Announcements

• First section tomorrow!
• Homework 0 due today (Wednesday) at 10 pm
  • Heads up: no late days for this one!
• Quiz 1 due tomorrow (Thursday) at 10 pm
• Homework 1 due Monday at 10 pm
  • Will be posted by tomorrow
• Message board
  • Use “needs-answer” tag on questions that need an answer
• Collaboration policy clarification
Overview

- Motivation
- Reasoning Informally
- Hoare Logic
- Weaker and Stronger Statements
- Variable Renaming

Note: This lecture has very helpful notes on the course website!
Why 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

\[ a^2 + b^2 = c^2 \]
\[ p + q + r = 180 \]

\[ \frac{AX}{XB} \cdot \frac{BY}{YC} \cdot \frac{CZ}{ZA} = -1 \]
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

• Formal reasoning tells us what’s true of a program’s state as it executes, given an initial assumption or a final goal

• What are some things we might want to know about certain programs?
  • If $x > 0$ initially, then $y == 0$ when loop exits
  • Contents of array $\text{arr}$ refers to are sorted
  • Except at one program point, $x + y == z$
  • For all instances of $\text{Node } n$,
    
    $n.next == \text{null} \lor n.next.prev == n$
  
  • ...
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”
Why Reason About Programs?

“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)
Why Reason About Programs?

• Re-explain to your neighbor (groups of 3-4)
• TAs may have some useful insights!
• Then share interesting thoughts/questions from your discussions.
Overview

- Motivation
- Reasoning Informally
- Hoare Logic
- Weaker and Stronger Statements
- Variable Renaming
Reasoning Informally
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, sequence, 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!
Informal Notation Warning

• The slides in this section have informal notation
• You will need to use more formal notation on your homework (after hw0)
Forward Reasoning Example

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

```plaintext
// w > 0
x = 17;
// w > 0 ∧ x == 17
y = 42;
// w > 0 ∧ x == 17 ∧ y == 42
z = w + x + y;
// w > 0 ∧ x == 17 ∧ y == 42 ∧ z > 59
...
```

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

// 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)
```
Overview

✓ Motivation
✓ Reasoning Informally
☐ Hoare Logic
☐ Weaker and Stronger Statements
☐ Variable Renaming
Hoare Logic
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, sequence, 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 \ }

Preconditions and Postconditions are two types of Formal Assertions.
Notation and Terminology

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

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

\[
\{ \, w > 0 \, \land \, x = 17 \, \}\]

\[
y = 42; \\
\{ \, w > 0 \, \land \, x = 17 \, \land \, y = 42 \, \} \\
\]
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
Hoare Triples

A *Hoare triple* is code wrapped in two assertions

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

- \(P\) 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

\[
\begin{align*}
\{x \neq 0\} & \quad y = x*x; \quad \{y > 0\} & \quad \text{valid} \\
\{z \neq 1\} & \quad y = z*z; \quad \{y \neq z\} & \quad \text{invalid} \\
\{x \geq 0\} & \quad y = 2*x; \quad \{y > x\} & \quad \text{invalid} \\
\{\text{true}\} & \quad (\text{if}(x > 7)\{y=4;\}\text{else}\{y=3;\}) \quad \{y < 5\} & \quad \text{valid} \\
\{\text{true}\} & \quad (x = y; \quad z = x;) \quad \{y=z\} & \quad \text{valid} \\
\{x=7 \land y=5\} & \quad \text{invalid} \\
(\text{tmp}=x; \quad x=\text{tmp}; \quad y=x;) & \quad \text{invalid} \\
\{y=7 \land x=5\} &
\end{align*}
\]
Aside: assert in Java

A Java assertion is a statement with a Java expression

```java
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:

\[
\{ \; z \; \geq \; 1 \; \}
\]

\[
y = z + 1;
w = y \times y;
\]

\[
\{ \; w \; > \; y \; \}
\]

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

\[
\{ P \} \text{if}(b) \ S1 \text{ else } S2 \{ Q \}
\]

Triple is valid iff there are assertions \( Q1, Q2 \) such that:

- \( \{ P \land b \} \ S1 \{ Q1 \} \) is valid
- \( \{ P \land \neg b \} \ S2 \{ Q2 \} \) is valid
- \( Q1 \lor Q2 \) implies \( Q \)

Example:

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

Let \( Q1 \) be \( \{ y > 7 \} \) and \( Q2 \) be \( \{ y = 20 \} \)
- Note: other choices work too!

1. Show \( \{ \text{true} \land x > 7 \} \ y = x \ \{ y > 7 \} \)
2. Show \( \{ \text{true} \land x \leq 7 \} \ y = 20 \ \{ y = 20 \} \)
3. Show \( y > 7 \lor y = 20 \) implies \( y > 5 \)
Combining Rule: Conditional

What if we change the code in a way that changes Q2 to “y=4”
Combining Rule: Conditional

```
{ P } if(b) S1 else S2 { Q }
```

Triple is valid iff there are assertions \( Q_1 \), \( Q_2 \) such that:

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

Example:

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

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

1. Show \( \{ true \land x > 7 \} y = x \{ y > 7 \} \)
2. Show \( \{ true \land x <= 7 \} y = 20 \{ y = 20 \} \)
3. Show \( y > 7 \lor y = 20 \) implies \( y > 5 \)
Overview

- Motivation
- Reasoning Informally
- Hoare Logic
- Weaker and Stronger Statements
- Variable Renaming
Weaker and Stronger Statements
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, sequence, 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 \( x > 0 \)

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

\( x \text{ is prime } \land \ x > 2 \) is stronger than \( x \text{ is odd } \land \ x > 2 \)
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 \( y \geq 0 \)
Strength and Hoare Logic

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

1. Show \( \{ P1 \} S \{ Q \} \), then
2. Show \( P \rightarrow P1 \)

Better, we could just show \( \{ P2 \} S \{ Q \} \) where \( P2 \) is the 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 \( P2 \), i.e., \( P \rightarrow P2 \)

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

- Like our general rules with backward reasoning
Weakest Precondition

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

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

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

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

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

\( (b \land \text{wp}(S1,Q)) \lor (!b \land \text{wp}(S2,Q)) \)

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

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

If $S$ is $y = x + 1; z = y - 3; \text{ and } Q$ 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

\[ S \text{ is } \begin{cases} \text{if } (x < 5) \{ \\
    x = x \times x; \\
\} \text{ else } \{ \\
    x = x + 1; \\
\} \end{cases} \]

\[ Q \text{ is } x \geq 9 \]

\[ \text{wp}(S, x \geq 9) = (x < 5 \land \text{wp}(x = x \times x; , x \geq 9)) \lor (x \geq 5 \land \text{wp}(x = x + 1; , x \geq 9)) \]

\[ = (x < 5 \land x \times x \geq 9) \lor (x \geq 5 \land x + 1 \geq 9) \]

\[ = (x \leq -3) \lor (x \geq 3 \land x < 5) \lor (x \geq 8) \]
Conditionals Review

Forward reasoning

\{P\}

if B

\{P \land B\}

S1

\{Q1\}

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)) \}

if B

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

S1

\{Q\}

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.

If our program state only has one variable, $x$, we can think of the true precondition as an assertion that holds for all values of $x$. 

\[ x \]

\[ -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 \]
Oops! Forward Bug...

With forward reasoning, our intuitive rule for assignment is **wrong**:  
- Changing a variable can affect other assumptions

Example:

```plaintext
{true}
w = x+y;
{w = x + y;}
x = 4;
{w = x + y ∧ x = 4}
y = 3;
{w = x + y ∧ x = 4 ∧ y = 3}
```

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:

```
{true}
w=x+y;
{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”

```plaintext
{x = x_pre ∧ y = y_pre}

tmp = x;
{x = x_pre ∧ y = y_pre ∧ tmp=x}

x = y;
{x = y ∧ y = y_pre ∧ tmp=x_pre}

y = tmp;
{x = y_pre ∧ y = tmp ∧ tmp=x_pre}
```
Announcements
Announcements

- First section tomorrow!
- Homework 0 due today (Wednesday) at 10 pm
  - Heads up: no late days for this one!
- Quiz 1 due tomorrow (Thursday) at 10 pm
- Homework 1 due Monday at 10 pm
  - Will be posted by tomorrow
- Read the Formal Reasoning Notes
  - posted on the course website
- Friday’s lecture is on one of the hardest topics