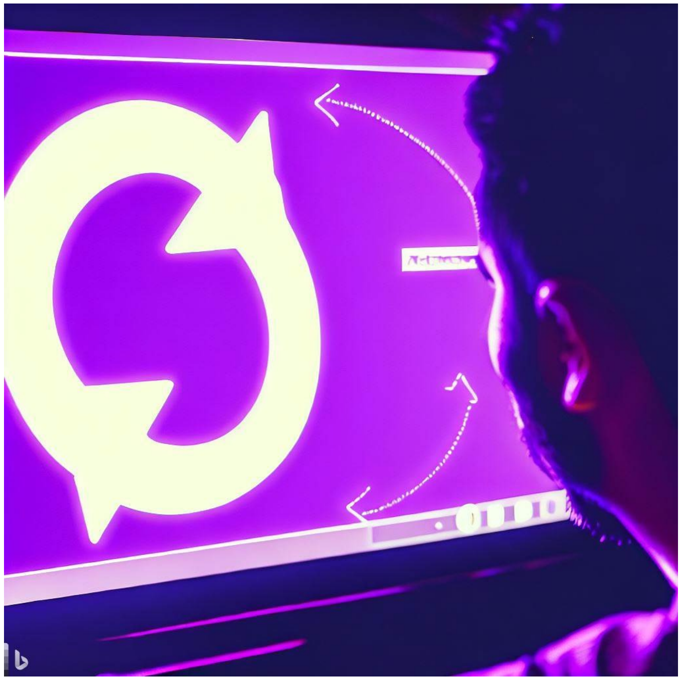

CSE 331

Loops in Floyd Logic

Kevin Zatloukal

Administrivia

- HW4 spells things out less than before
  - multiple solutions are often possible
- You make more decisions going forward…
- Reminder: HW9 has no details
  - just screen shots of a "potential UI"

Forward and Backward Reasoning

- Imperative code made up of
  - assignments (mutation)
  - conditionals
  - loops

- Anything can be rewritten with just these

- We will learn forward / backward rules for all three
  - will also learn a rule for function calls
  - once we have those, we are done

Backward Reasoning through Assignments

- For assignments, backward reasoning is substitution

$\{\{\ Q[x \mapsto y]\ \}\}$

```
  x = y;
```

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

  - just replace all the "x"s with "y"s
  - we will denote this substitution by $Q[x \mapsto y]$

- Mechanically simpler than forward reasoning

Forward Reasoning through Assignments

- For assignments, forward reasoning rule is

$\{\{\ P\ \}\}$
```
  x = y;
```
$\{\{\ P[x \mapsto x_0] \text{ and } x = y[x \mapsto x_0]\ \}\}$

  - replace all "x"s in P and y with "x_0"s (or any *new* name)

- This process can be simplified in many cases
  - no need for x_0 if we can write it in terms of new value
  - e.g., if "$x = x_0 + 1$", then "$x_0 = x - 1$"
  - assertions will be easier to read without old values

    (Technically, this is weakening, but it's usually fine
    Postconditions usually do not refer to old values of variables.)

Forward Reasoning through Assignments

- For assignments, forward reasoning rule is

$\{\{ P \}\}$

```
x = y;
```

$\{\{ P[x \mapsto x_0] \text{ and } x = y[x \mapsto x_0] \}\}$

x_0 is any **new** variable name

- If $x_0 = f(x)$, then we can simplify this to

$\{\{ P \}\}$

```
x = y;
```

$\{\{ P[x \mapsto f(x)] \}\}$

  - if the new part is "$x = x_0 + 1$", then "$x_0 = x - 1$"
  - if the new part is "$x = 2x_0$", then "$x_0 = x/2$"
  - does not work for integer division (an un-invertible operation)

Correctness Example

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
function f(n: number): number {
  n = n + 3;
  return n * n;
}
```

- **Check that value of n *at end* satisfies $\mathrm{n}^2 \geq 10$**

Correctness Example

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
function f(n: number): number {
  {{ n ≥ 1 }}
  n = n + 3;
  {{ n² ≥ 10 }}
  return n * n;
}
```

- Precondition and postcondition come from spec
- Remains to check that the triple is valid

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
function f(n: number): number {
  {{ n ≥ 1 }}
  n = n + 3;
  {{ n – 3 ≥ 1 }}
  {{ n² ≥ 10 }}
  return n * n;
}
```

$n = n_0 + 3$ **means** $n - 3 = n_0$

check this implication

$$
\begin{aligned}
n^2 &\geq 4^2 && \text{since } n - 3 \geq 1 \text{ (i.e., } n \geq 4) \\
&= 16 \\
&> 10
\end{aligned}
$$

Correctness Example by Backward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
function f(n: number): number {
  {{ n ≥ 1 }}
  {{ (n + 3)^2 ≥ 10 }}
  n = n + 3;
  {{ n^2 ≥ 10 }}
  return n * n;
}
```

check this implication

$$
\begin{aligned}
(n+3)^2 &\geq (1+3)^2 && \text{since } n \geq 1 \\
&= 16 \\
&> 10
\end{aligned}
$$

Function Calls

Reasoning about Function Calls

```
// @requires P₂            -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
function f(a: number, b: number): number
```

- **Forward reasoning rule is**

$\{\{ P \}\}$

```
x = f(a, b);
```

$\{\{ P[x \mapsto x_0] \text{ and } R \}\}$

Must also check that P implies P_2

- **Backward reasoning rule is**

$\{\{ Q_1 \text{ and } P_2 \}\}$

```
x = f(a, b);
```

$\{\{ Q_1 \text{ and } Q_2 \}\}$

Must also check that R implies Q_2

Q_2 is the part of postcondition using "x"

Conditionals

Forward Reasoning through Conditionals

- Forward reasoning on conditionals proceeds like this

```
{{ P }}
 if (cond) {
   S
 } else {
   T
 }
{{ ____________ }}
```

  - fill in the postcondition
  - will depend on what code is in `S` and `T`

Forward Reasoning through Conditionals

- Forward reasoning on conditionals proceeds like this

```
{{ P }}
 if (cond) {
  {{ P and cond }}
    S
 } else {
  {{ P and not cond }}
    T
 }
{{ ______________ }}
```

  – push P into top of **both** the then and else branches
  – add whether `cond` was true or false

    same facts are true but we gain one new fact (about cond)

Forward Reasoning through Conditionals

- **Forward reasoning on conditionals proceeds like this**

```
{{ P }}
 if (cond) {
  {{ P and cond }}
   S
  {{ Q1 }}
 } else {
  {{ P and not cond }}
   T
  {{ Q2 }}
 }
{{ ____________ }}
```

  - reason through `S` and `T`

    use whatever rules are appropriate to that code

Forward Reasoning through Conditionals

- **Forward reasoning on conditionals proceeds like this**

{{ P }}
```
if (cond) {
```
{{ P and cond **}}**
```
    S
```
{{ Q_1 **}}**
```
} else {
```
{{ P and not cond **}}**
```
    T
```
{{ Q_2 **}}**
```
}
```
{{ Q_1 or Q_2 }}

  - **pull the postconditions out, combine with "or"**

    either thing could be true since we could go through either branch

Backward Reasoning through Conditionals

- Backward reasoning on conditionals proceeds like this

```
{{ ____________ }}
 if (cond) {
   S
 } else {
   T
 }
{{ Q }}
```

  - fill in the precondition
  - will depend on what code is in `S` and `T`

Backward Reasoning through Conditionals

- Backward reasoning on conditionals proceeds like this

```
{{ ____________ }}
 if (cond) {
   S
  {{ Q }}
 } else {
   T
  {{ Q }}
 }
{{ Q }}
```

  – push Q into top of **both** the then and else branches

    Q needs to be true at the bottom of both

Backward Reasoning through Conditionals

- **Backward reasoning on conditionals proceeds like this**

```
{{ ______________ }}
if (cond) {
  {{ P1 }}
    S
  {{ Q }}
} else {
  {{ P2 }}
    T
  {{ Q }}
}
{{ Q }}
```

  - reason through `S` and `T`

    use whatever rules are appropriate to that code

Backward Reasoning through Conditionals

- **Backward reasoning on conditionals proceeds like this**

{{ (P_1 and cond) or (P_2 and not cond) }}
```
if (cond) {
  {{ P1 }}
   S
  {{ Q }}
} else {
  {{ P2 }}
   T
  {{ Q }}
}
{{ Q }}
```

  – **pull the preconditions out, combine as above**

    P_1 being true is only enough if cond is true, likewise for P_2

Example Reasoning through Conditionals

- Try this working forward

```
{{ }}
 if (x >= 0) {
   y = x;
 } else {
   y = -x;
 }
{{_______________}}
```

Example Reasoning through Conditionals

- **Try this working forward**

```
{{ }}
 if (x >= 0) {
   y = x;
 } else {
   y = -x;
 }
{{ ____________ }}
```

  - **precondition has no facts "{{ }}"**

    variables could have any legal values

Example Reasoning through Conditionals

- Try this working forward

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
  y = x;
} else {
```

$\{\{\ x < 0\ \}\}$

```
  y = -x;
}
```

$\{\{$ ________________ $\}\}$

Example Reasoning through Conditionals

- **Try this working forward**

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
y = x;
```

$\{\{\ x \geq 0 \text{ and } y = x\ \}\}$

```
} else {
```

$\{\{\ x < 0\ \}\}$

```
y = -x;
```

$\{\{\ x < 0 \text{ and } y = -x\ \}\}$

```
}
```

$\{\{$ ________________ $\}\}$

Example Reasoning through Conditionals

- **Try this working forward**

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
    y = x;
```

$\{\{\ x \geq 0 \text{ and } y = x\ \}\}$

```
} else {
```

$\{\{\ x < 0\ \}\}$

```
    y = -x;
```

$\{\{\ x < 0 \text{ and } y = -x\ \}\}$

```
}
```

$\{\{\ (x \geq 0 \text{ and } y = x) \text{ or } (x < 0 \text{ and } y = -x)\ \}\}$ **or equiv** $\{\{\ y = |x|\ \}\}$

Example Reasoning through Conditionals

- Try this working forward

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
  y = x;
```

$\{\{\ x \geq 0 \text{ and } y = x\ \}\}$

```
} else {
```

$\{\{\ x < 0\ \}\}$

```
  y = -x;
```

$\{\{\ x < 0 \text{ and } y = -x\ \}\}$

```
}
```

$\{\{$ ________________ $\}\}$

Warning: don't write $y \geq 0$ here!
That's true, but not strongest.

Example Reasoning through Conditionals

- Try this working forward

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
  y = x;
```

$\{\{\ y \geq 0\ \}\}$ **Wrong!**

```
} else {
```

$\{\{\ x < 0\ \}\}$

```
  y = -x;
```

$\{\{\ y > 0\ \}\}$ **Wrong!**

```
}
```

$\{\{\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\ \}\}$

Example Reasoning through Conditionals

- **Try this working forward**

$\{\{\ \}\}$

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

$\{\{\ x \geq 0\ \}\}$

```
    y = x;
```

$\{\{\ y \geq 0\ \}\}$ **Wrong!**

```
} else {
```

$\{\{\ x < 0\ \}\}$

```
    y = -x;
```

$\{\{\ y > 0\ \}\}$ **Wrong!**

```
}
```

$\{\{\ y \geq 0$ or $y > 0\ \}\}$ or equiv $\{\{\ y \geq 0\ \}\}$

- this is true, but it's not strong enough to show $y = |x|$

Example Reasoning through Conditionals

- Try this working backward

$\{\{ \underline{\hspace{10em}} \}\}$

```
if (x >= 0) {
  y = x;
} else {
  y = -x;
}
```

$\{\{\ y = |x|\ \}\}$

Example Reasoning through Conditionals

- Try this working backward

```
{{ ______________ }}
if (x >= 0) {
    y = x;
  {{ y = |x| }}
} else {
    y = -x;
  {{ y = |x| }}
}
{{ y = |x| }}
```

Example Reasoning through Conditionals

- Try this working backward

$\{\{$ ______________ $\}\}$

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

$\{\{\ x = |x|\ \}\}$ **or equiv** $\{\{\ x \geq 0\ \}\}$

```
    y = x;
```

$\{\{\ y = |x|\ \}\}$

```
} else {
```

$\{\{\ -x = |x|\ \}\}$ **or equiv** $\{\{\ x < 0\ \}\}$

```
    y = -x;
```

$\{\{\ y = |x|\ \}\}$

```
}
```

$\{\{\ y = |x|\ \}\}$

Example Reasoning through Conditionals

- Try this working backward

$\{\{\ (x \geq 0 \text{ and } x \geq 0) \text{ or } (x < 0 \text{ and } x < 0)\ \}\}$

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

$\{\{\ x = |x|\ \}\}$ **or equiv** $\{\{\ x \geq 0\ \}\}$

```
    y = x;
```

$\{\{\ y = |x|\ \}\}$

```
} else {
```

$\{\{\ -x = |x|\ \}\}$ **or equiv** $\{\{\ x < 0\ \}\}$

```
    y = -x;
```

$\{\{\ y = |x|\ \}\}$

```
}
```

$\{\{\ y = |x|\ \}\}$

Example Reasoning through Conditionals

- Try this working backward

$\{\{\ (x \geq 0 \text{ and } x \geq 0) \text{ or } (x < 0 \text{ and } x < 0)\ \}\}$ or equiv $\{\{\ x \geq 0 \text{ or } x < 0\ \}\}$

or equiv $\{\{\ \}\}$

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

$\{\{\ x = |x|\ \}\}$ or equiv $\{\{\ x \geq 0\ \}\}$

```
  y = x;
```

$\{\{\ y = |x|\ \}\}$

```
} else {
```

$\{\{\ -x = |x|\ \}\}$ or equiv $\{\{\ x < 0\ \}\}$

```
  y = -x;
```

$\{\{\ y = |x|\ \}\}$

```
}
```

$\{\{\ y = |x|\ \}\}$

Mixing Forward and Backward

- Avoid “or” by mixing forward & backward:

```
{{ }}
if (n >= 0) {
  m = 2*n + 1;
} else {
  m = 0;
}
{{ m > n }}
```

Mixing Forward and Backward

- **Avoid “or” by mixing forward & backward:**

```
{{ }}
if (n >= 0) {
  {{ n ≥ 0 }}
  m = 2*n + 1;
} else {
  {{ n < 0 }}
  m = 0;
}
{{ m > n }}
```

Mixing Forward and Backward

- **Avoid "or" by mixing forward & backward:**

```
{{ }}
if (n >= 0) {
  {{ n ≥ 0 }}
  m = 2*n + 1;
  {{ m > n }}
} else {
  {{ n < 0 }}
  m = 0;
  {{ m > n }}
}
{{ m > n }}
```

Mixing Forward and Backward

- **Avoid “or” by mixing forward & backward:**

```
{{ }}
if (n >= 0) {
  {{ n ≥ 0 }}
  m = 2*n + 1;
  {{ m > n }}
} else {
  {{ n < 0 }}
  {{ 0 > n }}
  m = 0;
  {{ m > n }}
}
{{ m > n }}
```

immediate

Mixing Forward and Backward

- **Avoid "or" by mixing forward & backward:**

$\{\{\ \}\}$
```
if (n >= 0) {
```
$\{\{ n \geq 0 \}\}$

$\{\{ 2n + 1 > n \}\}$

```
  m = 2*n + 1;
```
$\{\{ m > n \}\}$
```
} else {
```
$\{\{ n < 0 \}\}$

$\{\{ 0 > n \}\}$
```
  m = 0;
```
$\{\{ m > n \}\}$
```
}
```
$\{\{ m > n \}\}$

check this:

$$
\begin{aligned}
2n+1 &= n + n + 1 & \\
&\geq n + 1 & \text{since } n \geq 0 \\
&> n & \text{since } 1 > 0
\end{aligned}
$$

Mixing Forward and Backward

- **What happens if we reason only one direction:**

$\{\{\ \}\}$

```
if (n >= 0) {
  m = 2*n + 1;
} else {
  m = 0;
}
```

$\{\{\ (n \geq 0 \text{ and } m = 2n + 1) \text{ or } (n < 0 \text{ and } m = 0)\ \}\}$

$\{\{\ m > n\ \}\}$

- **How do we prove this implication?**
  - continue by cases ($n \geq 0$ or $n < 0$)
  - these fall out automatically if we use forward <u>and</u> backward

Loops

Correctness of Loops

- Assignment and condition reasoning is mechanical

- Loop reasoning <u>cannot</u> be made mechanical
  - no way around this

    (311 alert: this follows from Rice's Theorem)

- Thankfully, one *extra* bit of information fixes this
  - need to provide a "loop invariant"
  - with the invariant, reasoning is again mechanical

Loop Invariants

- Loop invariant is true <u>every time</u> at the top of the loop

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

  - must be true when we get to the top the first time
  - must remain true each time execute `S` and loop back up

- Use "**Inv:**" to indicate a loop invariant

  otherwise, this only claims to be true the first time at the loop

Loop Invariants

- Loop invariant is true <u>every time</u> at the top of the loop

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

  - must be true 0 times through the loop (at top the first time)
  - if true n times through, must be true $n+1$ times through

- Why do these imply it is always true?
  - follows by structural induction (on $\mathbb{N}$)

Checking Correctness with Loop Invariants

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

1. I holds initially

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits

Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
    S
  {{ I }}
}
{{ Q }}
```

1. I holds initially

2. S preserves I

Splits correctness into three parts

1. I holds initially
2. S preserves I
3. Q holds when loop exits

Checking Correctness with Loop Invariants

```
{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
    S
  {{ I }}
}
{{ I and not cond }}
{{ Q }}
```

1. I holds initially

2. S preserves I

3. Q holds when loop exits

Splits correctness into three parts

1.	**I holds initially**	implication
2.	**S preserves I**	forward/back then implication
3.	**Q holds when loop exits**	implication

Checking Correctness with Loop Invariants

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

Formally, invariant split this into three Hoare triples:

1. {{ P }} {{ I }} — I **holds initially**
2. {{ I and cond }} **S** {{ I }} — **S preserves** I
3. {{ I and not cond }} {{ Q }} — Q **holds when loop exits**

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := sum-to(n) + (n+1)   for any n : $\mathbb{N}$

- This loop claims to calculate it as well

{{ }}
```
let i: number = 0;
let s: number = 0;
```
{{ **Inv**: s = sum-to(i) }}
```
while (i != n) {
  i = i + 1;
  s = s + i;
}
```
{{ s = sum-to(n) }}

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

$$
\begin{aligned}
&\textbf{func}\ \text{sum-to}(0) &&:= 0 \\
&\phantom{\textbf{func}}\ \text{sum-to}(n+1) &&:= (n+1) + \text{sum-to}(n) \qquad \text{for any } n : \mathbb{N}
\end{aligned}
$$

- This loop claims to calculate it as well

```
{{ }}
let i: number = 0;
let s: number = 0;
{{ i = 0 and s = 0 }}
{{ Inv: s = sum-to(i) }}
while (i != n) {
  ...
```

$$
\begin{aligned}
s &= 0 \\
&= \text{sum-to}(0) && \textbf{def of } \text{sum-to} \\
&= \text{sum-to}(i) && \textbf{since } i = 0
\end{aligned}
$$

Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + \ldots + n$

$$
\begin{aligned}
&\textbf{func}\ \text{sum-to}(0) && := 0 \\
&\quad\ \text{sum-to}(n+1) && := (n+1) + \text{sum-to}(n) \qquad \text{for any } n : \mathbb{N}
\end{aligned}
$$

- **This loop claims to calculate it as well**

```
{{ Inv: s = sum-to(i) }}
while (i != n) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1;
  s = s + i;
  {{ s = sum-to(i) }}
}
```

Example Loop Correctness

- **Recursive function to calculate** $1 + 2 + \ldots + n$

$$\textbf{func}\ \text{sum-to}(0) := 0$$
$$\text{sum-to}(n+1) := (n+1) + \text{sum-to}(n) \qquad \text{for any } n : \mathbb{N}$$

- **This loop claims to calculate it as well**

{{ **Inv**: s = sum-to(i) }}

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

{{ s = sum-to(i) and i ≠ n }}

{{ s + i + 1 = sum-to(i+1) }}

```
i = i + 1;
```

{{ s + i = sum-to(i) }}

```
s = s + i;
```

{{ s = sum-to(i) }}

```
}
```

$$\begin{aligned}\text{sum-to}(i+1) &= (i+1) + \text{sum-to}(i) && \textbf{def of } \text{sum-to} \\ &= (i+1) + s && \textbf{since } s = \text{sum-to}(i)\end{aligned}$$

Example Loop Correctness

- Recursive function to calculate $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := (n+1) + sum-to(n)   for any n : $\mathbb{N}$

- This loop claims to calculate it as well

{{ **Inv**: s = sum-to(i) }}

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

{{ s = sum-to(i) and i = n }}
{{ s = sum-to(n) }}

s = sum-to(i)
= sum-to(n)   **since** i = n

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	no mutation	“	libraries	calculation induction
2	local variable mutation	“	“	Floyd logic
3	array / object mutation	“	“	(later)