CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2021

Lecture 3 – Reasoning about Loops

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

specification

method body

- 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 = expr
```

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

- Backward
  - just replace any “x” in the postcondition with expr (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 }}

Reasoning So Far

- "Turn the crank" reasoning for assignment and if statements

- All code (essentially) can be written just using:
  - assignments
  - if statements
  - while loops

- Only part we are missing is **loops**

Reasoning About Loops

- Loop reasoning is not as easy as with “=“ and “if”
  - recall Rice’s Theorem (from 311): checking any non-trivial semantic property about programs is **undecidable**

- We need help (more information) before the reasoning again becomes a turn-the-crank process

- That help comes in the form of a “loop invariant”

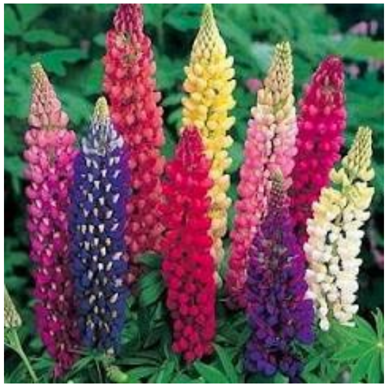
Loop Invariant

A **loop invariant** is an assertion that holds at the top of the loop:

{{ Inv: I }}
```
while (cond)
  S
```

- It holds when we **first get to** the loop.
- It holds each time we execute `S` and **come back to** the top.

Notation: I'll use "**Inv**:" to indicate a loop invariant.


Lupin variants

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

{{ P }}
```
S1
```

{{ Inv: **`I`** }}
```
while (cond)
  S2

S3
```
{{ Q }}

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

{{ P }}
```
 S1
```
{{ P1 }}

{{ Inv: **`I`** }}
```
 while (cond)
   S2

 S3
```
{{ Q }}

Need to check that P1 implies **`I`**
(i.e., that **`I`** is true the first time)

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

{{ P }}
```
S1
```
{{ Inv: **`I`** }}
```
while (cond)
```
  {{ **`I`** and cond }}
```
  S2
```
  {{ P2 }}

Need to check that P2 implies **`I`** again
(i.e., that **`I`** is true each time around)

```
S3
```
{{ Q }}

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

Let's try forward reasoning...

```
{{ P }}
  S1

{{ Inv: I }}
  while (cond)
     S2

{{ I and not cond }}
  S3
{{ P3 }}
{{ Q }}
```

Need to check that P3 implies Q

(i.e., Q holds after the loop)

Checking Correctness of a Loop

Consider a while-loop (other loop forms not too different) with a loop invariant **`I`**.

```
{{ P }}
 S1

{{ Inv: I }}
 while (cond)
   S2

 S3
{{ Q }}
```

Informally, we need:

- **`I`** holds initially
- **`I`** holds each time around
- **`Q`** holds after we exit

Formally, we need validity of:

- {{ P }} `S1` {{ **`I`** }}
- {{ **`I`** and cond }} `S2` {{ **`I`** }}
- {{ **`I`** and not cond }} `S3` {{ Q }}

(can check these with backward reasoning instead)

More on Loop Invariants

- Loop invariants are crucial information
  - needs to be provided before reasoning is “turn the crank”

- Pro Tip: always document your invariants for *non-trivial* loops
  - don’t make code reviewers guess the invariant

- Pro Tip: with a good loop invariant, the code is easy to write
  - all the creativity can be saved for finding the invariant
  - more on this in later lectures…

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Equivalent to this "for" loop:

```
s = 0;
for (int i = 0; i != n; i++)
  s = s + b[i];
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ s = 0 and i = 0 }}

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies
  s = b[0] + … + b[i-1] ?

  Yes. (An empty sum is zero.)

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

- (s = 0 and i = 0) implies **I**

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
{{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
  i = i + 1;
```
{{ s = b[0] + ... + b[i-1] }}
```
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **`I`**
- {{ **`I`** and i != n }} `S` {{ **`I`** }} ?

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = 0;
```

{{ Inv: s = b[0] + ... + b[i-1] }}

```
while (i != n) {
```

{{ s = b[0] + ... + b[i-1] and i != n }}

```
  s = s + b[i];
  i = i + 1;
```

{{ s = b[0] + ... + b[i-1] }}

```
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **I**
- {{ **I** and i != n }} `S` {{ **I** }} ?

{{ s + b[i] = b[0] + … + b[i] }}
{{ s = b[0] + … + b[i] }}

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = 0;
```
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = 0) implies **`I`**
- {{ **`I`** and i != n }} `S` {{ **`I`** }}
- {{ **`I`** and not (i != n) }} implies s = b[0] + … + b[n-1] ?

Example: sum of array

Consider the following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
i = 0;
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  s = s + b[i];
  i = i + 1;
}
{{ s = b[0] + ... + b[n-1] }}
```

- (s = 0 and i = 0) implies **`I`**
- {{ **`I`** and i != n }} `S` {{ **`I`** }}
- {{ **`I`** and i = n }} implies `Q`

These three checks verify the postcondition holds (i.e., the code is correct)

Termination

- Technically, this analysis does not check that the code **terminates**
  - it shows that the postcondition holds if the loop exits
  - but we never showed that the loop actually exits
- However, that follows from an analysis of the running time
  - e.g., if the code runs in $O(n^2)$ time, then it terminates
  - an infinite loop would be O(infinity)
  - any finite bound on the running time proves it terminates
- It is normal to also analyze the running time of code we write, so we get termination already from that analysis.

Reasoning So Far

- Forward and backward reasoning for...
  - assignments
  - if statements
  - while loops

- (essentially) all code can be rewritten to use just these

Example HW problem

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ ________________ }}
i = 0;
{{ ________________ }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
   {{ ________________ }}
   s = s + b[i];
   {{ ________________ }}
   i = i + 1;
   {{ ________________ }}
}
{{ ________________ }}
{{ s = b[0] + ... + b[n-1] }}
```

Example HW problem

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
   {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
   {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
   i = i + 1;
   {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does invariant hold initially?

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}

Holds initially? Yes: i = 0 implies s = b[0] + … + b[-1] = 0

```
while (i != n) {
```
{{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
{{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
{{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

i = 3: s = b[0] + b[1] + b[2]
i = 2: s = b[0] + b[1]
i = 1: s = b[0]
i = 0: s = 0

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
   {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
   {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
   i = i + 1;
   {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?
No, need to also check...

Does postcondition hold on termination?

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
   {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
   {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
   i = i + 1;
   {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Postcondition holds? Yes, since i = n.

Are we done?
No, need to also check...

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

Are we done?
No, need to also check...

Does loop body preserve invariant?

Yes. Weaken by dropping “i-1 != n”

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does loop body preserve invariant?

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Warning: not just filling in blanks

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
{{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
{{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
{{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

Does loop body preserve invariant?

{{ s + b[i] = b[0] + ... + b[i] }}
```
s = s + b[i];
```
{{ s = b[0] + ... + b[i] }}
```
i = i + 1
```
{{ s = b[0] + ... + b[i-1] }}

Yes. If Inv holds, then so does this (just add b[i] to both sides of Inv)

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = -1;
```
{{ Inv: s = b[0] + ... + b[i] }} Changed
```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```
{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

Changed from i = 0

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

Changed from n

Reordered

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

{{ s + b[i+1] = b[0] + ... + b[i+1] }}

{{ s + b[i] = b[0] + … + b[i] }}

{{ s = b[0] + ... + b[i] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = -1;
```
{{ Inv: s = b[0] + ... + b[i] }}
```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```
{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = -1) implies **`I`**
  - as before
- {{ **`I`** and i != n-1 }} `S` {{ **`I`** }}
  - reason backward:

{{ s + b[i+1] = b[0] + ... + b[i+1] }}
{{ s + b[i] = b[0] + … + b[i] }}

- (**`I`** and i = n-1) implies **`Q`**
  - as before

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we miss-order the assignments to `i` and `s`…

Where does the correctness check fail?

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
i = -1;
```
{{ Inv: s = b[0] + ... + b[i] }}
```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```
{{ s = b[0] + ... + b[n-1] }}

Suppose we miss-order the assignments to `i` and `s`…

We can spot this bug because the invariant does not hold:

{{ s + b[i] = b[0] + ... + b[i+1] }}
{{ s = b[0] + ... + b[i+1] }}
{{ s = b[0] + ... + b[i] }}

First assertion is not Inv.