CSE 331
Software Design & Implementation

Dan Grossman
Fall 2014

Lecture 3 – Reasoning About Loops

Reasoning about loops

So far, two things made all our examples much easier:

1. When running the code, each statement executed 0 or 1 times
2. (Therefore,) trivially the code always terminates

Neither of these hold once we have loops (or recursion)

– Will consider the key ideas with while-loops
– Introduces the essential and much more general concept of an *invariant*
– Will mostly ignore prove-it-terminates; brief discussion at end

Informal example

As before, consider high-level idea before the precise Hoare-triple definitions

```
// assume: x >= 0
y = 0; i=0;
// x >= 0 ∧  y = 0 ∧  i = 0
// invariant: y = sum(1,i)
while(i != x) {
  // y = sum(1,i) ∧ i != x
  i = i+1;
  // y = sum(1,i-1)
  y = y+i;
  // y = sum(1,i-1)+i
}
// i=x ∧ y = sum(1,i)
// assert: y = sum(1,x)
```


Key lessons

- To reason about a loop (that could execute any number of iterations), we need a loop invariant
- The precondition for the loop must imply the invariant
  – (Precondition stronger than (or equal to) invariant)
- Invariant plus loop-test-is-true must be enough to show the postcondition of the loop body ***also*** implies the invariant (!)
- Invariant and loop-test-is-false must be enough to show the postcondition of the loop

The Hoare logic

- Consider just a while-loop (other loop forms not so different)

`{P} while B S {Q}`

Such a triple is valid if there exists an invariant `I` such that:

- `P => I` invariant must hold initially
- `{I ∧ B}S{I}` body must re-establish invariant
- `(I ∧ !B) => Q` invariant must establish `Q` if test-is-false

The loop-test `B`, loop-body `S`, and loop-invariant `I` "fit together":

– There is often more than one correct loop, but with possibly different invariants

Note definition "makes sense" even in the zero-iterations case

Example, more precisely

`{P} while B S {Q}`

- `P => I` invariant must hold initially
- `{I ∧ B}S{I}` body must re-establish invariant
- `(I ∧ !B) => Q` invariant must establish `Q` if test-is-false

```
{x >= 0}
y = 0; i=0;
{pre: x >= 0 ∧  y = 0 ∧  i = 0}
{inv: y = sum(1,i)}
while(i != x) {
  i = i+1;
  y = y+i;
}
{post: i=x ∧ y = sum(1,i)}
 (so: y = sum(1,x))
```

A different approach

A different loop has a different invariant

```
{x >= 0}
y = 0; i=1;
{pre: x >= 0 ∧  y = 0 ∧  i = 1}
{inv: y = sum(1,i-1)}
while(i != x+1) {
  y = y+i;
  i = i+1;
}
{post: i=x+1 ∧ y = sum(1,i-1)}
 (so: y = sum(1,x))
```


And find bugs

And this third approach doesn't do what we want

```
{x >= 0}
y = 0; i=1;
{pre: x >= 0 ∧  y = 0 ∧  i = 1}
{inv: y = sum(1,i-1)}
while(i != x) {
  y = y+i;
  i = i+1;
}
{post: i=x ∧ y = sum(1,i-1)}
```
~~(so: y = sum(1,x))~~

More bugs

- And this approach has an invalid Hoare triple hidden in it

```
{x >= 0}
y = 0; i=0;
{pre: x >= 0 ∧  y = 0 ∧  i = 0}
{inv: y = sum(1,i)}
while(i != x) {
  y = y+i;
  i = i+1;
  // invariant not satsified – why?
}
{post: i=x ∧ y = sum(1,i)}
```


Neither too strong nor too weak

- If loop invariant is too *strong*, it could be false!
  - Won't be able to prove it holds either initially or after loop-body
- If loop invariant is too *weak*, it could
  - Leave the post-condition too weak to prove what you want
  - And/or be impossible to re-establish after the loop body
- This is the essence of why there is no complete automatic procedure for conjuring a loop-invariant
  - Requires *thinking* (or, sometimes, "guessing")
  - Often while writing the code
  - If proof doesn't work, invariant or code or both may need work
- There may be multiple invariants that "work" (neither too strong nor too weak), with some easier to reason about than others

A methodology

- Fortunately, programming is creative and inventive!
- Here, this means coming up with a loop and its invariant
- Won't advocate a hard-and-fast rule, but do want to avoid the natural approach of "always code first, dream up invariant second"
- Instead, often surprisingly effective to go in this order:
  1. Think up the invariant first, have it guide all other steps (!)
     - What describes the milestone of each iteration?
  2. Write a loop body to maintain the invariant
  3. Write the loop test so false-implies-postcondition
  4. Write initialization code to establish invariant

Example

Set `max` to hold the largest value in array `items`

1. Think up the invariant first, have it guide all other steps
   - Invariant: `max` holds largest value in range `0..k-1` of `items`
   - Other approaches possible: Homework 2

Example

Set **max** to hold the largest value in array **items**

2. Write a loop body to maintain the invariant

```
{inv: max holds largest value in items[0..k-1]}
while(  ) {
  // inv holds
  if(max < items[k]) {
    max = items[k]; // breaks inv temporarily
  } else {
    // nothing to do
  }
  // max holds largest value in items[0..k]
  k = k+1; // invariant holds again
}
```


Example

Set **max** to hold the largest value in array **items**

3. Write the loop test so false-implies-postcondition

```
{inv: max holds largest value in items[0..k-1]}
while(k != items.size) {
  // inv holds
  if(max < items[k]) {
    max = items[k]; // breaks inv temporarily
  } else {
    // nothing to do
  }
  // max holds largest value in items[0..k]
  k = k+1; // invariant holds again
}
```


Example

Set **max** to hold the largest value in array **items**

4. Write initialization code to establish invariant

```
k=1;
max = items[0];
{inv: max holds largest value in items[0..k-1]}
while(k != items.size) {
  …
}
```


Edge case

- Our initialization code has a precondition: **items.size > 0**

```
{items.size > 0}
k=1;
max = items[0];
{inv: max holds largest value in items[0..k-1]}
while(k != items.size) {
  …
}
```

- Such a (specified!) precondition may be appropriate
- Else need a different postcondition ("if size is 0, …") and a conditional that checks for the empty case
  - Or the **Integer.MIN_VALUE** "trick" and logical reasoning
- Neat: Precise preconditions should expose all this to you!

More examples

- Here:
  - Quotient and remainder
  - "Dutch national flag problem" (like Homework 0)
- More in reading notes:
  - Reverse an array (have to refer to "original" values)
  - Binary search (invariant about range of array left to search)
- More on Homework 2:
  - Enjoy!

Quotient and remainder

Set **q** to be the quotient of **x / y** and **r** to be the remainder

Pre-condition: **x > 0 ∧ y > 0**
Post-condition: **q*y + r = x ∧ r >= 0 ∧ r < y**

A possible loop invariant: **q*y + r = x ∧ r >= 0**

A loop body that preserves the invariant:

```
q = q + 1;
r = r - y;
```

The loop test that given the invariant implies the post: **y <= r**

Initialization to establish invariant: **q = 0; r = x;**

Put it all together

```
{x > 0 ∧ y > 0} // can this be weakened?
r = x;
q = 0;
{inv: q*y + r = x ∧ r >= 0 }
while (y <= r) {
  q = q + 1;
  r = r - y;
}
{q*y + r = x ∧ r >= 0 ∧ r < y}
```


Dutch National Flag (classic example)

Given an array of red, white, and blue pebbles, sort the array so the red pebbles are at the front, white are in the middle, and blue are at the end

– [Use only swapping contents rather than "count and assign"]

Edsgar Dijkstra

Pre- and post-conditions

Precondition: Any mix of red, white, and blue

Mixed colors: red, white, blue

Postcondition:

– Red, then white, then blue
– Number of each color same as in original array

Red	White	Blue

Some potential invariants

Any of these four choices can work, making the array more-and-more sorted as you go:

Red	White	Blue	Mixed
Red	White	Mixed	Blue
Red	Mixed	White	Blue
Mixed	Red	White	Blue

Middle two slightly better because at most one swap per iteration instead of two

More precise, and then some code

- Precondition **P**: **arr** contains **r** reds, **w** whites, and **b** blues
- Postcondition: `P ∧ 0 <= i <= j <= arr.size`
  `∧ arr[0..i-1] is red`
  `∧ arr[i..j-1] is white`
  `∧ arr[j..arr.size-1] is blue`
- Invariant: `P ∧ 0 <= i <= j <= k <= arr.size`
  `∧ arr[0..i-1] is red`
  `∧ arr[i..j-1] is white`
  `∧ arr[k..arr.size-1] is blue`
- Initializing to establish the invariant (could do before or after body): `i=0; j=0; k=arr.size;`

The loop test and body

Red	White	Mixed	Blue
	i	j	k

```
while(j!=k) {
  if(arr[j] == White) {
     j = j+1;
  } else if (arr[j] == Blue) {
     swap(arr,j,k-1);
     k = k-1;
  } else { // arr[j] == Red
     swap(arr,i,j)
     i = i+1;
     j = j+1;
   }
}
```

```
void swap(int[] x,
          int y,
          int z) {
  int tmp = x[y];
  x[y] = x[z];
  x[z] = tmp;
}
```

Aside: swap

- Reading notes write `swap(a[i],a[j])` and such
- This is not implementable in Java
  - But fine pseudocode
  - Great exercise: Write a coherent English paragraph *why* it is not implementable in Java (i.e., does not do what you want)
- You can implement `swap(a,i,j)` in Java
  - So previous slide and Homework 2 do it that way

When to use proofs for loops

- Most loops are so "obvious" that proofs are, in practice, overkill
  - `for(String name : friends) {…}`
- Use logical reasoning when intermediate state (invariant) is unclear or edge cases are tricky or you need inspiration, etc.
- Use logical reasoning as an intellectual debugging tool
  - What *exactly* is the invariant?
  - Is it satisfied on every iteration?
  - Are you sure? Write code to check?
  - Did you check all the edge cases?
  - Are there preconditions you did not make explicit?

Termination

- Two kinds of loops
  - Those we want to always terminate (normal case)
  - Those that may conceptually run forever (e.g., web-server)
- So, proving a loop correct usually also requires proving termination
  - We haven't been proving this: might just preserve invariant forever without test ever becoming false
  - Our Hoare triples say ***if*** loop terminates, postcondition holds
- How to prove termination (variants exist):
  - Map state to a natural number somehow (just "in the proof")
  - Prove the natural number goes down on every iteration
  - Prove test is false by the time natural number gets to 0

Termination examples

- Quotient-and-remainder: `r` (starts positive, gets strictly smaller)
- Binary search: size of range still considered
- Dutch-national-flag: size of range not yet sorted (`k-j`)
- Search in a linked list: length of list not yet considered
  - Don't know length of list, but goes down by one each time…
  - … unless list is cyclic in which case, termination not assured