Lecture 2
Formal Reasoning
Announcements

Please vote for midterm date

Homework 0 due Friday at 10am
  • No late days accepted for this assignment

Homework 1 due Wednesday at 11pm
  • Using program logic sans loops
Formal Reasoning
Formalization and Reasoning

Geometry gives us incredible power

- Lets us represent shapes symbolically
- Provides basic truths about these shapes
- Gives rules to combine small truths into bigger truths

Geometric proofs often establish *general* truths

\[
a^2 + b^2 = c^2 \quad \text{p + q + r = 180}
\]
Formalization and Reasoning

Formal reasoning provides tradeoffs

+ Establish truth for many (possibly infinite) cases
+ Know properties ahead of time, before object exists
- Requires abstract reasoning and careful thinking
- Need basic truths and rules for combining truths

Today: develop formal reasoning for programs

• What is true about a program’s state as it executes?
• How do basic constructs change what’s true?
• Two flavors of reasoning: forward and backward
Reasoning About Programs

What is true of a program’s state as it executes?
• Given initial assumption or final goal

Examples:
• If \( x > 0 \) initially, then \( y == 0 \) when loop exits
• Contents of array \( arr \) refers to are sorted
• Except at one program point, \( x + y == z \)
• For all instances of Node \( n \),
  \( n.next == \text{null} \lor n.next.prev == n \)
• ...
Why Reason About Programs?

Essential complement to testing
- Testing shows specific result for a specific input

Proof shows general result for entire class of inputs
- Guarantee code works for any valid input
- Can only prove correct code, proving uncovers bugs
- Provides deeper understanding of why code is correct

Precisely stating assumptions is essence of spec
- “Callers must not pass null as an argument”
- “Callee will always return an unaliased object”
“Today a usual technique is to make a program and then to test it. While program testing can be a very effective way to show the presence of bugs, it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.”

-- Dijkstra (1972)
Our Approach

Hoare Logic, an approach developed in the 70’s
  • Focus on core: assignments, conditionals, loops
  • Omit complex constructs like objects and methods

Today: the basics for assign and if in 3 steps
  1. High-level intuition for forward and backward reasoning
  2. Precisely define assertions, preconditions, etc.
  3. Define weaker/stronger and weakest precondition

Next lecture: loops
How Does This Get Used?

Current practitioners rarely use Hoare logic explicitly
• For simple program snippets, often overkill
• For full language features (aliasing) gets complex
• Shines for developing loops with subtle invariants
  • See Homework 0, Homework 2

Ideal for introducing program reasoning foundations
• How does logic “talk about” program states?
• How can program execution “change what’s true”?
• What do “weaker” and “stronger” mean in logic?

All essential for specifying library interfaces!
Forward Reasoning Example

Suppose we initially know (or assume) \( w > 0 \)

\[
\begin{align*}
// & \quad w > 0 \\
x &= 17; \\
// & \quad w > 0 \land x == 17 \\
y &= 42; \\
// & \quad w > 0 \land x == 17 \land y == 42 \\
z &= w + x + y; \\
// & \quad w > 0 \land x == 17 \land y == 42 \land z > 59 \\
\end{align*}
\]

Then we know various things after, e.g., \( z > 59 \)
Backward Reasoning Example

Suppose we want $z < 0$ at the end

```plaintext
// w + 17 + 42 < 0
x = 17;
// w + x + 42 < 0
y = 42;
// w + x + y < 0
z = w + x + y;
// z < 0
```

Then initially we need $w < -59$
Forward vs. Backward

Forward Reasoning
• Determine what follows from initial assumptions
• Useful for ensuring an invariant is maintained

Backward Reasoning
• Determine sufficient conditions for a certain result
• Desired result: assumptions need for correctness
• Undesired result: assumptions needed to trigger bug
Forward vs. Backward

Forward Reasoning
- Simulates the code for many inputs at once
- May feel more natural
- Introduces (many) potentially irrelevant facts

Backward Reasoning
- Often more useful, shows how each part affects goal
- May feel unnatural until you have some practice
- Powerful technique used frequently in research
Conditionals

```c
bool b = C
// initial assumptions
if(b) {
    ... // also know condition is true
} else {
    ... // also know condition is false
}
// either branch could have executed
```

Key ideas:

1. The precondition for each branch includes information about the result of the condition
2. The overall postcondition is the disjunction (“or”) of the postconditions of the branches
Conditionals

// initial assumptions
if(...) {
    ... // also know condition is true
} else {
    ... // also know condition is false
}
// either branch could have executed

Key ideas:
1. The precondition for each branch includes information about the result of the condition
2. The overall postcondition is the disjunction (“or”) of the postconditions of the branches
Conditional Example (Fwd)

```c
// x >= 0
z = 0;
// x >= 0 ∧ z == 0
if (x != 0) {
    // x >= 0 ∧ z == 0 ∧ x != 0 (so x > 0)
    z = x;
    // ... ∧ z > 0
}
else {
    // x >= 0 ∧ z == 0 ∧ !(x!=0) (so x == 0)
    z = x + 1;
    // ... ∧ z == 1
}
// ( ... ∧ z > 0) ∨ (... ∧ z == 1) (so z > 0)
```
Our Approach

Hoare Logic, an approach developed in the 70’s
  • Focus on core: assignments, conditionals, loops
  • Omit complex constructs like objects and methods

Today: the basics for assign and if in 3 steps

1. High-level intuition for forward and backward reasoning
2. Precisely define assertions, preconditions, etc.
3. Define weaker/stronger and weakest precondition

Next lecture: loops
Notation and Terminology

Precondition: “assumption” before some code

Postcondition: “what holds” after some code

Conventional to write pre/postconditions in “{...}”

{ w < -59 }

x = 17;

{ w + x < -42 }
Notation and Terminology

Note the “{ . . . }” notation is NOT Java

Within pre/postcondition “=” means mathematical equality, like Java’s “==” for numbers

\[
\{ \ w > 0 \ /\ x = 17 \ \} \\
y = 42; \\
\{ \ w > 0 \ /\ x = 17 \ /\ y = 42 \ \}
\]
Assertion Semantics (Meaning)

An *assertion* (pre/postcondition) is a logical formula that can refer to program state (variables)

Given a variable, a *program state* tells you its value
  • Or the value for any expression with no side effects

An assertion *holds* on a program state if evaluating the assertion using the program state produces *true*
  • An assertion represents the set of state for which it holds
A *Hoare triple* is code wrapped in two assertions

\[
\{ P \} \quad S \quad \{ Q \}
\]

- \( P \) is the precondition
- \( S \) is the code (statement)
- \( Q \) is the postcondition

Hoare triple \( \{ P \} \ S \ \{ Q \} \) is *valid* if:

- For all states where \( P \) holds, executing \( S \) always produces a state where \( Q \) holds
- “If \( P \) true before \( S \), then \( Q \) must be true after”
- Otherwise the triple is *invalid*
Hoare Triple Examples

Valid or invalid?

- Assume all variables are integers without overflow

  \{ x \neq 0 \} \ y = x^2; \ { y > 0 \} \quad \text{valid}

  \{ z \neq 1 \} \ y = z^2; \ { y \neq z \} \quad \text{invalid}

  \{ x \geq 0 \} \ y = 2x; \ { y > x \} \quad \text{invalid}

  \{ \text{true} \} \ (\text{if}(x > 7)\{ y=4; \} \text{else}\{ y=3; \}) \ { y < 5 \} \quad \text{valid}

  \{ \text{true} \} \ (x = y; \ z = x;) \ { y=z \} \quad \text{valid}

  \{ x=7 \land y=5 \}

  (\text{tmp}=x; \ x=\text{tmp}; \ y=x;) \quad \text{invalid}

  \{ y=7 \land x=5 \}

  valid

  invalid
Aside: assert in Java

A Java assertion is a statement with a Java expression

```
assert (x > 0 && y < x);
```

Similar to our assertions
- Evaluate with program state to get true or false

Different from our assertions
- Java assertions work at run-time
- Raise an exception if this execution violates assert
- … unless assertion checking disable (discuss later)

This week: we are reasoning about the code statically (before run-time), not checking a particular input
The General Rules

So far, we decided if a Hoare trip was valid by using our informal understanding of programming constructs.

Now we’ll show a general rule for each construct:

• The basic rule for assignments (they change state!)
• The rule to combine statements in a sequence
• The rule to combine statements in a conditional
• The rule to combine statements in a loop [next time]
Basic Rule: Assignment

\[
\{ \ P \ \} \ x = e; \ \{ \ Q \ \}
\]

Let \( Q' \) be like \( Q \) except replace \( x \) with \( e \)

Triple is valid if:

- For all states where \( P \) holds, \( Q' \) also holds
  - That is, \( P \) implies \( Q' \), written \( P \implies Q' \)

Example: \( \{ \ z > 34 \ \} \ y = z + 1; \ \{ \ y > 1 \ \} \)
  - \( Q' \) is \( \{ \ z + 1 > 1 \ \} \)
Combining Rule: Sequence

\[ \{ P \} \; S_1 ; \; S_2 \; \{ Q \} \]

Triple is valid iff there is an assertion \( R \) such that both the following are valid:

- \( \{ P \} \; S_1 \; \{ R \} \)
- \( \{ R \} \; S_2 \; \{ Q \} \)

Example:

\[
\begin{align*}
\{ \; z & \geq 1 \; \} \\
y & = z + 1; \\
w & = y \; * \; y; \\
\{ \; w & > y \; \}
\end{align*}
\]

Let \( R \) be \( \{ y > 1 \} \)

1. Show \( \{ z \geq 1 \} \; y = z + 1 \; \{ y > 1 \} \)
   Use basic assign rule:
   \[ z \geq 1 \; \text{implies} \; z + 1 > 1 \]

2. Show \( \{ y > 1 \} \; w = y \; * \; y \; \{ w > y \} \)
   Use basic assign rule:
   \[ y > 1 \; \text{implies} \; y \; * \; y > y \]
Combining Rule: Conditional

\{ P \} \text{ if}(b) S_1 \text{ else } S_2 \{ Q \}

Triple is valid iff there are assertions $Q_1$, $Q_2$ such that:

- $\{ P \land \neg b \} \ S_1 \{ Q_1 \}$ is valid
- $\{ P \land \neg \neg b \} \ S_2 \{ Q_2 \}$ is valid
- $Q_1 \land \neg Q_2$ implies $Q$

Example:

\{ true \}
if\( (x > 7) \)
y = x;
else
y = 20;
\{ y > 5 \}

Let $Q_1$ be $\{ \text{y} > 7 \}$ and $Q_2$ be $\{ \text{y} = 20 \}$
- Note: other choices work too!

1. Show $\{ \text{true} \land \neg x > 7 \} \ y = x \ \{ \text{y} > 7 \}$
2. Show $\{ \text{true} \land \neg x \leq 7 \} \ y = 20 \ \{ \text{y} = 20 \}$
3. Show $\text{y} > 7 \land \neg \text{y} = 20$ implies $\text{y} > 5$
Our Approach

Hoare Logic, an approach developed in the 70’s
  • Focus on core: assignments, conditionals, loops
  • Omit complex constructs like objects and methods

Today: the basics for assign and if in 3 steps
  1. High-level intuition for forward and backward reasoning
  2. Precisely define assertions, preconditions, etc.
  3. Define weaker/stronger and weakest precondition

Next lecture: loops
Weaker vs. Stronger

If \( P_1 \) implies \( P_2 \) (written \( P_1 \implies P_2 \)) then:

- \( P_1 \) is stronger than \( P_2 \)
- \( P_2 \) is weaker than \( P_1 \)

Whenever \( P_1 \) holds, \( P_2 \) is guaranteed to hold

- So it is at least as difficult to satisfy \( P_1 \) as \( P_2 \)
- \( P_1 \) holds on a subset of the states where \( P_2 \) holds
- \( P_1 \) puts more constraints on program states
- \( P_1 \) is a “stronger” set of obligations / requirements
Weaker vs. Stronger Examples

\[ x = 17 \] is stronger than \( x > 0 \)

\( x \text{ is prime} \) is neither stronger nor weaker than \( x \text{ is odd} \)

\( x \text{ is prime} \land x > 2 \) is stronger than \( x \text{ is odd} \lor x > 2 \)

...
Strength and Hoare Logic

Suppose:

• \( \{P\} S \{Q\} \) and
• \( P \) is weaker than some \( P_1 \) and
• \( Q \) is stronger than some \( Q_1 \)

Then \( \{P_1\} S \{Q\} \) and \( \{P\} S \{Q_1\} \) and \( \{P_1\} S \{Q_1\} \)

Example:

• \( P \) is \( x \geq 0 \)
• \( P_1 \) is \( x > 0 \)
• \( S \) is \( y = x + 1 \)
• \( Q \) is \( y > 0 \)
• \( Q_1 \) is \( y \geq 0 \)

“Wiggle Room”
Strength and Hoare Logic

For backward reasoning, if we want $\{P\}S\{Q\}$, we could:

1. Show $\{P_1\}S\{Q\}$, then
2. Show $P \Rightarrow P_1$

Better, we could just show $\{P_2\}S\{Q\}$ where $P_2$ is the \textit{weakest precondition} of $Q$ for $S$

- Weakest means the most lenient assumptions such that $Q$ will hold after executing $S$
- Any precondition $P$ such that $\{P\}S\{Q\}$ is valid will be stronger than $P_2$, i.e., $P \Rightarrow P_2$

Amazing (?) : Without loops/methods, for any $S$ and $Q$, there exists a unique weakest precondition, written $wp(S,Q)$

- Like our general rules with backward reasoning
Weakest Precondition

\[ wp(x = e, Q) \] is \( Q \) with each \( x \) replaced by \( e \)

- Example: \( wp(x = y*y;, x > 4) \) is \( y*y > 4 \), i.e., \( |y| > 2 \)

\[ wp(S1;S2, Q) \] is \( wp(S1, wp(S2, Q)) \)

- i.e., let \( R \) be \( wp(S2, Q) \) and overall \( wp \) is \( wp(S1, R) \)
- Example: \( wp((y=x+1; z=y+1;), z > 2) \) is \( (x + 1) + 1 > 2 \), i.e., \( x > 0 \)

\[ wp(if \ b \ S1 \ else \ S2, Q) \] is this logical formula:

\[ (b \wedge wp(S1,Q)) \vee (!b \wedge wp(S2,Q)) \]

- In any state, \( b \) will evaluate to either true or false...
- You can sometimes then simplify the result
Simple Examples

If $S$ is $x = y*y$ and $Q$ is $x > 4$, then $\wp(S,Q)$ is $y*y > 4$, i.e., $|y| > 2$.

If $S$ is $y = x + 1; z = y - 3;$ and $Q$ is $z = 10$, then $\wp(S,Q)$ ...

$= \wp(y = x + 1; z = y - 3; , z = 10)$
$= \wp(y = x + 1; , \wp(z = y - 3; , z = 10))$
$= \wp(y = x + 1; , y - 3 = 10)$
$= \wp(y = x + 1; , y = 13)$
$= x + 1 = 13$
$= x = 12$
Bigger Example

S is if (x < 5) {
    x = x*x;
} else {
    x = x+1;
}

Q is x >= 9

wp(S, x >= 9)
= (x < 5 ∧ wp(x = x*x; , x >= 9))
    ∨ (x >= 5 ∧ wp(x = x+1; , x >= 9))
= (x < 5 ∧ x*x >= 9)
    ∨ (x >= 5 ∧ x+1 >= 9)
= (x <= -3) ∨ (x >= 3 ∧ x < 5)
    ∨ (x >= 8)
Conditionals Review

Forward reasoning

\{P\}

\textbf{if} B

\{P \land B\}

S1

\{Q1\}

\textbf{else}

\{P \land \neg B\}

S2

\{Q2\}

\{Q1 \lor Q2\}

Backward reasoning

\{ (B \land \text{wp}(S1, Q)) \lor (\neg B \land \text{wp}(S2, Q)) \}

\textbf{if} B

\{\text{wp}(S1, Q)\}

S1

\{Q\}

\textbf{else}

\{\text{wp}(S2, Q)\}

S2

\{Q\}

\{Q\}
“Correct”

If $\text{wp}(S, Q)$ is true, then executing $S$ will always produce a state where $Q$ holds, since true holds for every program state.
Oops! Forward Bug…

With forward reasoning, our intuitive rule for assignment is wrong:

• Changing a variable can affect other assumptions

Example:

\[
\begin{align*}
\{ & \text{true} \} \\
&w = x + y; \\
x &= 4; \\
&w = x + y \land x = 4 \\
y &= 3; \\
&w = x + y \land x = 4 \land y = 3
\end{align*}
\]

But clearly we do not know \( w = 7 \) (!!!)
Fixing Forward Assignment

When you assign to a variable, you need to replace all other uses of the variable in the post-condition with a different “fresh” variable, so that you refer to the “old contents”

Corrected example:

```plaintext
{ true }
w = x + y;
{x = 4; }
w = x1 + y ∧ x = 4
{ y = 3; }w = x1 + y1 ∧ x = 4 ∧ y = 3
```
Useful Example: Swap

Name initial contents so we can refer to them in the post-condition

Just in the formulas: these “names” are not in the program

Use these extra variables to avoid “forgetting” “connections”

\[
\begin{align*}
\{ & x = x_{\text{pre}} \land y = y_{\text{pre}} \\
& \text{tmp} = x; \\
& x = y; \\
& \{ & x = y_{\text{pre}} \land y = y_{\text{pre}} \land \text{tmp} = x_{\text{pre}} \\
& y = \text{tmp}; \\
& \{ & x = y_{\text{pre}} \land y = \text{tmp} \land \text{tmp} = x_{\text{pre}}
\end{align*}
\]