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 \( x != \text{null}, \text{then } x.a > x.b \)

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
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: \( P \{ \text{code} \} Q \)
   P and Q are logical statements (about program values)
   \text{code} is Java code
   “\( P \{ \text{code} \} Q \)” means “if P is true and you execute \text{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) {
   z = z + 1;
   i = i - 1;
}  // x ≥ 0 & i = 0 & z = x
assert x == z;

⇐ What property holds here?
⇐ What property holds here?
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”

$wp(\text{“x=e;”}, Q) = Q$ 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 modifies
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 = \text{wp}("S1; S2;", Q) = \text{wp}("S1;", \text{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) =
( b \Rightarrow wp("S1", Q)
\land \neg b \Rightarrow wp("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 ≥ 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
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
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 \( x = y \)
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

\[
\begin{align*}
\text{Equivalently: } & \quad P \{ \text{while } (b) \ S; \} \ Q \\
\text{Find an invariant, LI, such that} & \\
1) \ & P \implies LI \quad \text{(true at start of first iteration)} \\
2) \ & LI \& b \ \{S\} \ LI \quad \text{(preserved by each iteration)} \\
3) \ & (LI \& \neg b) \implies Q \quad \text{(implies the desired post-condition)}
\end{align*}
\]

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
So, what is a suitable invariant?

What makes the loop work?

LI = x ≥ y

1) x ≥ 0 & y = 0 ⇒ LI
2) LI & x ≠ y {y = y + 1;} LI
3) (LI & ¬(x ≠ y)) ⇒ x = y
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: \textbf{while} (\( b \)) \textbf{S};

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

1. An execution of the loop reduces the function’s value:
   \( \text{LI} \& b \{s\} D(X_{\text{post}}) < D(X_{\text{pre}}) \)

2. If the function’s value is minimal, the loop terminates:
   \( (\text{LI} \& D(X) = \text{minVal}) \Rightarrow \neg b \)
Is “x-y” a good decrementing function?

1. Does the loop reduce the decrementing function’s value?
   
   // assert (y ≠ 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 ≥ y & x - y = 0) ⇒ (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