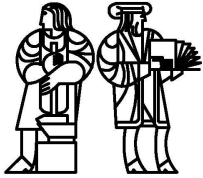




# 6.170 Lecture 15

## Reasoning About Code



MIT EECS



## Reasoning About Code

### Determine what facts are true during execution

- $x > 0$
- for all nodes  $n$ ,  $n.next.previous == n$
- array  $a$  is sorted
- $x + y = z$
- if  $x \neq null$ , then  $x.a > x.b$

### Applications:

- Ensure code is correct
- Understand why code is incorrect



## Forward reasoning

### You know what is true before running the code

What is true after running the code?

### Given precondition, what is postcondition?

#### Application:

- Rep invariant holds before running code
- Does it still hold after running code?

#### Example:

```
// precondition: x is even
x = x + 3;
y = 2x;
x = 5;
// postcondition: ??
```



## Backward reasoning

### You know what you want to be true after running the code

What must be true beforehand in order to ensure that?

### Given postcondition, what is the corresponding precondition?

#### Application:

- Desire to (re-)establish rep invariant at end of a method
- What should the method require?

#### Example:

```
// precondition: ??
x = x + 3;
y = 2x;
x = 5;
// postcondition: y > x
```

### How did you (informally) compute this?



### Forward reasoning is more intuitive for many people

- Introduces facts that may be irrelevant to goal
- Set of current facts may get large
- Takes longer to realize that the task is hopeless

### Backward reasoning is usually more helpful

- Given a specific goal, indicates how to achieve it



### General plan

- Eliminate code a statement at a time
- Rely on knowledge of logic and types
- Convert assertions about programs to logic

### There is a (forward and backward) rule for each statement in the programming language

- Loops have no rule: you have to *guess* a loop invariant

### Jargon: $P \{ \text{code} \} Q$

- P and Q are logical statements (about program values)
- code is Java code
- Means “if P is true and you execute code, then Q is true afterward”



```
// assert x ≥ 0
i = x;
  // x ≥ 0 & i = x
z = 0;
  // x ≥ 0 & i = x & z = 0
while (i != 0) {
  z = z + 1;
  i = i - 1;
}
  // x ≥ 0 & i = 0 & z = x
// assert x = z
```

### Now, on to backward reasoning rules for Java statements



```
// precondition: ??
x = e;
// postcondition: Q
```

### Precondition = Q with all (free) occurrences of x replaced by e

### Example:

```
// assert: ??
x = x + 1;
// assert x > 0
```

Precondition =  $(x+1) > 0$

### We write this as wp for “weakest precondition”

$wp(“x=e;”, Q) = Q$  with x replaced by e



```
// precondition: ??
x = foo();
// postcondition: Q
```

**If the method has no side effects: just like ordinary assignment**

**If it has side effects: an assignment to every var in modifies**

Use the method specification to determine the new value



```
// precondition: ??
S1;      // some statement
S2;      // another statement
// postcondition: Q
```

**Work from back to front**

**Postcondition = wp("s1; s2;", Q) = wp("s1;", wp("s2;", Q))**

**Example:**

```
// precondition: ??
x = 0;
y = x+1;
// postcondition: y > 0
```



```
// precondition: ??
if (b) S1 else S2
// postcondition: Q
```

**Essentially case analysis**

**wp("if b s1 else s2", Q) =**

**$b \Rightarrow \text{wp}("s1", Q) \wedge \neg b \Rightarrow \text{wp}("s2", Q)$**



```
// precondition: ??
if (x == 0) {
    x = x + 1;
} else {
    x = (x/x);
}
// postcondition: x ≥ 0
```

**Precondition**

**= wp("if (x==0) {x = x+1;} else {x = x/x}", x ≥ 0)**  
**= x = 0 ⇒ (wp("x = x+1", x ≥ 0) & x ≠ 0 ⇒ wp("x = x/x", x ≥ 0))**  
**= (x = 0 ⇒ x + 1 ≥ 0) & (x ≠ 0 ⇒ x/x ≥ 0)**  
**= 1 ≥ 0 & 1 ≥ 0**  
**= true**



### A loop represents an unknown number of paths

- Case analysis is problematic
- Recursion presents the same issue

### Cannot enumerate all paths

- What makes testing and reasoning hard



```
// assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

- 1) Pre-assertion guarantees that  $x \geq y$
- 2) Every time through loop
  - y incremented by 1
  - x unchanged
  - Therefore, y is closer to x
- 3) Since there are only a finite number of integers between x and y, y will eventually equal x
- 4) Execution exits the loop as soon as  $x = y$



### Just made an inductive argument

### What are we inducting over?

- Number of iterations

### Computation induction

- Show that conjecture holds if zero iterations
- Show that it holds after  $n+1$  iterations  
(assuming that it holds after  $n$  iterations)

### Two things to prove

- Some property is preserved (known as “partial correctness”)
  - Loop invariant is preserved by each iteration
- The loop completes (known as “termination”)
  - The “decrementing function” is reduced by each iteration



```
// assert P
while (b) S;           Equivalently: P {while (b) S;} Q
// assert Q
```

### Find an invariant, I, such that

- 1)  $P \Rightarrow I$
- 2)  $I \ \& \ b \ \{S\} \ I$
- 3)  $(I \ \& \ \neg b) \Rightarrow Q$

### Sufficient to know that if loop terminates, Q will hold

### Finding the invariant is the key to reasoning about loops

### Inductive assertions

- Complete method of proof
- If loop does satisfy pre/post conditions, there exists an invariant sufficient to show it



```
//assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
//assert x = y
```

So, what is a suitable invariant?

What makes the loop work?

$$I = x \geq y$$

- 3)  $x \geq 0 \& y = 0 \Rightarrow I$
- 4)  $I \& x \neq y \{y = y+1;\} I$
- 5)  $(I \& \neg(x \neq y)) \Rightarrow x = y$



Nothing I have done so far should convince you that the loop terminates

## Decrementing function

Maps subset of program variables to well-ordered set

## Well-ordered set

Ordered set

Every non-empty subset has least element

## Which of the following is well-ordered?

Natural numbers **Yes**

Non-negative reals **No,  $\{x \mid x \neq 0\}$**

Integers **No**



$$P\{\text{while } (b) \text{ } s;\} Q$$

D(X), where X is some subset of state, such that

- 1)  $I \& b \{S\} D(X') < D(X)$
- 2)  $(I \& D(X) = \text{minVal}) \Rightarrow \neg b$



```
// assert x ≥ 0 & y = 0
// Invariant: x ≥ y
// Decrements (x-y)
while (x != y) {
    y = y + 1;
}
// assert x = y
```

- 1) // assert  $(x-y) = d_0 \& (y \neq x)$   
 $y = y + 1;$   
// assert  $(x - (y+1)) < d_0$
- 2)  $(x \geq y \& x - y = 0) \Rightarrow (x = y)$



**For straight-line code, the wp (weakest precondition) function gives us the appropriate property**

**For loops, you have to guess**

The loop invariant

The decrementing function

**Then, use reasoning techniques to prove the goal property**

**If the proof doesn't work:**

Maybe you chose a bad invariant or decrementing function

Choose another and try again

Maybe the loop is incorrect

Fix the code

**Automatically choosing loop invariants is a research topic**



**I don't routinely write**

Loop invariants and decrementing functions

**I do write them when I am unsure about a loop**

**When I have evidence that a loop is not working**

Add invariant and decrementing function if missing

Write code to check them

Understand why the code doesn't work

Reason to ensure that no similar bugs remain