CSE 331
Software Design & Implementation

Fall 2020

Section 2 – Code Reasoning

Administrivia

- HW1 due tonight.
- Any questions before we dive in?
  – What are the most interesting/confusing/puzzling things so far in the course?

Agenda

- Review logical reasoning about code with Hoare Logic
- Practice both forward and backward modes
  - Just assignment, conditional (“if-then-else”), and sequence
  - Logical rules from yesterday’s lecture/notes
- Review logical strength of assertions (weaker *vs.* stronger)
- Practice determining stronger/weaker assertions
- Practice checking correctness of loops.
- Practice writing loops (if time allows)

Why reason about code?

- Prove that code is correct
- Understand *why* code is correct
- Diagnose why/how code is *not* correct
- Specify code behavior

Logical reasoning about code

- Determine facts that hold of program state between statements
  - "Fact" ~ assertion (logical formula over program state, informally "value(s) of some/all program variables)
  - Driven by assumption (precondition) or goal (postconditon)

- Forward reasoning
  - What facts follow from initial assumptions?
  - Go from <u>pre</u>condition to <u>post</u>condition

- Backward reasoning
  - What facts need to be true to reach a goal?
  - Go from <u>post</u>condition to <u>pre</u>condition

Hoare Logic: Validity by Reasoning

- Checking validity of `{`P`}` S `{`Q`}`
  - Valid iff, starting from any state satisfying P, executing S results in a state satisfying Q
- Forward reasoning:
  - Reason from P to strongest postcondition `{`P`}` S `{`R`}`
  - Check that R implies Q (i.e., Q is weaker)
- Backward reasoning:
  - Reason from Q to get weakest precondition `{`R`}` S `{`Q`}`
  - Check that P implies R (i.e., P is stronger)

Implication (=>)

- Logic formulas with *and* (&, &&, or ∧), *or* (|, ||, or ∨) and *not* (! or ¬) have the same meaning they do in programs

- Implication might be a bit new, but the basic idea is pretty simple. Implication p=>q is true as long as q is always true whenever p is

p	q	p => q
T	T	T
T	F	F
F	T	T
F	F	T

Assignment Statements

- Reasoning about `x = y;`

- Forward reasoning:
  - add “x = y” as a new fact
  - (also rewrite any existing references to “x” to use new value)

- Backward reasoning:
  - replace all instances of “x” in the postcondition with “y”

Conditionals, more closely

Forward reasoning

$\{P\}$
if (b)
  $\{P \wedge \mathtt{b}\}$
  S_1
  $\{Q_1\}$
else
  $\{P \wedge \mathtt{!b}\}$
  S_2
  $\{Q_2\}$
$\{Q_1 \vee Q_2\}$

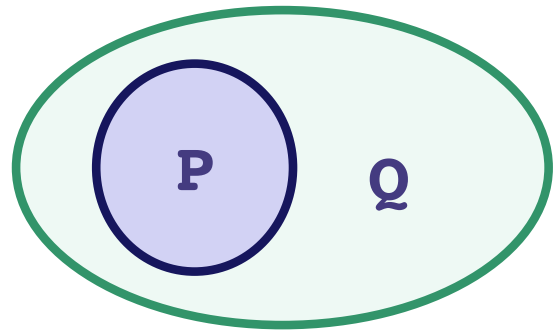
Backward reasoning

$\{ (\mathtt{b} \wedge P_1) \vee (\mathtt{!b} \wedge P_2) \}$
if (b)
  $\{P_1\}$
  S_1
  $\{Q\}$
else
  $\{P_2\}$
  S_2
  $\{Q\}$
$\{Q\}$

Weaker vs. stronger

Formal definition:

- If $P \Rightarrow Q$, then
  - Q is weaker than P
  - P is stronger than Q



Intuitive definition:

- "Weak" means unrestrictive; a weaker assertion has a larger set of possible program states (*e.g.*, **`x != 0`**)
- "Strong" means restrictive; a stronger assertion has a smaller set of possible program states (*e.g.*, **`x = 1`** or **`x > 0`** are both stronger than **`x != 0`**).

Worksheet

- Take ~10 minutes to get where you can
- Find a partner and work with them
- Let me know if you feel stuck
- We’ll walk through some solutions afterwards

Worksheet – problem 2

```
{ true }
if (x>0) {
  { x > 0 }
  y = 2*x;
  { x > 0 ∧ y = 2x }
} else {
  { x <= 0 }
  y = -2*x;
  { x <= 0 ∧ y = -2x }
}
{ (x > 0 ∧ y = 2x) ∨ (x <= 0 ∧ y = -2x) }
⇒ { y = 2|x| }
```

Worksheet – problem 4

```
{ y > 15 ∨ (y <= 5 ∧ y + z > 17) }
if (y > 5) {
  { y > 15 }
  x = y + 2
  { x > 17 }
} else {
  { y + z > 17 }
  x = y + z;
  { x > 17 }
}
{ x > 17 }
```

Worksheet – problem 6 (forward)

```
{ true }
if (x < y) {
  { true ∧ x < y }
  m = x;
  { x < y ∧ m = x }
} else {
  { true ∧ x >= y }
  m = y;
  { x >= y ∧ m = y }
}
{ (x < y ∧ m = x) ∨ (x >= y ∧ m = y) }
⇒ { m = min(x, y) }
```

Worksheet – problem 6 (backward)

```
{ true } ⇔
{ (x <= y ∧ x < y) ∨ (y <= x ∧ x >= y) }
if (x < y) {
  { x = min(x, y) } ⇔ { x <= y }
  m = x;
  { m = min(x, y) }
} else {
  { y = min(x, y) } ⇔ { x >= y }
  m = y;
  { m = min(x, y) }
}
{ m = min(x, y) }
```

Worksheet – problem 7

`{ y > 23 }`	`{ y >= 23 }`
`{ y = 23 }`	`{ y >= 23 }`
`{ y < 0.23 }`	`{ y < 0.00023 }`
`{ x = y * z }`	`{ y = x / z }`
`{ is_prime(y) }`	`{ is_odd(y) }`

Worksheet – problem 7

`{ y > 23 }`	is stronger than	`{ y >= 23 }`
`{ y = 23 }`		`{ y >= 23 }`
`{ y < 0.23 }`		`{ y < 0.00023 }`
`{ x = y * z }`		`{ y = x / z }`
`{ is_prime(y) }`		`{ is_odd(y) }`

Worksheet – problem 7

`{ y > 23 }`	is stronger than	`{ y >= 23 }`
`{ y = 23 }`	is stronger than	`{ y >= 23 }`
`{ y < 0.23 }`		`{ y < 0.00023 }`
`{ x = y * z }`		`{ y = x / z }`
`{ is_prime(y) }`		`{ is_odd(y) }`

Worksheet – problem 7

`{ y > 23 }`	is stronger than	`{ y >= 23 }`
`{ y = 23 }`	is stronger than	`{ y >= 23 }`
`{ y < 0.23 }`	is weaker than	`{ y < 0.00023 }`
`{ x = y * z }`		`{ y = x / z }`
`{ is_prime(y) }`		`{ is_odd(y) }`

Worksheet – problem 7

`{ y > 23 }`	is stronger than	`{ y >= 23 }`
`{ y = 23 }`	is stronger than	`{ y >= 23 }`
`{ y < 0.23 }`	is weaker than	`{ y < 0.00023 }`
`{ x = y * z }`	is **incomparable** with	`{ y = x / z }`
`{ is_prime(y) }`		`{ is_odd(y) }`

Worksheet – problem 7

`{ y > 23 }` is stronger than `{ y >= 23 }`

`{ y = 23 }` is stronger than `{ y >= 23 }`

`{ y < 0.23 }` is weaker than `{ y < 0.00023 }`

`{ x = y * z }` is **incomparable** with `{ y = x / z }`

`{ is_prime(y) }` is **incomparable** with `{ is_odd(y) }`

Questions?

- What is the most surprising thing about this?
- What is the most confusing thing?
- What will need a bit more thinking to digest?

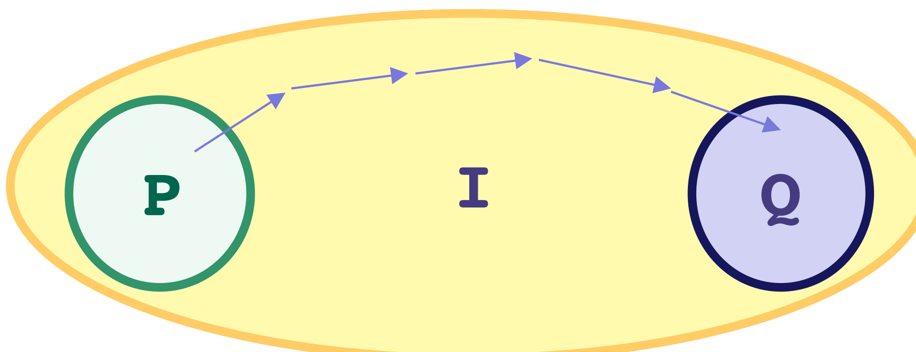
Previously on CSE 331...

{{ **P** }} `while (cond) S` {{ **Q** }}

This triple is valid iff

{{ **P** }}
{{ Inv: **I** }}
```
while (cond)
  S
```
{{ **Q** }}

- **I** holds initially
- **I** holds each time we execute `S`
- **Q** holds when **I** holds and `cond` is false



Worksheet – problem 8

Need to check each of these parts:

- code before the loop
- body of the loop
- exit of the loop gives claimed assert
- postcondition holds when returning true
- postcondition holds when returning false

We'll go through these in that order…

Worksheet – problem 8

{{ Precondition: $x >= 1$ }}

```
int k = 0;
int y = 1;
```

{{ $k = 0$ and $y = 1$ }}

{{ Inv: $y = 2^k$ and $y/2 < x$ }}

When $k = 0$, we have $y = 2^k = 2^0 = 1$, which is true.

We also have $y/2 = 1/2 < 1 <= x$, so that part is also true

Worksheet – problem 8

{{ Inv: $y = 2^k$ and $y/2 < x$ }}
while (y < x) {
  {{ $2y = 2^{k+1}$ and $2y/2 < x$ }} or equiv {{ $y = 2^k$ and $y < x$ }}
  y = y * 2;
  {{ $y = 2^{k+1}$ and $y/2 < x$ }}
  k = k + 1;
  {{ $y = 2^k$ and $y/2 < x$ }}
}

Inv and loop condition (y < x) include both of the two facts we need for correctness.

Worksheet – problem 8

{{ Inv: $y = 2^k$ and $y/2 < x$ }}

```
while (y < x) {
  y = y * 2;
  k = k + 1;
}
```

{{$y = 2^k$ and $y/2 < x$ and not $(y < x)$ }}

{{ $y = 2^k$ and $y/2 < x <= y$ }}

Last fact is $y >= x$,
so we have $y/2 < x <= y$ as required.

Worksheet – problem 8

{{ $y = 2^k$ and $y/2 < x <= y$ }}

```
if (y == x) {
```

{{ $y = 2^k$ and $y = x$ }}

```
  {{ Postcondition: x is a power of 2 }}
  return true;
} else {
 …
}
```

y is a power of 2 and y = x, so x is a power of 2

Worksheet – problem 8

{{ $y = 2^k$ and $y/2 < x <= y$ }}

```
if (y == x) {
  …
} else {
```

{{ $y = 2^k$ and $y/2 < x <= y$ and $y != x$ }}

```
  {{ Postcondition: x is not a power of 2 }}
  return false;
}
```

y != x tells us we have $y/2 < x < y$

So x lies strictly between two subsequent powers of 2, which means it is not a power of 2.

Loop Invariants

- Loop invariant comes out of the algorithm idea
  - describes partial progress toward the goal
  - how you will get from start to end

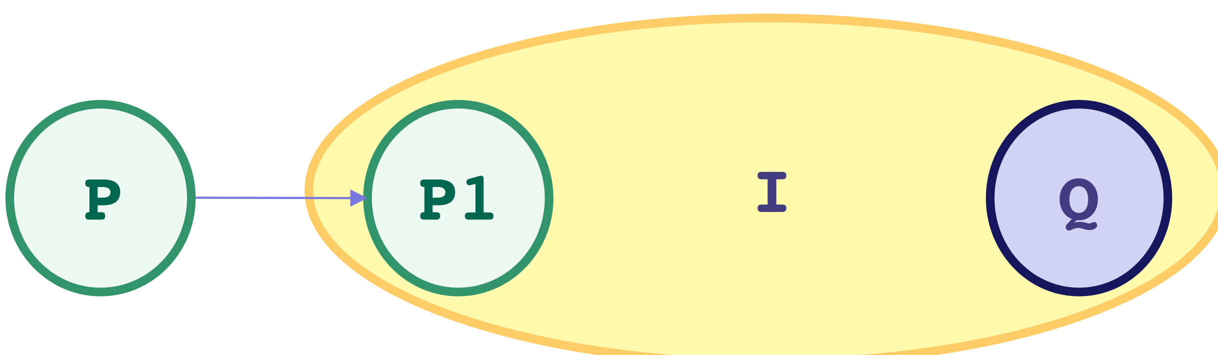
- Essence of the algorithm idea is:
  - invariant
  - how you make progress on each step (e.g., `i = i + 1`)

- Code is *ideally* just details that follow from that idea...

Loop Invariant → Code

In fact, can usually deduce the code from the invariant:

- When does loop invariant satisfy the postcondition?
  - gives you the termination condition
- What is the easiest way to satisfy the loop invariant?
  - gives you the initialization code
- How does the invariant change as you make progress?
  - gives you the rest of the loop body



Another Example

Example: quotient and remainder

Problem: Set q to be the quotient of x/y and r to be the remainder

Precondition: x >= 0 and y > 0

Postcondition: q*y + r = x and 0 <= r < y

- i.e., y doesn't go into x any more times

Example: quotient and remainder

Problem: Set q to be the quotient of x/y and r to be the remainder

Precondition: x >= 0 and y > 0

Postcondition: q*y + r = x and 0 <= r < y

– i.e., y doesn't go into x any more times

Loop invariant: q*y + r = x and 0 <= r

– postcondition is special case when we also have r < y
– this suggests a loop condition…

This is (again) just a weakening of the postcondition. (We just drop r < y.)

Example: quotient and remainder

We want "r < y" when the conditions fails

- so the condition is r >= y
- can see immediately that the postcondition holds on loop exit

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {


}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

Need to make the invariant hold initially…

- search for an easy way to satisfy q*y + r = x and 0 <= r

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {


}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

Need to make the invariant hold initially…

- search for an easy way to satisfy q*y + r = x and 0 <= r
- how about q = 0?
  - then we need r = x... and that is okay since 0 <= x

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {


}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

Need to make the invariant hold initially…

- search for the simplest way that works

```
int q = 0;
int r = x;
```

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {


}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {

  r = r - y;
}
{{ q*y + r = x and 0 <= r < y }}
```

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
```

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {
```

{{ q*y + r = x and 0 <= r and y <= r }}

{{ q*y + r-y = x and 0 <= r-y }}

```
  r = r - y;
```

{{ q*y + r = x and 0 <= r }}

```
}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

add and subtract y

```
int q = 0;
int r = x;
```

{{ Inv: q*y + r = x and 0 <= r }}

{{ q*y+y + r-y = x and 0 <= r and y <= r }}

```
while (r >= y) {
```

{{ q*y + r = x and 0 <= r and y <= r }}

```
  r = r - y;
}
```

{{ q*y + r-y = x and 0 <= r-y }}

{{ q*y + r = x and 0 <= r }}

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

top has q+1 where bottom has q, so we need code that changes q to q+1

```
int q = 0;
int r = x;
```

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {
```

{{ (q+1)*y + r-y = x and 0 <= r and y <= r }}

{{ q*y+y + r-y = x and 0 <= r and y <= r }}

{{ q*y + r = x and 0 <= r and y <= r }}

```
  r = r - y;
```

{{ q*y + r-y = x and 0 <= r-y }}

{{ q*y + r = x and 0 <= r }}

```
}
```

{{ q*y + r = x and 0 <= r < y }}

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
{{ Inv: q*y + r = x and 0 <= r }}
while (r >= y) {
  q = q + 1;
  r = r - y;
}
{{ q*y + r = x and 0 <= r < y }}
```

let's double-check this, just to be sure…

Example: quotient and remainder

We have r large initially.

Need to shrink r on each iteration in order to terminate…

- if r >= y, then y goes into x at least one more time

```
int q = 0;
int r = x;
```

{{ Inv: q*y + r = x and 0 <= r }}

```
while (r >= y) {
  q = q + 1;
  r = r - y;
}
```

{{ q*y + r = x and 0 <= r < y }}

+y and -y cancel to give exactly Inv

{{ (q+1)*y + r-y = x and y <= r }}
{{ q*y + r-y = x and 0 <= r-y }}
{{ q*y + r = x and 0 <= r }}

Aside on Efficiency

- This is not an efficient agorithm
  - runs in $O(x/y)$ time, which could be huge (e.g. $x/y = 2^{63}$)
  - but it is correct

- Grade school “long division“ is much more efficient
  - runs in $O((\log x)^2)$ time
  - makes progress in larger steps
    - (needs a more complex invariant)