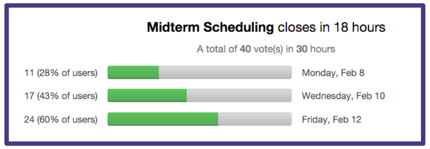
CSE 331
Software Design and Implementation

Lecture 2
Formal Reasoning

Zach Tatlock / Winter 2016

Announcements

Please vote for midterm date



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

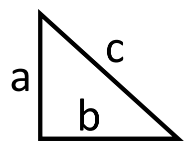
Homework 1 due Wednesday at 11pm
- Using program logic sans loops

Formal Reasoning

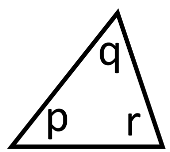
Formalization and Reasoning

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

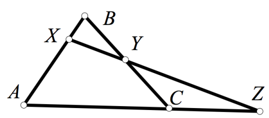
Geometric proofs often establish *general* truths



$a^2 + b^2 = c^2$



$p + q + r = 180$



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

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

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 == null` ∨ `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)

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`

```
// 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

```
// 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

```
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)

```
// 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

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

```
{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;)
{y=7 ∧ x=5}
```
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 => Q'`

Example: `{ z > 34 } y = z + 1; { y > 1 }`

- `Q'` is `{ z + 1 > 1 }`

Combining Rule: Sequence

`{ P } S1; S2 { Q }`

Triple is valid iff there is an assertion `R` such that both the following are valid:

- `{ P } S1 { R }`
- `{ R } S2 { Q }`

Example:

```
{ z >= 1 }
y = z + 1;
w = y * y;
{ w > y }
```

Let `R` be `{y > 1}`

1. Show `{z >= 1} y = z + 1 {y > 1}`
   Use basic assign rule:
   `z >= 1` implies `z + 1 > 1`

2. Show `{y > 1} w = y * y {w > y}`
   Use basic assign rule:
   `y > 1` implies `y * y > y`

Combining Rule: Conditional

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

Triple is valid iff there are assertions `Q1, Q2` such that:

- `{ P /\ b } S1 { Q1 }` is valid
- `{ P /\ !b } S2 { Q2 }` is valid
- `Q1 \/ Q2` implies `Q`

Example:

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

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

1. Show `{true /\ x > 7} y = x {y > 7}`

2. Show `{true /\ x <= 7} y = 20 {y = 20}`

3. Show `y > 7 \/ y = 20` implies `y > 5`

Our Approach

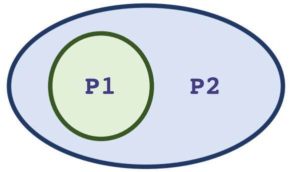
Hoare Logic, an approach developed in the 70's
- Focus on core: assignments, conditionals, loops
- Omit complex constructs like objects and methods

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 **`P1`** implies **`P2`** (written **`P1 => P2`**) then:
- **`P1`** is stronger than **`P2`**
- **`P2`** is weaker than **`P1`**



Whenever **`P1`** holds, **`P2`** is guaranteed to hold
- So it is at least as difficult to satisfy **`P1`** as **`P2`**
- **`P1`** holds on a subset of the states where **`P2`** holds
- **`P1`** puts more constraints on program states
- **`P1`** is a "stronger" set of obligations / requirements

Weaker vs. Stronger Examples

`x = 17` is stronger than **`x > 0`**

`x is prime` is neither stronger nor weaker than
`x is odd`

`x is prime /\ x > 2` is stronger than
`x is odd /\ x > 2`

…

Strength and Hoare Logic

Suppose:
- **`{P} S {Q}`** 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}`**

Example:
- **`P`** is **`x >= 0`**
- **`P1`** is **`x > 0`**
- **`S`** is **`y = x+1`**
- **`Q`** is **`y > 0`**
- **`Q1`** is **`y >= 0`**

"Wiggle Room"

Strength and Hoare Logic

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

1. Show `{P1}S{Q}`, then
2. 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 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` ∧ 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

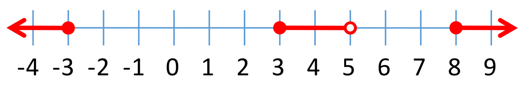
If `S` is `x = y*y` and `Q` is `x > 4`,
then wp(`S`,`Q`) is `y*y > 4`, i.e., `|y| > 2`

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`)



Conditionals 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 *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 intuitve rule for assignment is wrong:

- 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` (!!!)

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”

```
{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}
```