y))
Extraction: λg.(letrec f(i) = (g(i)(λj.f(j))))
```

Thus, once a measure function $\boldsymbol{\rho}$: `T` $\longrightarrow \mathbb{N}$ is shown to exist we can apply well-founded induction on the type `T`. Note that the measure function $\boldsymbol{\rho}$ does not occur in the body of the extract. It is used to provide evidence for termination, but of course, termination measures are not typically included as arguments to functions.

Using a squash stable well-founded relation we are able to define an induction scheme for the lexicographic ordering of inverse images onto natural numbers.

```
⊢ ∩T:𝕌.∩T':𝕌.∩ρ:T → ℕ.∩ρ':T' → ℕ.
  WF{i}(T × T'; x,y.ρ(x.1) < ρ(y.1)
                    ∨ (ρ(x.1) = ρ(y.1) ∧ ρ'(x.2) < ρ'(y.2)))
Extraction: λg.(letrec f(i) = (g(i)(λj.f(j))))
```

3 Final Remarks

In the full paper we will show applications of these theorems to program extraction and will include other induction principles.

3.1 Remarks on the Proofs and the Limitations of the Method

The proofs having the efficient recursion schemes as their extracts would perhaps more correctly be called verifications. The extract terms are explicitly provided using the `UseWitness` tactic. After this step, the remaining proof is to show that the extract does indeed inhabit type. These proofs are surprisingly intricate. Also, proofs of well-foundedness of the lexicographic measure induction are widely published [5, 10], but they rely on the least element principle which is non-constructive. The proof used here is by nested inductions on the natural numbers.

The method of applying well-founded induction principles in proofs has a limitation; lemmas can not be used in proving well-formedness goals. This is because the instantiation of the property being proved (`P` in the definition squash stable well-foundedness) will generate a well-formedness subgoal which is equivalent to the original goal. In practice, this limitation has not been a burden since well-formedness goals are typically proved by following the structure of the type. A mechanism for incorporating derived rules into Nuprl has been designed. Once implemented, this feature will eliminate this restriction on the use of these principles.

3.2 Future Work

We have only formalized a few of Paulson's well-founded induction schemes, however, we have applied them extensively. Some of the more complex schemes that we have not formalized (yet) have been used elsewhere, notably the application of lexicographic exponentiation in the certification of Buchberger's algorithm [13].

3.3 Thanks

The author thanks the anonymous reviewers for their detailed and thoughtful comments. Suggestions that have not been addressed in this preliminary version of the paper will be fully addressed in the final paper.

References

1. S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, Ithaca, NY, 1987. TR 87-866.
2. Stuart Allen. *NuprlPrimitives - Explanations of Nuprl Primitives and Conventions.* Computer Science Department, Cornell University, Ithaca, NY. Manuscript available at `http://www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/Welcome.html`, 2001.
3. James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*, pages 10–17. IEEE Computer Society, 1997.
4. James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1-2):55–90, Feb. 2000.
5. J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper and Row, 1986.
6. D. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, Ithaca, NY, April 1988.
7. Douglas 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.
8. Paul Jackson. The Nuprl proof development 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/manual/it.html`, July 1995.
9. Paul B. Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra.* PhD thesis, Cornell University, 1995.
10. Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming: Volume II: Deductive Systems.* Addison Wesley, 1990.
11. L. C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4):325–355, 1986.
12. Anne Salvesen. On specifications, subset types and interpretation of propositions in type theory. In *Proceedings of the Workshop on Programming Logic*, pages 209–230, Båstad, Sweden, May 1989. Programming Methodology Group, University of Göteborg and Chalmers University of Technology.
13. Laurent Théry. A Certified Version of Buchberger's Algorithm. In H. Kirchner and C. Kirchner, editors, *15th International Conference on Automated Deduction*, LNAI 1421, pages 349–364, Lindau, Germany, July 5–July 10, 1998. Springer-Verlag.
14. Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.