COMS22202: 2015/16

Language Engineering

Dr Oliver Ray
(csxor@Bristol.ac.uk)

Department of Computer Science
University of Bristol

Tuesday 26th April, 2016

Lab Sheet Feedback

- Intuitions, semantics and syntax of “for loops” differ a lot between languages: für (Superplan), for..do (Algol,Pascal), do/for all (Fortran), for..next (Basic), for(..;..;..) (Java, C), downto, step,...
- The labsheet uses “for x:=a1 to a2 do S” to denote a simple for loop
- When a1 and a2 are numerals and x is not used outside the loop or modified within the loop, then most people can agree on the meaning; but opinions can differ in almost all other cases
- One view is that these are shorthand for C-style loops of the form “for(x:=a1; x<=a2; x++)” – where a2 is re-evaluated each iteration
- Another view is they are loops where the number of iterations is known in advance – where a2 is fixed at the start along with a1 – more like “l:=a1; u=a2; for(x:=l; x<=u; x++)” where l and u are fresh variables (which are needed in case a2 depends on x)

For Loop Examples

- What do you think the following loops should do?

- for x:=0 to 1 do write x — probably ‘0 1’
- for x:=1 to 0 do write x — probably ‘’
- for x:=0 to 0 do write x — probably ‘0’ not ‘’
- x:=1; for x:=0 to x do write x — probably ‘0 1’ not ‘0’
- y:=1; for x:=0 to y do y:=y+1 — probably terminate not loop
- x:=1; (for x:=0 to 2 do skip); write x — probably ‘2’ not ‘1’ or undef

- Does your axiomatic semantics give the results you expected?

Aside: Computational Power

- Note: It can be shown that the fixed-bound interpretation of for loops is less powerful than the general interpretation (in the sense that the former can only be used to compute primitive recursive functions while the latter can compute other partial recursive functions e.g. Ackermann's function)

Coursework Feedback

- I see many of you still trying to apply axiomatic proof rules forwards (starting at the top of the program, working downward) and coming up with proofs that are much much more verbose than necessary!
- Many of you are also struggling with the fact that n>0 is not in the precondition of coursework Q.3(a) – which raises some quite subtle points that we explored in the week 19 tutorial in our discussion of labsheet Q.1(b) and noted below
- That question (whose solution is given in example 6.9 of the book) involves showing that n>0 in the postcondition of the factorial program; and this needs a more complex invariant such as $x>0 \rightarrow (y.x!=n! \wedge n>=x)$
- But in the coursework we don't need to prove that n>0 if the loop terminates (although that must be the case) so, as I've explained on the board a couple of times, we can use a simpler invariant like $x>0 \rightarrow y.x!=n!$ or even $x\neq0 \rightarrow y.x!=n!$
- In the labs I may have misled one or two of you by pointing out that the even simpler invariant y.x!=n! works for both positive x>0 (whose factiorial is always defined) and negative x<0 (whose factiorial is always undefined) …
- …but, as shown in the week 18 slides, this is not an invariant in case x=0 (for, if y=1 and x=0 initially then y.x!=1.0!=1 but, after the first iteration, y=0 and x=-1 so y.x!=0.-1! which is either 0 or undefined, but certainly not 1)

Axiomatic Proofs

- In order to discharge proof obligations for the consequence rule, you will need to insert additional comments into your program to explain your proofs; and there are multiple ways of doing this
- The next two slides give another variant of the factorial case study and show two ways of discharging proof obligations: the first is a detailed proof and the latter is more informal – but both would be OK
- Remember that even when reasoning “backwards” (from postconditions or invariants) any proof obligations that you generate will be in the “forwards” direction
- Note that you are free to simplify logical expressions at any point in a proof if it makes your life easier
- Remember that to prove an implication a-> b you need to assume a and show b
- Bear in mind that although you read the proofs downwards it is easier to write them upwards!

Factorial Example: formal proofs

```
{x=n ^ y=1}
{x>0 -> y*x!=n!}
while !(x=1) do (
  {x>0 -> y*x!=n! ^ !(x=1)}
  {(x-1)>0 -> (y*x)*(x-1)!=n!}
  y := y * x;
  {(x-1)>0 -> y*(x-1)!=n!}
  x := x - 1
  {x>0 -> y*x!=n!}
);
{x>0 -> y*x!=n! ^ !!(x=1)}
{y=n!}
```

{ 1. x=n ^ y=1 given
2. x=n by ^elimination on 1
3. y=1 by ^elimination on 1
4. x!=n! taking ! of both sides of 2
5. x!=1*x! by basic arithmetic
6. x!=y*x! substituting 3 in 5
7. x>0 assumption
8. y*x!=n! substituting 6 in 4
9. x>0 -> y*x!=n! ->introduction 7-8 }

{ 1. x>0 -> n!=yx! given
2. (x-1)>0 assumption
3. x>1 from 2 by adding 1 to both sides
4. 1>0 by basic arithmetic
5. x>0 from 3 and 4 by transitivity of <
6. n!=yx! from 5 and 1 by ->elimination
7. x! = x(x-1)! by definition of ! using 5
8. n! = y*x*(x-1)! substituting 7 in 6 }

{ 1. x>0 -> y*x!=n! ^ !!(x=1) given
2. x>0 -> y*x!=n! by ^elimination on 1
3. !!(x=1) by ^elimination on 1
4. x=1 by !! elimination on 3
5. x>0 from 4
6. y*x!=n! from 5 and 2 by ->elimination
7. x!=1!=1*0!=1*1=1 from 4 using definition of !
8. y*1=n! substituting 7 in 6
9. y=n! by simple arithmetic }

Factorial Example: informal proofs

```
{x=n ^ y=1}
  { as n! = x! =1*x! = y*x! }
{y*x!=n!}
  { as anything -> true }
{x>0 -> y*x!=n!}
while !(x=1) do (
  {x>0 -> y*x!=n! ^ !(x=1)}
    { as if (x-1)>0 then x>1 so x>0 and y*x!=n! and (y*x)*(x-1)!=n! }
  {(x-1)>0 -> (y*x)*(x-1)!=n!}
  y := y * x;
  {(x-1)>0 -> y*(x-1)!=n!}
  x := x - 1
  {x>0 -> y*x!=n!}
);
{x>0 -> y*x!=n! ^ !!(x=1)}
  { as x=1 so n! = y*x! = y.x! = y.1 = y }
{y=n!}
```