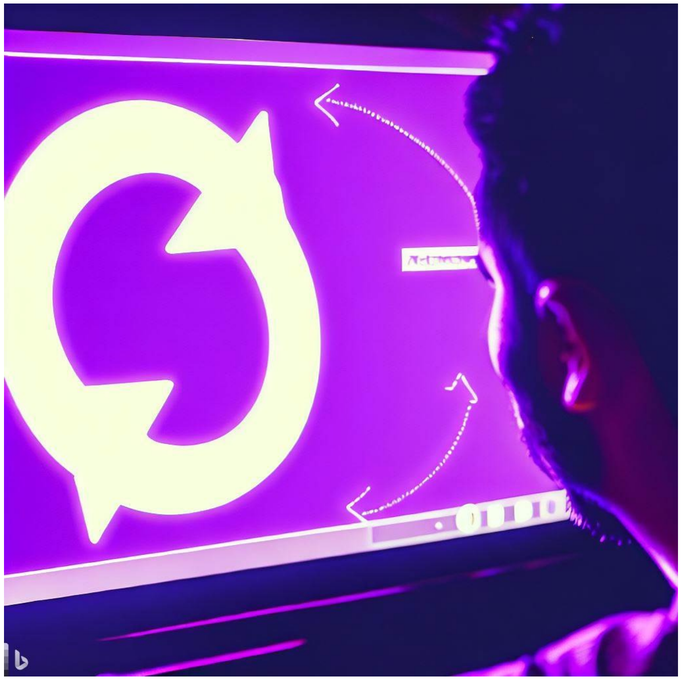


CSE 331

Loops in Floyd Logic

Kevin Zatloukal

Recall: Hoare Triples

- A **Hoare triple** has two assertions and some code

$$\{\{\ P\ \}\}$$
$$S$$
$$\{\{\ Q\ \}\}$$

  - P is the precondition, Q is the postcondition
  - S is the code

- Triple is "**valid**" if the code is correct:
  - S takes *any* state satisfying P into a state satisfying Q

    does not matter what the code does if P does not hold initially
  - otherwise, the triple is invalid

Recall: Correctness via Forward / Backward Reasoning

- Turn correctness into checking an implication

Forward

{{ P }}
S
{{ R }}
{{ Q }}

R implies Q ?

Backward

{{ P }}
{{ R }}
S
{{ Q }}

P implies R ?

- Check the implication by calculation (as before)

Recall: Forward and Backward Reasoning

- Imperative code made up of
  - assignments
  - conditionals
  - loops

- Anything can be rewritten with just these

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

Assignments

Example Forward Reasoning through Assignments

{{ $w > 0$ }}
```
x = 17;
```
{{ $w > 0$ and $x = 17$ }}
```
y = 42;
```
{{ $w > 0$ and $x = 17$ and $y = 42$ }}
```
z = w + x + y;
```
{{ $w > 0$ and $x = 17$ and $y = 42$ and $z = w + x + y$ }}

- **With no mutation, rule is** {{ P }} `x = y;` {{ **P and $x = y$** }}
- **That rule does not work if P refers to "x"**
  - need to invent a new name, x_0, to refer to x's old value
  - change the "x"s in P into "x_0"s since they mean the old value

Forward Reasoning through Assignments

- For assignments, general 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)

Correctness Example by Forward Reasoning

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

- Code is correct if this triple is valid…

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: number): number =? {
  {{ n ≥ 1 }}
  n = n + 3;
  {{ n₀ ≥ 1 and n = n₀ + 3 }}
  {{ n² ≥ 10 }}
  return n * n;
};
```

check this implication

$n^2 = (n_0 + 3)^2$ since $n = n_0 + 3$

$\geq 4^2$ since $n_0 \geq 1$

$= 16$

≥ 10

Forward Reasoning through Assignments

- For assignments, general 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 old value 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 we can write $x_0 = f(x)$, then we can simplify this to

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

```
x = … x …;
```

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

no need for, e.g., "and $x = x_0 + 1$"

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

Correctness Example by Forward Reasoning

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const 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

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

This is the preferred approach.
Avoid subscripts when possible.

Example Backward Reasoning with Assignments

{{ ________________ }}
```
x = 17;
```
{{ ________________ }}
```
y = 42;
```
{{ ________________ }}
```
z = w + x + y;
```
{{ $z < 0$ }}

- **What must be true before** $z = w + x + y$ **so** $z < 0$ **?**
  - want the weakest postcondition (most allowed states)

Example Backward Reasoning with Assignments

{{ ____________________ }}

```
x = 17;
```

{{ ____________________ }}

```
y = 42;
```

{{ $w + x + y < 0$ }}

```
z = w + x + y;
```

{{ $z < 0$ }}

- **What must be true before** `z = w + x + y` **so** $z < 0$ **?**
  - **must have** $w + x + y < 0$ **beforehand**

- **What must be true before** `y = 42` **for** $w + x + y < 0$ **?**

Example Backward Reasoning with Assignments

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

```
x = 17;
```

$\{\{\ w + x + 42 < 0\ \}\}$

```
y = 42;
```

$\{\{\ w + x + y < 0\ \}\}$

```
z = w + x + y;
```

$\{\{\ z < 0\ \}\}$

- **What must be true before** `y = 42` **for** $w + x + y < 0$ **?**
  - **must have** $w + x + 42 < 0$ **beforehand**
- **What must be true before** `x = 17` **for** $w + x + 42 < 0$ **?**

Example Backward Reasoning with Assignments

$\{\{\ w + 17 + 42 < 0\ \}\}$

```
x = 17;
```

$\{\{\ w + x + 42 < 0\ \}\}$

```
y = 42;
```

$\{\{\ w + x + y < 0\ \}\}$

```
z = w + x + y;
```

$\{\{\ z < 0\ \}\}$

- **What must be true before `x = 17` for $w + x + 42 < 0$?**
  - **must have $w + 59 < 0$ beforehand**
- **All we did was <u>substitute</u> right side for the left side**
  - **e.g., substitute "$w + x + y$" for "z" in "$z < 0$"**
  - **e.g., substitute "42" for "y" in "$w + x + y < 0$"**
  - **e.g., substitute "17" for "x" in "$w + x + 42 < 0$"**

Backward Reasoning through Assignments

- For assignments, backward reasoning is substitution

↑
```
{{ Q[x ↦ 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
  - no need for subscripts

Correctness Example by Forward Reasoning

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

- Code is correct if this triple is valid…

Correctness Example by Backward Reasoning

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

check this implication (between $\{\{ n \geq 1 \}\}$ and $\{\{ (n+3)^2 \geq 10 \}\}$)

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

Conditionals

Conditionals in Functional Programming

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: number, b: number): number => {
  if (a >= 0 && b >= 0) {
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

- Prior reasoning also included *conditionals*
  - what does that look like in Floyd logic?

Conditionals in Floyd Logic

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: number, b: number): number => {
  {{ }}
  if (a >= 0 && b >= 0) {
    {{ a ≥ 0 and b ≥ 0 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

- Conditionals introduce extra facts in forward reasoning
  - simple "and" case since nothing is mutated

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  return m;
}
```

- Code like this was impossible without mutation
  - cannot write to a "`const`" after its declaration

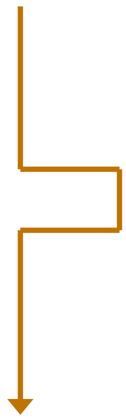
- How do we handle it now?

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  return m;
}
```

- Reason *separately* about each path to a `return`
  - handle each path the same as before
  - but now there can be multiple paths to one `return`

Conditionals in Floyd Logic



```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

- Check correctness path through “then” branch

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2*n + 1;
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2*n + 1;
    {{ n ≥ 0 and m = 2n + 1}}
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    {{ n ≥ 0 }}
    m = 2*n + 1;
    {{ n ≥ 0 and m = 2n + 1}}
  } else {
    m = 0;
  }
  {{ n ≥ 0 and m = 2n + 1 }}
  {{ m > n }}
  return m;
}
```

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

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  {{ n ≥ 0 and m = 2n + 1 }}
  {{ m > n }}
  return m;
}
```

- Note: no mutation, so we can do this in our head
  - read along the path, and collect all the facts

Conditionals in Floyd Logic

```
// Returns a number m with m > n
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  {{ n < 0 and m = 0 }}          m  = 0
  {{ m > n }}                       > n          since 0 > n
  return m;
}
```

- Check correctness path through “else” branch
  - note: no mutation, so we can do this in our head

Function Calls

Reasoning about Function Calls

```
// @requires P₂             -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
const f = (a: number, b: number): number => {..}
```

- **Forward reasoning rule is**

{{ P }}
```
  x = f(a, b);
```
{{ $P[x \mapsto x_0]$ and R }}

Must also check that P implies P_2

- **Backward reasoning rule is**

{{ Q_1 and P_2 }}
```
  x = f(a, b);
```
{{ Q_1 and Q_2 }}

Must also check that R implies Q_2

Q_2 is the part of postcondition using "x"

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 **every time** 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 **every time** 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 }}
```

- How do we check validity with a loop invariant?
  - intermediate assertion splits into *three* triples to check

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 : 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 $1 + 2 + \ldots + n$

func sum-to(0) := 0
sum-to(n+1) := sum-to(n) + (n+1)   for any n : 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) }}
```

Easy to get this wrong!
- might be initializing "i" wrong ($i = 1$?)
- might be exiting at the wrong time ($i \neq n-1$?)
- might have the assignments in wrong order
- ...

Fact that we need to check 3 implications is a strong indication that more bugs are possible.

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

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

sum-to(i)
= sum-to(0)   **since** i = 0
= 0   **def of** sum-to
= s

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

Example Loop Correctness

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

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) {
  {{ s = sum-to(i) and i ≠ n }}
  i = i + 1;
  {{ s = sum-to(i–1) and i–1 ≠ n }}
  s = s + i;
  {{ s = sum-to(i) }}
}
```

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 : ℕ

- **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 = sum-to(i–1) and i–1 ≠ n }}
  s = s + i;
  {{ s – i = sum-to(i–1) and i–1 ≠ n }}
  {{ s = sum-to(i) }}
}
```

s = i + sum-to(i-1) **since** s – i = sum-to(i-1)
= sum-to(i) **def of** sum-to

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 : 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) }}

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

Termination

- 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 does exit
- Termination follows from the running time analysis
  - 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
- Normal to also analyze the running time of our code, and we get termination already from that analysis