Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Verification of cycles (inference - rule R5)

- Before cycle:
  - The invariant should be checked before the 1st test to the condition of the while loop.
    - Before the loop, i = 0. There is a j (index of A []) such as 0 <= j & & j <0, so the invariant is true.
- During the cycle:
  - Variant:
    - The variant function starts: v = V = a.length.
    - After each iteration v = v-1 <V
  - Invariant: (must be checked in the final state of the sequence of expressions of the loop body)
    - To run the loop body,
      A [j]! = R forall j: 0 <= j <= i, before increasing i.
    - What is equivalent to
      A [j]! = R forall j: 0 <= j <i after incrementing i.
    - So the invariant is true.
- End of the cycle:
  - The invariant should be checked at the end of the loop body execution
    - When the cycle ends, A [i] == r.
    - The array A has the value r at index i.

Exercise:

Find the postcondition and the loop invariant:

```
class Add {

  // pre A != null && B != null && A.length == B.length;
  // post

     int[] add(int A[], int B[]) {

          int C[]=new int[A.length];
          int i=0;

          // loop_invariant
          for(i=0; i<A.length; i++)
                C[i] = A[i]+B[i];
          return C;
     }
}
```


Tools

PVS (Specification and Verification Systems)
http://www-step.stanford.edu/

STeP
http://pvs.csl.sri.com/

HOL (Higher order logic)
http://www.cl.cam.ac.uk/Research/HVG/HOL/

The Logics Workbench
http://www.lwb.unibe.ch

Coq
http://pauillac.inria.fr/coq/

Example in PVS

```
|-------
f1g FORALL (n: nat): sum(n) = n * (n + 1) / 2
Rule? (induct "n")
Inducting on n on formula 1,
this yields 2 subgoals:
closed_form.1 :
|-------
f1g sum(0) = 0 * (0 + 1) / 2
Rule? (postpone)
Postponing closed_form.1.
closed_form.2 :
|-------
f1g FORALL (j: nat):
sum(j) = j * (j + 1) / 2 IMPLIES sum(j + 1) = (j + 1) * (j + 1 + 1) / 2
Rule? (postpone)
Postponing closed_form.2.
closed_form.1 :
|-------
f1g sum(0) = 0 * (0 + 1) / 2
Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
closed_form.1 :
|-------
f1g 0 = 0 / 2
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of closed_form.1.
```

Prova por indução

Mostra a subprova seguinte ainda não provada

Faz alguns cálculos

Aplica procedimentos de decisão para transformar o consequente em verdade

Passa para a subprova seguinte

```
closed_form.2 :
14 2.3 Proving
|-------
f1g FORALL (j: nat):
sum(j) = j * (j + 1) / 2 IMPLIES sum(j + 1) = (j + 1) * (j + 1 + 1) / 2
Rule? (skolem!)
Skolemizing,
this simplifies to:
closed_form.2 :
|-------
f1g sum(j!1) = j!1 * (j!1 + 1) / 2 IMPLIES
sum(j!1 + 1) = (j!1 + 1) * (j!1 + 1 + 1) / 2
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
closed_form.2 :
f-1g sum(j!1) = j!1 * (j!1 + 1) / 2
|-------
f1g sum(j!1 + 1) = (j!1 + 1) * (j!1 + 1 + 1) / 2
Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
closed_form.2 :
[-1] sum(j!1) = j!1 * (j!1 + 1) / 2
|-------
f1g 1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of closed_form.2.
Q.E.D.
Run time = 0.81 secs.
Real time = 223.01 secs.
```

Eliminar o quantificador universal

Transforma o consequente num antecedente com consequente

Faz alguns cálculos

Aplica procedimentos de decisão para transformar o consequente em verdade

Model-Checking versus theorem proving

Model Checking	Theorem proving
Good for control	Good for state
Do not deal with infinite state spaces	Use of induction techniques to deal with infinite spaces
Automated analysis	Semi-automated analysis
Good for checking temporal properties	Not really
Easier to use	Less easy to use but more generic

Loops

- {P} while B do S {Q}
- Partial correctness
  - P => I, the invariant is initially true
  - {Inv /\ B} S {Inv}, each execution of the loop preserves the invariant
  - (Inv /\ ~B)=>Q, the invariant and the loop exit condition imply the postcondition
- Total correctness
  - (Inv /\ B)=> v>0, if we are entering the loop body (i.e., the loop condition B evaluates to true) and the invariant holds, then v must be strictly positive
  - {Inv /\ B /\ v=V} S {v<V}, the value of the variant function decreases each time the loop executes (here V is a constant)

Loop example (1)

```
r:=1;
i:=0;
While i<m do
        r:= r*n;
        i:=i+1;
```

Prove that this function computes the n^{th} power of *m* and leaves the result in *r*.

Postcondition: r = n^m

Precondition: m>=0 /\ n>0

Loop invariant:

(1)a good heuristic for choosing a loop invariant is often to modify the postcondition of the loop to make it depend on the loop index instead of some other variable, such as r=n^i, but (2) this invariant is not strong enough…

Loop example (2)

- … loop invariant conjoined with the loop exit condition should imply the postcondition. The loop exit condition is **i>=m**, but we know that **i=m**. We can get this if we add **i<=m** to the loop invariant. In addition, for proving the loop body correct, it is convenient to add **0<=i** and **n>0** to the loop invariant as well. Thus our complete loop invariant will be
  - **$r=n^i$ /\ 0<=i<=m /\ n>0**

Loop example (3)

- In order to prove total correctness, we need to state a variant function for the loop that can be used to show that the loop will terminate. In this case **m-i** is a natural choice, because it is positive at each entry to the loop and decreases with each loop iteration.

Loop example (4)

- Now, we use the weakest precondition to generate proof obligations that will verify the correctness of the specification. First, we will ensure that the invariant is initially true when the loop is reached, by propagating that invariant past the first two statements in the program:

  {m>=0 /\ n>0}
  r:=1;
  i:=0;
  {$r=n^i$ /\ 0<=i<=m /\ n>0}

- We propagate the loop invariant past i:=0 to get $r=n^0$ /\ 0<=0<=m /\ n>0. Thus our proof obligation is to show that:

  m>=0 /\ n>0 => $1=n^0$ /\ 0<=0<=m /\ n>0

Loop example (5)

- We prove this with the following logic:
  - m>=0 /\ n>0, by assumption
  - $1=n^0$, because $n^0=1$ for all n>0 and we know n>0
  - 0<=0, by definition of <=
  - 0<=m, because m>=0 by assumption
  - n>0, by assumption above
  - $1=n^0$ /\ 0<=0<=m /\ n>0, by conjunction of the above

Loop example (6)

- We now apply weakest precondition to the body of the loop. We will first prove the invariant is maintained, then prove the variant function decreases. To show the invariant is preserved, we have:

  $\{r=n^i \wedge 0<=i<=m \wedge n>0 \wedge i<m\}$
  r:= r*n
  i:=i+1
  $\{r=n^i \wedge 0<=i<=m \wedge n>0\}$

  It comes from the loop condition

- We propagate the invariant past i:=i+1 to get
  - $r=n^{i+1} \wedge 0<=i+1<=m \wedge n>0$.
- We propagate this past r:=r*n to get:
  - $r*n=n^{i+1} \wedge 0<=i+1<=m \wedge n>0$.
- Our proof obligation is therefore:
  - $r=n^i \wedge 0<=i<=m \wedge n>0 \wedge i<m \Rightarrow r*n=n^{i+1} \wedge 0<=i+1<=m \wedge n>0$

Loop example (7)

- We can prove this as follows:

$r=n^i \wedge 0<=i<=m \wedge n>0 \wedge i<m$ => $r*n=n^{i+1} \wedge 0<=i+1<=m \wedge n>0$

$r = n^i \wedge 0<=i<=m \wedge n > 0 \wedge i < m$, by assumption
$r*n = n^i *n$, multiplying by n
$r*n = n^{i+1}$, definition of exponentiation
0 <= i+1, because 0<=i
i+1 < m+1, by adding 1 to inequality
i+1 <= m, by definition of <=
n>0, by assumption
$r*n = n^{i+1} \wedge 0<=i+1<=m \wedge n>0$, by conjunction of the above

Loop example (8)

- We have a prove obligation to show that the variant function is positive when we enter the loop. The obligation is to show that the loop invariant and the entry condition imply this:
- **$r=n^i \wedge 0<=i<=m \wedge n>0 \wedge i<m$** => **$m-i > 0$**
- The proof is trivial

  $r = n^i \wedge 0 <= i <= m \wedge n > 0 \wedge i < m$, by assumption
  i < m, by assumption
  m – i > 0, subtracting i from both sides

Loop example (9)

- We also need to show that the variant function decreases. We generate the proof obligation using weakest preconditions:

  $\{r = n^i \wedge 0<=i<=m \wedge n > 0 \wedge i < m \wedge m - i = V\}$
  r := r*n;
  i := i + 1;
  $\{m - i < V\}$

- We propagate the condition past i := i + 1 to get m – (i + 1) < V. Propagating past the next statement has no effect. Our proof obligation is therefore:

  $r = n^i \wedge 0<=i<=m \wedge n>0 \wedge i<m \wedge m - i = V \Rightarrow m - (i + 1) < V$

Loop example (10)

- Again the proof is easy:

  r=n^i /\ 0<=i<=m /\ n>0 /\ i<m /\ m−i=V, by assumption

  m − i = V, by assumption

  m − i − 1 < V, by definition of <

  m − (i + 1) < V, by arithmetic rules

- Last we need to prove that the postcondition holds when we exit the loop.

  r=n^i /\ 0<=i<=m /\ n>0 /\ i>=m => r = n^m

- We can prove it as follows:

  r = n^i /\ 0<=i<=m /\ n>0 /\ i>=m, by assumption

  i = m, because i<=m and i>=m

  r = n^m, substituting m for i in assumption