CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
REASONING I

Autumn 2011

From last lecture

Ways to get your code right

- Verification/quality assurance
  - Purpose is to uncover problems and increase confidence
  - Combination of reasoning and test
- Debugging
  - Finding out why a program is not functioning as intended
- Defensive programming
  - Programming with validation and debugging in mind
- Testing ≠ debugging
  - test: reveals existence of problem; test suite can also increase overall confidence
  - debug: pinpoint location + cause of problem

Today and Wednesday

Reasoning about code

- Determine what facts are true during execution – we've seen these as assertions, representation invariants, preconditions, postconditions, etc.
  - `x > 0`
  - `for all nodes n:  n.next.previous == n`
  - `array a is sorted`
  - `x + y == z`
  - `if  x != null then  x.a > x.b`
- These can help…
  - … increase confidence that code is correct
  - … understand why code is incorrect

Forward reasoning

- Given a precondition, what is the postcondition?
- Example

```
// precondition:  x is even
x = x + 3;
y = 2x;
x = 5;
// postcondition:  ??
```

- One use: rep invariant holds before running code, does it still hold after running code?

Backward reasoning

- Given a postcondition, what is the corresponding precondition?
- Example

```
// precondition:  ??
x = x + 3;
y = 2x;
x = 5;
// postcondition:  y > x
```

- Uses include: what is needed to re-establish rep invariant, to reproduce a bug, to exploit a bug?

Ex: SQL injection attack

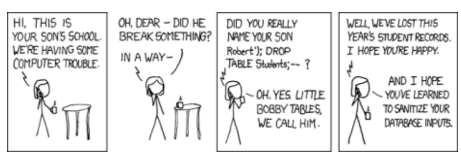
- SQL query constructed using unfiltered user input

```
query = "SELECT * FROM users "
      + "WHERE name='" + userInput + "';";
```

- If the user inputs `a' or '1'='1` this results in

```
query ⇒ SELECT * FROM users
         WHERE name='a' or '1'='1';
```

- This query returns information about all users – bad!



http://xkcd.com/327/

Forward vs. backward reasoning

7

- Forward reasoning is more intuitive for most people
  - Helps you understand what will happen (simulates the code)
  - Introduces facts that may be irrelevant to your task
- Backward reasoning is usually more helpful
  - Helps you understand what should happen
  - Given a specific task, indicates how to achieve it – for example, it can help creating a test case that exposes a specific error

Reasoning about code statements

8

- Convert assertions about programs into logic
- One logic representation is a *Hoare triple*:
  `P {Java* code} Q`
- `P` and `Q` are logical assertions about program values
- The triple means "if `P` is true and you execute code, then `Q` is true afterward"
- A Hoare triple is a boolean – true or false

* Or YFPL: Your favorite programming language

Tiny examples

9

```
true
  {y = x * x}
y ≥ 0
```

T or F?

```
x ≠ 0
  {y = x * x}
y > 0
```

T or F?

```
x > 0
  {y = x + 1}
y > 1
```

T or F?

Partial examples

10

```
x = k
  {if x < 0 x = -x}
?
```

Replace ? with what to get **true**

```
?
  {x = 3}
x = 8
```

Replace ? with what to get **true**

Longer example

11

```
x ≥ 0 {
  z = 0;
  if (x != 0) z = x; else z = z + 1;}
z > 0
```

Hoare triple: T or F?

```
assert x >= 0;    //  x ≥ 0
z = 0;            //  x ≥ 0 ∧ z = 0
if (x != 0)
  z = x;          //  x > 0 ∧ z = x
else
  z = z + 1;      //  x = 0 ∧ z = 1
assert z > 0;     //  (x > 0 ∧ z = x) ∧
                  //  (x = 0 ∧ z = 1)
```

Reasoning: what we know after each program point

⇒ z > 0

QED

Strongest or weakest conditions?

12

- `x = 5`
  `{x = x * 2}`
  `true`
- `x = 5`
  `{x = x * 2}`
  `x > 0`
- `x = 5`
  `{x = x * 2}`
  `x = 10 ∨ x = 5`
- `x = 5`
  `{x = x * 2}`
  `x = 10`
- All are true Hoare triples – which postcondition is most valuable, and why?

- `x = 5 ∧ y = 10`
  `{z = x/y}`
  `z < 1`
- `x < y ∧ y > 0`
  `{z = x/y}`
  `z < 1`
- `y ≠ 0 ∧ x / y < 1`
  `{z = x/y}`
  `z < 1`
- All are true Hoare triples – which precondition is most valuable, and why?

Weakest precondition

13

- `y ≠ 0 ∧ x / y < 1`
  `{z = x/y}`
  `z < 1`
  (the last one) is the most useful because it allows us to invoke the program in the most general condition
- It is called the *weakest precondition*, **wp(S,Q)** of **S** with respect to **Q**
- If **P {S} Q** and for all **P'** such that **P' ⇒ P**, then **P** is **wp(S,Q)**

A rule for each language construct

14

- The above examples use intuition to discuss the Hoare triples
- Specifically to understand how the code affects the
  - precondition to determine the (strongest) postcondition, using forward reasoning
  - postcondition to determine the (weakest) precondition, using backward reasoning
- To replace the intuition with a mechanical transformation – needed for precision and for automation – each language construct must be explicitly defined using the logic

Sequential execution or: What does ; really mean?

- $P\ \{S_1;S_2\}\ Q$
- Compute the intermediate assertion
  $A = wp(S_2,Q)$
- This means that
  $P\ \{S_1\}\ (A\ \{S_2\}\ Q)$
- Compute the assertion
  $T = wp(S_1,A)$
- This means that
  $T\ \{S_1\}\ (A\ \{S_2\}\ Q)$
- If **P⇒T** the triple is true
- We reason backwards to compose the statements

```
x > 0
  {y = x*2;
   z = y/2
  }
z > 0
```

```
x > 0
  {y = x*2}
y > 0
  {z = y/2}
z > 0
```


Conditional execution

- $P\ \{\texttt{if } C\ S_1 \texttt{ else } S_2\}\ Q$
- Must consider both branches – consider

```
true
 {
  if x >= y
     z = x;
  else
     z = y;
 }
z = x ∨ z = y
```

- But something is missing – knowledge about the value of the condition

$P\ \{\texttt{if } C\ S_1 \texttt{ else } S_2\}\ Q$

17

- The precise definition of a conditional (if-then-else) statement takes into account the condition's value and both branches
  $(P \wedge C\ \{S_1\}\ Q) \wedge$
  $(P \wedge \neg C\ \{S_2\}\ Q)$
- Even though at execution only one branch is taken, the proof needs to show that both will satisfy Q
- Or $wp(\texttt{if } C\ S_1; \texttt{ else } S_2;, Q)$ is equal to
  $C \Rightarrow wp(S_1,Q) \wedge \neg C \Rightarrow wp(S_2,Q)$

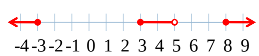
Example

$C \Rightarrow wp(S_1,Q) \wedge \neg C \Rightarrow wp(S_2,Q)$

18

```
?
 {
 if (x < 5)
     x = x*x;
 else
     x = x+1;
 }
x ≥ 9
```

- wp(if (x<5) {x = x*x;} else {x = x+1}, 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



Assignment statements

19

- What does the statement `x = E` really mean?
- `Q(E) {x = E} Q(x)`
- That is, if we knew something to be true about `E` before the assignment, then we know it to be true about `x` after the assignment
  - Assuming no side-effects
- `wp(x=E;, Q)` is `Q` with `x` replaced by `E`

Examples
`Q(E) {x = E} Q(x)`

20

```
y > 0
  {x = y}
x > 0
```

Reason backwards

```
Q(y)  ≡ y > 0
Q(x)  ≡ x > 0
```

```
x > 0
  {x = x + 1}
x > 1
```

```
Q(x+1)  ≡ x + 1 > 1
        ≡ x > 0
Q(x)    ≡ x > 1
```


More examples

21

```
?
  {x = y + 5}
x > 0
```

Replace ? with what to get **true**

```
x = A ∧ y = B
  {
    t = x;
    x = y;
    y = t;
  }
x = B ∧ y = A
```

true or false?

Method calls

22

```
?
  {x = foo()}
Q
```

- If the method has no side effects, it's just like ordinary assignment

```
(y = 22 ∨ y = -22)
    {x = Math.abs(y)}
x = 22
```


With side effects

23

- If it has side effects it also needs an assignment to every variable in `modifies`
- Use the method specification to determine the new value

`z+1 = 22`

`{incrZ()}   // spec:` $z_{post} = z_{pre} + 1$

`z = 22`

Loops: `P {while B do S} Q`

24

- A loop represents an unknown number of paths (and recursion presents the same problem as loops
- Cannot enumerate all paths – this is what makes testing and reasoning hard
- Trying to unroll the loop doesn't work, since we don't know how many times the loop can execute

```
(P ∧ ¬ B  {S}  Q)  ∧
(P ∧ B    {S}  Q ∧ ¬B)  ∧
(P ∧ B    {S}  Q ∧  B)  {S}  Q ∧ ¬B ∧ …
```

Loop invariant

- The most common approach to this is to find a *loop invariant*, a predicate that is
  - true each time the loop head is reached (on entry and after each iteration)
  - **and** helps us prove the postcondition of the loop
- Essentially, we will prove the properties inductively
- Find a loop invariant `I` such that
  - `P ⇒ I          //Invariant is correct on entry`
  - `B ∧ I {S} I    //Invariant is maintained`
  - `¬B ∧ I ⇒ Q     //Loop termination proves Q`

Example


```
x ≥ 0 ∧ y = 0 {
  while (x != y)
    y = y + 1;
x = y
```

```
P ⇒ I          //Invariant is correct on entry
B ∧ I {S} I    //Invariant is maintained
¬B ∧ I ⇒ Q     //Loop termination proves Q
```

- An invariant that works: **LI = x ≥ y**
  1. **x≥0 ∧ y=0 ⇒ LI**
  2. **LI ∧ x≠y {y = y+1} LI**
  3. **(LI ∧ ¬(x≠y)) ⇒ x=y**

Example


```
P ⇒ I          //Invariant is correct on entry
B ∧ I {S} I    //Invariant is maintained
¬B ∧ I ⇒ Q     //Loop termination proves Q
```

```
n > 0
  {
    x = a[1];
    i = 2;
    while i <= n {
      if a[i] > x
        x = a[i];
      i = i + 1;
    }
  }
x = max(a[1],…,a[n])
```

Ideas for an effective loop invariant?

Termination

- Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper postcondition if it terminates – this is called *weak correctness*
- A Hoare triple for which termination has been proven is *strongly correct*
- Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets
  - In the **max** example it's easy, since **i** is bounded by **n**, and **i** increases at each iteration

Choosing loop invariants

- For straightline code, the **wp** gives us the appropriate property
- For loops, you have to guess the loop invariant and then apply the proof techniques
- If the proof doesn't work
  - Maybe you chose an incorrect or ineffective invariant – choose another and try again
  - Maybe the loop is incorrect – gotta fix the code
- Automatically choosing loop invariants is a research topic

When to use code proofs for loops

- Most of your loops need no proofs
  - **for (String name : friends) { ... }**
- Write loop invariants and decrementing functions when you are unsure about a loop
- If a loop is not working
  - Add invariant
  - Write code to check them
  - Understand why the code doesn't work
  - Reason to ensure that no similar bugs remain

Next steps

31

- Wednesday: reasoning II; Friday: usability; Monday: UML; Wednesday: TBA
- A5 and A6