CS3110 Spring 2016 Lecture 9: Specifications

Lecture Plan

- – Discuss *specifications* for programming tasks

Using	Logic	(HOL, PVS, Hoare Logic,...)
Using	PL types	
Using	Dependent types	(Agda, Coq, Nuprl, F^*)

  – Experience:

  "Folk wisdom" about PL types, "if it type checks, it works."

  – Data about *dependent types*

CompCert C compiler	– Proved correct in Coq
seL4 microkernel	– Proved correct in HOL
Formal mathematics	– Proved correct in Coq, HOL, Nuprl
HACMS drone	– Proved correct in Coq
Multi-Paxos protocol	– Proved corect in Nuprl

  – This course

  – we will use OCaml Type Theory and relate it to logic.

  – we will discuss *dependent types*, used by Coq, Agda, Nuprl, F^*,...

  PS3 will cover some logic using OCaml types and the *constructive real numbers*, as a module with field structure.

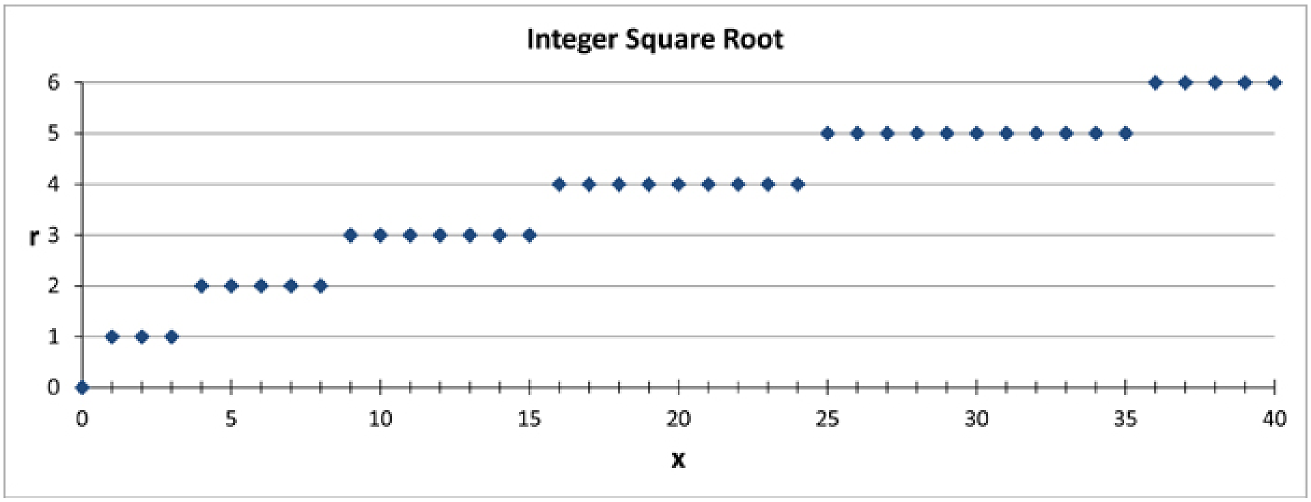
  * Thursday March 3 lecture by Abhishek Anand – motivational. *

- Sample programming specification and a proof

- Seeing the "proof as a program"

Integer Square Root Specification

$\forall$x:$\mathbb{N}$.($\exists$r:{$\mathbb{N}$ |(((r * r) $\leq$ x) $\wedge$ x < (r + 1) * (r + 1))})



Derivation of a Fast Integer Square Root Algorithm

by Christoph Kreitz

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

$$\texttt{NatInd}\qquad \forall P{:}\mathbb{N}{\to}\mathbb{P}.\ \ (P(0)\ \wedge\ (\forall i{:}\mathbb{N}^{+}.\ \ P(\underline{i\text{-}1})\ \Rightarrow\ P(i)))\quad \Rightarrow\quad (\forall i{:}\mathbb{N}.\ \ P(i))$$

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 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≤ i 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≤ i 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
```