Derivation of a Fast Integer Square Root Algorithm

Christoph Kreitz
Department of Computer Science, Cornell-University, Ithaca, NY 14853-7501
kreitz@cs.cornell.edu

Abstract

In a constructive setting, the formula $\forall n\,\exists r\ r^2{\le}n \wedge n{<}(r{+}1)^2$ specifies an algorithm for computing the integer square root r of a natural number x. A proof for this formula implicitly contains an integer square root algorithm that mirrors the way in which the formula was proven correct. In this note we describe the formal derivation of several integer square root algorithms within the Nuprl proof development system and show how efficient algorithms can be derived using advanced induction schemes.

1 Deriving a Linear Algorithm

The standard approach to proving $\forall n\,\exists r\ r^2{\le}n \wedge n{<}(r{+}1)^2$ is induction on n, which will lead to the following two proof goals

Base Case: prove $\exists r\ r^2{\le}0 \wedge 0{<}(r{+}1)^2$
Induction Step: prove $\exists r\ r^2{\le}n{+}1 \wedge n{+}1{<}(r{+}1)^2$ assuming $\exists r_n\ r^2{\le}n \wedge n{<}(r_n{+}1)^2$.

The base case can be solved by choosing $r = 0$ and using standard arithmetical reasoning to prove the resulting proof obligation $0^2{\le}0 \wedge 0{<}(0{+}1)^2$.

In the induction step, one has to analyze the root r_n. If $(r_n{+}1)^2{\le}n{+}1$, then choosing $r = r_n{+}1$ will solve the goal. Again, the proof obligation $(r_n{+}1)^2{\le}n{+}1 \wedge n{+}1{<}((r_n{+}1){+}1)^2$ can be shown by standard arithmetical reasoning. $(r_n{+}1)^2 > n{+}1$, then one has to choose $r = r_n$ and prove $r_n^2{\le}n{+}1 \wedge n{+}1{<}(r_n{+}1)^2$ using standard arithmetical reasoning.

Figure 1 shows the trace of a formal proof in the Nuprl system [CAB+86, ACE+00] that uses exactly this line of argument. It initiates the induction by applying the library theorem

```
NatInd        ∀P:ℕ→ℙ.  (P(0) ∧ (∀i:ℕ+.  P(i-1) ⇒ P(i)))   ⇒  (∀i:ℕ.  P(i))
```

The base case is solved by assigning 0 to the existentially quantified variable and using Nuprl's autotactic (trivial standard reasoning) to deal with the remaining proof obligation. In the step case (from $i{-}1$ to i) it analyzes the root r for $i{-}1$, introduces a case distinction on $(r{+}1)^2{\le}i$ and then assigns either r or $r{+}1$, again using Nuprl's autotactic on the rest of the proof.

Nuprl is capable of extracting an algorithm from the formal proof, which then may be run within Nuprl's computation environment or be exported to other programming systems. The algorithm is represented in Nuprl's extended lambda calculus.

Depending on the formalization of the existential quantifier there are two kinds of algorithms that may be extracted. In the standard formalization, where $\exists$ is represented as a (dependent)

```
∀n:ℕ.  ∃r:ℕ.  r² ≤ n < (r+1)²
BY  allR

  n:ℕ
  ⊢ ∃r:ℕ.  r² ≤ n < (r+1)²
  BY  NatInd 1

  .....basecase.....
      ⊢ ∃r:ℕ.  r² ≤ 0 < (r+1)²
  √ BY  existsR ⌜0⌝ THEN Auto

  .....upcase.....
      i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²
      ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      BY  Decide ⌜(r+1)² ≤ i⌝ THEN Auto

      .....Case 1.....
          i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²,  (r+1)² ≤ i
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √ BY  existsR ⌜r+1⌝ THEN Auto'

      .....Case 2.....
          i:ℕ⁺,  r:ℕ,  r² ≤ i-1 < (r+1)²,  ¬((r+1)² ≤ i)
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √ BY  existsR ⌜r⌝ THEN Auto
```

Figure 1: Proof of the Specification Theorem using Standard Induction.

product type, the algorithm – shown on the left[1] – computes both the integer square root r of a given natural number n and a proof term, which verifies that r is in fact the integer square root of n. If $\exists$ is represented as a set type, this verification information is dropped during extraction and the algorithm – shown on the right – only performs the computation of the integer square root.

```
let rec sqrt i
= if i=0 then <0,pf_0>
  else let <r,pf_{i-1}>  = sqrt (i-1)
       in
         if (r+1)²≤n  then <r+1,pf_i>
         else <r,pf_i'>
```

```
let rec sqrt i
= if i=0 then 0
  else let r = sqrt (i-1)
       in
         if (r+1)²≤n  then r+1
         else r
```

Using standard conversion mechanisms, Nuprl can then transform the algorithm into any programming language that supports recursive definition and export it to the corresponding programming environment. As this makes little sense for algorithms containing proof terms, we only convert the algorithm on the right. A conversion into SML, for instance, yields the following program.

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n-1)
                  in
                    if n<(r+1)^2  then r
                    else r+1
                  end
```

[1]The place holders pf_k represent the actual proof terms that are irrelevant for the computation.

2 Deriving an Algorithm that runs in $\mathcal{O}(\sqrt{n})$

Due to the use of standard induction on the input variable, the algorithm derived in the previous section is linear in the size of the input n, which is reduced by 1 in each step. Obviously, this is not the most efficient way to compute an integer square root. In the following we will derive more efficient algorithms by proving $\forall n\, \exists r\; r^2 \leq n \wedge n<(r+1)^2$ in a different way. These proof, however, will have to rely on more complex induction schemes to ensure a more efficient computation.

A more common method to compute the integer square root of a given number n is to start a search for a possible result r. One starts with $r{=}0$ and then increases r until $(r+1)^2 > n$. In the context of a proof, this means that we need to introduce an auxiliary variable k for the search and perform induction on this variable instead of n.

```
∀n:ℕ.  ∃r:ℕ.  r² ≤ n < (r+1)²
BY  allR  THEN  Assert ⌜∀j:ℕ.  (n-j)²≤n  ⇒  ∃r≥n-j. r² ≤ n < (r+1)²⌝
.....Assertion.....
   n:ℕ,  j:ℕ,  (n-j)²≤n
   ⊢ ∃r≥n-j. r² ≤ n < (r+1)²
   BY  NatInd 2
   .....basecase.....
      n:ℕ,  (n-0)²≤n
      ⊢ ∃r≥n-0. r² ≤ n < (r+1)²
   √ BY  existsR ⌜n⌝ THEN Auto'
   .....upcase.....
      n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r² ≤ n < (r+1)²,  (n-j)²≤n
      ⊢ ∃r≥n-j. r² ≤ n < (r+1)²
      BY  Decide ⌜n < (n-j+1)²⌝ THEN Auto
      .....Case 1.....
         n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r² ≤ n < (r+1)²,  (n-j)²≤n,
         n < (n-j+1)²
         ⊢ ∃r≥n-j. r² ≤ n < (r+1)²
      √ BY  existsR ⌜n-j⌝ THEN Auto'
      .....Case 2.....
         n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r² ≤ n < (r+1)²,  (n-j)²≤n
         ¬(n < (n-j+1)²)
         ⊢ ∃r≥n-j. r² ≤ n < (r+1)²
         BY  impL 3 THEN Auto

           n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r² ≤ n < (r+1)²,  (n-j)²≤n
           ¬(n < (n-j+1)²)
           ⊢ ∃r≥n-j. r² ≤ n < (r+1)²
        √  BY  existsR ⌜r⌝ THEN Auto'
.....Main.....
   n:ℕ,  ∀j:ℕ.  (n-j)²≤n  ⇒  ∃r≥n-j. r² ≤ n < (r+1)²
   ⊢ ∃r:ℕ. r² ≤ n < (r+1)²
   BY  allL 2 ⌜n⌝ THEN Auto

      n:ℕ,  r:ℕ,  r≥n-n,  r² ≤ n < (r+1)²
      ⊢ ∃r:ℕ. r² ≤ n < (r+1)²
√     BY  existsR ⌜r⌝ THEN Auto
```

Figure 2: Proof of the Specification Theorem using Search

A naive approach would be to prove the theorem $\forall n\,\forall k\,\exists r{\geq}k\;\; r^2{\leq}n \wedge n{<}(r{+}1)^2$ using induction on k and then to instantiate this theorem with $k{=}0$. This approach, however, has two major flaws. First, the induction on k expresses a solution for k in terms of a solution for $k{-}1$, which is less efficient than a forward search. Second, the search must begin at some $k{>}\sqrt{n}$ but the theorem obviously does not hold for $k{>}\sqrt{n}$.

To fix these problems, we need to change the direction of the search into one that starts at 0 and recursively solves the problem for k by consulting a solution for $k{+}1$ until the square root has been found, which can be expressed by a standard induction over $j = n{-}k$. We also need to add a limit to the search, i.e. $(n{-}j)^2{=}k^2{\leq}n$.

The formal Nuprl proof begins by asserting $\forall j\; (n{-}j)^2{\leq}n \Rightarrow \exists r{\geq}(n{-}j)\quad r^2{\leq}n \wedge n{<}(r{+}1)^2$, proves this statement by induction, and then instantiates it with $j{=}n$. Extracting the algorithm from this proof, depicted in Figure 2, and converting it into SML leads to the following program, which now runs in $\mathcal{O}(\sqrt{n})$.

```
fun sqrt n = let fun aux j =
                   if j=0  then n
                   if n < (n-j+1)^2  then n-j
                   else aux (j-1)
             in
               aux n
             end
```

Note that the case $j{=}0$ is never reached unless n is 0.

By using different induction schemes it is possible to modify this algorithm into a more conventional form that uses an auxiliary variable k that is increasing instead of the term $n{-}j$, where j is decreasing. This induction scheme, however, needs to make explicit that choices for the auxiliary variable have an upper bound (i.e. n), whereas the lower bound zero is implicit in the other induction schemes that quantify over natural numbers. The induction scheme

```
RevNatInd    ∀P:ℕ→ℙ.  (∀i:{...t}.  (∀j:{i+1..t}.P(j)) ⇒ P(i))   ⇒   (∀i:{...t}.P(i))
```

which can easily be derived from the scheme `NatInd`, enables us to begin our proof by asserting $\forall k\; k^2{\leq}n \Rightarrow \exists r{\geq}k\; r^2{\leq}n \wedge n{<}(r{+}1)^2$ and then to proceed as in Figure 2, replacing every occurrence of $n{-}j$ by k. The extracted algorithm would now be

```
fun sqrt n = let fun aux k =
                   if k=n  then n
                   if n < (k+1)^2  then y
                   else aux (k+1)
             in
               aux 0
             end
```

Actually, the search algorithm is an instance of a generic search method that is implicitly contained in the following theorem

```
NatSearch    ∀P:ℕ→ℙ.  ∀n:ℕ.  P(n) ⇒ (∃k:{0..n}.  P(k) ∧ (∀j:{0..k-1}.  ¬P(j)))
```

which states the (bounded) existence of a minimal k with some property P. Instantiating this theorem with $P(k)$ replaced by $(k+1)^2 > n$ immediately gives us the desired search algorithm.

```
∀n:ℕ.  ∃r:ℕ.  r² ≤ n < (r+1)²
BY  allR

  n:ℕ
  ⊢ ∃r:ℕ.  r² ≤ n < (r+1)²
  BY  NatInd4 1

  .....basecase.....
      ⊢ ∃r:ℕ.  r² ≤ 0 < (r+1)²
  √  BY  existsR ⌜0⌝ THEN Auto

  .....upcase.....
      i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²
      ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      BY  Decide ⌜((2*r)+1)² ≤ i⌝ THEN Auto

      .....Case 1.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ((2*r)+1)² ≤ i
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌜(2*r)+1⌝ THEN Auto'

      .....Case 2.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ¬(((2*r)+1)² ≤ i)
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌜2*r⌝ THEN Auto
```

Figure 3: Proof of the Specification Theorem using Binary Induction

3 Deriving a Logarithmic Algorithm

One of the most efficient forms of computation on numbers is to operate on their binary representation and to construct a value bit by bit. The corresponding induction scheme requires proving a conclusion $P(x)$ from an induction hypothesis $P(x \div 2)$, where $\div$ denotes integer division. For the integer square root problem, this induction scheme would have to be used on the *output variable* similarly to the way linear induction was used in the previous section.

It is much easierr, however, to use *4-adic induction* on the input variable instead, as this leads to a simpler proof. In fact, it is possible to mirror the proof given in Section 1 by applying the library theorem

```
NatInd4        ∀P:ℕ→ℙ.  (P(0) ∧ (∀i:ℕ.  P(i÷4) ⇒ P(i)))   ⇒   (∀i:ℕ.  P(i))
```

and then replacing every occurrence of `r` in the arguments of a proof tactic by `2*r`. Apart from these differences, which are emphasized in both proofs, the proof in Figure 3 is identical to the one in Figure 1. Accordingly, the generated algorithms have exactly the same structure. Extracting the algorithm from the proof in Figure 3 and converting it into SML leads to the following program, which now runs in *logarithmic time* (assuming that division by 4 is implemented as bit-shift operation).

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n/4)
                  in
                    if n < (2*r+1)^2  then 2*r
                    else 2*r+1
                  end
```

Final Remarks

The algorithms and derivations presented in this note are contained in Nuprl's formal digital library that is now available online for interactive browsing at `http://www.nuprl.org`.

Using the proof strategies for inductive reasoning described in [KP01] it is possible to automatically construct all the proofs presented here. The implementation of this method as well as the proofs generated by it will be posted as part of the formal digital library in the future.

This work was supported in part by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765 (Building Interactive Digital Libraries of Formal Algorithmic Knowledge) and by NSF Grant CCR 0204193 (Proof Automation in Constructive Type Theory).

References

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl proof development system.* Prentice Hall, 1986.

[KP01] Christoph Kreitz and Brigitte Pientka. Connection-driven inductive theorem proving. *Studia Logica*, 69(2):293–326, 2001.