Reasoning About Programs

Week 8 Tutorial - Loop Invariants and Variants

Sophia Drossopoulou and Mark Wheelhouse

1st Question: Single Loops

Consider the following recursive method `gcd`, which returns the greatest common divisor of two integers:

```
int gcd(int m, int n)
// PRE: m ≥ 1  ∧  n ≥ 1
// VAR: m + n
{
    if ( n==m ) {
        return n;
    } else if ( m > n) {
        return gcd(m-n,n);
    } else {
        return gcd(m,n-m);
    }
}
```

Note that we can provide a variant to help us argue about the termination of a recursively defined function. Now consider the following iterative method `GcdCal`, which also calculates the greatest common divisor of two integers:

```
int GcdCal(int m, int n)
// PRE: m ≥ 1  ∧  n ≥ 1
// POST: r = gcd(m,n)
{
    int m1 = m;
    int n1 = n;
    // INV: I
    // VAR: V
    while( m1 != n1 ) {
        if (m1>n1) {
            m1 = m1 - n1;
        } else {
            n1 = n1 - m1;
        }
    }
    return m1;
}
```

Give an appropriate loop invariant I and a variant V that are strong enough to prove total correctness.

A possible answer:

$$\begin{array}{rcl} I & \longleftrightarrow & \texttt{m1} \geq 1 \ \wedge \ \texttt{n1} \geq 1 \ \wedge \ \texttt{gcd(m,n)} = \texttt{gcd(m1,n1)} \\ V & = & \texttt{m1} + \texttt{n1} \end{array}$$

Note that the variant V stays positive, but does not reach 0. Still, since the variant is bounded, termination is guaranteed.

2nd Question: Consecutive Loops

Consider the following recursive method `abs`, which returns the absolute value of an integer:

```
 1  int abs(int m)
 2  // PRE: true
 3  // POST: r = |m|
 4  {
 5      if( m > 0 ) {
 6          return m;
 7      } else {
 8          return -m;
 9      }
10  }
```

Now consider the following iterative method `CalcTerm`, which calculates the term $3m-2n$ using only increments and decrements of 1, and comparisons with 0:

```
 1  int CalcTerm(int m, int n)
 2  // PRE: n ≥ 0
 3  // POST: r = 3m − 2n
 4  {
 5      int m1 = abs(m);
 6      int n1 = n;
 7      int res = 0;
 8      // INV: I₁
 9      // VAR: V₁
10      while( m1 != 0 ) {
11          res = res + 3;
12          m1 = m1 - 1;
13      }
14      // MID: M₁
15      if( m < 0 ) {
16          res = -res;
17      }
18      // MID: M₂
19      // INV: I₂
20      // VAR: V₂
21      while( n1 != 0 ) {
22          res = res - 2;
23          n1 = n1 - 1;
24       }
25      // MID: M₃
26      return res;
27  }
```

Give appropriate loop invariants and variants that are strong enough to prove total correctness. Also, give midconditions for lines 14, 18, and 25, which show the cumulative effect of execution so far and are appropriate to show partial correctness.

A possible answer:

$$\begin{array}{rcl}
I_1 & \longleftrightarrow & \texttt{res} = 3(|\texttt{m}| - \texttt{m1}) \\
V_1 & = & \texttt{m1} \\
M_1 & \longleftrightarrow & \texttt{res} = 3|\texttt{m}| \\
M_2 & \longleftrightarrow & \texttt{res} = 3\texttt{m} \\
I_2 & \longleftrightarrow & \texttt{res} = 3\texttt{m} - 2(\texttt{n} - \texttt{n1}) \\
V_2 & = & \texttt{n1} \\
M_3 & \longleftrightarrow & \texttt{res} = 3\texttt{m} - 2\texttt{n}
\end{array}$$

3rd Question: Nested Loops

Consider the following iterative method `CalcProd`, which calculates the product $m * n$ using only increments and decrements of 1, and comparisons with 0:

```
1   int CalcProduct(int m, int n)
2   // PRE: m ≥ 0  ∧ n ≥ 0
3   // POST: r = m ∗ n
4   {
5       int m1 = m;
6       int res = 0;
7       // INV: I_1
8       // VAR: V_1
9       while( m1 != 0 ) {
10          int n1 = n;
11          // INV: I_2
12          // VAR: V_2
13          while( n1 != 0 ) {
14              res = res + 1;
15              n1 = n1 - 1;
16          }
17          // MID: M_1
18          m1 = m1 - 1;
19      }
20      // MID: M_2
21      return res;
22  }
```

Give appropriate loop invariants and variants that are strong enough to prove total correctness. Also, give midconditions for lines 17 and 20, which show the cumulative effect of execution so far and are appropriate to show partial correctness.

A possible answer:

$$\begin{array}{rcl}
I_1 & \longleftrightarrow & \texttt{res} = (\texttt{m} - \texttt{m1}) * \texttt{n} \\
V_1 & = & \texttt{m1} \\
I_2 & \longleftrightarrow & \texttt{res} = (\texttt{m} - \texttt{m1}) * \texttt{n} + (\texttt{n} - \texttt{n1}) \\
V_2 & = & \texttt{n1} \\
M_1 & \longleftrightarrow & \texttt{res} = (\texttt{m} - (\texttt{m1} - 1)) * \texttt{n} \\
M_2 & \longleftrightarrow & \texttt{res} = \texttt{m} * \texttt{n}
\end{array}$$

4th Question: Triply Nested Loops

Consider the following iterative method `CalcThreeProd`, which calculates the product $m * n * p$ using only increments and decrements of 1, and comparisons with 0:

```
int CalcThreeProduct(m: nat, n: nat, p: nat)
// PRE: $m \geq 0 \ \wedge n \geq 0 \ \wedge p \geq 0$
// POST: $r = m * n * p$
{
    int m1 = 0;
    int res := 0;
    // INV: $I_1$
    // VAR: $V_1$
    while( m1 != m) {
        int n1 = 0;
        // INV: $I_2$
        // VAR: $V_2$
        while( n1 != n ) {
            int p1 = 0;
            // INV: $I_3$
            // VAR: $V_3$
            while( p1 != p ) {
                res = res + 1;
                p1 = p1 + 1;
            }
            // MID: $M_1$
            n1 = n1 + 1;
        }
        // MID: $M_2$
        m1 = m1 + 1;
    }
    // MID: $M_3$
    return res ;
}
```

Give appropriate loop invariants and variants that are strong enough to prove total correctness. Also, give midconditions for lines 21, 24 and 27, which show the cumulative effect of execution so far and are appropriate to show partial correctness.

A possible answer:

$$
\begin{aligned}
I_1 &\longleftrightarrow \texttt{res} = \texttt{m1} * \texttt{n} * \texttt{p} \ \wedge \ \texttt{m1} < \texttt{m} \\
V_1 &= \texttt{m} - \texttt{m1} \\
I_2 &\longleftrightarrow \texttt{res} = (\texttt{m1} * \texttt{n} * \texttt{p}) + (\texttt{n1} * \texttt{p}) \ \wedge \ \texttt{n1} < \texttt{n} \\
V_2 &= \texttt{n} - \texttt{n1} \\
I_3 &\longleftrightarrow \texttt{res} = (\texttt{m1} * \texttt{n} * \texttt{p}) + (\texttt{n1} * \texttt{p}) + \texttt{p1} \ \wedge \ \texttt{p1} < \texttt{p} \\
V_3 &= \texttt{p} - \texttt{p1} \\
M_1 &\longleftrightarrow \texttt{res} = (\texttt{m1} * \texttt{n} * \texttt{p}) + (\texttt{n1} + 1) * \texttt{p} \ \wedge \ \texttt{m1} < \texttt{m} \ \wedge \ \texttt{n1} < \texttt{n} \\
M_2 &\longleftrightarrow \texttt{res} = (\texttt{m1} + 1) * \texttt{n} * \texttt{p} \ \wedge \ \texttt{m1} < \texttt{m} \\
M_3 &\longleftrightarrow \texttt{res} = \texttt{m} * \texttt{n} * \texttt{p}
\end{aligned}
$$

5th Question: The 4 Stripe Flag Problem

Given an enumeration type `Colour` with members `Red` and `White`, and an array of colours, say `Colour[] a`, rearrange `a` so that there are 4 stripes, (in the order `Red`, `White`, `Red`, `White`) where the size of the two `Red` stripes differ by at most 1 and similarly the size of the two `White` stripes differ by at most 1.

Any modification to the array has to be through the swap method given in the lecture slides. We use the notation $|\mathtt{m}|$, to indicate the absolute value of `m`, for example: $|-3| = 3 = |3|$. We also use the permutation notation $\mathtt{a} \sim \mathtt{a}_0$ as given in the lecture slides.

a) Assume that the method `fourSort` has the following specification:

```
enum Colour = {Red, White}
void fourSort(Colour[] a)
// PRE: a ≠ null
// POST: a ~ a₀
//          ∧  ∃k₁, k₂, k₃ ∈ [0..a.length].
//              [ a[0..k₁) = Red = a[k₂..k₃)
//                  ∧  a[k₁..k₂) = White = a[k₃..a.length)
//                  ∧  |k₁ − (k₃ − k₂)| ≤ 1  ∧  |(k₂ − k₁) − (a.length − k₃)| ≤ 1 ]
{ ... }
```

a) What would be the consequence of removing the $\mathtt{a} \sim \mathtt{a}_0$ conjunct from the post-condition? Would the `fourSort` method still solve the 4 Stripe Flag Problem?

b) What would be the consequence of changing the conjunct of the post-condition from

$$|k_1 - (k_3 - k_2)| \leq 1 \ \wedge\ |(k_2 - k_1) - (\mathtt{a.length} - k_3)| \leq 1$$

to

$$k_1 - (k_3 - k_2) \leq 1 \ \wedge\ (k_2 - k_1) - (\mathtt{a.length} - k_3) \leq 1$$

Would the `fourSort` method still solve the 4 Stripe Flag Problem?

b) Give all the possible outputs of calling the sort method (as specified above) on the following inputs:

i) `a` = `[R|W|R|W|R|W|R|W]`

ii) `a` = `[R|R|R|W|R|W|R|W]`

c) Consider the specification of `fourSort` as given in part (a). The code for `fourSort` will have the following outline:

```
void fourSort(Colour[] a)
// PRE: a ≠ null > 0
// POST: a ~ a₀
//          ∧  ∃k₁, k₂, k₃ ∈ [0..a.length].
//              [ a[0..k₁) = Red = a[k₂..k₃)
//                  ∧  a[k₁..k₂) = White = a[k₃..a.length)
//                  ∧  |k₁ − (k₃ − k₂)| ≤ 1  ∧  |(k₂ − k₁) − (a.length − k₃)| ≤ 1 ]
{
  ...
  // INV: ???
  // VAR: ???
  while( ??? ) {
    ...
  }
  // MID ⟷ POST
}
```

In the following, we shall systematically develop the code:

i) **From post-condition to invariant**: Find an invariant for the while loop.

   You will probably find it useful to use the boolean values `evenR` and `evenW` to track the size of the two `Red` and `White` stripes that have been created so far. When `evenR` is *true* then the two `Red` stripes are the same size, similarly for `evenW`.

ii) **From invariant to loop condition**: Find a loop condition the negation of the loop condition and the invariant imply the midcondition.

iii) **From invariant to loop code**: Write some code which preserves the invariant when the loop condition holds and which also "makes progress".

   You will probably want to use a case split similar to that in the `reorder` method from the lecture notes. You may also find that you need to case split within this depending on the values of `evenR` or `evenW`.

iv) **From invariant to initialization**: Write some code which establishes the invariant.

v) **Array Accesses**: Informally check that any array accesses in the code will be valid.

vi) **Termination**: Find a variant which decreases with *every* loop iteration and give its lower bound.

vii) **Putting it all together**: Write out the full code and give in comments all invariants, mid-conditions and variants.

viii) **Prove that the code satisfies its specification**:

   a) Prove that the initialization code establishes the invariant.

   b) Prove that the loop body preserves the invariant.
      (Remember that the precondition of swap must hold before it is called.)

   c) Prove that the midcondition holds immediately on termination of the loop.

   d) Prove that the variant is bounded and decreases on *every* loop iteration.

   e) Prove that all array access in the method are valid.

A possible answer:

See other sample answer document.