CS 5860 Program Verification

Tue Nov 1, 2011

We return now to the example with which we started the course, the iterative form of Euclid's division algorithm presented in David Gries' Science of Programming, 1981, pp 1-4.

Here is essentially Gries' account.

Euclid's Algorithm

{ Assume $x : \mathbb{N}$, $y : \mathbb{N}^+$ e.g. $0 < y$ }

$r := x$
$q := 0$

{ $Inv(x,y,r,q)$ iff $x = y*q + r$ & $0 \le r$ }

while $r \ge y$ do

{ $Inv(x,y,r,q)$ } { $r \ge y$ }

$r := r - y$ /* defined since $y \le r$ */
$q := q+1$

{ $Inv(x,y,r,q)$ since $0 \le r-y$ & $x = y*(q+1) + (r-y)$ }
by Arithmetic

od

{ $Inv(x,y,r,q)$ & $r < y$ }

{ $Out(x,y,r,q)$ iff $x = y*q + r$ & $r < y$ }

Exercise: Write a similar while program to compute $x!$ (factorial(x)) and give an abstract version as we do next for Euclid's algorithm. Also write the program as a recursive procedure (iterative) and a recursive function.

CS 5860 Program Verification continued

Tue Nov 1, 2011

We say that this algorithm is partially correct if it satisfies $\forall x:\mathbb{N}. \forall y:\mathbb{N}^+.\ x = y*q+r \ \&\ r<y$ if it halts.

We say that the algorithm is (totally) correct if it is partially correct and it halts.

We can see that the above algorithm halts, but there is no "record" of our reasoning to see this from the code. We notice that the algorithm can take at most x steps in the loop.

Here is an abstract form of this algorithm. We call this a program scheme (or schema), over domain D.

```
{ Assume In(x,y) }
   r := x
   q := a        /* a is some constant in domain D */
   { Inv(x,y,r,q) }
   while b(r,y) do
      { Inv(x,y,r,q) & b(r,y) }
      r := f1(r,y)
      q := f2(q)
      { Inv(x,y,r,q) }
   od
      { Inv(x,y,r,q) & ~b(r,y) }
      { Out(x,y,r,q) }
```

CS 5860

Tue Nov 1, 2011

Program Verification continued

If we express the input assumptions $In(x,y)$ and the invariant and output conditions, $Inv(x,y,r,q)$, $Out(x,y,r,q)$, in first-order logic, then we can interpret the while-loop as a partial realizer (or partial evidence) for

$$\forall x,y.\,(In(x,y) \Rightarrow Out(x,y,q,r))$$

where q, r are "program variables." To avoid this new kind of variable we can do the following.

- use a recursive definition and avoid program variables and assignment; the recursion would be "tail recursive."
- convert the program variables to ordinary variables at the end using the existential quantifier, e.g.

  $$\forall x,y.\,(In(x,y) \Rightarrow \exists u,v.\,Out(x,y,u,v))$$

  This solution works when the loop terminates.

- in Boolean logic one can offer an equivalent form starting from

  $$\forall x,y.\,\exists u,v.\,(In(x,y) \Rightarrow Out(x,y,u,v))$$

  and use Boolean rules to say the following is equivalent (not so computationally)

  $$\forall x,y.\sim \forall u,v.\sim(In(x,y) \Rightarrow Out(x,y,u,v))$$

  This is a Boolean equivalent to total correctness.

  Zohar Manna developed a theory around this in his book Mathematical Theory of Computation, 1974.

CS 5860 Program Verification continued

Tue Nov 1, 2011

Let us examine the recursive solution to the Euclid specification $\forall x, y. (y > 0 \Rightarrow \exists q, r. (x = y * q + r) \& r < y)$

$$euclid(x, y, q, r) = \text{if } r < y \text{ then } \langle q, r \rangle$$
$$\text{else } euclid(x, y, q+1, r-y)$$

This procedure achieves the specification when executed on $euclid(x, y, 0, x)$. We see that $x = y * 0 + x$, and the procedure terminates when $r < y$. This happens since we decrease r on each recursive call, since $y > 0$.

To establish correctness, we assume that initially the inputs satisfy $y > 0$ and $x = y * q + r$, and if $x = y * q + r$ before the call, then it holds of the result. This syntax is suggestive

~~let rec~~

~~letrec euclid(x, y, q, r)~~

$\{$assuming $y > 0 \ \& \ x = y * q + r\}$

letrec $euclid(x, y, q, r) =$

if $r < y$ then $\langle q, r \rangle$ $\{x = y * q + r \ \& \ r < y\}$

else $\{x = y * q + r\}$

$euclid(x, y, q+1, r-y)$

attains $\{x = y * q + r \ \& \ r < y\}$

CS 5860

Tue Nov 1, 2011

Program Verification continued

The recursive procedure euclid is a recursive realizer for the Euclid specification. It is another very useful form of induction. To see how useful, try to prove the specification using any of the alternative forms we discussed in the previous lecture. Computer science abounds in varieties of induction especially on recursive data types such as natural numbers, lists, and trees.

Interestingly, we can express partial correctness in first-order logic by introducing various kinds of realizers: recursive and iterative.

We will now look at the recursive solution to the problem of finding integer square roots.

A note on program variables

If our logic allowed function constants, we could avoid program variables by defining the state of a program to be a mapping from Identifiers, say $x, y, z, \ldots$ to values

$$s : \text{Identifiers} \to D.$$

Then $x := f(y)$ maps a state s to a new state s' where $s'(x) = f(s(y))$.

6


```
∀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.