CSE 331
Software Design and Implementation

Lecture 3
Loop Reasoning

Leah Perlmutter / Summer 2018

Announcements

Announcements

- Casual Friday
- Congrats on making it through HW0 and Quiz1
- Thank you TAs for making section go!
  - Thoughts or ideas for section? Share with your sec TAs!
- Sorry for access issues with message board / gradescope / quiz1
  - TAs are working hard to straighten it all out
  - Post on discussion board if you still have issues
- HW1 due Monday June 25 at 10pm
- Reporting collaborators
  - You will get credit if you write something on the "Collaborators" line, can be "none" or a list of names.
  - No credit if you leave it blank
- Quiz1
  - accidental question about future lecture will be invalidated

Follow up

- Answer question on Combining Rule: Conditional
- Finish slides from Wednesday
  - I also got the feedback to go faster, so I practiced my lecture timing more for today

Loop Reasoning

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

Loop-Related Questions

Is this a valid Hoare Triple?

```
  {P}          while(B) S;          {Q}
```

```
{x = y ∧ x < 10} while(x != 10) x = x+1; {x > y}
```

Write code that does blah and prove that it is correct

```
{R} LotsOfCode; {Q}
```

Suppose LotsOfCode includes a loop. Proof may resemble this:

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

```
{R}init;{P} {I} while(B)  {I∧B} S {I}  {I∧!B} {Q}
```

Whiteboarding!

(See lecture notes)

:)

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

Need "Little Bear" Invariants

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

Recap: Do it backwards!

1. Start with postcondition (from spec)
2. Write an invariant (often a weaker form of the postcondition)
3. Implement the loop body
4. Backward reasoning to prove `{I ∧ B} S {I}`
5. Figure out `B` to fulfill `(I ∧ !B) => Q`
6. Initialization code to make `{I}` true before loop header
7. Precondition `{R}`

`{R}init;{P} {I} while(B) {I∧B} S {I} {I∧!B} {Q}`

Dutch National Flag Problem

Dutch National Flag (classic)

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 partitioned 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 code

Precondition **`P`**: **`arr`** contains **`r`** reds, **`w`** whites, and **`b`** blues

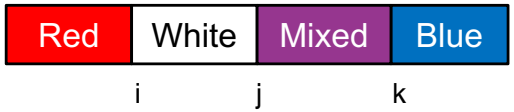
Postcondition: **`P ∧ 0 <= i <= j <= arr.size`**
`∧ arr[0..i-1] is red`
`∧ arr[i..j-1] is white`
`∧ arr[j..arr.size-1] is blue`

Invariant: **`P ∧ 0 <= i <= j <= k <= arr.size`**
`∧ arr[0..i-1] is red`
`∧ arr[i..j-1] is white`
`∧ arr[j..k-1] is unsorted`
`∧ arr[k..arr.size-1] is blue`

Exit when unsorted segment is empty, i.e. **`j=k`**

Initializing to establish the invariant: **`i=0; j=0; k=arr.size;`**

The loop test and body




```
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

In Practice ...

Many loops are so "obvious" that proofs are, in practice, overkill

- **`for(String name : friends) {…}`**

Often the intermediate state (invariant) is unclear or edge cases are tricky – use invariants here!

- Can draw a picture

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

Dutch-national-flag: size of unsorted range (`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

Closing

Closing

Recap of announcements

- HW1 due Monday June 25 at 10pm
- Reporting collaborators
  - You will get credit if you write something on the "Collaborators" line.
  - Can be "none" or a list of names, but don't leave blank

Are there any general questions on HW1?

- Office hours today
  - Wei: 2-3 pm
  - Joyce: 5-6 pm