CS3110 Fall 2013 Lecture 19: Computational Complexity and Recursion

Robert Constable

1 Lecture Plan

1. Computational complexity analysis requires clean execution model
2. Integer square root example, specification in OCaml SL
3. Using fixed point operators for clean recursive definitions
4. Implementing the specification, counting steps
5. Type checking and "proving" the spec
6. Fast integer square root and "fast induction"

2 Review, Overview, and Comments

We have established that dependent types allow us to encode more details of a specification into the OCaml SL types. We have seen that dependent types for some functions look like induction principles.

In this lecture we will see that dependent types can help us discover functions with improved performance. To reason about performance of functions we need clean computation rules. For recursion, this leads us to

use *fixed point operators* which we will define here. They give us a very precise set of rules that allow us to define measures of computational complexity.

We compare fast induction with standard induction.

3 Reading and Sources

Along with this lecture we will include notes written by Professor Christoph Kreitz showing how to prove properties of programs. We can see these notes as another view of type checking, but more importantly, they show a way to derive better performing algorithms in a systematic way. We can think of this method as discovering *fast induction principles*.

Professor Kozen covers many of these topics in his 2009 fall lectures, specifically these: CS3110 fall 2009 Lecture 18 and Recitation 18. You should read those notes carefully to review or learn the concept of Big-Oh notation commonly used and the hierarchy of complexity bounds $O(1)$ for constant algorithms, $O(log\ n)$ for log n algorithms, $O(n)$, for linear, $O(n\ log\ n)$, for "n log n," $O(n^2)$ for quadratic, $O(n^3)$ for cubic, etc.

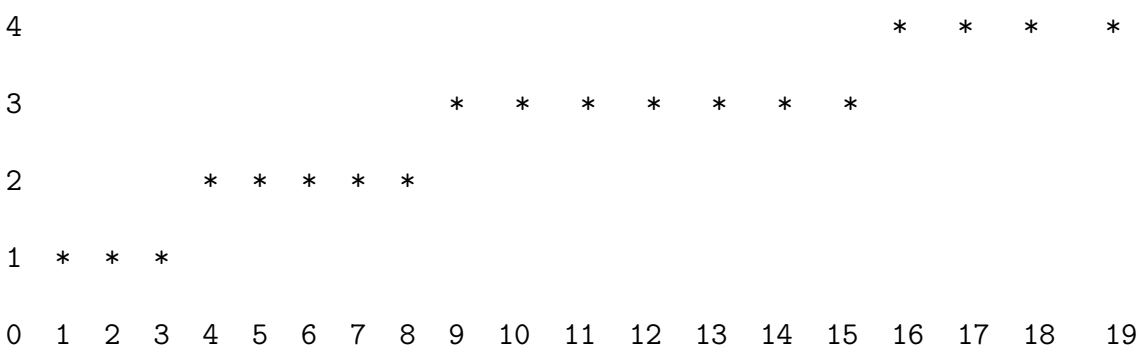
The Big-Oh notation $O(exp)$ means that the complexity of the algorithm is bounded by a positive constant times the bound, e.g. for $O(log\ n)$ the computational complexity of $f(n)$ is bounded by $c \times log\ n$. The parameter n is often the *magnitude* in the case of numbers and usually the *size* of the input in the case of algorithms on lists, trees, graphs, etc. However, at times for numbers people use n as the *length* or size of the input not the value or magnitude. In our examples for the integer square root, we are using the n as the value (magnitude).

Be sure to study the example of multiplication versus fast multiplication in Kozen's 2009 notes. We will examine the same idea for an algorithm to find the integer square root of a natural number. We will use dependent types to specify the algorithm.

4 Program Development from Specifications

4.1 An example to illustrate the issues

The example we use to illustrate the main ideas of this lecture is the task of finding the integer square root of a natural number. The clearest "specification" might be a graph of the function we have in mind.



We can see from the graph of the integer square root function that when the input is a perfect square, we require the exact result, e.g. the integer square root of 4 is 2 or 9 is 3 or 16 is 4, etc. What do we require when the input is not a perfect square? This graph shows that we want the largest positive integer r such that $\mathtt{r} * \mathtt{r} \leq \mathtt{n}$. This means that $\mathtt{n} < (\mathtt{r}+1) * (\mathtt{r}+1)$. So the specification is quite clear, given n, find the largest natural number r whose square is less than or equal to n. This is the same thing as the least number r such that $\mathtt{n} < (\mathtt{r}+1) * (\mathtt{r}+1)$.

These observations lead to the following development of a clean precise OCaml SL specification.

```
(n:nat -> (r:nat where r*r <= n < (r+1)*(r+1)) )
```

The simplest algorithm for finding the least number such that some condition holds is to start the search at 0 and test the condition, increasing the candidate value until the condition is met. We only need a guarantee that the search will stop. As we learned in one of our earliest lectures we can write a simple recursive program to implement this loop. Below is the simple loop, not in OCaml, and the corresponding tail recursive program.

```
  r:=0; while (r+1)*(r+1) <= n do r:= r+1 od

  (* while program version of square root *)
  let rec whl (r:int) (n:int) : int =
  if (r+1)*(r+1) <= n then whl (r+1) n else r ;;

  We initialize this program at whl 0 n as in

  # whl 0 15;;
- : int = 3
```

In OCaml it is more natural to simply write a recursive function with the dependent type specification as a comment and type check the function. Here is how it is done.

```
(*  (n:nat -> (r:nat where r*r <= n < (r+1)*(r+1)) )  *)

# let rec sqrt (n:int) :int =
  if n = 0 then 0
           else let r = sqrt (n-1) in
           if (r+1)*(r+1) <= n then r+1 else r ;;

val sqrt : int -> int = <fun>
# sqrt 3;;
- : int = 1
# sqrt 4;;
- : int = 2
```

```
# sqrt 164 ;;
- : int = 12
# sqrt 10257;;
- : int = 101
# sqrt 111111;;
- : int = 333

(* now try 1,111,111 *)
# sqrt 1111111;;
Stack overflow during evaluation (looping recursion?).
```

We can see from this example that the execution of this algorithm is not so impressive. In Nuprl we can compute the real number square root of 2 to hundreds of decimal digits in a split second. How do we write a more efficient algorithm for this task. How do we analyze the number of steps that this algorithm takes?

To perform an analysis, we need a simple definition for the execution of a recursive procedure. We can simply perform the substitution of the code body for the recursive call as a first step, but we can't be sure that OCaml executes this way. It would be better to have a simple internal model of how recursion works.

4.2 Recursion using a fixed point operator

Here is a single internal operator, `efix` for eager fixed point, which can serve as our definition of a recursive procedure. The idea might seem a bit strange at first, but it is a remarkable small result that we can build on to analyze recursion and prove a number of interesting properties of OCaml functions.

```
# let rec efix f x = f (efix f) x ;;
val efix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

```
# efix (fun (f:int -> int) ->
             fun (x:int) -> if x = 0 then 0
                            else f (x-1) + x ) ;;
- : int -> int = <fun>
# let sigma =
efix (fun (f:int -> int) ->
           fun (x:int) ->
           if x = 0 then 0
           else f (x-1) + x ) ;;

val sigma : int -> int = <fun>
# sigma 2 ;;
- : int = 3
# sigma 5 ;;
- : int = 15
# sigma 100 ;;
- : int = 5050
```

Here is the square root example done using *efix*.

```
# let rec efix f x = f (efix f) x ;;
val efix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

```
# let rec sqrt (n:int) :int = if n = 0 then 0 else let r = sqrt (n-1) in
  if (r+1)*(r+1) <= n then r+1 else r ;;

val sqrt : int -> int = <fun>

# sqrt 3;;
- : int = 1
# sqrt 4;;
- : int = 2
```

```
# let f_rt = fun f -> fun n ->
             if n = 0 then 0
             else let r = f (n-1) in
             if (r+1)*(r+1) <= n then r+1 else r;;

val f_rt : (int -> int) -> int -> int = <fun>

# let sqrt2 = efix f_rt ;;


# sqrt2 10257;;
- : int = 101
# sqrt2 111111;;
- : int = 333
```

4.3 Counting steps

```
let rec sqrt_steps (p: int*int) : int*int =
let n,s = p in if n = 0 then (0,1)
               else let p2 = sqrt_steps (n-1,s+1) in
                    let r,stps = p2 in
                    if (r+1)*(r+1) <= n then (r+1,stps+2)
                    else (r,stps+1)


val sqrt_steps : int * int -> int * int = <fun>
# sqrt_steps (14,0);;
- : int * int = (3, 18)
# sqrt_steps (125,0);;
- : int * int = (11, 137)

# sqrt_steps (1155,0);;
- : int * int = (33, 1189)
```

```
# sqrt_steps (27667,0);;
- : int * int = (166, 27834)
```

We can also count the number of steps and the depth of the recursive calls.

```
# let rec sqrt_steps_dpth (inp:(int*int*int) ) : (int*int*int) =
  let n,s,d = inp in if n = 0 then (0,1,d)
                     else let s2 = sqrt_steps_dpth (n-1,s+1,d+1) in
                     let r,stps,dd = s2 in if (r+1)*(r+1) <= n
                                           then (r+1,stps+2,dd)
                                           else (r,stps+1,dd);;

val sqrt_steps_dpth : int * int * int -> int * int * int = <fun>

# sqrt_steps_dpth (1267,0,0);;
- : int * int * int = (35, 1303, 1267)

(* Note that the depth of recursion in this case is exactly
the input value, and that's a lot!! *)
```

4.4 Speeding up the integer square root.

Here is an algorithm that is exponentially better for finding the integer square root since its run time is logarithmic in the input assuming that the division operation is fast – say by bit shifting.

```
# let rec fst_rt (n:int) :int =
if n = 0 then 0 else let r = fst_rt (n/4) in
                if n < (2*r + 1)*(2*r + 1)
                then 2*r else 2*r + 1 ;;

val fst_rt : int -> int = <fun>
```

```
# fst_rt 10257 ;;
- : int = 101
# fst_rt 111111;;
- : int = 333
# fst_rt 1111111;;
- : int = 1054
# fst_rt 11111111;;
- : int = 3333
# fst_rt 123456789;;
- : int = 11111


# fst_rt 1234567890;;
Characters 7-17:
  fst_rt 1234567890;;
         ^^^^^^^^^^
Error: Integer literal exceeds the range of
representable integers of type int
```

```
let rec fst_rt_stp (p:int*int) :(int*int) =
let (n,s) = p in if n = 0 then (0,1) else let (r1,r2) =
fst_rt_stp ((n/4),s+1) in
if n < (2*r1 + 1)*(2*r1 + 1) then (2*r1, r2+3)
                             else (2*r1 + 1, r2+4) ;;

val fst_rt_stp : int * int -> int * int = <fun>
#

# fst_rt_stp (123456789,0);;
- : int * int = (11111, 43)
```

```
(* compare these next two *)
# sqrt_steps (123456,0);;
- : int * int = (351, 123808)

# fst_rt_stp (123456,0);;
- : int * int = (351, 28)


# sqrt_steps (123456789,0);;
Stack overflow during evaluation (looping recursion?).

# fst_rt_stp (1234567,0);;
- : int * int = (1111, 34)
# sqrt_steps (1234567,0);;
Stack overflow during evaluation (looping recursion?).
```

4.5 Fast induction

How do we justify the correctness of the fast integer square root? We can see that sqrt is correct by simple induction on its input. We can easily imagine a standard mathematical proof for this, indeed, there is a Nuprl proof included with this lecture.

Can we justify fast integer square root using some matching form of induction, a kind of *fast induction*? Does this approach make sense? Professor Kreitz illustrated the idea of fast induction using the square root examples in an appendix to this article [1] available on the Nuprl web page (www.nuprl.org). Here is the principle of fast induction that he used.

Fast Induction:

$(P(0)\ \&\ \forall\, n : \mathbb{N}.\ P(n/4)\ \Rightarrow\ P(n))\ \Rightarrow\ \forall\, n : \mathbb{N}.\ P(n).$

The value 4 is arbitrary here, and you can formulate many interesting versions of induction that are easily proved correct from our standard induction. Professor Kozen discusses this idea carefully in Lecture 18 from 2009 fall. He uses strong induction in those notes to derive other forms of induction like this one.

We have attached to the pdf of this lecture notes by Professor Kreitz on how to prove the fast square root algorithm correct using fast induction. We will explore this idea further in Lecture 21.

References

[1] Robert L. Constable. Information-intensive proof technology; lecture notes for the marktoberdorf nato summer school. Cornell University, Ithaca, NY, 2003. `http://www.nuprl.org/documents/Constable/marktoberdorf03.html`.

Appendix A

Derivation of a Fast Integer Square Root Algorithm

by Christoph Kreitz

A.1 Deriving a Linear Algorithm

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

Base Case: prove $\exists r\ r^2{\leq}0 \wedge 0{<}(r{+}1)^2$

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

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

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

Figure A.1 shows the trace of a formal proof in the Nuprl system [40, 10] that uses exactly this line of argument. It initiates the induction by applying the library theorem

`NatInd` $\forall$`P:`$\mathbb{N}{\to}\mathbb{P}$`.` `(P(0)` $\wedge$ `(`$\forall$`i:`$\mathbb{N}^+$`.` `P(i-1)` $\Rightarrow$ `P(i)))` $\Rightarrow$ `(`$\forall$`i:`$\mathbb{N}$`.` `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 $(r{+}1)^2{\leq}i$ 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) product type, the algorithm – shown on the left* – 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.

*The place holders pf_k represent the actual proof terms that are irrelevant for the computation.

```
∀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 A.1: Proof of the Specification Theorem using Standard Induction.

```
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
```

A.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 A.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 $\forall j\ (n{-}j)^2{\le}n \Rightarrow \exists r{\ge}(n{-}j)\ \ r^2{\le}n \wedge n{<}(r{+}1)^2$, proves this statement by induction, and then instantiates it with $j{=}n$. Extracting the algorithm from this proof, depicted in Figure A.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 $\quad \forall P{:}\mathbb{N}{\to}\mathbb{P}.\ \ (\forall i{:}\{...t\}.\ \ (\forall j{:}\{i{+}1..t\}.P(j)) \Rightarrow P(i)) \ \Rightarrow\ (\forall i{:}\{...t\}.P(i))$

which can easily be derived from the scheme NatInd, enables us to begin our proof by asserting $\forall k\ k^2{\le}n \Rightarrow \exists r{\ge}k\ r^2{\le}n \wedge$ and then to proceed as in Figure A.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 $\quad \forall P{:}\mathbb{N}{\to}\mathbb{P}.\ \ \forall n{:}\mathbb{N}.\ \ P(n) \Rightarrow (\exists k{:}\{0..n\}.\ \ P(k) \wedge (\forall j{:}\{0..k{-}1\}.\ \ \neg P(j)))$

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.

A.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 A.1 by applying the library theorem

NatInd4 $\quad \forall P{:}\mathbb{N}{\to}\mathbb{P}.\ \ (P(0) \wedge (\forall i{:}\mathbb{N}.\ \ P(i\underline{\div 4}) \Rightarrow P(i))) \ \Rightarrow\ (\forall i{:}\mathbb{N}.\ \ P(i))$

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 A.3 is identical to the one in Figure A.1. Accordingly, the generated algorithms have exactly the same structure. Extracting the 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 A.3: Proof of the Specification Theorem using Binary Induction

from the proof in Figure A.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 [74] 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.