


6.170

6.170 Lecture 16

Reasoning About Code



MIT EECS

©MIT 6.170 Slide 1

6.170 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 $x != null$, then $x.a > x.b$

Applications:

- Ensure code is correct
- Understand why code is incorrect

©MIT 6.170 Slide 2

6.170 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: ??
```

©MIT 6.170 Slide 3

6.170 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?

©MIT 6.170 Slide 4

6.170 Forward vs. backward reasoning

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

©MIT 6.170 Slide 5

6.170 Reasoning about code statements

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: $P \{code\} Q$
 P and Q are logical statements (about program values)
`code` is Java code
“ $P \{code\} Q$ ” means “if P is true and you execute `code`, then Q is true afterward”

©MIT 6.170 Slide 6

6.170 Forward reasoning example

```

// 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

```

← What property holds here?

← What property holds here?

Now, on to backward reasoning rules for Java statements

©MIT 6.170 Slide 7

6.170 Assignment

```

// 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

©MIT 6.170 Slide 8

6.170 Method calls

```

// 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

©MIT 6.170 Slide 9

6.170 Composition (statement sequences; blocks)

```

// 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

```

©MIT 6.170 Slide 10

6.170 If statements

```

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

```

Essentially case analysis

$wp("if (b) s1 else s2", Q) =$
 $b \Rightarrow wp("s1", Q) \ \wedge \ \neg b \Rightarrow wp("s2", Q)$

©MIT 6.170 Slide 11

6.170 If, an Example

```

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

```

Precondition

$= wp("if (x==0) {x = x+1;} else {x = x/x};", x \ge 0)$
 $= x = 0 \Rightarrow (wp("x = x+1", x \ge 0) \ \& \ x \ne 0 \Rightarrow wp("x = x/x", x \ge 0))$
 $= (x = 0 \Rightarrow x + 1 \ge 0) \ \& \ (x \ne 0 \Rightarrow x/x \ge 0)$
 $= 1 \ge 0 \ \& \ 1 \ge 0$
 $= \text{true}$

©MIT 6.170 Slide 12

6.170 Reasoning About Loops

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

©MIT 6.170 Slide 13

6.170 Reasoning About Loops: values and termination

```
// 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
 $x \geq y$ still holds
 y is incremented by 1
 x is 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 $x = y$

©MIT 6.170 Slide 14

6.170 Understanding loops by induction

We 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

©MIT 6.170 Slide 15

6.170 Properties of Loop Invariant, LI

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

Find an invariant, LI, such that

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

It is sufficient to know that if loop terminates, Q will hold
Finding the invariant is the key to reasoning about loops
Inductive assertions is a complete method of proof:
 If a loop satisfies pre/post conditions, there exists an invariant sufficient to prove it

©MIT 6.170 Slide 16

6.170 Loop invariant for the example

```
//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?
 $LI = x \geq y$

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

©MIT 6.170 Slide 17

6.170 Total correctness via well-ordered sets

We have not established that the loop terminates
Suppose that the loop always reduces some variable's value. Does the loop terminate if the variable is a

- Natural number?
- Integer?
- Non-negative real number?
- Boolean?
- ArrayList?

The loop terminates if the variable values are (a subset of) a well-ordered set
 Ordered set
 Every non-empty subset has least element

©MIT 6.170 Slide 18

Decrementing function $D(X)$

Maps state (program variables) to some well-ordered set
This greatly simplifies reasoning about termination

Consider: **while** (b) s;

We seek $D(X)$, where X is the state, such that

1. An execution of the loop reduces the function's value:
 $LI \ \& \ b \ \{s\} \ D(X_{\text{post}}) < D(X_{\text{pre}})$
2. If the function's value is minimal, the loop terminates:
 $(LI \ \& \ D(X) = \text{minVal}) \Rightarrow \neg b$

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

Is this a good decrementing function?

1. Does the loop reduce the decrementing function's value?

// assert (y ≠ x); let $d_{\text{pre}} = (x-y)$

$y = y + 1;$

// assert $(x_{\text{post}} - y_{\text{post}}) < d_{\text{pre}}$

2. If the function has minimum value, does the loop exit?

$(x \geq y \ \& \ x - y = 0) \Rightarrow (x = y)$

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