CS 220: Discrete Structures and their Applications

Loop Invariants

Chapter 3 in zybooks

Colorado State University

Program verification

How do we know our program works correctly?

In this lecture we will focus on a tool for verifying the correctness of programs that involve loops.

pre- and post-conditions

Precondition: what's true before a block of code

Postcondition: what's true after a block of code

Example: computing the square root of a number

Precondition: the input x is a positive real number

Postcondition: the output is a number y such that $y^2=x$

pre- and post-conditions

Precondition: what's true before a block of code

Postcondition: what's true after a block of code

Example: sorting a list of numbers

```
def sort(a_list) :
   # code for sorting the list
```

Precondition: a_list is a list of n numbers in arbitrary order

Postcondition: the list is permutation of the input list and is sorted in ascending order, i.e.

$$a\_list[i] \leq a\_list[i+1] \text{ for } i \in \{0,\ldots,n-2\}$$

programming as a contract

Specifying what each method does

- Specify it in a comment before/after method's header

Precondition

- What is assumed to be true before the method is executed
- **Caller obligation**

Postcondition

- Specifies what will happen if the preconditions are met – what the method guarantees to the caller
- **Method obligation**

Example

```
def factorial(n) :
    '''
    precondition:  n >= 0
    postcondition:  return value equals n!
    '''
```

Enforcing preconditions

```
def factorial(n) :
    '''
    precondition:  n >= 0
    postcondition:  return value equals n!
    '''
    if (n < 0) : raise ValueError
```

What about postconditions?

```
def factorial(n) :
      '''
      precondition:  n >= 0
      postcondition:  return value equals n!
      '''
      if (n < 0) : raise ValueError


assert factorial(5)==120
```

Can use assertions to verify that postconditions hold

Loop invariants as a way of reasoning about the state of your program

```
<pre-condition: n>0>
i = 0
 while (i < n) :
    i = i+1

<post-condition: i==n>
```

We want to prove the post-condition: i==n right after the loop

Example: loop index value after a loop

```
// precondition: n>=0
i = 0
// i<=n    loop invariant
 while (i < n) :
      // i < n  test passed
      //   AND
      //   i<=n  loop invariant
       i = i + 1
      // i <= n  loop invariant

// i>=n          WHY?
//    AND
// i <= n
//              → i==n
```

So we can conclude the obvious:

i==n right after the loop

But what if the body were:

i = i+2 ?

Loop invariants

A way to reason about the correctness of a program

A **loop invariant** is a predicate

- that is true directly before the loop executes
- that is true before and after each repetition of the loop body
- and that is true directly after the loop has executed

i.e., it is kept invariant by the loop.

What does it mean...

```
<loop invariant>
while(test) :
    <test AND loop
     invariant>
    // loop body
    <loop invariant>

<not test AND loop
invariant>
```

If we can prove that
the loop invariant holds before the loop
and that
the loop body keeps the loop invariant true

then we can infer that

not test AND loop invariant
holds after the loop terminates

Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop.

Example: sum of elements in an array

```
def total(a_list) :
    sum = 0
    i = 0
    // sum == sum of elements from 0...i-1
    while (i < n) :
        // sum == sum of elements 0...i-1
        sum += a_list[i]
        i++
        // sum == sum of elements 0...i-1
    // i==n (previous example) AND
    // sum == sum elements 0...i-1
    // →  sum == sum of elements 0...n-1
    return sum
```

Loop invariant for selection sort

```
def selection_sort (a_list) :
    for i in range(len(a_list) - 1) :
        min = i
        for j in range(i+1, len(a_list)) :
            if (a_list[j] < a_list[min]) :
                min = j
        a_list[i],a_list[min] = a_list[min],a_list[i]
```

Invariant?

Loop invariant for selection sort

```
def selection_sort (a_list) :
    for i in range(len(a_list) - 1) :
        min = i
        for j in range(i+1, len(a_list)) :
            if (a_list[j] < a_list[min]) :
                min = j
        a_list[i],a_list[min] = a_list[min],a_list[i]
```

Invariant: a_list[0]…a_list[i-1] are in sorted order

```
for i in range(n) :
    body
```

is equivalent with

```
i = 0
while(i<n) :
 body
 i = i+1
```

Closed Curve Game

There are two players, Red and Blue. The game is played on a rectangular grid of points:

```
6   . . . . . . .
5   . . . . . . .
4   . . . . . . .
3   . . . . . . .
2   . . . . . . .
1   . . . . . . .
    1 2 3 4 5 6 7
```

Red draws a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by a line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red's goal is to form a closed curve of red line segments. Blue's goal is to prevent Red from doing so.

See http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

Closed Curve Game

We can express this game as a computer program:

```
while (more line segments can be drawn):
        Red draws line segment
        Blue draws line segment
```

Question: Does either Red or Blue have a winning strategy?

Closed Curve Game

Answer: Yes! Blue is guaranteed to win the game by responding to each turn by Red in the following manner:

```
if (Red drew a horizontal line segment) {
    let i and j be such that Red's line segment connects (i,j) with (i,j+1)
    if (i>1) {
       draw a vertical line segment connecting (i-1,j+1) with (i,j+1)
    } else {
       draw a line segment anywhere
    }
 } else  // Red drew a vertical line segment
    let i and j be such that Red's line segment connects (i,j) with (i+1,j)
    if (j>1) {
       draw a horizontal line segment connecting (i+1,j-1) with (i+1,j)
    } else  {
       draw a line segment anywhere
    }
 }
```

Closed Curve Game

By following this strategy Blue guarantees that Red does not have an "upper right corner" at any step.

So, the invariant is:

There does not exist on the grid a pair of red line segments that form an upper right corner.

And in particular, Red has no closed curve!

Egyptian multiplication

	A	B	
	19	5	
/2	9	10	*2
/2	4	20	*2
/2	2	40	*2
/2	1	80	*2

throw away all rows with even A:

	A	B
	19	5
	9	10
	1	80
		————
add B's		95

--> the product !!

Can we show it works? Loop invariants!!

```
def egyptian_multiply(left, right) :
    # precondition: left>0 AND right>0
    a=left; b=right; p=0  #p: the product computed stepwise
    # p + (a*b) == left * right  loop invariant
    while (a!=0) :
        # a!=0 and p + (a*b) == left * right
        # loop condition and loop invariant
        if odd(a):
            p+=b
        a = a//2
        b = b * 2
        # p + (a*b) == left*right
    # a==0 and p+a*b == left*right --> p == left*right
    return p
```

Try it on 7 * 8

left	right	a	b	p
7	8	7	8	0
		3	16	+=b: 8
		1	32	+=b: 24
		0	64	+=b: 56

Try it on 8*7

left	right	a	b	p
8	7	8	7	0
		4	14	0
		2	28	0
		1	56	0
		0	118	+=b: 56

Relation to binary representation 19*5

```
  00101
  10011
 -------
    101   5
   1010  10
  00000
 000000
1010000  80
 -------
1011111  95
```

Incorporating loop invariants into your code

```
def egyptian_multiply(left, right) :
    # precondition: left>0 AND right>0
    a=left; b=right; p=0  #p:  the product
    assert p + (a*b) == left * right
    while (a!=0) :
        assert a!=0 and p + (a*b) == left * right
        # loop condition and loop invariant
        if not (a/2 == a//2) :
            p+=b
        a = a//2
        b = b * 2
        assert p + (a*b) == left*right
    assert a==0 and p+a*b == left*right
    return p
```