CSE 331
Software Design & Implementation

Hal Perkins

Winter 2022

Lecture 2 – Reasoning About Code With Logic

Administrivia (1)

- Gradescope and Ed discussion accounts created now. If you're registered but not set up yet, send first & last name, id # (7 digits), and @uw.edu email address to cse331-staff[at]cs

- Office hours: schedule for this week posted; rest coming soon
  - Goal: help you get "unstuck" when you are stuck and not making progress after a reasonable time
    - i.e., come up with ideas for how you can make progress, not necessarily fix/solve everything right then
  - Goal: clear up questions or confusions
  - Analogy: going down the hall to see if a colleague has any ideas
  - Help us help you: organize what you want to talk about, be sure you can explain what you've already done and where the problems seem to be, …
  - See info sheet on web resources page

Administrivia (2)

- HW1 out now, due Tuesday night, 11 pm
  - Reasoning about code; programming logic without loops
  - Today's lecture and tomorrow's sections
    - Look on canvas calendar for section zoom links

- Reminder: readings to the calendar – sections (items) in *Pragmatic Programmer* (PP) and *Effective Java* (EJ)
  - Free access to books online via UW library's institutional license – see the syllabus or other course resources for access details
    - But alas, Core Java seems to have disappeared at the request of the publisher. We'll see if there's anything we can do about that…

Overview

- Next few lectures: two presentations linked to course calendar on the web:
  - Lecture notes – primary source
    - Must read/study
  - Powerpoint slides – summary & supplement

  They are complementary and you should understand both of them

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` ∨ **`n.next.prev == n`**
- …

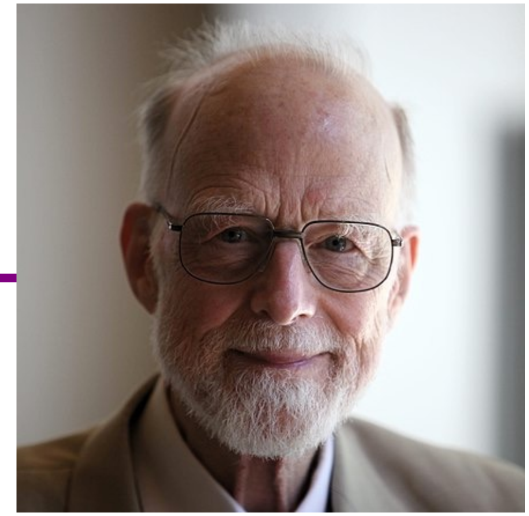
- Notation: In logic we often use ∧ for "and" and ∨ for "or". Concise and convenient, but we're not dogmatic about it

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), or (even better) develop program and proof together to get a program that is correct by construction
    - Understand *why* code is correct

- Stating assumptions is the essence of specification
  - "Callers must not pass **`null`** as an argument"
  - "Method will always return an unaliased object"
  - …

Our approach



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

- 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? (1)

- 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 can be very useful to develop and reason about loops and data with subtle *invariants*
    - Examples: Homework 0, Homework 2

- Most professionals can do reasoning like this in their head
  - Eventually it will be the same for you

- Overkill for simple problems, essential for really hard ones

Why? (2)

- Formal reasoning is 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* and *data invariants*, 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`**

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 and to write correct code

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:

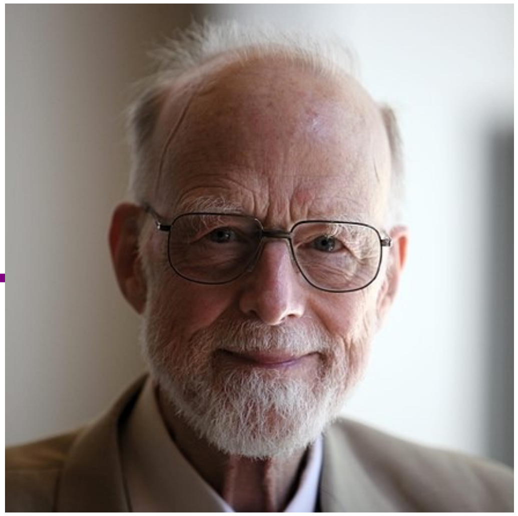
1. The precondition for each branch includes information about the result of the test-expression
2. The overall postcondition is the disjunction (“or”) of the postcondition of the branches

Example (Forward)

Assume initially **`x >= 0`**

```
// 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 classic approach to logical reasoning about code
  - Named after its inventor, Tony Hoare
  - We will only consider 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  ∧  x = 17 }
y = 42;
{ w > 0  ∧  x = 17 ∧  y = 42 }
```

What an assertion means

- An *assertion* (including pre/postconditions) 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*
  - (informally, this is just the current values of all variables)
- 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

Aside: assert *statement* in Java

- An Java **`assert`** is a statement with a Java expression, e.g.,

```
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 – but really useful to detect bugs early

- This week: we are reasoning about the code, not running it on some input

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 != 0} y = x*x; {y > 0}` valid
- `{z != 1} y = z*z; {y != z}` invalid
- `{x >= 0} y = 2*x; {y > x}` invalid
- `{true} (if(x > 7) {y=4;} else {y=3;}) {y < 5}` valid
- `{true} (x = y; z = x;) {y=z}` valid
- `{x=7 ∧ y=5}`
  `(tmp=x; x=tmp; y=x;)` invalid
  `{y=7 ∧ x=5}`

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(s):] A rule for loops
  - …

Basic rule: Assignment

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

- Let **`Q′`** be the same as **`Q`** except replace every **`x`** with **`e`**
- Triple is valid if: For all program states, if **`P`** holds, then **`Q′`** holds (i.e., if **`P`** guarantees that **`Q′`** is true, then execution of **`x=e;`** will guarantee that **`Q`** is true)

- Example: **`{z > 34} y=z+1; {y > 1}`**
  - **`Q′`** is **`{z+1 > 1}`**
  - Triple is valid because if **`{z > 34}`** is true then **`{z+1 > 1}`** is guaranteed to be true

Combining rule: Sequence

`{P} S1;S2 {Q}`

- Triple is valid if and only if there is an assertion **`R`** such that
  - `{P}S1{R}` is valid, and
  - `{R}S2{Q}` is valid
- Example: `{z >= 1} y=z+1; w=y*y; {w > y}` (integers)
  - Let **`R`** be `{y > 1}` (this particular **`R`** picked because "it works")
  - Show `{z >= 1} y=z+1; {y > 1}`
    - Use rule for assignments: `z >= 1` implies `z+1 > 1`
  - Show `{y > 1} w=y*y; {w > y}`
    - Use rule for assignments: `y > 1` implies `y*y > y`

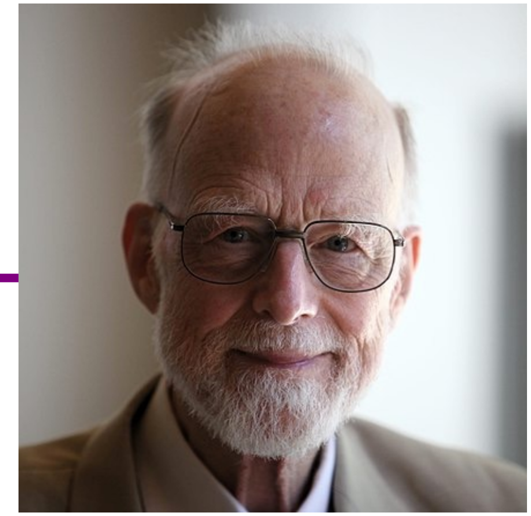
Combining rule: Conditional

`{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` (i.e., if either of `Q1` or `Q2` is valid then `Q` is also)

- 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 classic approach to logical reasoning about code
  - Considering just variables, assignments, if-statements, while-loops
    - So no objects or methods

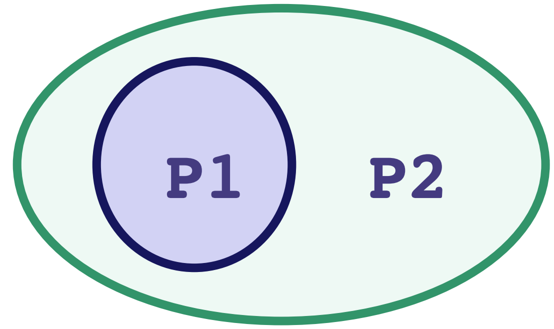
- This lecture: The idea, without loops, in 3 passes
  1. High-level intuition of forward and backward reasoning
  2. Precise definition of logical assertions, preconditions, etc.
  3. Definition of weaker/stronger and weakest-precondition

- Next lecture: Loops

Weaker vs. Stronger

If P1 implies P2 (written P1 => P2), then:

- P1 is stronger than P2
- P2 is weaker than P1



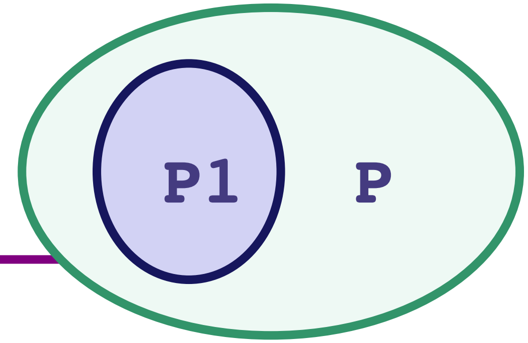
This means:

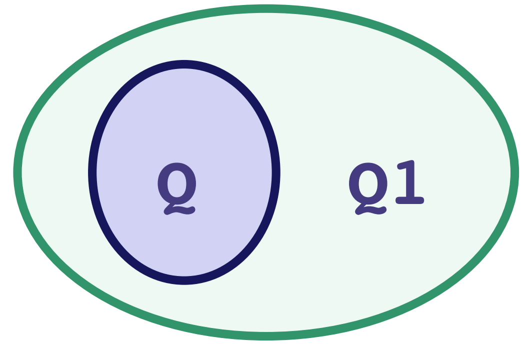
- Whenever P1 holds, P2 also holds
- So it is more (or at least as) “difficult” to satisfy P1
  - The program states where P1 holds are a subset of the program states where P2 holds
- So P1 puts more constraints on program states
- So it’s a stronger set of obligations/requirements

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`
- …

Why this matters to us



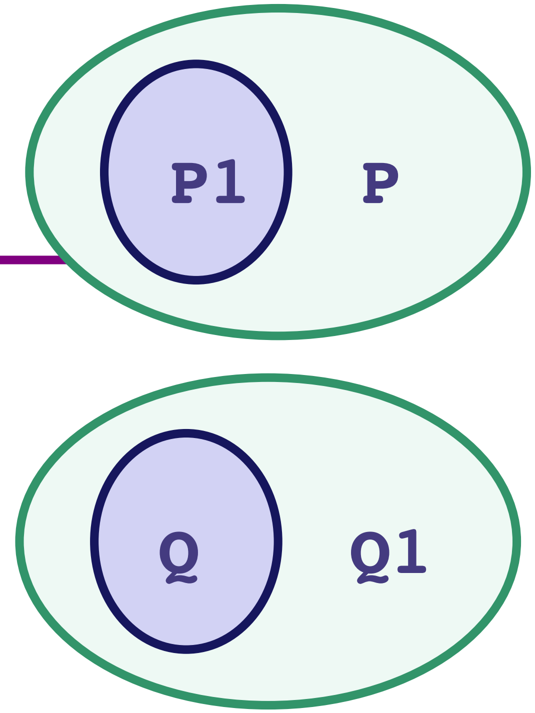


- Suppose we have:
  - `{P}S{Q}` is valid, and
  - `P` is weaker than some `P1`, and
  - `Q` is stronger than some `Q1`

- Then: `{P1}S{Q}` and `{P}S{Q1}` and `{P1}S{Q1}`

- What???
  - `{P}` weaker than `{P1}` means whenever `{P1}` is true than `{P}` is also true, so if `{P}S{Q}` is valid then so is `{P1}S{Q}`
  - `{Q}` stronger than `{Q1}` means whenever `{Q}` is true then `{Q1}` is also true, so if `{P}S{Q}` is valid then so is `{P}S{Q1}`
  - Combine to show if `{P}S{Q}` is valid then so is `{P1}S{Q1}`

Example



Suppose we have

- P is x >= 0
- S is y = x+1
- Q is y > 0

Then: {P}S{Q} is valid: {x >= 0} y = x+1 { y >0 }

Let P1 be x > 0. P1 is stronger than P (i.e., P1 => P)

Then: {P1}S{Q} is valid: {x > 0} y = x+1 {y > 0}

Let Q1 be y>=0. Q1 is weaker than Q (i.e., Q=>Q1)

Then: {P}S{Q1} is valid: {x >= 0} y = x+1 { y >= 0}

And: {P1}S{Q1} is also valid: {x > 0} y = x+1 {y >= 0}

So…

- For backward reasoning, if we want **{P}S{Q}**, we could instead:
  - Show **{P1}S{Q}**, and
  - Show **P => 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 => P2**
- 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

- wp(`x = e;`, `Q`) is `Q` with each `x` replaced by `e`
  - Example: wp(`x = y*y;`, `x > 4`) = `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`) =
    `(x + 1)+1 > 2`, i.e., `x > 0`

- wp(`if b S1 else S2`, `Q`) is this logic formula:

  (b ∧ wp(S1,Q)) ∨ (!b ∧ 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`))

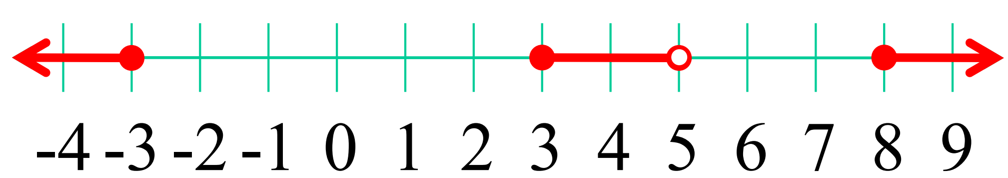
∨ (`x >= 5` ∧ wp(`x = x+1;`, `x >= 9`))

= (`x < 5 ∧ x*x >= 9`)

∨ (`x >= 5 ∧ x+1 >= 9`)

= (`x <= -3`) ∨ (`x >= 3 ∧ x < 5`)

∨ (`x >= 8`)



If-statements review

Forward reasoning

$\{P\}$

```
if B
```

$\{P \wedge B\}$

```
S1
```

$\{Q1\}$

```
else
```

$\{P \wedge !B\}$

```
S2
```

$\{Q2\}$

$\{Q1 \vee Q2\}$

Backward reasoning

$\{ (B \wedge wp(S1, Q)) \vee$

$(!B \wedge wp(S2, Q)) \}$

```
if B
```

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

```
S1
```

$\{Q\}$

```
else
```

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

```
S2
```

$\{Q\}$

$\{Q\}$

“Correct”

- If wp(**`S`**,**`Q`**) is **`true`**, then executing **`S`** will always produce a state where **`Q`** holds
  - **`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"
    - But only do this if you actually use the "old contents" from that variable later in the proof – omit otherwise
- 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

- 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”

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