CSE 331
Software Design & Implementation

Hal Perkins
Winter 2015
Lecture 2 – Reasoning About Code With Logic
(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins)
Administrivia

• May need to move the midterm: Fri. 2/13 collides with 341 and 421. How many people are in both?

• Website reminders – if you haven’t already, please:
  – Fill in the office hours doodle
  – Post a followup to the welcome message on the discussion board

• If you’re not registered and want to add, be sure your name is on the signup sheet

• Next few lectures: read lecture notes posted on website in addition to flipping through slides
Reasoning about code

Determine what facts are true as a program executes
  – Under what assumptions

Examples:
  – If $x$ starts positive, then $y$ is 0 when the loop finishes
  – Contents of the array that $arr$ refers to are sorted
  – Except at one code point, $x + y == z$
  – For all instances of $Node\ n$, $n.next == null \lor n.next.prev == n$
  – ...
Why do this?

• Essential complement to testing, which we will also study
  – Testing: Actual results for some actual inputs
  – Logical reasoning: Reason about whole classes of inputs/states at once (“If $x > 0$, …”)
    • Prove a program correct (or find bugs trying)
    • Understand why code is correct

• Stating assumptions is the essence of specification
  – “Callers must not pass null as an argument”
  – “Callee will always return an unaliased object”
  – …
Our approach

• Hoare Logic: a 1970s approach to logical reasoning about code
  – For now, consider just variables, assignments, if-statements, while-loops
    • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Why?

- Programmers rarely “use Hoare logic” in this much detail
  - For simple snippets of code, it’s overkill
  - Gets very complicated with objects and aliasing
  - But it can be useful for loops and data with subtle invariants
    - Examples: Homework 0, Homework 2

- Also it’s an ideal setting for the right logical foundations
  - How can logic “talk about” program states?
  - How does code execution “change what is true”??
  - What do “weaker” and “stronger” mean?

This is all essential for specifying library-interfaces, which does happen All the Time in The Real World® (coming lectures)
Example

Forward reasoning:

- Suppose we initially know (or assume) $w > 0$
  
  ```
  // 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, including $z > 59$
  ```

CSE 331 Winter 2015
Example

Backward reasoning:

- Suppose we want $z$ to be negative at the end
  
  ```
  // w + 17 + 42 < 0
  x = 17;
  // w + x + 42 < 0
  y = 42;
  // w + x + y < 0
  z = w + x + y;
  // z < 0
  ```

- Then we know initially we need to know/assume $w < -59$
  
  • Necessary and sufficient
Forward vs. Backward, Part 1

- **Forward reasoning:**
  - Determine what follows from initial assumptions
  - Most useful for *maintaining an invariant*

- **Backward reasoning**
  - Determine sufficient conditions for a certain result
    - If result desired, the assumptions suffice for correctness
    - If result undesired, the assumptions suffice to trigger bug
Forward vs. Backward, Part 2

• Forward reasoning:
  – Simulates the code (for many “inputs” “at once”)
  – Often more intuitive
  – But introduces [many] facts irrelevant to a goal

• Backward reasoning
  – Often more useful: Understand what each part of the code contributes toward the goal
  – “Thinking backwards” takes practice but gives you a powerful new way to reason about programs
Conditionals

// initial assumptions
if(...) {
    ... // also know test evaluated to true
} else {
    ... // also know test evaluated to false
}
// either branch could have executed

Two key ideas:

1. The precondition for each branch includes information about the result of the test-expression

2. The overall postcondition is the disjunction (“or”) of the postcondition of the branches
Assume initially $x \geq 0$

```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, a 1970s approach to logical reasoning about code
  – [Named after its inventor, Tony Hoare]
  – Considering just variables, assignments, if-statements, while-loops
  • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Some notation and terminology

• The “assumption” before some code is the **precondition**
• The “what holds after (given assumption)” is the **postcondition**

• Instead of writing pre/postconditions after //, write them in {...}
  – This is not Java
  – How Hoare logic has been written “on paper” for 40ish years
    \[
    \{ \ w < -59 \ \} \\
    x = 17; \\
    \{ \ w + x < -42 \ \}
    \]
  – In pre/postconditions, = is equality, not assignment
    • Math’s “=”, which for numbers is Java’s ==
      \[
      \{ \ w > 0 \ \land \ x = 17 \ \} \\
      y = 42; \\
      \{ \ w > 0 \ \land \ x = 17 \ \land \ y = 42 \ \}
      \]
What an assertion means

• An assertion (pre/postcondition) is a logical formula that can refer to program state (e.g., contents of variables)

• A program state is something that “given” a variable can “tell you” its contents
  – Or any expression that has no side-effects

• An assertion holds for a program state, if evaluating using the program state produces true
  – Evaluating a program variable produces its contents in the state
  – Can think of an assertion as representing the set of (exactly the) states for which it holds
A Hoare Triple

- A **Hoare triple** is two assertions and one piece of code:
  \[
  \{P\} \ S \ {Q}\]
  - $P$ the precondition
  - $S$ the code (statement)
  - $Q$ the postcondition

- A Hoare triple $\{P\} \ S \ {Q}$ is (by definition) **valid** if:
  - For all states for which $P$ holds, executing $S$ always produces a state for which $Q$ holds
  - Less formally: If $P$ is true before $S$, then $Q$ must be true after
  - Else the Hoare triple is **invalid**
Examples

Valid or invalid?

- (Assume all variables are integers without overflow)

• \{x \neq 0\} y = x^2; \{y > 0\}
• \{z \neq 1\} y = z^2; \{y \neq z\}
• \{x \geq 0\} y = 2x; \{y > x\}
• \{true\} (if(x > 7) \{y=4;\} else \{y=3;\}) \{y < 5\}
• \{true\} (x = y; z = x;) \{y=z\}
• \{x=7 \land y=5\}
  (tmp=x; x=tmp; y=x;)
  \{y=7 \land x=5\}
Examples

Valid or invalid?

- (Assume all variables are integers without overflow)

- \{x \neq 0\} y = x \times x; \{y > 0\} \quad \text{valid}

- \{z \neq 1\} y = z \times z; \{y \neq z\} \quad \text{invalid}

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

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

\{y=7 \ \land \ x=5\}
Aside: assert in Java

• An assertion in Java is a statement with a Java expression, e.g.,
  ```java
  assert x > 0 && y < x;
  ```

• Similar to our assertions
  – Evaluate using a program state to get true or false
  – Uses Java syntax

• In Java, this is a run-time thing: Run the code and raise an exception if assertion is violated
  – Unless assertion-checking is disabled
  – Later course topic

• This week: we are reasoning about the code, not running it on some input
The general rules

- So far: Decided if a Hoare triple was valid by using our understanding of programming constructs

- Now: For each kind of construct there is a general rule
  - A rule for assignment statements
  - A rule for two statements in sequence
  - A rule for conditionals
  - [next lecture:] A rule for loops
  - ...
Assignment statements

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

- Let \(Q'\) be like \(Q\) except replace every \(x\) with \(e\)
- Triple is valid if:
  - For all program states, if \(P\) holds, then \(Q'\) holds
  - That is, \(P\) implies \(Q'\), written \(P \Rightarrow Q'\)

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

\{P\} S1;S2 \{Q\}

- Triple is valid if and only if there is an assertion \(R\) such that
  - \(\{P\}S_1\{R\}\) is valid, and
  - \(\{R\}S_2\{Q\}\) is valid

- Example: \(\{z \geq 1\} y=z+1; w=y*y; \{w > y\}\) (integers)
  - Let \(R\) be \(\{y > 1\}\)
  - Show \(\{z \geq 1\} y=z+1; \{y > 1\}\)
    - Use rule for assignments: \(z \geq 1\) implies \(z+1 > 1\)
  - Show \(\{y > 1\} w=y*y; \{w > y\}\)
    - Use rule for assignments: \(y > 1\) implies \(y*y > y\)
{P} if(b) S1 else S2 {Q}

- Triple is valid if and only if there are assertions Q1, Q2 such that
  - {P ∧ b}S1{Q1} is valid, and
  - {P ∧ !b}S2{Q2} is valid, and
  - Q1 ∨ Q2 implies Q

- Example: {true} (if(x > 7) y=x; else y=20;) {y > 5}
  - Let Q1 be {y > 7} (other choices work too)
  - Let Q2 be {y = 20} (other choices work too)
  - Use assignment rule to show {true ∧ x > 7}y=x;{y>7}
  - Use assignment rule to show {true ∧ x <= 7}y=20;{y=20}
  - Indicate y>7 ∨ y=20 implies y>5
Our approach

• Hoare Logic, a 1970s approach to logical reasoning about code
  – Considering just variables, assignments, if-statements, while-loops
    • So no objects or methods

• This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

• Next lecture: Loops
Weaker vs. Stronger

If P1 implies P2 (written P1 => P2), then:
- P1 is stronger than P2
- P2 is weaker than P1

• Whenever P1 holds, P2 also holds
• So it is more (or at least as) “difficult” to satisfy P1
  - The program states where P1 holds are a subset of the program states where P2 holds
• So P1 puts more constraints on program states
• So it’s a stronger set of obligations/requirements
Examples

• $x = 17$ is stronger than $x > 0$

• $x$ is prime is neither stronger nor weaker than $x$ is odd

• $x$ is prime and $x > 2$ is stronger than $x$ is odd and $x > 2$

• ...

CSE 331 Winter 2015 26
Why this matters to us

• 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 \( y \geq 0 \)
So…

• For backward reasoning, if we want \{P\}S\{Q\}, we could instead:
  – Show \{P_1\}S\{Q\}, and
  – Show \(P \Rightarrow P_1\)

• Better, we could just show \{P_2\}S\{Q\} where \(P_2\) is the weakest precondition of \(Q\) for \(S\)
  – Weakest means the most lenient assumptions such that \(Q\) will hold
  – 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 preconditions

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

- \( \text{wp}(S1;S2, Q) \) 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: \( \text{wp}((y=x+1; z=y+1;), z > 2) = (x + 1) + 1 > 2 \), i.e., \( x > 0 \)

- \( \text{wp}(\text{if } b \ S1 \ \text{else } S2, Q) \) is this logic 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 $x = y*y$ 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 \((x < 5)\) {
  \(x = x \times x;\)
} else {
  \(x = x + 1;\)
}

Q is \(x \geq 9\)

\(wp(S, x \geq 9)\)

\(= (x < 5 \land wp(x = x \times x;, x \geq 9))\)
\(\lor (x \geq 5 \land wp(x = x + 1;, x \geq 9))\)

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

\(= (x \leq -3) \lor (x \geq 3 \land x < 5)\)
\(\lor (x \geq 8)\)
If-statements review

Forward reasoning

{P}
if B
{P ∧ B}
S1
{Q1}
else
{P ∧ !B}
S2
{Q2}
{Q1 ∨ Q2}

Backward reasoning

{(B ∧ wp(S1, Q)) ∨ (!B ∧ wp(S2, Q))}
if B
{wp(S1, Q)}
S1
{Q}
else
{wp(S2, Q)}
S2
{Q}
{Q}
“Correct”

• If $wp(S,Q)$ is $\text{true}$, then executing $S$ will always produce a state where $Q$ holds
  – $\text{true}$ holds for every program state
One more issue

• With forward reasoning, there is a problem with assignment:
  – Changing a variable can affect other assumptions

• Example:
  
  ```
  {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!
  ```
The fix

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

• Corrected example:

```plaintext
{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 contents
  - Give a name to 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”

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

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

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