Reasoning About Code

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 (via reasoning or testing)

Understand why code is incorrect

Forward reasoning

You know what is true before running the code

What is true after running the code?

Given a precondition, what is the postcondition?

Applications:

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 a postcondition, what is the corresponding precondition?

Application:

(Re-)establish rep invariant at method exit: what requires?

Reproduce a bug: what must the input have been?

Example:

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

How did you (informally) compute this?

Forward vs. backward reasoning

Forward reasoning is more intuitive for most people

- Helps you understand what will happen (simulates the code)
- 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

- Helps you understand what should happen
- Given a specific goal, indicates how to achieve it
- Given an error, gives a test case that exposes it

Reasoning about code statements

Goal: Convert assertions about programs into logic

General plan

- Eliminate code a statement at a time
- Rely on knowledge of logic and types

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"
- Is this notation good for forward or for backward reasoning?

Forward reasoning example

```
assert x >= 0;
i = x;
    // x ≥ 0  &  i = x
z = 0;
    // x ≥ 0  &  i = x  &  z = 0
while (i != 0) {        ⇐ What property holds here?
  z = z + 1;
  i = i –1;             ⇐ What property holds here?
}
    // x ≥ 0  &  i = 0  &  z = x
assert x == z;
```

Backward reasoning

Technique for backward reasoning:

Compute the weakest precondition (“wp”)

There is a wp rule for each statement in the programming language

Weakest precondition yields strongest specification for the computation (analogous to function specifications)

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

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 <u>modifies</u>

Use the method specification to determine the new value

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
```

If statements

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

Essentially case analysis

wp("`if (b) S1 else S2`", Q) =

$$(\quad b \Rightarrow wp(\text{“S1”}, Q) \quad \wedge \quad \neg b \Rightarrow wp(\text{“S2”}, Q) \ )$$

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}`", $x \geq 0$)

= ($x = 0 \Rightarrow$ wp("`x = x+1`", $x \geq 0$)

& $x \neq 0 \Rightarrow$ wp("`x = x/x`", $x \geq 0$))

= $(x = 0 \Rightarrow x + 1 \geq 0)$ & $(x \neq 0 \Rightarrow x/x \geq 0)$

= $1 \geq 0$ & $1 \geq 0$

= true

Reasoning About Loops

A loop represents an unknown number of paths

Case analysis is problematic

Recursion presents the same issue

Cannot enumerate all paths

That is what makes testing and reasoning hard

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

$x \geq y$ holds – and if body is entered, $x > y$
y is incremented by 1
x is unchanged
Therefore, y is closer to x (but $x \geq y$ still holds)

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 $x = y$

Understanding loops by induction

We just made an inductive argument

Inducting over the 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

Properties of Loop Invariant, LI

```
// assert P
while (b) S;
// assert Q
```

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

Find an invariant, LI, such that

1) $P \Rightarrow LI$ (true at start of first iteration)
2) LI & b {**S**} LI (preserved by each iteration)
3) $(LI \,\&\, \neg b) \Rightarrow Q$ (implies the desired post-condition)

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, then there exists an invariant sufficient to prove it

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 & $\neg(x \neq y)$) $\Rightarrow$ $x = y$

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

Decrementing Function

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

Proving Termination

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

Is "x-y" a good decrementing function?

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

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

   ```
   y = y + 1;
   ```

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

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

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

Choosing loop invariants

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

In Practice

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