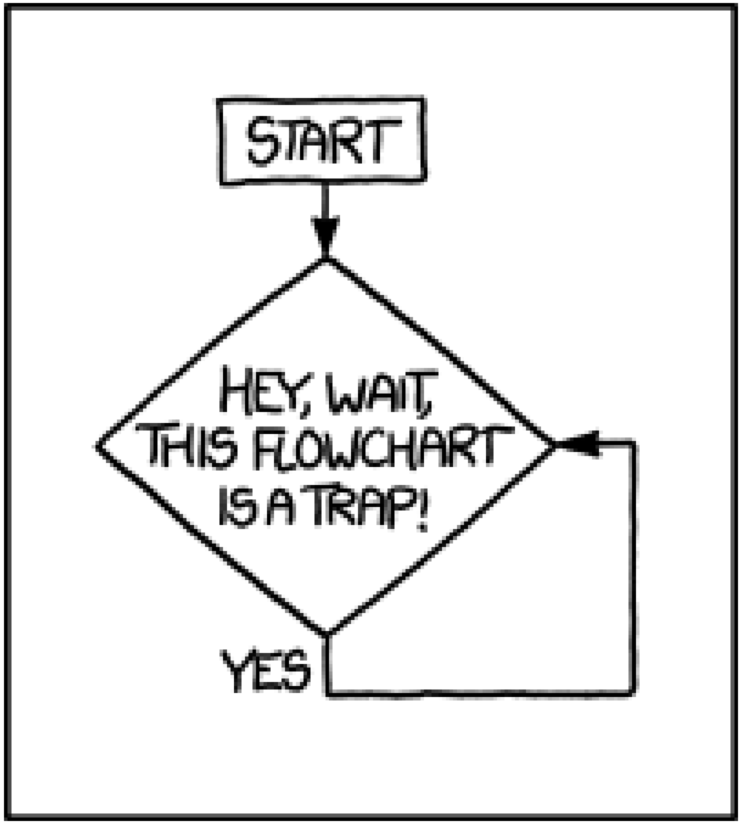
CSE 331
Spring 2025

Floyd Logic, Part I



xkcd #1195

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/02)

- HW5 is out!
  - note on **new** “require invariants” flag & option; we won’t require “//Inv:” on your submissions!

Closing the Loop on Feedback: Wins

- lectures:
  - interactivity through questions & peer activities
  - live code demos with code posted ahead of time
  - reasonably energetic/engaging/fun, remembers names
- section:
  - hands-on review of course materials
  - generally helpful for homework*
- infrastructure:
  - available lecture resources (slides, code, notes, videos)
  - support via office hours & EdStem

Closing the Loop on Feedback: Rough Spots

- relative consensus on:
  - micro & macro course pacing (especially Weeks 1 & 2)
  - steep learning curve into homeworks
  - particular disconnect on debugging & debugging log
  - coding & math conventions clarity
- discussed, but not with consensus:
  - homework itself
  - how helpful lecture is for homework
  - balance of problem walkthrough vs work time in section
  - organization of lecture content by day vs topic
  - non-ideal relationship with 12X & 311*

Closing the Loop on Feedback: What’s Next

- some concrete commitments:
  - matt is trying to slow down lecture, focus on depth
  - more consistent & deliberate use of section as practice
- course staff actively discussing & working on…
  - lecture-section-homework relationship (HW6 onwards)
  - clarity & precision of specs, coding & math conventions
- micro-changes
  - listing day “breakdowns” for lecture slides on website
  - better live-coding “hygiene” (e.g. visibility, contrast, …)
  - matt needs to get better at iPad :’)

Beyond the Loop

- note: matt is (likely) *not* teaching 331 next year – so these are all suggestions
- things for the near future (e.g. 25su, 25au)
  - revisiting Weeks 1 – 3 (debugging materials + pacing)
  - improving live coding (in lecture & section)
  - comfy-* tooling improvements
- things that are more up in the air
  - pre-class readings or videos
  - macro-level course shifts (e.g. breadth of topics)
  - relationship with 12X, 311, other courses

Misc Thoughts from Matt + Course Staff

- advice on note-taking (w.r.t. pace)
  - don't write down everything I say/type (we give it to you)
  - don't write nothing!
  - actively write down what doesn't make sense!
  - reminder: slides are *not* comprehensive review
- please continue to engage in interactive learning
  - interaction *is* the learning (and the emphasis)
  - helps everybody around you :))
- please reach out to us if you're past "productive struggle" – we want to support you!
- please keep on giving us feedback!

Reasoning So Far

- Code so far made up of three elements
  - straight-line code
  - conditionals
  - recursion

- All code without mutation looks like this

Recall: Finding Facts at a Return Statement

- Consider this code

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

find facts by reading along <u>path</u> from top to return statement

- Known facts include "$a \geq 0$", "$b \geq 0$", and "$L = \text{cons}(\ldots)$"

- Prove that postcondition holds: "$\text{sum}(L) \geq 0$"

Finding Facts at Returns, with Mutation

- Consider this code

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    a = a - 1n;
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

$a \geq 0$

$a \geq 0$? No!

- Facts no longer hold throughout the function

- When we state a fact, we have to say <u>where</u> it holds

Correctness Levels

Description	Testing	Tools	Reasoning
no mutation	coverage	type checking	calculation induction
local variable mutation	“”	“”	Floyd logic
array mutation	“”	“”	for-any facts
heap state mutation	“”	“”	rep invariants

Notation: Facts at a Point in Time

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    {{ a ≥ 0 }}
    a = a - 1n;
    {{ a ≥ -1 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
```

- When we state a fact, we have to say <u>where</u> it holds
- {{ .. }} notation indicates facts true at that point
  - cannot assume those are true anywhere else

Forwards & Backwards Reasoning, Informally

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    {{ a ≥ 0 }}
    a = a - 1n;
    {{ a ≥ -1 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
```

- There are <u>mechanical</u> tools for moving facts around
  - "forward reasoning" says how they change as we move down
  - "backward reasoning" says how they change as we move up

Reasoning and Programming

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    {{ a ≥ 0 }}
    a = a - 1n;
    {{ a ≥ -1 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
```

- Professionals are *absurdly* good at forward reasoning
  - "programmers are the Olympic athletes of forward reasoning"
  - you'll have an edge by learning backward reasoning too

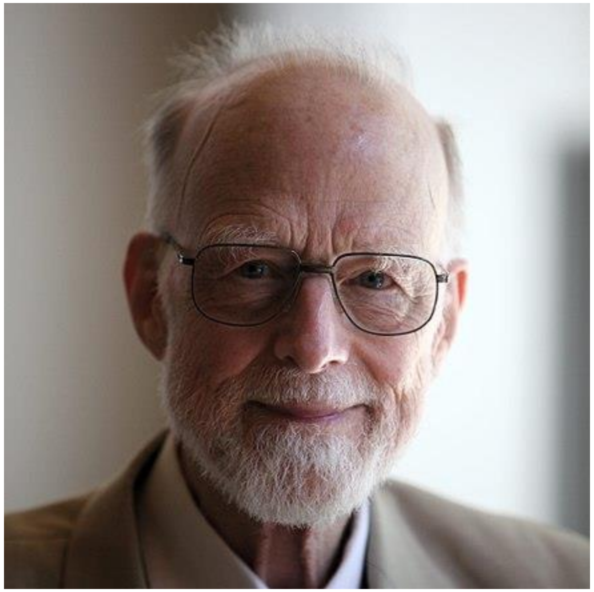
Floyd Logic

History of Floyd Logic

- Invented by Robert Floyd and Sir Anthony Hoare
  - Floyd won the Turing award in 1978
  - Hoare won the Turing award in 1980

Robert Floyd

picture from Wikipedia



Tony Hoare

picture from Wikipedia

Floyd Logic Terminology

- The **program state** is the values of the variables

- An **assertion** (in {{ .. }}) is a T/F claim about the state
  - an assertion "holds" if the claim is true
  - assertions are *math* not code

    (we do our reasoning in math)

- Most important assertions:
  - **precondition**: claim about the state when the function starts
  - **postcondition**: claim about the state when the function ends

Hoare Triples

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

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

S

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

  - 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

Correctness with Mutation Example (Setup)

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

- **Check that value returned,** $m = n^2$**, satisfies** $m \geq 10$

Correctness with Mutation Example (Triples)

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

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

Hoare Triples with No Code

- Code could be empty:

  {{ P }}
  {{ Q }}

- When is such a triple valid?
  - valid iff P implies Q
  - we already know how to check validity in this case: prove each fact in Q by calculation, using facts from P

Hoare Triples with No Code: Example

- **Code could be empty:**

$$\{\{\ a \geq 0,\ \ b \geq 0,\ \ L = \text{cons}(a, \text{cons}(b, \text{nil}))\ \}\}$$
$$\{\{\ \text{sum}(L) \geq 0\ \}\}$$

- **Check that P implies Q by calculation**

$$\begin{aligned}
\text{sum}(L) \quad &= \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) && \text{since } L = \ldots \\
&= a + \text{sum}(\text{cons}(b, \text{nil})) && \text{def of sum} \\
&= a + b + \text{sum}(\text{nil}) && \text{def of sum} \\
&= a + b && \text{def of sum} \\
&\geq 0 + b && \text{since } a \geq 0 \\
&\geq 0 + 0 && \text{since } b \geq 0 \\
&= 0
\end{aligned}$$

Hoare Triples with Multiple Lines of Code

- Code with multiple lines:

{{ P }}
`S`
`T`
{{ Q }}

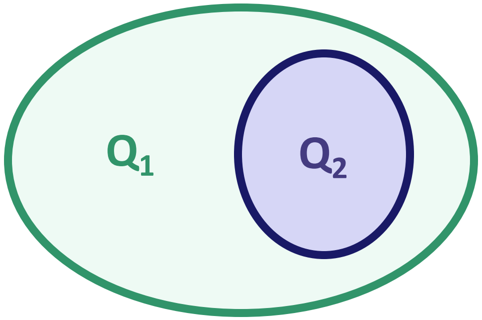
{{ P }}
`S`
{{ R }}
`T`
{{ Q }}

- Valid iff there exists an R making both triples valid
  - i.e., {{ P }} `S` {{ R }} is valid and {{ R }} `T` {{ Q }} is valid

- Will see next how to put these to good use…

Stronger Assertions vs Specifications

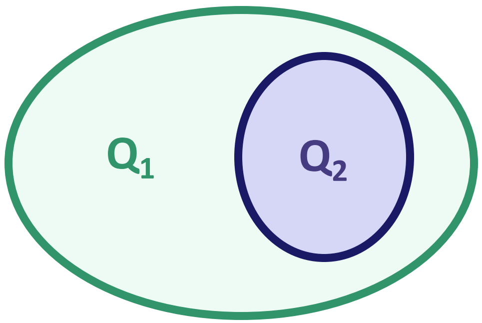
- **Assertion** is **stronger** iff it holds in a subset of states



- **Stronger** assertion <u>implies</u> the **weaker** one
  - stronger is a synonym for "implies"
  - weaker is a synonym for "is implied by"

Weakest & Strongest Assertions

- **Assertion** is **stronger** iff it holds in a subset of states



- **Weakest** possible assertion is “true” (all states)
  – an empty assertion (“”) also means “true”

- **Strongest** possible assertion is “false” (no states!)

Defining Forward & Backward Reasoning

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples

- **Forward** reasoning fills in postcondition

  {{ P }} S {{ ___ }}

  - gives *strongest* postcondition making the triple valid

- **Backward** reasoning fills in precondition

  {{ ___ }} S {{ Q }}

  - gives *weakest* precondition making the triple valid

Correctness via Forward Reasoning

- Apply forward reasoning

{{ P }}
S
{{ Q }}

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

1

2

  - first triple is always valid
  - only need to check second triple

    just requires proving an implication (since no code is present)

- If second triple is invalid, the code is **incorrect**
  - true because R is the strongest assertion possible here

Correctness via Backward Reasoning

- Apply backward reasoning

{{ P }}
S
{{ Q }}

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

1
2

  – second triple is always valid
  – only need to check first triple
    just requires proving an implication (since no code is present)

- If first triple is invalid, the code is **incorrect**
  – true because R is the weakest assertion possible here

Using Mechanical Reasoning Tools

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples
- Reduce correctness to proving implications
  - this was already true for functional code
  - will soon have the same for imperative code
- Implication will be false if the code is **incorrect**
  - reasoning can verify correct code
  - reasoning will never accept incorrect code

Correctness via Forward & Backward Reasoning

- Can use both types of reasoning on longer code

```
{{ P }}
   S          1
{{ R1 }}
{{ R2 }}           2
   T          3
{{ Q }}
```

  - first and third triples is always valid
  - only need to check second triple

    verify that R_1 implies R_2

Forward & Backward Reasoning

Forward and Backward Reasoning in Practice

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

Ex: Forward Reasoning with Assignments (1/6)

```
{{ w > 0 }}
 x = 17n;
{{ ______________ }}
 y = 42n;
{{ ______________ }}
 z = w + x + y;
{{ ______________ }}
```

- **What do we know is true after `x = 17` ?**
  - want the strongest postcondition (most precise)

Ex: Forward Reasoning with Assignments (2/6)

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

- **What do we know is true after `x = 17`?**
  - w **was not changed, so** $w > 0$ **is still true**
  - x **is now** 17

- **What do we know is true after `y = 42`?**

Ex: Forward Reasoning with Assignments (3/6)

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

- **What do we know is true after `y = 42`?**
  - w and x were not changed, so previous facts still true
  - y is now 42

- **What do we know is true after `z = w + x + y`?**

Ex: Forward Reasoning with Assignments (4/6)

{{ $w > 0$ }}

```
x = 17n;
```

{{ $w > 0$ and $x = 17$ }}

```
y = 42n;
```

{{ $w > 0$ and $x = 17$ and $y = 42$ }}

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

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

- **What do we know is true after `z = w + x + y`?**
  - w, x, and y were not changed, so previous facts still true
  - z is now $w + x + y$
- **Could also write $z = w + 59$ (since $x = 17$ and $y = 42$)**

Ex: Forward Reasoning with Assignments (5/6)

$\{\{\ w > 0\ \}\}$

```
x = 17n;
```

$\{\{\ w > 0 \text{ and } x = 17\ \}\}$

```
y = 42n;
```

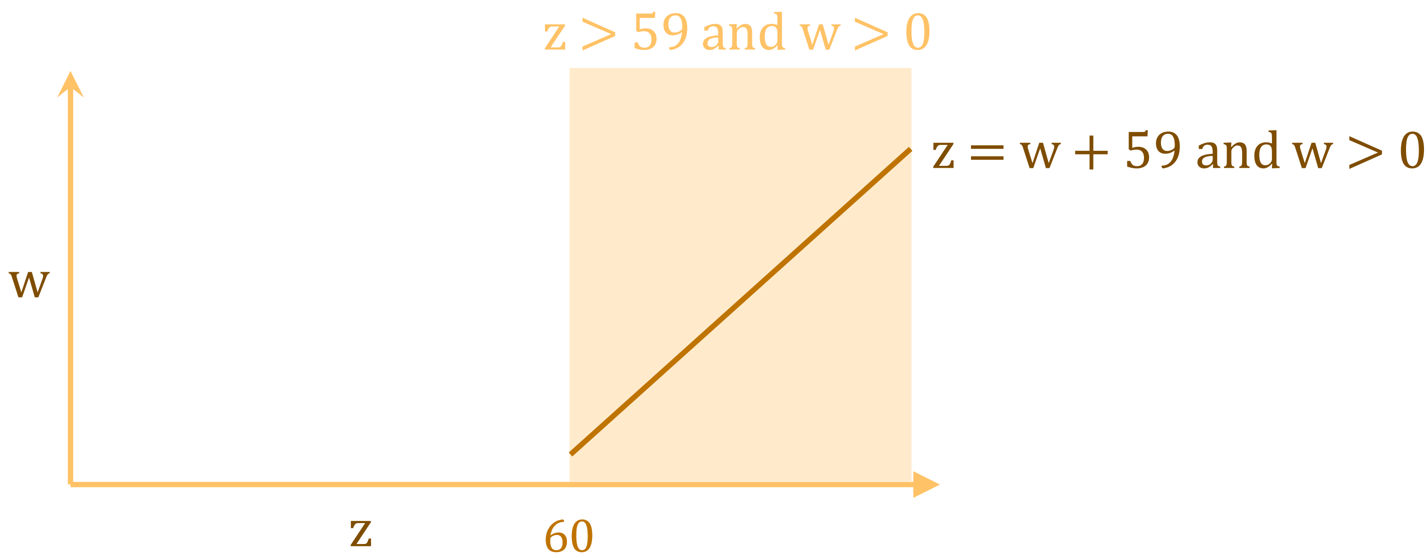
$\{\{\ w > 0 \text{ and } x = 17 \text{ and } y = 42\ \}\}$

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

$\{\{\ w > 0 \text{ and } x = 17 \text{ and } y = 42 \text{ and } z = w + x + y\ \}\}$

- **Could write $z = w + 59$, but <u>do not</u> write $z > 59$!**
  - **that is true since $w > 0$, but…**

Ex: Forward Reasoning with Assignments (6/6)



- **Could write** $z = w + 59$**, but** **<u>do not</u>** **write** $z > 59$ **!**
  - **that is true since** $w > 0$**, but…**

Picking the Strongest Postcondition

{{ $w > 0$ }}

```
x = 17n;
```

{{ $w > 0$ and $x = 17$ }}

```
y = 42n;
```

{{ $w > 0$ and $x = 17$ and $y = 42$ }}

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

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

- **Could write $z = w + 59$, but do not write $z > 59$!**
  - **that is true since $w > 0$, but…**
  - **that is not the strongest postcondition**

    correctness check could now fail even if the code is right

Forward Reasoning with Code (1/4)

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  return z;
};
```

- Let's check correctness using Floyd logic…

Forward Reasoning with Code (2/4)

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  {{ w > 0 }}
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  {{ z > 59 }}
  return z;
};
```

- Reason forward…

Forward Reasoning with Code (3/4)

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  {{ w > 0 }}
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  {{ w > 0 and x = 17 and y = 42 and z = w + x + y }}
  {{ z > 59 }}
  return z;
};
```

- **Check implication:**

$$
\begin{aligned}
z &= w + x + y && \\
&= w + 17 + y && \text{since } x = 17 \\
&= w + 59 && \text{since } y = 42 \\
&> 59 && \text{since } w > 0
\end{aligned}
$$

Forward Reasoning with Code (4/4)

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  return z;
};
```

find facts by reading along path
from top to return statement

- **How about if we use our old approach?**
- **Known facts:** $w > 0$, $x = 17$, $y = 42$, **and** $z = w + x + y$
- **Prove that postcondition holds:** $z > 59$

Finding Facts at Returns *is* Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  return z;
};
```

- We've been doing forward reasoning already!
  - forward reasoning is (only) "and" with *no mutation*

- Line-by-line facts are for "`let`" (not "`const`")

Forward Reasoning with Mutation (1/2)

- **Forward reasoning is trickier with mutation**
  - gets harder if we mutate a variable

```
 w = x + y;
```
$\{\{\ w = x + y\ \}\}$
```
 x = 4n;
```
$\{\{\ w = x + y \text{ and } x = 4\ \}\}$
```
 y = 3n;
```
$\{\{\ w = x + y \text{ and } x = 4 \text{ and } y = 3\ \}\}$

- **Final assertion is not necessarily true**
  - $w = x + y$ is true with their old values, not the new ones
  - changing the value of "x" can invalidate facts about x
    - facts refer to the old value, not the new value
  - avoid this by using different names for old and new values

Notation: Subscripts for Variables Across Time

- Can use subscripts to refer to values at different times

```
… (x: bigint) => …
  …
  x = …
  …
  x = …
  …
  x = …
  …
  x = …
  …
```

x_0 $\quad x = x_0 \quad$ "x" means current value

x_1 $\quad x = x_1$

x_2 $\quad x = x_2$

x_3 $\quad x = x_3$

x_4 $\quad x = x_4$

Forward Reasoning with Mutation (2/2)

- Rewrite existing facts to use names of earlier values
  - will use "x" and "y" to refer to current values
  - can use "x_0" and "y_0" (or other subscripts) for earlier values

$\{\{\ \mathrm{w} = \mathrm{x} + \mathrm{y}\ \}\}$

```
x = 4n;
```

$\{\{\ \mathrm{w} = \mathbf{x_0} + \mathrm{y} \text{ and } \mathbf{x} = 4\ \}\}$

```
y = 3n;
```

$\{\{\ \mathrm{w} = \mathrm{x}_0 + \mathbf{y_0} \text{ and } \mathrm{x} = 4 \text{ and } \mathbf{y} = 3\ \}\}$

- Final assertion is now accurate
  - w is equal to the sum of the initial values of x and y

Generalized Forward Reasoning Rule

- **For assignments, general forward reasoning rule is**

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

```
x = y;
```

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

  - **replace all "**x**"s in** P **and** y **with "**x_k**"s**

- **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.)

Example of "Shortcut" for Invertible Operations

- For assignments, general forward reasoning rule is

$\{\{ P \}\}$

```
x = y;
```

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

x_k is name of previous value

- If $x_0 = f(x)$, 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)

Revisiting Correctness with Forward Reasoning

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

Mutation in Straight-Line Code

- Alternative ways of writing this code:

```
n = n + 3n;
return n * n;
```

```
const n1 = n + 3n;
return n1 * n1;
```

- Mutation in *straight-line* code is unnecessary
  – can always use different names for each value

- Why would we prefer the former?
  – seems like it might save memory…
  – but it doesn't!

    most compilers will turn the left into the right on their own (SSA form)

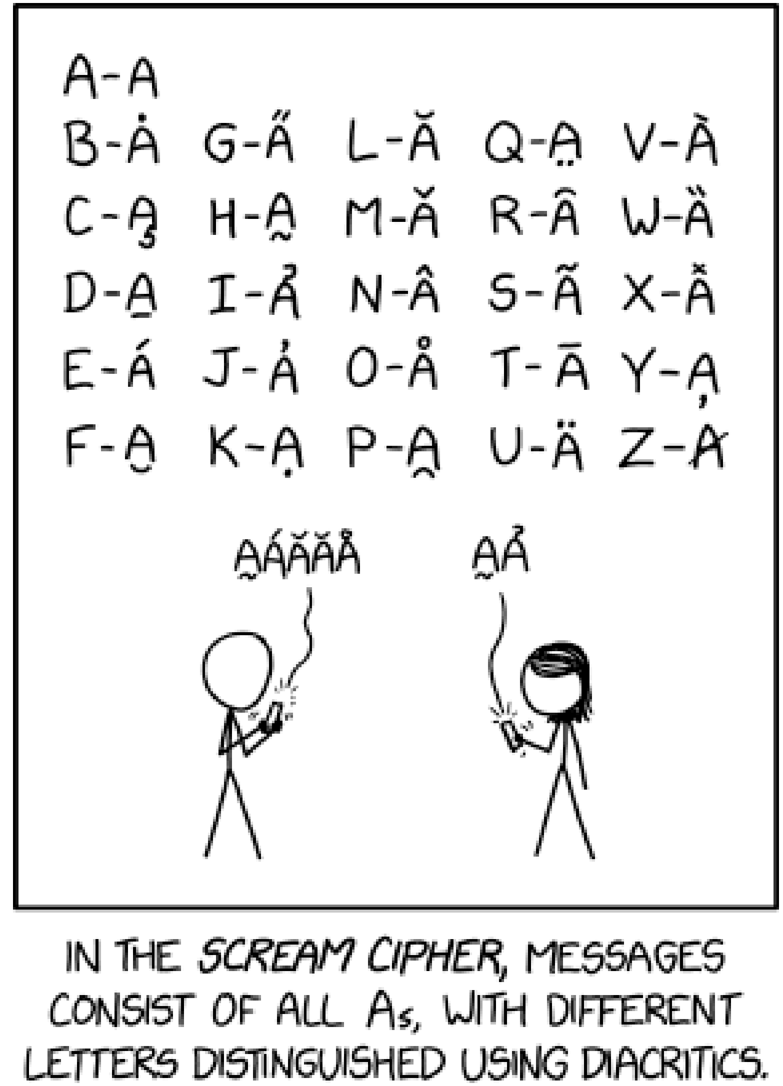
    it's better at saving memory than you are, so it does it itself

CSE 331
Spring 2025

Floyd Logic, Part II

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong



xkcd #3054

Floyd Logic Agenda

- Last Friday:
  – vocab: Hoare triple, “stronger” assertions
  – forward reasoning
- Today:
  – backwards reasoning
  – conditionals
  – function calls
- Wednesday:
  – loops & loop invariants (the “capstone”)

Recall: Forward Reasoning (adding facts)

- Each assignment just adds one new fact ("and")

{{ $w > 0$ }}

```
x = 4n;
```

{{ $w > 0$ and $x = 4$ }}

```
y = 3n;
```

{{ $w > 0$ and $x = 4$ and $y = 3$ }}

Recall: Forward Reasoning (with code)

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: bigint): bigint => {
  {{ w > 0 }}
  const x = 17n;
  const y = 42n;
  const z = w + x + y;
  {{ w > 0 and x = 17 and y = 42 and z = w + x + y }}
  {{ z > 59 }}
  return z;
};
```

- "Collecting the facts" was forward reasoning
  - only this simple because there was *no mutation*

Recall: Forward Reasoning (with mutation)

- **Forward reasoning is trickier with mutation**
  - gets harder if we mutate a variable

```
w = x + y;
```
$\{\{\ w = x + y\ \}\}$
```
x = 4n;
```
$\{\{\ w = x + y \text{ and } x = 4\ \}\}$
```
y = 3n;
```
$\{\{\ w = x + y \text{ and } x = 4 \text{ and } y = 3\ \}\}$

- **Final assertion is not necessarily true!**
  - fact $w = x + y$ was about the *old values* of x and y
  - still true if we clarify which value of x and y we mean

Recall: Forward Reasoning (with subscripts)

- Rewrite existing facts to use names of earlier values
  - will use "x" and "y" to refer to current values
  - can use "x_0" and "y_0" (or other subscripts) for earlier values

{{ $w = x + y$ }}

```
x = 4n;
```

{{ $w = \mathbf{x_0} + y$ and $\mathbf{x} = 4$ }}

```
y = 3n;
```

{{ $w = x_0 + \mathbf{y_0}$ and $x = 4$ and $\mathbf{y} = 3$ }}

- Final assertion is now accurate
  - w is equal to the sum of the initial values of x and y

Recall: General Forward Reasoning Rule

- For assignments, general forward reasoning rule is

{{ P }}

```
  x = y;
```

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

  – replace all "x"s in P and y with "x_k"s

- This process can be simplified in many cases…

Recall: Shorthand for Invertible Mutation

- For assignments, general forward reasoning rule is

{{ P }}
```
  x = x + 1;
```
{{ P and $x = x_0 + 1$ }}

- Can express the old value x_0 in terms of new value
  - if assignment is "$x = x_0 + 1$", then "$x_0 = x - 1$"
  - if assignment is "$x = 2x_0$", then "$x_0 = x/2$"

Recall: Full Forward Reasoning Example (on code)

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

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

check this implication

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

This is the preferred approach.
Avoid subscripts when possible.

Recall: Defining Forward & Backward Reasoning

- Forward / backward reasoning fill in assertions
  - mechanically create valid triples

- **Forward** reasoning fills in postcondition

  $$\{\{\, P \,\}\}\ \ S\ \ \{\{\, \_\_\_ \,\}\}$$

  - gives *strongest* postcondition making the triple valid

- **Backward** reasoning fills in precondition

  $$\{\{\, \_\_\_ \,\}\}\ \ S\ \ \{\{\, Q \,\}\}$$

  - gives *weakest* precondition making the triple valid

Backwards Reasoning by Example (1/4)

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

```
x = 17n;
```

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

```
y = 42n;
```

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

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

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

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

Backwards Reasoning by Example (2/4)

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

```
x = 17n;
```

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

```
y = 42n;
```

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

```
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** $w + x + y < 0$ **?**

Backwards Reasoning by Example (3/4)

$\{\{$ __________________ $\}\}$

```
x = 17n;
```

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

```
y = 42n;
```

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

```
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** $w + x + 42 < 0$ **?**

Backwards Reasoning by Example (4/4)

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

```
 x = 17n;
```

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

```
 y = 42n;
```

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

```
 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 "$w + x + 42 < 0$"

Generalized Backwards Reasoning Rule

- **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**
  - no need for subscripts

Backwards Reasoning with Code (1/2)

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

- Code is correct if this triple is valid…

Backwards Reasoning with Code (2/2)

```
/**
 * @param n an integer with n >= 1
 * @returns an integer m with m >= 10
 */
const f = (n: bigint): bigint => {
  {{ n ≥ 1 }}
  {{ (n + 3)² ≥ 10 }}
  n = n + 3n;
  {{ n² ≥ 10 }}
  return n * n;
};
```

check this implication

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

Recall: Forwards Reasoning with Code

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

check this implication

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

Forward reasoning produces known facts.
Backward reasoning produces facts to prove.

Conditionals

Conditionals in Floyd Logic (1/2)

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  if (a >= 0n && b >= 0n) {
    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 (2/2)

```
// Inputs a and b must be integers.
// Returns a non-negative integer.
const f = (a: bigint, b: bigint): bigint => {
  {{ }}
  if (a >= 0n && b >= 0n) {
    {{ 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" since nothing is mutated

Conditionals Worked Example: Setup

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

- Code like this was impossible without mutation
  - cannot write to a "`const`" after its declaration
- How do we handle it now?

Conditionals Worked Example: Cases

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  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 Worked Example: “Then” (1/5)

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

- Check correctness path through “then” branch

Conditionals Worked Example: “Then” (2/5)

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

Conditionals Worked Example: “Then” (3/5)

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

Conditionals Worked Example: “Then” (4/5)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    {{ n ≥ 0 }}
    m = 2n * n + 1n;
    {{ n ≥ 0 and m = 2n + 1}}
  } else {
    m = 0n;
  }
  {{ 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 Worked Example: “Then” (5/5)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  {{ 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 Worked Example: “Else”

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

Conditionals Worked Example: Join (1/2)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
    {{ n ≥ 0 and m = 2n + 1 }}
  } else {
    m = 0n;
    {{ n < 0 and m = 0 }}
  }
  {{ ________________________________ }}
  {{ m > n }}
  return m;
}
```

What do we know is true even if we don't know which branch was taken?

Conditionals Worked Example: Join (2/2)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    m = 0n;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and m = 0) }}
  {{ m > n }}
  return m;
}
```

- The "or" means we must reason by cases anyway!

Generalizing Conditional Floyd Logic (1/2)

```
{{ P }}
if (cond) {
    {{ P and cond }}
    S1
} else {
    {{ P and not cond }}
    S2
}
{{ R }}
{{ Q }}
```

- 2 possible paths to execute
- R is in the form of {{A or B}}
  - A being what we know if we had taken the `if` branch

Generalizing Conditional Floyd Logic (2/2)

```
{{ P }}
if (cond) {
    {{ P and cond }}
    S1
} else {
    {{ P and not cond }}
    S2
}
{{ R }}
{{ Q }}
```

- 2 possible paths to execute
- R is in the form of {{A or B}}
  - A being what we know if we had taken the `if` branch
  - B being what we know if we had taken the `else`

Conditionals and Early Returns (1/2)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    return 0n;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and ??) }}
  {{ m > n }}
  return m;
}
```

- What is the state after a "`return`"?

Conditionals and Early Returns (2/2)

```
// Returns an integer m with m > n
const g = (n: bigint): bigint => {
  {{ }}
  let m;
  if (n >= 0n) {
    m = 2n * n + 1n;
  } else {
    return 0n;
  }
  {{ (n ≥ 0 and m = 2n + 1) or (n < 0 and false) }}
  {{ m > n }}
  return m;
}
```

simplifies to just $n \geq 0$ and $m = 2n + 1$

- State after a "`return`" is false (no states)

Generalizing Early Returns and Forward Reasoning

- Latter rule for "`if` .. `return`" is useful:

```
{{ P }}
if (cond)
  return something;
{{ P and not cond }}
…
return something else;
```

- Only reach the line after the "`if`" if `cond` was false

- Only one path to each "`return`" statement
  - forward reason to the "`return`" inside the "`if`"
  - forward reason to the "`return`" after the "`if`"

Complex Conditionals Example: Paths? (1/2)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  }
  {{ ________________________________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

How many paths can the code take?

Complex Conditionals Example: Paths? (2/2)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  } else {
    // do nothing
  }
  {{ __________ or __________ or __________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

3 paths! `else` branch is not written out, but it's there implicitly

After the conditional, there are 3 sets of facts that could be true

Complex Conditionals Example: “Then” (1/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    {{ ______________ }}
    m = m * -1n;
    {{ ______________ }}
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing
  {{ ___________ or ___________ or ___________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: "Then" (2/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    {{ m = x and x < 0 }}
    m = m * -1n;
    {{ ______________ }}
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing
  {{ ___________ or ___________ or ___________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: "Then" (3/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    {{ m = x and x < 0 }}
    m = m * -1n;
    {{ m = - x and x < 0 }}
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing
  {{ (m = - x and x < 0) or ____________ or ____________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: “Else If” (1/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    {{ ______________ }}
    return 1n;
  } // else: do nothing
  {{ (m = - x and x < 0) or ___________ or ___________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: “Else If” (2/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    {{ x = 0 and m = x }}
    return 1n;
  } // else: do nothing
  {{ (m = - x and x < 0) or ____________ or ____________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: “Else If” (3/3)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    {{ x = 0 and m = x }}
    return 1n;
  } else {
    //
  }
  {{ (m = - x and x < 0) or (x = 0 and m = x  and false) or ________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Must prove that post condition holds here

false: no states can reach beyond return

Complex Conditionals Example: Implicit Else (1/2)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing

  {{ (m = - x and x < 0) or ____________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

What do we know in implicit else case?

When *neither* of the then cases were entered

Complex Conditionals Example: Implicit Else (2/2)

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing

  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Complex Conditionals Example: Backwards Step

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing
  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  {{ ________________ }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

Can reason backward and forward and meet in the middle

Complex Conditionals Example: Prove Implication

```
// Returns an integer m, with m > 0
const h = (x: bigint): bigint => {
  {{ }}
  let m = x;
  if (x < 0n) {
    m = m * -1n;
  } else if (x === 0n) {
    return 1n;
  } // else: do nothing
  {{ (m = - x and x < 0) or (x > 0 and m = x) }}
  {{ m + 1 > 0 }}
  m = m + 1n;
  {{ m > 0 }}
  return m;
}
```

check this implication

Does the set of facts we know at this point in the program satisfy what must be true to reach our post condition

Aside: Proving "Or" Implications by Cases

- Prove by cases

  {{ (m = - x and x < 0) or (x > 0 and m = x) }}

  {{ m + 1 > 0 }}

  Case 1: m = - x and x < 0

  m + 1 = -x + 1 since m = -x
  > 1 since x < 0
  > 0

  Case 2: x > 0 and m = x

  m + 1 = x + 1 since m = x
  > 1 since x > 0
  > 0

- Already proved for the branch with the return, so proved the postcondition holds, in general

Function Calls

Reasoning about Function Calls

- Causes no extra difficulties if…
  1. defined for all inputs
  2. no inputs are mutated (much, much harder with mutation)

- Forward reasoning rule is

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

```
x = Math.sin(a);
```

$\{\{\ P[x \mapsto x_0] \text{ and } x = \sin(a)\ \}\}$

- Backward reasoning rule is

$\{\{\ Q[x \mapsto \sin(a)]\ \}\}$

```
x = Math.sin(a);
```

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

Pause & Think: “Debugging Logs” (1/2)

```
// Returns float (~ real number) a * b
// (note to reader: JS Math.log2 ~ Java Math.log2)
const mult = (a: number, b: number): number => {
    {{ }}
    a = Math.log2(a);

    b = Math.log2(b);

    const sum = a + b;

    const prod = Math.pow(2, sum);

    {{ a · b }}
    return prod;
}
```

Pause & Think: “Debugging Logs” (2/2)

```
// Returns float (~ real number) a * b
// (note to reader: JS Math.log2 ~ Java Math.log2)
const mult = (a: number, b: number): number => {
    {{ }}
    a = Math.log2(a);
    {{ a = log2 a0 }}
    b = Math.log2(b);
    {{ a = log2 a0 and b = log2 b0}}
    const sum = a + b;
    {{ a = log2 a0 and b = log2 b0 and sum = a + b}}
    const prod = Math.pow(2, sum);
    {{ a = log2 a0 and b = log2 b0 and sum = a + b and prod = 2^sum }}
    {{ a0 · b0 }}
    return prod;
}
```

This code is wrong! Why? Hint: it’s not the log rules, but does involve log…

$$\log_2 \boldsymbol{a_0} + \log_2 \boldsymbol{b_0} = \log_2 \boldsymbol{a_0} \cdot \boldsymbol{b_0} \qquad \text{(log rules)}$$

$$2^{\log_2 \boldsymbol{a_0} \cdot \boldsymbol{b_0}} = \boldsymbol{a_0} \cdot \boldsymbol{b_0} \qquad \text{(def of log)}$$

Reasoning about Function Calls: Preconditions

- **Preconditions must be checked**
  - not valid to call the function on disallowed inputs
- **Forward reasoning rule is**

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

```
x = Math.log(a);
```

$\{\{\ P[x \mapsto x_0] \text{ and } x = \log(a)\ \}\}$

Must also check $a > 0$

- **Backward reasoning rule is**

$\{\{\ Q[x \mapsto \log(a)] \text{ and } \mathbf{a > 0}\ \}\}$

```
x = Math.log(a);
```

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

Function Calls with Imperative Specs

- Applies to functions we define with imperative specs

```
// @param n a non-negative integer
// @returns square(n), where
//        square(0) := 0
//      square(n+1) := square(n) + 2n + 1
const square = (n: bigint): bigint => {..}
```

- Reasoning is the same. E.g., forward rule is

{{ P }}
```
  x = square(n);
```
{{ $P[x \mapsto x_0]$ and $x = \text{square}(n)$ }}

Must also check that `n` is non-negative

Function Call with Imperative Spec: Forward (1/5)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
    {{ x ≥ 0 }}
    let r = x + 2;
    {{ ____________________ }}
    r = Math.sqrt(r);
    {{ ____________________ }}
    r = r + 1;
    {{ ____________________ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

check this implication

Function Call with Imperative Spec: Forward (2/5)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

```
    let r = x + 2;
```

$\{\{\ x \geq 0 \text{ and } r = x + 2\ \}\}$

```
    r = Math.sqrt(r);
```

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

```
    r = r + 1;
```

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

$\{\{\ r = \sqrt{x+2} + 1\ \}\}$

```
    return r;
}
```

x: **"A number greater than or equal to 0."**

Returns $\sqrt{x}$, a unique y $\geq$ 0, $y^2 = x$

$$\begin{aligned} r &= x + 2 \\ &\geq 0 + 2 && \text{since } x \geq 0 \\ &= 2 \end{aligned}$$

Function Call with Imperative Spec: Forward (3/5)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

```
    let r = x + 2;
```

$\{\{\ x \geq 0 \text{ and } r = x + 2\ \}\}$

```
    r = Math.sqrt(r);
```

$\{\{\ x \geq 0 \text{ and } r = \sqrt{x + 2}\ \}\}$

```
    r = r + 1;
```

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

$\{\{\ r = \sqrt{x + 2} + 1\ \}\}$

```
    return r;
}
```

`x:` "A number greater than or equal to 0."

Returns $\sqrt{x}$, a unique $y \geq 0$, $y^2 = x$

$$\begin{aligned} r &= x + 2 \\ &\geq 0 + 2 && \text{since } x \geq 0 \\ &= 2 \end{aligned}$$

Function Call with Imperative Spec: Forward (4/5)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
    {{ x ≥ 0 }}
    let r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{x ≥ 0 and r – 1 = √(x + 2) }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

check this implication

Function Call with Imperative Spec: Forward (5/5)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

```
    let r = x + 2;
```

$\{\{\, x \geq 0 \text{ and } r = x + 2 \,\}\}$

```
    r = Math.sqrt(r);
```

$\{\{\, x \geq 0 \text{ and } r = \sqrt{x + 2} \,\}\}$

```
    r = r + 1;
```

$\{\{x \geq 0 \text{ and } r = \sqrt{x + 2} + 1\}\}$

$\{\{\, r = \sqrt{x + 2} + 1 \,\}\}$

holds!

```
    return r;
}
```

Function Call w/ Imperative Spec: Backward (1/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
    {{ x ≥ 0 }}
    {{ ____________________ }}
    let r = x + 2;
    {{ ____________________ }}
    r = Math.sqrt(r);
    {{ ____________________ }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Function Call w/ Imperative Spec: Backward (2/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
    {{ x ≥ 0 }}
    {{ _______________ }}
    let r = x + 2;
    {{ _______________ }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Function Call w/ Imperative Spec: Backward (3/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
    {{ x ≥ 0 }}
    {{ ____________________ }}
    let r = x + 2;
    {{ ____________________ }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

x: "A number greater than or equal to 0."

Returns $\sqrt{x}$, a unique y ≥ 0, $y^2 = x$

Function Call w/ Imperative Spec: Backward (4/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

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

```
    let r = x + 2;
```

$\{\{\ \sqrt{r} + 1 = \sqrt{x+2}\ + 1$ **and** $r \geq 0\ \}\}$

```
    r = Math.sqrt(r);
```

$\{\{\ r + 1 = \sqrt{x+2}\ + 1\ \}\}$

```
    r = r + 1;
```

$\{\{\ r = \sqrt{x+2} + 1\ \}\}$

```
    return r;
}
```

Function Call w/ Imperative Spec: Backward (5/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

$\{\{\ \sqrt{x+2} + 1 = \sqrt{x+2} + 1 \text{ and } x + 2 \geq 0\ \}\}$

```
    let r = x + 2;
```

$\{\{\ \sqrt{r} + 1 = \sqrt{x+2} + 1 \text{ and } r \geq 0\ \}\}$

```
    r = Math.sqrt(r);
```

$\{\{\ r + 1 = \sqrt{x+2} + 1\ \}\}$

```
    r = r + 1;
```

$\{\{\ r = \sqrt{x+2} + 1\ \}\}$

```
    return r;
}
```

Function Call w/ Imperative Spec: Backward (6/6)

```
// Evaluates polynomial with given input
// @param x a non-negative integer
// @returns sqrt(x + 2) + 1
const f = (x: number): number => {
```

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

$\{\{\ \sqrt{x+2} + 1 = \sqrt{x+2} + 1 \text{ and } x + 2 \geq 0\ \}\}$

```
    let r = x + 2;
```

$\{\{\ \sqrt{r} + 1 = \sqrt{x+2} + 1 \text{ and } r \geq 0\ \}\}$

```
    r = Math.sqrt(r);
```

$\{\{\ r + 1 = \sqrt{x+2} + 1\ \}\}$

```
    r = r + 1;
```

$\{\{\ r = \sqrt{x+2} + 1\ \}\}$

```
    return r;
}
```

$\{\{$ **true** and $x + 2 \geq 0\ \}\}$

$\{\{x + 2 \geq 0\ \}\}$

$x \geq 0$ **implies** $x + 2 \geq 0$

Function Calls with Declarative Specs

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

- **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"

CSE 331
Spring 2025

Floyd Logic,
Part III

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

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 (1/2)

- 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 (2/2)

- 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 $\mathbb{N}$)

Loop Invariants as Three Distinct Triples (1/5)

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

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

Loop Invariants as Three Distinct Triples (2/5)

{{ 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

Loop Invariants as Three Distinct Triples (3/5)

```
{{ 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

Loop Invariants as Three Distinct Triples (4/5)

{{ 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

Loop Invariants as Three Distinct Triples (5/5)

```
{{ 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**

Loop Invariant Example: Square (1/8)

- This loop claims to calculate n^2

```
{{ }}
let j: bigint = 0n;
let s: bigint = 0n;
{{ Inv: s = j² }}
while (j !== n) {
  j = j + 1n;
  s = s + j + j - 1;
}
{{ s = n² }}
```

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

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

Loop Invariant Example: Square (2/8)

- This loop claims to calculate n^2

```
{{ }}
let j: bigint = 0n;
let s: bigint = 0n;
{{ Inv: s = j^2 }}
while (j !== n) {
  j = j + 1n;
  s = s + j + j - 1;
}
{{ s = n^2 }}
```

Loop Idea

- move j from 0 to n
- keep track of j^2 in s

j	s
0	0
1	1
2	4
3	9
4	16
…	…

Loop Invariant formalizes the **Loop Idea**

Loop Invariant Example: Square (3/8)

- **This loop claims to calculate** n^2

```
{{ }}
let j: bigint = 0n;
let s: bigint = 0n;
{{ j = 0 and s = 0 }}
{{ Inv: s = j^2 }}
while (j !== n) {
  j = j + 1n;
  s = s + j + j - 1;
}
{{ s = n^2 }}
```

$$\begin{aligned} s &= 0 \\ &= 0^2 && \text{since } j = 0 \\ &= j^2 \end{aligned}$$

Loop Invariant Example: Square (4/8)

- **This loop claims to calculate** n^2

{{ **Inv**: $s = j^2$ }}

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

{{ $s = j^2$ and $j = n$ }}

{{ $s = n^2$ }}

$s = j^2$

$\quad = n^2$ **since** $j = n$

Loop Invariant Example: Square (5/8)

- **This loop claims to calculate** n^2

{{ **Inv**: $s = j^2$ }}

```
while (j !== n) {
  {{ s = j² and j ≠ n }}
  j = j + 1n;
  s = s + j + j - 1;
  {{ s = j² }}
}
```

{{ $s = n^2$ }}

Loop Invariant Example: Square (6/8)

- This loop claims to calculate n^2

```
{{ Inv: s = j² }}
while (j !== n) {
  {{ s = j² and j ≠ n }}
  j = j + 1n;
  {{ s = (j – 1)² and j – 1 ≠ n }}
  s = s + j + j - 1;
  {{ s = j² }}
}
{{ s = n² }}
```

$j = j_0 + 1$ means $j_0 = j - 1$

Loop Invariant Example: Square (7/8)

- **This loop claims to calculate** n^2

$\{\{$ **Inv**: $s = j^2$ $\}\}$

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

$\{\{\ s = j^2 \text{ and } j \neq n\ \}\}$

```
  j = j + 1n;
```

$\{\{\ s = (j - 1)^2 \text{ and } j - 1 \neq n\ \}\}$

```
  s = s + j + j - 1;
```

$s = s_0 + 2j - 1$ **means** $s_0 = s - 2j + 1$

$\{\{\ s - 2j + 1 = (j - 1)^2 \text{ and } j - 1 \neq n\ \}\}$

$\{\{\ s = j^2\ \}\}$

```
}
```

$\{\{\ s = n^2\ \}\}$

Loop Invariant Example: Square (8/8)

- **This loop claims to calculate n^2**

$\{\{$ **Inv**: $s = j^2$ $\}\}$

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

$\{\{\ s = j^2 \text{ and } j \neq n\ \}\}$

```
    j = j + 1n;
```

$\{\{\ s = (j - 1)^2 \text{ and } j - 1 \neq n\ \}\}$

```
    s = s + j + j - 1;
```

$\{\{\ s - 2j + 1 = (j - 1)^2 \text{ and } j - 1 \neq n\ \}\}$

$\{\{\ s = j^2\ \}\}$

```
}
```

$\{\{\ s = n^2\ \}\}$

$$
\begin{aligned}
s &= 2j - 1 + (j - 1)^2 \\
&= 2j - 1 + j^2 - 2j + 1 \\
&= j^2
\end{aligned}
$$

since $s - 2j + 1 = (j - 1)^2$

Loop Invariant Example: Sum of List (1/8)

- Recursive function to calculate sum of list

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- This loop claims to calculate it as well:

$\{\{\ L = L_0\ \}\}$

```
let s: bigint = 0n;
```

$\{\{\ \textbf{Inv}: \text{sum}(L_0) = s + \text{sum}(L)\ \}\}$

```
while (L.kind !== "nil") {
  s = s + L.hd;
  L = L.tl;
}
```

$\{\{\ s = \text{sum}(L_0)\ \}\}$

Loop Idea

- move through L front-to-back
- keep sum of *prior* part in s

Loop Invariant Example: Sum of List (2/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Check that the invariant holds initially**

$\{\{\ L = L_0\ \}\}$

```
let s: bigint = 0n;
```

$\{\{\ L = L_0 \textbf{ and } s = 0\ \}\}$

$\{\{\ \textbf{Inv}\text{: } \mathrm{sum}(L_0) = s + \mathrm{sum}(L)\ \}\}$

```
while (L.kind !== "nil") {
  ...
```

$$\begin{aligned} &\mathrm{sum}(L_0) \\ &\quad = \mathrm{sum}(L) && \textbf{since } L = L_0 \\ &\quad = 0 + \mathrm{sum}(L) \\ &\quad = s + \mathrm{sum}(L) && \textbf{since } s = 0 \end{aligned}$$

Loop Invariant Example: Sum of List (3/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Check that the postcondition holds at loop exit**

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L.kind !== "nil") {
  s = s + L.hd;
  L = L.tl;
}
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ and $L = \mathrm{nil}$ }}

{{ $s = \mathrm{sum}(L_0)$ }}

$$\begin{aligned} &\mathrm{sum}(L_0) \\ &= s + \mathrm{sum}(L) \\ &= s + \mathrm{sum}(\mathrm{nil}) && \textbf{since } L = \mathrm{nil} \\ &= s && \textbf{def of } \mathrm{sum} \end{aligned}$$

Loop Invariant Example: Sum of List (4/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ and $L \neq \mathrm{nil}$ }}

```
  s = s + L.hd;
  L = L.tl;
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
}
```

$L \neq \mathrm{nil}$ **means** $L = L.\mathrm{hd} :: L.\mathrm{tl}$

Loop Invariant Example: Sum of List (5/8)

- **Recursive function to calculate sum of list**

$$
\begin{aligned}
\mathrm{sum}(\mathrm{nil}) &:= 0 \\
\mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L)
\end{aligned}
$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ and $L = L.hd :: L.tl$ }}

```
  s = s + L.hd;
  L = L.tl;
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
}
```

Loop Invariant Example: Sum of List (6/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ and $L = L.\mathrm{hd} :: L.\mathrm{tl}$ }}

```
  s = s + L.hd;
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L.\mathrm{tl})$ }}

```
  L = L.tl;
```

{{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
}
```

Loop Invariant Example: Sum of List (7/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ and $L = L.\text{hd} :: L.\text{tl}$ }}

{{ $\text{sum}(L_0) = s + L.\text{hd} + \text{sum}(L.\text{tl})$ }}

```
  s = s + L.hd;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L.\text{tl})$ }}

```
  L = L.tl;
```

{{ $\text{sum}(L_0) = s + \text{sum}(L)$ }}

```
}
```

Loop Invariant Example: Sum of List (8/8)

- **Recursive function to calculate sum of list**

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &:= 0 \\ \mathrm{sum}(x :: L) &:= x + \mathrm{sum}(L) \end{aligned}$$

- **Check that the loop body preserves the invariant**

{{ **Inv**: $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
while (L.kind !== "nil") {
```

  {{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ and $L = L.hd :: L.tl$ }}

  {{ $\mathrm{sum}(L_0) = s + L.hd + \mathrm{sum}(L.tl)$ }}

```
  s = s + L.hd;
```

  {{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L.tl)$ }}

```
  L = L.tl;
```

  {{ $\mathrm{sum}(L_0) = s + \mathrm{sum}(L)$ }}

```
}
```

$$\begin{aligned} &\mathrm{sum}(L_0) \\ &\quad = s + \mathrm{sum}(L) \\ &\quad = s + \mathrm{sum}(L.hd :: L.tl) && \textbf{since } L = L.hd :: L.tl \\ &\quad = s + L.hd + \mathrm{sum}(L.tl) && \textbf{def of } \mathrm{sum} \end{aligned}$$

Loop Invariant Example: List Contains (1/7)

- **Recursive function to check if y appears in list L**

$$
\begin{aligned}
&\mathrm{contains}(y, \mathrm{nil}) &&:= \mathrm{false} && \\
&\mathrm{contains}(y, x :: L) &&:= \mathrm{true} && \text{if } x = y \\
&\mathrm{contains}(y, x :: L) &&:= \mathrm{contains}(y, L) && \text{if } x \neq y
\end{aligned}
$$

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

{{ **Inv**: $\mathrm{contains}(y, L_0) = \mathrm{contains}(y, L)$ }}

```
while (L.kind !== "nil") {
  if (L.hd === y)
    return true;
  L = L.tl;
}
return false;
```

Loop Idea
- move through L front-to-back
- answer remains the same as on the original list L_0
- can only do that if y is *not* found

Loop Invariant Example: List Contains (2/7)

- **Check that the invariant holds initially**

$\{\{\ L_0 = L\ \}\}$

$\{\{\ \textbf{Inv}: \text{contains}(y, L_0) = \text{contains}(y, L)\ \}\}$

```
while (L.kind !== "nil") {
  if (L.hd === y)
    return true;
  L = L.tl;
}
return false;
```

$\text{contains}(y, L_0)$

$\quad = \text{contains}(y, L)$ **since** $L_0 = L$

$\text{contains}(y, \text{nil})$	$:= \text{false}$	
$\text{contains}(y, x :: L)$	$:= \text{true}$	**if** $x = y$
$\text{contains}(y, x :: L)$	$:= \text{contains}(y, L)$	**if** $x \neq y$

Loop Invariant Example: List Contains (3/7)

- **Check that the invariant implies the postcondition**

{{ **Inv**: contains(y, L_0) = contains(y, L) }}

```
while (L.kind !== "nil") {
  if (L.hd === y)
    return true;
  L = L.tl;
}
```

{{ contains(y, L_0) = contains(y, L) and L = nil }}

{{ contains(y, L_0) = false }}

```
return false;
```

contains(y, L_0)
 = contains(y, L)
 = contains(y, nil) **since** L = nil
 = false **def of** contains

contains(y, nil)	:= false	
contains(y, x :: L)	:= true	**if** x = y
contains(y, x :: L)	:= contains(y, L)	**if** x ≠ y

Loop Invariant Example: List Contains (4/7)

- **Check that the body preserves the invariant**

{{ **Inv**: $\text{contains}(y, L_0) = \text{contains}(y, L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\text{contains}(y, L_0) = \text{contains}(y, L)$ and $L \neq \text{nil}$ }}

```
  if (L.hd === y)
    return true;
  L = L.tl;
```

{{ $\text{contains}(y, L_0) = \text{contains}(y, L)$ }}

```
}
return false;
```

$L \neq \text{nil}$ **means** $L = L.hd :: L.tl$

contains(y, nil)	:= false	
contains(y, x :: L)	:= true	**if** $x = y$
contains(y, x :: L)	:= contains(y, L)	**if** $x \neq y$

Loop Invariant Example: List Contains (5/7)

- **Check that the body preserves the invariant**

{{ **Inv**: contains(y, L_0) = contains(y, L) }}

```
while (L.kind !== "nil") {
```

{{ contains(y, L_0) = contains(y, L) and L = L.hd :: L.tl }}

```
  if (L.hd === y)
```

{{ contains(y, L_0) = contains(y, L) and L = L.hd :: L.tl and L.hd = y }}

{{ contains(y, L_0) = true }}

```
    return true;
  L = L.tl;
```

{{ contains(y, L_0) = contains(y, L) }}

```
}
return false;
```

contains(y, L_0)
= contains(y, L)
= contains(y, L.hd :: L.tl) **since** L = L.hd :: L.tl
= true **since** y = L.hd

contains(y, nil)	:= false	
contains(y, x :: L)	:= true	**if** x = y
contains(y, x :: L)	:= contains(y, L)	**if** x ≠ y

Loop Invariant Example: List Contains (6/7)

- **Check that the body preserves the invariant**

{{ **Inv**: $\text{contains}(y, L_0) = \text{contains}(y, L)$ }}

```
while (L.kind !== "nil") {
```

{{ $\text{contains}(y, L_0) = \text{contains}(y, L)$ and $L = L.hd :: L.tl$ }}

```
  if (L.hd === y)
```

{{ $\text{contains}(y, L_0) = \text{true}$ }}

```
    return true;
```

{{ $\text{contains}(y, L_0) = \text{contains}(y, L)$ and $L = L.hd :: L.tl$ and $L.hd \neq y$ }}

```
  L = L.tl;
```

{{ $\text{contains}(y, L_0) = \text{contains}(y, L)$ }}

```
}
return false;
```

contains(y, nil)	:= false	
contains(y, x :: L)	:= true	**if** $x = y$
contains(y, x :: L)	:= contains(y, L)	**if** $x \neq y$

Loop Invariant Example: List Contains (7/7)

- **Check that the body preserves the invariant**

```
{{ Inv: contains(y, L0) = contains(y, L) }}
while (L.kind !== "nil") {
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl }}
  if (L.hd === y)
    {{ contains(y, L0) = true }}
    return true;
  {{ contains(y, L0) = contains(y, L) and L = L.hd :: L.tl and L.hd ≠ y }}
  {{ contains(y, L0) = contains(y, L.tl) }}
  L = L.tl;
  {{ contains(y, L0) = contains(y, L) }}
}
return false;
```

$\text{contains}(y, L_0)$
$= \text{contains}(y, L)$
$= \text{contains}(y, L.hd :: L.tl)$ **since** $L = L.hd :: L.tl$
$= \text{contains}(y, L.tl)$ **since** $y \neq L.hd$

$\text{contains}(y, \text{nil})$	$:= \text{false}$	
$\text{contains}(y, x :: L)$	$:= \text{true}$	**if** $x = y$
$\text{contains}(y, x :: L)$	$:= \text{contains}(y, L)$	**if** $x \neq y$

Hoare Logic & 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

Evaluating Correctness of Loops

- **With straight-line code and conditionals, if the triple is not valid…**
  - **the code is wrong**
  - **there is *some* test case that will prove it**

    (doesn't mean we found that case in our tests, but it exists)

- **With loops, if the triples are not valid…**
  - **the code is wrong *with that invariant***
  - **there may <u>not</u> be any test case that proves it**

    the code may behave correctly on all inputs
  - **the code could be right but with a *different* invariant**

- **Loops are inherently more complicated**

Loop Invariant Example: sqrt (1/9)

- **Declarative spec of** $\mathrm{sqrt}(\mathrm{x})$

  return $\mathrm{y} \in \mathbb{Z}$ **such that** $(\mathrm{y} - 1)^2 < \mathrm{x} \leq \mathrm{y}^2$

  - **precondition that** x **is positive:** $0 < \mathrm{x}$
  - **precondition that x is not too large:** $\mathrm{x} < 10^{12} = (10^6)^2$

Loop Invariant Example: sqrt (2/9)

return $y \in \mathbb{Z}$ such that $(y - 1)^2 < x \leq y^2$

- This loop claims to calculate it:

```
let a: bigint = 0;
let b: bigint = 1000000;
{{ Inv: a² < x ≤ b² }}
while (a !== b - 1) {
  const m = (a + b) / 2n;
  if (m*m < x) {
    a = m;
  } else {
    b = m;
  }
}
return b;
```

Loop Idea

- maintain a range a … b with x in the range a^2 … b^2

Loop Invariant Example: sqrt (3/9)

return $y \in \mathbb{Z}$ such that $(y - 1)^2 < x \leq y^2$

- Check that the invariant holds initially:

```
{{ Pre: 0 < x ≤ 10^12 }}
let a: bigint = 0;
let b: bigint = 1000000;
{{ Inv: a^2 < x ≤ b^2 }}
while (a !== b - 1) {
  …
}
return b;
```

Loop Invariant Example: sqrt (4/9)

return $y \in \mathbb{Z}$ **such that** $(y - 1)^2 < x \le y^2$

- **Check that the invariant holds initially:**

{{ **Pre**: $0 < x \le 10^{12}$ }}

```
let a: bigint = 0;
let b: bigint = 1000000;
```

{{ $0 < x \le 10^{12}$ and $a = 0$ and $b = 10^6$ }}

{{ **Inv**: $a^2 < x \le b^2$ }}

```
while (a !== b - 1) {
  …
}
return b;
```

$$\begin{aligned} a^2 &= 0^2 && \text{since } a = 0 \\ &= 0 \\ &< x \end{aligned}$$

$$\begin{aligned} x &< 10^{12} \\ &= (10^6)^2 \\ &= b^2 && \text{since } b = 10^6 \end{aligned}$$

Loop Invariant Example: sqrt (5/9)

return $y \in \mathbb{Z}$ **such that** $(y-1)^2 < x \le y^2$

- **Check that the postcondition hold after exit**

$\{\{$ **Inv**: $a^2 < x \le b^2$ $\}\}$

```
while (a !== b - 1) {
  ...
}
```

$\{\{\ a^2 < x \le b^2 \text{ and } a = b - 1\ \}\}$

$\{\{\ (b-1)^2 < x \le b^2\ \}\}$

```
return b;
```

Does $(y-1)^2 < x < y^2$ **hold with** $y = b$**?**

$(b-1)^2$

$= a^2$ **since** $a = b - 1$

$< x$

Loop Invariant Example: sqrt (6/9)

return $y \in \mathbb{Z}$ **such that** $(y - 1)^2 < x \leq y^2$

- **Check that the body preserves the invariant:**

{{ **Inv**: $a^2 < x \leq b^2$ }}

```
while (a !== b - 1) {
  {{ a² < x ≤ b²  and a ≠ b - 1 }}
  const m = (a + b) / 2n;
  if (m*m < x) {
    a = m;
  } else {
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

Loop Invariant Example: sqrt (7/9)

return $y \in \mathbb{Z}$ such that $(y - 1)^2 < x \leq y^2$

- Check that the body preserves the invariant:

$\{\{$ **Inv**: $a^2 < x \leq b^2$ $\}\}$

```
while (a !== b - 1) {
```

$\{\{\ a^2 < x \leq b^2$ and $a \neq b - 1\ \}\}$

```
  const m = (a + b) / 2n;
  if (m*m < x) {
```

$\{\{\ a^2 < x \leq b^2$ and $a \neq b - 1$ and $m = (a + b) / 2$ and $m^2 < x\ \}\}$

```
    a = m;
  } else {
```

$\{\{\ a^2 < x \leq b^2$ and $a \neq b - 1$ and $m = (a + b) / 2$ and $x \leq m^2\ \}\}$

```
    b = m;
  }
```

$\{\{\ a^2 < x \leq b^2\ \}\}$

```
}
```

Loop Invariant Example: sqrt (8/9)

return $y \in \mathbb{Z}$ such that $(y - 1)^2 < x \le y^2$

- **Check that the body preserves the invariant:**

$\{\{$ **Inv**: $a^2 < x \le b^2$ $\}\}$

```
while (a !== b - 1) {
  const m = (a + b) / 2n;
  if (m*m < x) {
```

$\{\{\ a^2 < x \le b^2 \text{ and } a \ne b - 1 \text{ and } m = (a + b) / 2 \text{ and } m^2 < x\ \}\}$

$\{\{\ m^2 < x \le b^2\ \}\}$

Immediate!

```
    a = m;
  } else {
```

$\{\{\ a^2 < x \le b^2 \text{ and } a \ne b - 1 \text{ and } m = (a + b) / 2 \text{ and } x \le m^2\ \}\}$

```
    b = m;
  }
```

$\{\{\ a^2 < x \le b^2\ \}\}$

```
}
```

Loop Invariant Example: sqrt (9/9)

return $y \in \mathbb{Z}$ such that $(y - 1)^2 < x \le y^2$

- **Check that the body preserves the invariant:**

```
{{ Inv: a² < x ≤ b² }}
while (a !== b - 1) {
  const m = (a + b) / 2n;
  if (m*m < x) {
    a = m;
  } else {
    {{ a² < x ≤ b²  and a ≠ b – 1 and m = (a + b) / 2 and x ≤ m² }}
    {{ a² < x ≤ m² }}
    b = m;
  }
  {{ a² < x ≤ b² }}
}
```

Immediate!

Correctness of binary search is pretty easy *once* you have the invariant clear!