CS 520

Theory and Practice of Software Engineering
Spring 2021

Reasoning about programs

April 29, 2021

Updates about assignments

- Homework 2 grades and a possible solution were posted yesterday.
- In-class exercise 4 about Model Inference will be graded over the weekend.
- The final project presentations and demonstrations will be held during next Tuesday's lecture 5/4.
- The Week 12 questionnaire about Automated Theorem Proving is due Friday night. There will also be a Week 13 questionnaire about the final projects.

What is Software Engineering?

More than just writing code

The complete process of specifying, designing, developing, analyzing, deploying, and maintaining a software system.

Common Software Engineering tasks include:

- Requirements engineering
- Software architecture and design
- Programming
- Verification & Validation
- Debugging

What is Software Engineering?

More than just writing code

The complete process of specifying, designing, developing, analyzing, deploying, and maintaining a software system.

Common Software Engineering tasks include:

- Requirements engineering
- Software architecture and design
- Programming
- Verification & Validation
- Debugging

Ways to verify your code

- The hard way:
  - Make up some inputs for a given system
  - If that system doesn't crash, ship it
  - When the system fails in the field, attempt to **debug**

- The easier way:
  - Manually reason about possible system behavior and desired outcomes (e.g., **code review**)
  - Automate that reasoning (e.g., **testing**, **model checking**)

Another way to verify your code (that can be easier)

- Prove that the system does what you want
  - Representation (rep) invariants are preserved
  - Implementation satisfies specification
- Proof can be formal (e.g., **theorem prover**) or informal (today, will be informal)
- Complementary to manual and automated reasoning (e.g., **code review**, **testing**, **model checking**)

Reasoning about code

- Determine what facts are true during system execution, e.g.,
  - 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

Possible uses of such facts

- Ensure code is correct (via reasoning or testing)
- Understand why code is incorrect

Code example

```
int y = 100;
for (int x = 0; x < 100; x++)
  y--;
```

What is true about the above?

Code example

```
int y = 100;
for (int x = 0; x < 100; x++)
  y--;
```

What is true about the above?

- *Before loop: x = 0, y = 100*
- *During loop: Each iteration x increases by 1 and y decreases by 1*
- *After loop: x = 100, y = 0*

Forward reasoning

- **Key idea:**
  - You know what is true before running the code. What is true after running the code?
  - Given a precondition, what is the postcondition?
- **Possible uses:**

  Rep invariant holds before running code

  Does it still hold after running code?

- **Example:**

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

Forward reasoning example

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

Forward reasoning example

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

Forward reasoning example

```
// precondition: x is even
x = x + 3;
// x is odd
y = 2x;
// y is even and thus divisible by 2 (but not by 4)
x = 5;
// postcondition: ??
```

Forward reasoning example

```
// precondition: x is even
x = x + 3;
// x is odd
y = 2x;
// y is even and thus divisible by 2 (but not by 4)
x = 5;
// postcondition:  x = 5, y is divisible by 2 (but not by 4)
```

Another forward reasoning example

```
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;
```

⇐ What property holds here?

⇐ What property holds here?

Another forward reasoning example

```
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;
```

⇐ What property holds here? **i + z = x**

⇐ What property holds here? **i + z = x**

Advantages of forward reasoning

- 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

- **Key idea:**
  - 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?
- **Possible uses:**

  (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
```

Backward reasoning example

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

Backward reasoning example

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

Backward reasoning example

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

Backward reasoning example

```
// precondition:  x >= 0
x = x + 3;
// x >= 3
y = 2x;
// y > 5
x = 5;
// postcondition:  y > x
```

Advantages of backward reasoning

- 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

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

```
// 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 = ??

Assignment

```
// precondition: ??
x = e;
// postcondition: Q
```

Precondition: Q with all (free) occurrences of x replaced by e

- Example:

```
// assert:  (x+1) > 0
x = x + 1;
// assert: x > 0
```

Precondition = $(x+1) > 0$

Method calls

```
// 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

Use the method specification to determine the new value

If statements

// precondition: ??
if (b) S1 else S2
// postcondition: Q

Essentially case analysis:

wp(“if (b) S1 else S2”, Q) =

$$\begin{aligned} (\quad & b \Rightarrow wp(\text{“S1”}, Q) \\ \wedge\ \neg & b \Rightarrow wp(\text{“S2”}, Q) \ ) \end{aligned}$$

If example

```
// precondition: ??
if (x == 0) {
   x = x + 1;
} else {
   x = (x/x);
}
// postcondition:  x ≥ 0
```

Precondition:

wp("if (x==0) {x = x+1} else {x = x/x}", $x \geq 0$) =

= ($x = 0 \Rightarrow$ wp("x = x+1", $x \geq 0$)

& $x \neq 0 \Rightarrow$ wp("x = x/x", $x \geq 0$))

= $(x = 0 \Rightarrow x + 1 \geq 0)$ & $(x \neq 0 \Rightarrow x/x \geq 0)$

= $1 \geq 0$ & $1 \geq 0$

= true

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 part of what makes testing and reasoning hard

Loops: values and termination

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

1) **Pre-assertion guarantees that x ≥ y**

2) **Every time through loop**

   x ≥ 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 ≥ 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**

Understanding loops by induction (1)

- **We just made an inductive argument**

  Inducting over the number of iterations

- **Computation induction**

  Show that conjecture holds if 0 iterations

  Assume it holds after n iterations and show it holds after n+1

Understanding loops by induction (2)

- There are two things to prove:
  1. **Some property is preserved (known as "partial correctness")**
     - loop invariant is preserved by each iteration
  2. **The loop completes (known as "termination")**
     - The "decrementing function" is reduced by each iteration

Loop invariant for the example

```
// 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?

Loop Invariant (LI) = $x \geq y$

When you start, LI holds

While in the loop, LI holds

After the loop, postcondition holds

Loop invariant for the example

```
// 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?

Loop Invariant (LI) = $x \geq y$

When you start, LI holds

1) $x \geq 0$ & $y = 0 \Rightarrow$ LI

While in the loop, LI holds

After the loop, postcondition holds

Loop invariant for the example

```
// 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?

Loop Invariant (LI) = $x \geq y$

When you start, LI holds

While in the loop, LI holds

After the loop, postcondition holds

1) $x \geq 0 \;\&\; y = 0 \Rightarrow LI$

2) $LI \;\&\; x \neq y$ {`y = y+1;`} LI

Loop invariant for the example

```
// 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?

Loop Invariant (LI) = $x \geq y$

When you start, LI holds

While in the loop, LI holds

After the loop, postcondition holds

1) $x \geq 0$ & $y = 0 \Rightarrow$ LI

2) LI & $x \neq y$ {`y = y+1;`} LI

3) (LI & $\neg(x \neq y)$) $\Rightarrow$ $x = y$

Is anything missing?

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

Does the loop terminate?

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 ≥ 0 & y = 0
// Loop invariant: x ≥ y
// Loop decrements:  (x-y)
while (x != 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 != x); let d_{pre} = (x – y)
   y = y + 1;
   // ??

2. If the function has minimum value, does the loop exit?
   ??

Proving Termination

```
// assert x ≥ 0 & y = 0
// Loop invariant: x ≥ y
// Loop decrements:  (x-y)
while (x != 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 != x); let $d_{pre} = (x – y)$
   y = y + 1;
   // assert $(x_{post} – y_{post}) < d_{pre}$

2. If the function has minimum value, does the loop exit?
   ??

Proving Termination

```
// assert x ≥ 0 & y = 0
// Loop invariant: x ≥ y
// Loop decrements:  (x-y)
while (x != 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 != x); let $d_{pre} = (x – y)$
   y = y + 1;
   // assert $(x_{post} – y_{post}) < d_{pre}$

2. If the function has minimum value, does the loop exit?
   $(x >= y \ \& \ x – y = 0) \rightarrow (x = y)$

Choosing Loop Invariant (1)

- 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

Choosing Loop Invariant (2)

- Then, use reasoning techniques to prove the goal property
- If the proof doesn't work:
  - Maybe you chose a bad invariant or decrementing function
    - Choose another and try again
  - Maybe the loop is incorrect
    - Fix the code
- Automatically choosing loop invariants is a research topic

In practice

Often don’t routinely write loop invariants

Do write them when unsure about a loop and when 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

$$2^n = 1 + \sum_{k=1}^{n} 2^{k-1}$$

Proof by induction: Base Case

For n=1, $1 + \sum_{k=1}^{1} 2^{k-1} = 1 + 2^0 = 1 + 1 = 2 = 2^1$

Inductive Step

Assume $2^m = 1 + \sum_{k=1}^{m} 2^{k-1}$ and show that $2^{m+1} = 1 + \sum_{k=1}^{m+1} 2^{k-1}$

$$2^{m+1} = 1 + \sum_{k=1}^{m+1} 2^{k-1} = 1 + \sum_{k=1}^{m} 2^{k-1} + 2^m = 2^m + 2^m = 2 \times 2^m = 2^{m+1}$$

Final project presentations and demonstrations

- Will be held during next Tuesday's lecture
- Each final project group will be assigned a Zoom breakout room
- The group will also be randomly assigned a 6 minute timeslot
  - talk for 3-5 minutes
  - Q&A for ~1 minute

Final project deliverables

- **Due:** Tuesday May 11 at 11:59 PM (a little before midnight)
- **Submission:**
  - **Research (MSR and ML development toolkits):** A short paper (at least 5 pages) describing the problem, approach, experimental evaluation, references. Also a presentation.
  - **Development (EleNa systems):** A project folder including the documentation (e.g., README, architecture, design, manual test results, etc.), source code, automated test suites, etc. Also README for how to run your system or a video showing running the system.