CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Lecture 2 – Reasoning About Straight-Line Code

Outline

- Adminstrivia
- Recap (highlights only)
- Q & A
- Exercises
- More Examples (bit more complex)

Administrivia: HW

- HW0 may have been a struggle
  - will show you how to make this easy

- HW1 posted shortly
  - worksheet
  - practice applying these ideas
    - verifying correctness of short, non-loop code
  - due on Monday by 11pm

Administrivia: Section Splits

Sections

Each section will split into two sub-sections. For example, section AA on the calendar becomes AA-1 and AA-2 that both meet at 8:30am. The table below shows which students should go to which of the two subsections based on one of the digits in their **UW Student Number**.

See the Zoom page to find the link to the meeting for that section (e.g., "Section AA-1").

Time	Name	Split	Value	TA
8:30	AA-1	**last** digit	odd	Yihang
	AA-2	last digit	even	Chloe
9:30	AB-1	last digit	odd	Alexey
	AB-2	last digit	even	Rachel
10:30	AC-1	last digit	odd	Andrew
	AC-2	last digit	even	Manchen
11:30	AD-1	last digit	odd	Dmitriy
	AD-2	last digit	even	Chanwut
12:30	AE-1	**second** digit	odd	Frank
	AE-2	second digit	even	Jasmine

https://canvas.uw.edu/courses/1370605/pages/sections

Administrivia: Section

- Each section has 16-20 students
  - hopefully, you will get to know the other students

- Section plan: Q&A, review, worksheet
  - may want to print the worksheet beforehand (if you can)
  - worksheet is similar to HW1

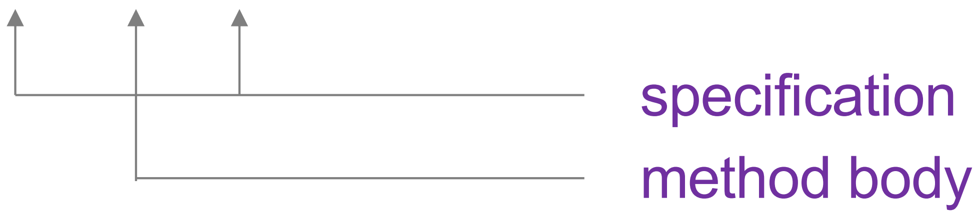
Quick Recap (10 min)

Correctness Toolkit

Hoare Logic

- A Hoare triple is two assertions and one piece of code:

  {*P*} *S* {*Q*}

  – *P* the precondition
  – *S* the code
  – *Q* the postcondition

- A Hoare triple **{*P*} *S* {*Q*}** is called valid if:
  – in any state where P holds,
    executing S produces a state where Q holds
  – i.e., if *P* is true before *S*, then *Q* must be true after it
  – otherwise the triple is called invalid
  – code is correct iff triple is valid

Reasoning Forward & Backward

- Forward:
  - start with the **given** precondition
  - fill in the **strongest** postcondition

$\{P\}\ S\ \{?\}$ →

- Backward
  - start with the **required** postcondition
  - fill in the **weakest** precondition

$\{?\}\ S\ \{Q\}$ ←

- Finds the "best" assertion that makes the triple valid

Reasoning: Assignments

```
x = ...
```

- Forward
  - add the fact “x = ...” to what is known
  - BUT you must *fix* any existing references to “x”

- Backward
  - just replace “x” with “...” in the postcondition (substitution)

Reasoning: If Statements

Forward reasoning

```
{{ P }}
if (cond)
    {{ P and cond }}
    S1
    {{ P1 }}
else
    {{ P and not cond }}
    S2
    {{ P2 }}
{{ P1 or P2 }}
```

Backward reasoning

```
{{ cond and Q1 or
    not cond and Q2 }}
if (cond)
    {{ Q1 }}
    S1
    {{ Q }}
else
    {{ Q2 }}
    S2
    {{ Q }}
{{ Q }}
```

Validity with Fwd & Back Reasoning

Reasoning in either direction gives valid assertions

Just need to check adjacent assertions:

- top assertion must imply bottom one

```
{{ P }}
  S1
  S2
{{ P1 }}
{{ Q }}
```

```
{{ P }}
{{ Q1 }}
  S1
  S2
{{ Q }}
```

```
{{ P }}
  S1
{{ P1 }}
{{ Q1 }}
  S2
{{ Q }}
```

Q & A

Dropping Irrelevant Facts

- Forward reasoning often adds many irrelevant facts
- Dropping them is *usually* okay

{{ P }}	{{ P }}
`S1`	`S1`
{{ Q }}	{{ Q1 }}
`S2`	`S2`
{{ _ }}	{{ R1 }}

- Result is still a valid triple (ok to weaken postcondition)
- BUT no longer the **strongest** postcondition

- May get a final postcondition that doesn't imply the given one
- In that case, put them back and try again...

Exercises

Hoare Triples

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
- `{} if(x > 7) {y=4;} else {y=3;} {y < 5}` valid
- `{} x = y; z = x; {y=z}` valid
- `{x=7 ∧ y=5}`
  `tmp=x; x=tmp; y=x;` invalid
  `{y=7 ∧ x=5}`

Forward Reasoning

{{ x >= 0 }}

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

{{ ________________________________ }}

Forward Reasoning

{{ x >= 0 }}

```
 if (x != 0) {
```

{{ x > 0 }}

```
   z = x;
```

{{ x > 0 and z = x }}

```
 } else {
```

{{ x = 0 }}

```
   z = x + 1;
```

{{ x = 0 and z = 1 }}

```
 }
```

{{ (x > 0 and z = x) or (x = 0 and z = 1) }} ⇒ {{ z > 0 }} but strictly weaker

Backward Reasoning

{{ ______________________________________ }}

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

{{ y > 5 }}

Backward Reasoning

{{ (x > 7 and x > 5) or (x <= 7) }} ⇔ {{ (x > 7) or (x <= 7) }}

⇔ {{ }}

```
 if (x > 7) {
```

{{ x > 5 }}

```
    y = x;
```

{{ y > 5 }}

```
 } else {
```

{{ 20 > 5 }} ⇔ {{ }}

```
    y = 20;
```

{{ y > 5 }}

```
 }
```

{{ y > 5 }}

More Examples

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
  y = x/2;
else
  y = -((-x+1)/2);
```

{{ 2y = x or 2y = x - 1 }}

Note that, in Java, a/b rounds *toward zero*.

Harder Example

Compute x/2 rounded toward minus infinity.

```
{{ }}
if (x >= 0)
    {{ x >= 0 }}
    y = x/2;
else
    {{ x < 0 }}
    y = -((-x+1)/2);
{{ 2y = x or 2y = x - 1 }}
```

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
```

{{ x >= 0 }}

```
y = x/2;
```

{{ 2y = x or 2y = x - 1 }}

```
else
```

{{ x < 0 }}

```
y = -((-x+1)/2);
```

{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}
```
if (x >= 0)
```
{{ x >= 0 }}
```
y = x/2;
```
{{ 2y = x or 2y = x - 1 }}
```
else
```
{{ x < 0 }}
```
y = -((-x+1)/2);
```
{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

?

?

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
```

{{ x >= 0 }}

```
y = x/2;
```

{{ 2y = x or 2y = x - 1 }}

since x >= 0, "/" rounds down so this is valid

```
else
```

{{ x < 0 }}

```
y = -((-x+1)/2);
```

{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
```

...

```
else
```

{{ x < 0 }}

```
y = (x+1)/2;  // was y = -((-x+1)/2);
y = -y;
```

{{ 2y = x or 2y = x - 1 }}

{{ 2y = x or 2y = x - 1 }}

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
```

...

```
else
```

{{ x < 0 }}

```
y = (-x+1)/2;
```

{{ 2y = -x or 2y = -x + 1 }}

```
y = -y;
```

{{ 2y = x or 2y = x - 1 }}

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}
```
if (x >= 0)
```
...
```
else
```
{{ x < 0 }}
```
y = (-x+1)/2;
```
{{ 2y = (-x + 1) - 1 or 2y = -x + 1 }}
```
y = -y;
```
{{ 2y = x or 2y = x - 1 }}

Harder Example

Compute x/2 rounded toward minus infinity.

{{ }}

```
if (x >= 0)
```

...

```
else
```

{{ x < 0 }}

```
y = (-x+1)/2;
```

{{ 2y = (-x + 1) - 1 or 2y = -x + 1 }}

since -x > 0, “/” rounds down so this is valid

```
y = -y;
```

{{ 2y = x or 2y = x - 1 }}

Useful Subscripts Example: swap

- Consider code for a swapping x and y

{{ }}
```
 tmp = x;
```
{{ tmp = x }}
```
 x = y;
```
{{ tmp = x_0 and x = y }}
```
 y = tmp;
```
{{ tmp = x_0 and x = y_0 and y = tmp }}

- Post condition implies x = y_0 and y = x_0
- I.e., their final values are equal to the original values swapped