CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2020
Lecture 3 – Reasoning about Loops
Reasoning So Far

• “Turn the crank” reasoning for assignment and if statements

• All code (essentially) can be written just using:
  – assignments
  – if statements
  – 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:

```markdown
{{ 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.
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...

\[
\begin{align*}
\{ P \} \\
S1 \\
\{ \text{Inv: } I \} \\
\text{while (cond)} \\
S2 \\
S3 \\
\{ Q \}
\end{align*}
\]
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...

```latex
\begin{align*}
\{\{ P \}\} & \quad S1 \\
\{\{ \text{Inv: } I \}\} & \quad \text{while (cond)} \\
\quad & \quad \{\{ I \text{ and cond} \}\} \\
\downarrow & \\
\quad & \quad \{\{ P2 \}\} \\
\quad & \quad S2 \\
\{\{ Q \}\} & \quad S3
\end{align*}
```

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

{\text{$\{ P \}$}}

\hspace{1cm} S1

{\text{$\{ \text{Inv: } I \}$}}

\hspace{1cm} while (cond)

\hspace{2cm} S2

{\text{$\{ I \text{ and not cond} \}$}}

\hspace{1cm} S3

{\text{$\{ P3 \}$}}

{\text{$\{ Q \}$}}

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 \( I \).

\[
\begin{align*}
\{ \{ P \} \} & \quad S_1 \\
\{ \{ \text{Inv: } I \} \} & \quad \text{while (cond)} \\
& \quad S_2 \\
& \quad S_3 \\
\{ \{ Q \} \}
\end{align*}
\]

Informally, we need:
- \( I \) holds initially
- \( I \) holds each time around
- \( Q \) holds after we exit

Formally, we need validity of:
- \( \{ \{ P \} \} S_1 \{ \{ I \} \} \)
- \( \{ \{ I \text{ and cond} \} \} S_2 \{ \{ I \} \} \)
- \( \{ \{ I \text{ and not cond} \} \} S_3 \{ \{ Q \} \} \)

(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
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
    s = 0;
    i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
{{ s = b[0] + \ldots + b[n-1] }}
```

Equivalent to this “for” loop:

```java
s = 0;
for (int i = 0; i != n; i++)
    s = s + b[i];
```
Example: sum of array

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    s = s + b[i];
i = i + 1;
}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
{{ b.length >= n }}
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 $b[0] + \ldots + b[n-1]$: 

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

• (s = 0 and i = 0) implies $s = b[0] + \ldots + b[i-1]$?
  Yes. (An empty sum is zero.)
```
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

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

- \((s = 0 \text{ and } i = 0) \implies \)
Consider the following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = 0;

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

- $(s = 0$ and $i = 0) \implies \mathcal{I}$
- $\{\{ \mathcal{I} \text{ and } i != n \}\} \implies \{\{ \mathcal{I} \}\}$?

```
Example: sum of array

Consider the following code to compute $b[0] + ... + b[n-1]$: 

```java
{{ b.length >= n }}
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 $i != n )\} \implies s \{( I )\}^?$

$\{( s + b[i] = b[0] + ... + b[i] )\}$
$\{( s = b[0] + ... + b[i] )\}$
Example: sum of array

Consider the following code to compute \( b[0] + \ldots + b[n-1] \):

\[
\begin{align*}
\{ \text{b.length }\geq n \} \\
& s = 0; \\
& i = 0; \\
\{ \text{Inv: s = b[0] + \ldots + b[i-1]} \} \\
\text{while (i }\neq \text{n)} \{ \\
& s = s + b[i]; \\
& i = i + 1; \\
\} \\
\{ \text{s = b[0] + \ldots + b[i-1] and not (i }\neq \text{n)} \} \\
\{ \text{s = b[0] + \ldots + b[n-1]} \}
\end{align*}
\]

- \( (s = 0 \text{ and } i = 0) \) implies \( I \)
- \( \{ I \text{ and } i \neq n \} \) \( s \} \{ I \} \)
- \( \{ I \text{ and not (}i \neq n) \} \) implies \( s = b[0] + \ldots + b[n-1] \)?
Example: sum of array

Consider the following code to compute \(b[0] + \ldots + b[n-1]\):

\[
\begin{align*}
\{ \text{b.length} \geq n \} \\
&\quad s = 0; \\
&\quad i = 0; \\
\{ \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
&\quad \text{while (i != n) } \{ \\
&\quad\quad s = s + b[i]; \\
&\quad\quad i = i + 1; \\
&\quad \} \\
\{ \text{s = b[0] + ... + b[n-1]} \}
\end{align*}
\]

- \((s = 0 \text{ and } i = 0)\) implies \(I\)
- \(\{ I \text{ and } i != n \} \) \(S\) \(\{ I \} \)
- \(\{ I \text{ and } i = n \} \) 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 $O(\text{infinity})$
  – 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
  – loops

• (essentially) all code can be rewritten to use just these
Example HW problem

The following code to compute $b[0] + \ldots + b[n-1]$:

```java
{{ }}
s = 0;
{{ ____________ }}
i = 0;
{{ ____________ }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
    {{ ____________ }}
s = s + b[i];
    {{ ____________ }}
i = i + 1;
    {{ ____________ }}
}
{{ ____________ }}
{{ s = b[0] + \ldots + b[n-1] }}
```
Example HW problem

The following code to compute $b[0] + \ldots + b[n-1]$:

```c
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + \ldots + b[i-1] }}
while (i != n) {
  {{ s = b[0] + \ldots + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + \ldots + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + \ldots + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + \ldots + b[i-1] and not (i != n) }}
{{ s = b[0] + \ldots + b[n-1] }}
```
The following code to compute $b[0] + \ldots + b[n-1]$:

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

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

Does invariant hold initially?
Warning: not just filling in blanks

The following code to compute $b[0] + \ldots + b[n-1]$:  

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

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

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

- $i = 2$: $s = b[0] + b[1]$
- $i = 1$: $s = b[0]$
- $i = 0$: $s = 0$
Warning: not just filling in blanks

The following code to compute $b[0] + \ldots + b[n-1]$:

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

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

Does postcondition hold on termination?
The following code to compute $b[0] + ... + b[n-1]$:


define

$$s = 0;$$
$$i = 0;$$
$$s = 0 + b[0] + ... + b[i-1]$$
$$i = i + 1;$$
$$s = s + b[i];$$
$$i = i + 1;$$
$$s = s + b[i];$$
$$i = i + 1;$$
$$s = s + b[i];$$

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

Postcondition holds? Yes, since $i = n$. 

CSE 331 Spring 2020
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] }}
```
Warning: not just filling in blanks

The following code to compute \( b[0] + \ldots + b[n-1] \):

\[
\begin{align*}
&\{ \} \\
&s = 0; \quad \{ s = 0 \} \\
&i = 0; \quad \{ s = 0 \text{ and } i = 0 \} \\
&\{ \text{Inv: } s = b[0] + \ldots + b[i-1] \} \\
&\text{while (i != n) } \{ \\
&\quad \{ s = b[0] + \ldots + b[i-1] \text{ and } i != n \} \\
&\quad s = s + b[i]; \quad \{ s = b[0] + \ldots + b[i] \text{ and } i != n \} \\
&\quad i = i + 1; \quad \{ s = b[0] + \ldots + b[i-2] + b[i-1] \text{ and } i-1 != n \} \\
&\} \\
&\{ s = b[0] + \ldots + b[i-1] \text{ and not } (i != n) \} \\
&\{ s = b[0] + \ldots + b[n-1] \} \\
\end{align*}
\]

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

Does loop body preserve invariant?

\[
\begin{align*}
&\{ s + b[i] = b[0] + \ldots + b[i] \} \\
&s = s + b[i]; \quad \{ s = b[0] + \ldots + b[i] \} \\
&i = i + 1 \quad \{ s = b[0] + \ldots + b[i-1] \} \\
\end{align*}
\]
Warning: not just filling in blanks

The following code to compute \( b[0] + \ldots + b[n-1] \):

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

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

Does loop body preserve invariant?

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 $b[0] + \ldots + b[n-1]$:

```java
{{ b.length >= n }}
s = 0;
i = -1;
while (i != n - 1) {
    i = i + 1;
    s = s + b[i];
}
{{ s = b[0] + \ldots + b[n-1] }}
```

- Changed from $i = 0$
- Changed from $n$
- Reordered
Consider the following code to compute $b[0] + \ldots + b[n-1]$:

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

Changed
Consider the following code to compute $b[0] + \ldots + b[n-1]$:

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

```
{{ s + b[i+1] = b[0] + \ldots + b[i+1] }}
{{ s + b[i] = b[0] + \ldots + b[i] }}
{{ s = b[0] + \ldots + b[i] }}
```
Example: sum of array (attempt 2)

Consider the following code to compute $b[0] + \ldots + b[n-1]$:

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

- $(s = 0 \text{ and } i = -1)$ implies $I$
  - as before

- $\{\{ I \text{ and } i != n-1 \}\} \supseteq \{\{ I \}\}$
  - reason backward:
    - $(s + b[i+1] = b[0] + \ldots + b[i+1])$
    - $(s + b[i] = b[0] + \ldots + b[i])$

- $(I \text{ and } i = n-1)$ implies $Q$
  - as before
Example: sum of array (attempt 3)

Consider the following code to compute $b[0] + ... + b[n-1]$:

```java
{{ b.length >= n }}
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 \( b[0] + \ldots + b[n-1] \):

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

Suppose we miss-order the assignments to \( i \) and \( s \)... We can spot this bug because the invariant does not hold:

```java
{{ s + b[i] = b[0] + \ldots + b[i+1] }}
{{ s = b[0] + \ldots + b[i+1] }}
{{ s = b[0] + \ldots + b[i] }}
```

First assertion is not Inv.