CORRECTNESS ISSUES AND LOOP INVARIANTS

Lecture 8
CS2110 – Fall 2014

About A2 and feedback. Recursion

2

S2 has been graded. If you got 30/30, you will probably have no feedback.

If you got less than full credit, there should be feedback showing you which function(s) is incorrect.

If you don't see feedback, ask for a regrade on the CMS. Please don't email anyone asking for a regrade.

We will put on the course website some recursive functions for you to write, to get practice with recursion. This will not be an assignment. But if you know you need practice, practice!

Preconditions and Postconditions

3

precondition Q — {x >= 0}
statement S — x= x + 1;
postcondition R — {x > 0}

This Hoare triple is true!

Write it like this: {Q} S {R}
Called a Hoare Triple, after Sir Tony Hoare. Introduced notation in a paper in 1969.

In a Java program, you have to make have to make the assertion comments: // {x >= 0}

{Q} S {R} is a true-false statement. Read it as follows:
Execution of S begun in a state in which Q is true is guaranteed to terminate, and in a state in which R is true.

Preconditions and Postconditions

4

precondition Q — {x >= 0}
statement S — x= x + 1;
postcondition R — {x = 0}

This Hoare triple is false!

Write it like this: {Q} S {R}
Called a Hoare Triple, after Sir Tony Hoare. Introduced notation in a paper in 1969.

In a Java program, you have to make have to make the assertion comments: // {x >= 0}

{Q} S {R} is a true-false statement. Read it as follows:
Execution of S begun in a state in which Q is true is guaranteed to terminate, and in a state in which R is true.

Annotating more completely with assertions

5

```
/** Return b^c. Precondition 0 <= c */
public static int exp(int b, int c) {
   {0 <= c}                          precondition!
   int ans;
   if (c == 0) {
       {0 = c}  ans= 1;  {ans = b^c}  —Hoare triples
   } else {
       {0 < c} ans= b * exp(b, c-1); {ans = b^c}
   }
   {ans = b^c}
   return ans;
}
```

The blue things are assertions –pre- and post-conditions. They help prove that IF 0 <= c at the beginning, ans = b^c before the return.

Axiomatic definition of a language

6

```
/** Return b^c. Precondition 0 <= c */
public static int exp(int b, int c) {
   {0 <= c}
   int ans;
   if (c == 0) {
     {0 = c} ans= 1; {ans = b^c}
   } else {
     {0 < c}
     ans= b * exp(b, c-1);
     {ans = b^c}
   }
   {ans = b^c}
   return ans;
}
```

Hoare gave rules for deciding whether a Hoare triple {Q} S {R} was correct. Defined the language in terms of correctness instead of execution.

See that in later courses.
We concentrate on one aspect: how to "prove" loops correct.

Axiomatic definition of a language

7

```
/** Return b^c. Precondition 0 <= c */
public static int exp(int b, int c) {
  {0 <= c}
  int ans;
  if (c == 0)  {
    {0 = c} ans= 1; {ans = b^c}
  } else {
    {0 < c}
    ans= b * exp(b, c-1);
    {ans = b^c}
  }
  {ans = b^c}
  return ans;
}
```

Hoare gave rules for deciding whether a Hoare triple {Q} S {R} was correct. Defined the language in terms of correctness instead of execution.

See that in later courses. We concentrate on one aspect: how to “prove” loops correct.

Reason for introducing loop invariants

8

```
Given c >= 0, store b^c in x
z= 1;  x= b;  y= c;
while (y != 0) {
   if (y is even) {
      x= x*x;  y= y/2;
   } else {
      z= z*x;  y= y - 1;
   }
}
{z  =  b^c}
```

Algorithm to compute b^c.

Can’t understand any piece of it without understanding everything.

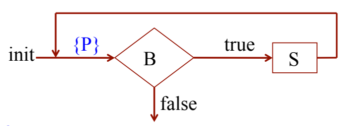
In fact, only way to get a handle on it is to execute it on some test case.

Need to understand initialization without looking at any other code.
Need to understand condition y != 0 without looking at method body
Etc.

Invariant: is true before and after each iteration

9

```
initialization;
// invariant P
while (B) {S}
```



Upon termination, we know P true, B false

“invariant” means unchanging. Loop invariant: an assertion —a true-false statement— that is true before and after each iteration of the loop —every time B is to be evaluated.

Help us understand each part of loop without looking at all other parts.

Simple example to illustrate methodology

10

```
Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

First loopy question.
Does it start right?
Does initialization make invariant true?

Yes!
s = sum of 0..k-1
= <substitute initialization>
0 = sum of 0..1-1
= <arithmetic>
0 = sum of 0..0

We understand initialization without looking at any other code

Simple example to illustrate methodology

11

```
Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Second loopy question.
Does it stop right?
Upon termination, is postcondition true?

Yes!
inv && ! k <= n
=> <look at inv>
inv && k = n+1
=> <use inv>
s = sum of 0..n+1-1

We understand that postcondition is true without looking at init or repetend

Simple example to illustrate methodology

12

```
Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv:  s = sum of 0..k-1 &&
//         0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Third loopy question.
Progress?
Does the repetend make progress toward termination?

Yes! Each iteration increases k, and when it gets larger than n, the loop terminates

We understand that there is no infinite looping without looking at init and focusing on ONE part of the repetend.

Simple example to illustrate methodology

13

```
Store sum of 0..n in s
Precondition: n >= 0
// { n >= 0}
k= 1; s= 0;
// inv: s = sum of 0..k-1 &&
//        0 <= k <= n+1
while (k <= n) {
    s= s + k;
    k= k + 1;
}
{s = sum of  0..n}
```

Fourth loopy question.
Invariant maintained by each iteration?

Is this Hoare triple true?
{inv && k <= n} repetend {inv}

Yes!

```
{s = sum of 0..k-1}
s= s + k;
{s =  sum of 0..k}
k= k+1;
{s = sum of 0..k-1}
```

4 loopy questions to ensure loop correctness

14

```
{precondition Q}
init;
// invariant P
while (B) {
    S
}
{R}
```

Four loopy questions: if answered yes, algorithm is correct.

First loopy question;
Does it start right?
Is {Q} init {P} true?

Second loopy question:
Does it stop right?
Does P && ! B imply R?

Third loopy question:
Does repetend make progress?
Will B eventually become false?

Fourth loopy question:
Does repetend keep invariant true?
Is {P && ! B} S {P} true?

Note on ranges m..n

15

Range m..n contains n+1–m ints: m, m+1, ..., n
(Think about this as "follower (n+1) minus first (m)")

2..4 contains 2, 3, 4: that is 4 + 1 – 2 = 3 values
2..3 contains 2, 3: that is 3 + 1 – 2 = 2 values
2..2 contains 2: that is 2 + 1 – 2 = 1 value
2..1 contains : that is 1 + 1 – 2 = 0 values

Convention: notation m..n implies that m <= n+1
Assume convention even if it is not mentioned!
If m is 1 larger than n, the range has 0 values

m n

array segment b[m..n]: b

Can't understand this example without invariant!

16

```
Given c >= 0, store b^c in z

z= 1;  x= b;  y= c;
// invariant  y >= 0  &&
//              z*x^y = b^c
while (y != 0) {
    if (y is even) {
       x= x*x;  y= y/2;
    } else {
       z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

First loopy question.
Does it start right?
Does initialization make invariant true?

Yes!
z*x^y
= <substitute initialization>
1*b^c
= <arithmetic>
b^c

We understand initialization without looking at any other code

For loopy questions to reason about invariant

17

```
Given c >= 0, store b^c in x

z= 1;  x= b;  y= c;
// invariant y >= 0  AND
//              z*x^y = b^c
while (y != 0) {
    if (y is even) {
       x= x*x;  y= y/2;
    } else {
       z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

Second loopy question.
Does it stop right?
When loop terminates, is z = b^c?

Yes! Take the invariant, which is true, and use fact that y = 0:
z*x^y = b^c
= <y = 0>
z*x^0 = b^c
= <arithmetic>
z = b^c

We understand loop condition without looking at any other code

For loopy questions to reason about invariant

18

```
Given c >= 0, store b^c in x

z= 1;  x= b;  y= c;
// invariant y >= 0  AND
//              z*x^y = b^c
while (y != 0) {
    if (y is even) {
       x= x*x;  y= y/2;
    } else {
       z= z*x;  y= y - 1;
    }
}
{z  =  b^c}
```

Third loopy question.
Does repetend make progress toward termination?

Yes! We know that y > 0 when loop body is executed. The loop body decreases y.

We understand progress without looking at initialization

For loopy questions to reason about invariant

19

Given c >= 0, store b^c in x

```
z= 1;  x= b;  y= c;
// invariant y >= 0  AND
//               z*x^y = b^c
while (y != 0) {
    if (y is even) {
        x= x*x;  y= y/2;
    } else {
        z= z*x;  y= y - 1;
    }
}
{z = b^c}
```

Fourth loopy question.
Does repetend keep invariant true?

Yes! Because of properties:

- For y even, x^y = (x*x)^(y/2)
- z*x^y = z*x*x^(y-1)

We understand invariance without looking at initialization

Designing while-loops or for-loops

20

Many loops process elements of an array b (or a String, or any list) in order: b[0], b[1], b[2], …

If the postcondition is

R: b[0..b.length-1] has been processed

Then in the beginning, nothing has been processed, i.e.

b[0..-1] has been processed

After k iterations, k elements have been processed:

P: b[0..k-1] has been processed

	0	k	b.length
invariant P: b	processed	not processed	

Developing while-loops (or for loops)

21

```
Task: Process b[0..b.length-1]
k= 0;
{inv P}
while (   k != b.length    ) {
    Process b[k];     // maintain invariant
    k= k + 1;         // progress toward termination
}
{R:  b[0..b.length-1]   has been processed}
```

Replace b.length in postcondition by fresh variable k to get invariant
b[0..k-1] has been processed

or draw it as a picture

	0	k	b.length
inv P: b	processed	not processed	

Developing while-loops (or for loops)

22

```
Task: Process b[0..b.length-1]
k= 0;
{inv P}
while (   k != b.length    ) {
    Process b[k];     // maintain invariant
    k= k + 1;         // progress toward termination
}
{R:  b[0..b.length-1]   has been processed}
```

Most loops that process the elements of an array in order will have this loop invariant and will look like this.

	0	k	b.length
inv P: b	processed	not processed	

Counting the number of zeros in b. Start with last program and refine it for this task

23

```
Task: Set s to the number of 0's in b[0..b.length-1]
k= 0;    s= 0;
{inv P}
while (   k != b.length    ) {
    if (b[k] == 0)  s= s + 1;
    k= k + 1;         // progress toward termination
}
{R: s = number of 0's in b[0..b.length-1]}
```

	0	k	b.length
inv P: b	s = # 0's here	not processed	

Be careful. Invariant may require processing elements in reverse order!

24

This invariant forces processing from beginning to end

	0	k	b.length
inv P: b	processed	not processed	

This invariant forces processing from end to beginning

	0	k	b.length
inv P: b	not processed	processed	

Process elements from end to beginning

25

```
k= b.length–1;              // how does it start?
while (k >= 0) {            // how does it end?
    Process b[k];           // how does it maintain invariant?
    k= k – 1;               // how does it make progress?
}
```

{R: b[0..b.length-1] is processed}

0 k b.length

inv P: b | not processed | processed |

Process elements from end to beginning

26

```
k= b.length–1;
while (k >= 0) {
    Process b[k];
}   k= k – 1;
```

Heads up! It is important that you can look at an invariant and decide whether elements are processed from beginning to end or end to beginning!

For some reason, some students have difficulty with this. A question like this could be on the prelim!

{R: b[0..b.length-1] is processed}

0 k b.length

inv P: b | not processed | processed |

Develop binary search for v in sorted array b

27

0 b.length

pre: b | ? |

0 h b.length

post: b | <= v | > v |

Example:

0 4 5 6 7 b.length

pre: b | 2 2 4 4 4 4 7 9 9 9 9 |

If v is 4, 5, or 6, h is 5

If v is 7 or 8, h is 6

If v in b, h is index of rightmost occurrence of v.
If v not in b, h is index before where it belongs.

Develop binary search in sorted array b for v

28

0 b.length

pre: b | ? |

Store a value in h to make this true:

0 h b.length

post: b | <= v | > v |

Get loop invariant by combining pre- and post- conditions, adding variable t to mark the other boundary

0 h t b.length

inv: b | <= v | ? | > v |

How does it start (what makes the invariant true)?

29

0 b.length

pre: b | ? |

0 h t b.length

inv: b | <= v | ? | > v |

Make first and last partitions empty:

h= -1; t= b.length;

When does it end (when does invariant look like postcondition)?

30

0 h b.length

post: b | <= v | > v |

0 h t b.length

inv: b | <= v | ? | > v |

```
h= -1; t= b.length;
while ( h != t-1) {

}
```

Stop when ? section is empty. That is when h = t-1.
Therefore, continue as long as h != t-1.

How does body make progress toward termination (cut ? in half) and keep invariant true?

31

inv:

	0	h	t	b.length
b	<= v	?	> v	

	0	h	e	t	b.length
b	<= v	?		> v	

```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
}
```

Let e be index of middle value of ? Section. Maybe we can set h or t to e, cutting ? section in half

How does body make progress toward termination (cut ? in half) and keep invariant true?

32

inv:

	0	h	t	b.length
b	<= v	?	> v	

	0	h	e		t	b.length
b	<= v	?		?	> v	

	0	h	e		t	b.length
b	<= v	<= v		?	> v	

```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
    if (b[e] <= v)  h= e;

}
```

If b[e] <= v, then so is every value to its left, since the array is sorted. Therefore, h= e; keeps the invariant true.

How does body make progress toward termination (cut ? in half) and keep invariant true?

33

inv:

	0	h	t	b.length
b	<= v	?	> v	

	0	h	e		t	b.length
b	<= v	?		?	> v	

	0	h	e		t	b.length
b	<= v	?		> v	> v	

```
h= -1;  t= b.length;
while ( h != t-1 ) {
    int  e= (h+t)/2;
    if (b[e] <= v)  h= e;
    else  t= e;
}
```

If b[e] > v, then so is every value to its right, since the array is sorted. Therefore, t= e; keeps the invariant true.