Coming up

• Final projects:
  – User report are graded
  – All 1.0 presentations will be held on
    Thursday, December 8, in class
  – 1.0 Release due data extended:
    Friday, December 9, 11:59PM
• Final exam:
  – Wednesday, December 14, 10:30 AM, in this room

Today’s plan

• Exam review
• Evaluations
• Reasoning about programs

What’ll be on the exam?
(12/14, 10:30AM, here)

• Regular questions:
  – testing
  – debugging
  – working in groups
  – reasoning about programs
• high-level questions only:
  – software bias and formal verification of software
  – guest lectures on safety in software design

Exam: What kind of questions?

• True/False
• Multiple Choice
  (most are “choose all that apply”)
• That’s it. No other types of questions.

Testing

• Know about different kinds of tests
  – unit, integration, regression, etc.
• Know about different kinds of coverage
  – statement, path, etc.
• Know what’s hard about testing
  – GUI, usability, covering all behavior, etc.

Debugging

• Know four kinds of defense against bugs
  – make impossible
  – don’t introduce
  – make errors visible
  – last resort: debugging
• Representation (rep) invariants
• Assertions
Working in groups

- What's hard?
  - corner cases
  - complete specification covers A LOT of behavior
  - unless a spec is concise, it's hard to understand
  - precision is hard: language is ambiguous
  - communication is important

Reasoning about programs

- Ways to verify your code
  - testing, reasoning, proving
- Forward reasoning
- Backward reasoning
- Loop invariants
- Induction
- Practice some examples!

Loop example

Find the weakest precondition

```
for (int x = 1; x <> y;) {
  if (y > x) {
    y = y / 2;
    x=2*x;
  }
  // postcondition: x=8, y=8, and x and y are ints
```

you can also find the loop invariant and decrement function

Bias in evaluations

- Ample scientific evidence that there are biases in evaluations.
- Women and minority faculty get statistically lower scores even when the teaching style is controlled to be exactly the same.
- Being aware is one of the best ways to combat the problem.

Evaluations

- We’ll take 15 minutes to do evaluations
- They are anonymous and I don’t see them until (long) after the grades are posted
- I actually use them to improve my teaching
- UMass uses them to decide if I am a good teacher and whether to let me keep teaching

Evaluations

- If we get 80% participation by tomorrow:
  - Everyone gets 0.5 points of extra credit.

References:


Reasoning about programs

Ways to verify your code

- The hard way:
  - Make up some inputs
  - If it doesn’t crash, ship it
  - When it fails in the field, attempt to debug
- The easier way:
  - Reason about possible behavior and desired outcomes
  - Construct simple tests that exercise that behavior
- Another way that can be easy
  - Prove that the system does what you want
    - Rep invariants are preserved
    - Implementation satisfies specification
  - Proof can be formal or informal (we will be informal)
  - Complementary to testing

Reasoning about code

- Determine what facts are true during execution
  - 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
- Applications:
  - Ensure code is correct (via reasoning or testing)
  - Understand why code is incorrect

Forward reasoning

- You know what is true before running the code
  What is true after running the code?
- Given a precondition, what is the postcondition?
- Applications:
  - (Re-)establish rep invariant at method exit: what’s required?
  - Reproduce a bug: what must the input have been?
- Example:
  // precondition: ??
  x = x + 3;
y = 2x;
x = 5;
// postcondition: ??
- How did you (informally) compute this?

Backward reasoning

- You know what you want to be true after running the code
  What must be true beforehand in order to ensure that?
- Given a postcondition, what is the corresponding precondition?
- Applications:
  - (Re-)establish rep invariant at method exit: what’s required?
  - Reproduce a bug: what must the input have been?
- Example:
  // precondition: ??
  x = x + 3;
y = 2x;
x = 5;
// postcondition: y > x
- How did you (informally) compute this?

Forward vs. backward reasoning

- Forward reasoning is more intuitive for most people
  - Helps understand what will happen (simulates the code)
  - Introduces facts that may be irrelevant to goal
    - Set of current facts may get large
  - Takes longer to realize that the task is hopeless
- Backward reasoning is usually more helpful
  - Helps you understand what should happen
  - Given a specific goal, indicates how to achieve it
  - Given an error, gives a test case that exposes it
Forward reasoning example

```plaintext
assert x >= 0;
i = x;
// x ≥ 0 & i = x
z = 0;
// x ≥ 0 & i = x & z = 0
while (i != 0) {
 z = z + 1;
i = i - 1;
}
// x ≥ 0 & i = 0 & z = x
assert x == z;
```

Backward reasoning

Technique for backward reasoning:
- Compute the weakest precondition (wp)
- There is a wp rule for each statement in the programming language
- Weakest precondition yields strongest specification for the computation (analogous to function specifications)

Assignment

```plaintext
// precondition: ?
x = e;
// postcondition: Q
Precondition: Q with all (free) occurrences of x replaced by e
- Example:
  // assert: ??
x = x + 1;
  // assert x > 0
Precondition = (x+1) > 0
```

Method calls

```plaintext
// precondition: ??
x = foo();
// postcondition: Q
- If the method has no side effects: just like ordinary assignment
- If it has side effects: an assignment to every variable it modifies
```

If: an example

```plaintext
// precondition: ??
if (b) S1 else S2
// postcondition: Q
Essentially case analysis:
wp("if (b) S1 else S2", Q) =
( b ⇒ wp("S1", Q)
Λ ¬ b ⇒ wp("S2", Q) )
```

If statements

```plaintext
// precondition: ??
if (b) S1 else S2
// postcondition: Q
Essentially case analysis:
wp("if (b) S1 else S2", Q) =
( b ⇒ wp("S1", Q)
Λ ¬ b ⇒ wp("S2", Q) )
```

Use the method specification to determine the new value
Reasoning About Loops

- A loop represents an unknown number of paths
  - Case analysis is problematic
  - Recursion presents the same issue
- Cannot enumerate all paths
  - That is what makes testing and reasoning hard

Loops: values and termination

1) Pre-assertion guarantees that $x \geq y$
2) Every time through loop
   - $x \geq y$ holds and, if body is entered, $x > y$
   - $y$ is incremented by 1
   - $x$ is unchanged
   - Therefore, $y$ is closer to $x$ (but $x \geq y$ still holds)
3) Since there are only a finite number of integers between $x$ and $y$, $y$ will eventually equal $x$
4) Execution exits the loop as soon as $x = y$

Loops: values and termination

```java
// assert x >= 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

Understanding loops by induction

- We just made an inductive argument
  Inducting over the number of iterations
- Computation induction
  Show that conjecture holds if zero iterations
  Assume it holds after $n$ iterations and show it holds after $n+1$
- There are two things to prove:
  - Some property is preserved (known as “partial correctness”)
    Loop invariant is preserved by each iteration
  - The loop completes (known as “termination”)
    The “decrementing function” is reduced by each iteration

Loop invariant for the example

```java
// assert x >= 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y
```

- So, what is a suitable invariant?
- What makes the loop work?
  $LI = x \geq y$

  1) $x \geq 0 \& y = 0 \Rightarrow LI$
  2) $LI \& x \neq y \{ y = y + 1; \} LI$
  3) $(LI \& \neg(x \neq y)) \Rightarrow x = y$

Decrementing Function

- Decrementing function $D(X)$
  - Maps state (program variables) to some well-ordered set
  - This greatly simplifies reasoning about termination
- Consider: while (b) S;
- We seek $D(X)$, where $X$ is the state, such that
  1. An execution of the loop reduces the function’s value:
     $LI \& b \{ S \} D(X_{post}) < D(X_{pre})$
  2. If the function’s value is minimal, the loop terminates:
     $(LI \& D(X) = minVal) \Rightarrow \neg b$
Proving Termination

// assert \(x \geq 0 \& \& y = 0\)
// Loop invariant: \(x \geq y\)
while \((x \neq y)\) {
  \(y = y + 1\);
}
// assert \(x = y\)

• Is “\(x - y\)” a good decrementing function?
  1. Does the loop reduce the decrementing function’s value?
     // assert (\(y \leq x\)); let \(d_{\text{pre}} = (x - y)\)
     \(y = y + 1;\)
     // assert (\(d_{\text{post}} - d_{\text{post}}\) < \(d_{\text{pre}}\))
  2. If the function has minimum value, does the loop exit?
     \((x \leq y \& \& x - y = 0)\) \((x = y)\)

Choosing Loop Invariant

• For straight-line code, the wp (weakest precondition) function gives us the appropriate property
• For loops, you have to guess:
  – The loop invariant
  – The decrementing function
• Then, use reasoning techniques to prove the goal property
• If the proof doesn’t work:
  – Maybe you chose a bad invariant or decrementing function
  – Maybe the loop is incorrect
• Fix the code
• Automatically choosing loop invariants is a research topic

In practice

I don’t routinely write loop invariants

I do write them when I am unsure about a loop and when I have evidence that a loop is not working
  – Add invariant and decrementing function if missing
  – Write code to check them
  – Understand why the code doesn’t work
  – Reason to ensure that no similar bugs remain

More on Induction

• Induction is a very powerful tool

\[2^n = 1 + \sum_{i=1}^{n} 2^{i-1}\]

Proof by induction: Base Case

For \(n=1\),

\[1 + \sum_{i=1}^{1} 2^{i-1} = 1 + 2^0 = 1 + 1 = 2 = 2^1\]

Inductive Step

Assume \(2^n = 1 + \sum_{i=1}^{n} 2^{i-1}\) and show that \(2^{n+1} = 1 + \sum_{i=1}^{n+1} 2^{i-1}\)

\[2^{n+1} = 1 + \sum_{i=1}^{n+1} 2^{i-1} = 1 + \sum_{i=1}^{n} 2^{i-1} + 2^n = 2^{n+1} + 2^n = 2 \times 2^n = 2^{n+1}\]

Is Induction Too Powerful?