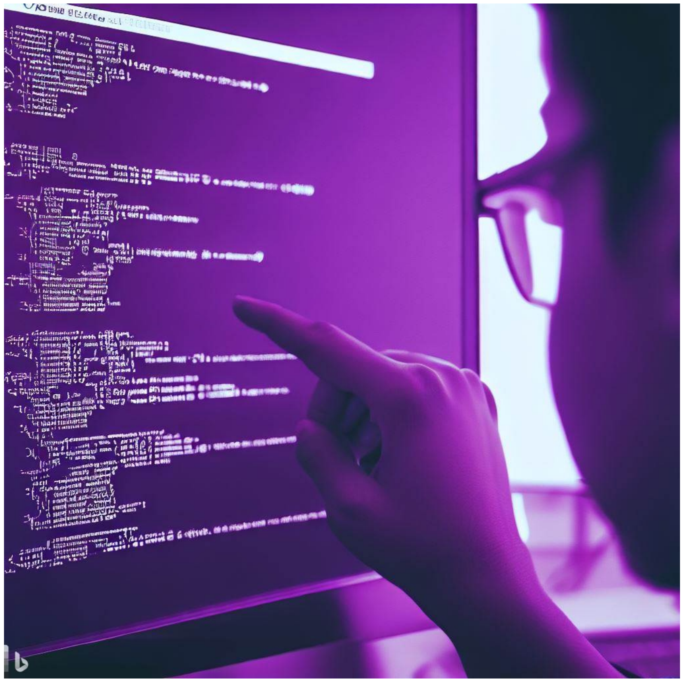


CSE 331

Floyd Logic

Kevin Zatloukal

Reasoning So Far

- Code so far made up of three elements
  - straight-line code
  - conditionals
  - recursion
- Know how to reason (**think**) about these already
  - saw the first two already
  - we reasoned about recursion in math,
    but this can be done in code also

    our code is direct translation of math, so easy to switch between

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: number, b: number): number => {
  if (a >= 0 && b >= 0) {
    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 "$L = cons(\ldots)$"

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

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: number): number => {
  if (n === 0) {
    return 0;
  } else {
    return square(n - 1) + n + n - 1;
  }
};
```

- How do we check correctness?
- Option 1: translate this to math

$$
\begin{aligned}
&\textbf{func}\ \text{square}(0) &&:= 0 \\
&\quad \text{square}(n+1) &&:= \text{square}(n) + 2(n+1) - 1 \qquad \text{for any } n : \mathbb{N}
\end{aligned}
$$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: number): number => { … };
```

func $\text{square}(0) := 0$
$\text{square}(n+1) := \text{square}(n) + 2(n+1) - 1$ for any $n : \mathbb{N}$

- **Prove that** $\text{square}(n) = n^2$ **for any** $n : \mathbb{N}$

- **Structural induction requires proving two implications**
  - **base case: prove** $\text{square}(0) = 0^2$
  - **inductive step: prove** $\text{square}(n+1) = (n+1)^2$

    can use the fact that $\text{square}(n) = n^2$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: number): number => {
  if (n === 0) {
    return 0;
  } else {
    return square(n - 1) + n + n - 1;
  }
};
```

- **Option 2: reason directly about the code**
- **Known fact at top return:** $n = 0$

$$\begin{aligned} \text{square}(0) &= 0 && \text{(code)} \\ &= 0^2 \end{aligned}$$

Reasoning About Recursion

```
// @param n a natural number
// @returns n*n
const square = (n: number): number => {
  if (n === 0) {
    return 0;
  } else {
    return square(n - 1) + n + n - 1;
  }
};
```

why is it okay to assume square is correct when we're checking it?

Inductive Hypothesis

- **Known fact at bottom return:** $n > 0$

$\text{square}(n)$	$= \text{square}(n - 1) + 2n - 1$	(code)
	$= (n - 1)^2 + 2n - 1$	spec of square
	$= n^2 - 2n + 1 + 2n + 1$	
	$= n^2$	

Reasoning So Far

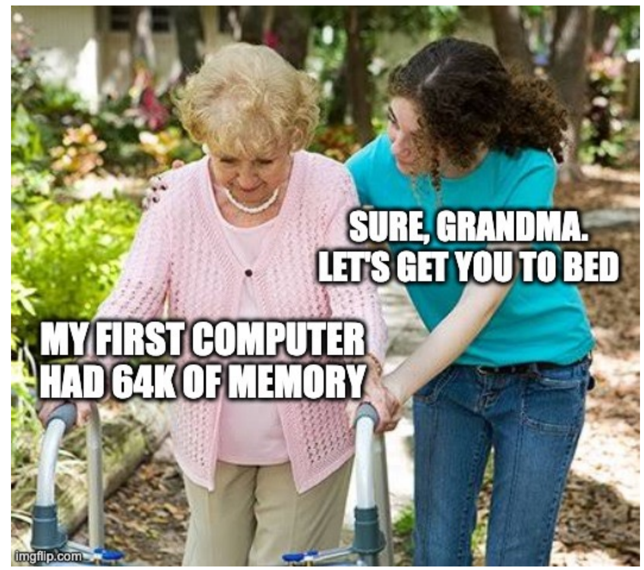
- Code so far made up of three elements
  - straight-line code
  - conditionals
  - structural recursion
- Any[1] program can be written with just these
  - we could stop the course right here!
- For performance reasons, we often use more
  - this week: mutation of local variables
  - next week: mutation of heap data

[1] only exception is code with infinite loops

Brief History of Software

- **Computers used to be very slow**

  my first computer had 64k of memory



- **Software <u>had</u> to be extremely efficient**
  - loops, mutation all over the place
  - very hard to write correctly, so it did *very little*

Brief History of Software

- Computers used to be very slow
  - software had to be extremely efficient
- Today, programmers are the scarcest resource
  - we have enormous computing resources
- Anti-pattern: favoring efficiency over correctness
  - programmers overestimate importance of efficiency

    "programmers are notoriously bad" at guessing what is slow — B. Liskov

    "premature optimization is the root of all evil" — D. Knuth
  - programmers are overconfident about correctness

Brief History of Software

- Computers used to be very slow
  - software had to be extremely efficient

- Today, programmers are the scarcest resource
  - we have enormous computing resources
  - programmers biased toward efficiency over correctness

- Modern systems focus on programmer efficiency
  - e.g., React / angular UI tries to be *functional*
  - e.g., web workers use message passing, not locks

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	“	“	?

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: number, b: number): number => {
  if (a >= 0 && b >= 0) {
    a = a - 1;
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
  …
```

$a \geq 0$? Yes

$a \geq 0$? No!

- Facts no longer hold throughout the function

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

Recall: Finding Facts at a Return Statement

```
// 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 }}
    a = a - 1;
    {{ a ≥ -1 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
```

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

Recall: Finding Facts at a Return Statement

```
// 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 }}
    a = a - 1;
    {{ 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

Recall: Finding Facts at a Return Statement

```
// 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 }}
    a = a - 1;
    {{ a ≥ -1 }}
    const L: List = cons(a, cons(b, nil));
    return sum(L);
  }
```

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

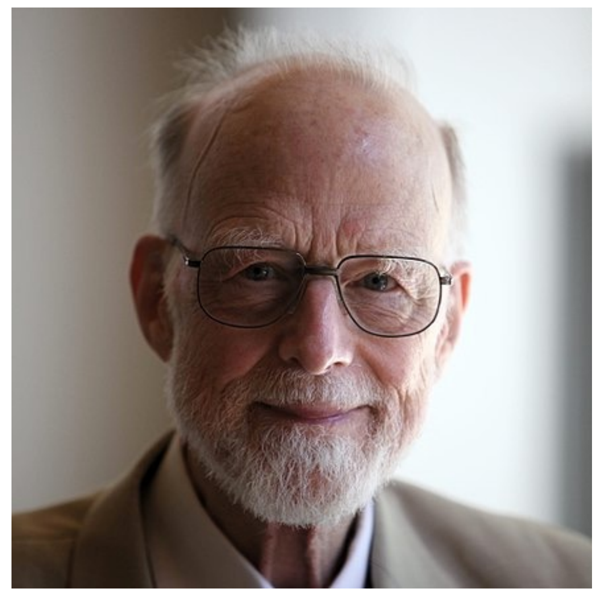
Floyd Logic

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

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

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

  - 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

Correctness Example

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

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

Correctness Example

```
/**
 * @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;
};
```

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

Hoare Triples with No Code

- Code could be empty:

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

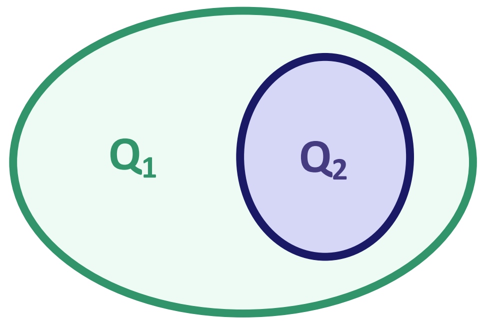
- When is such a triple valid?
  - valid = Q follows from P
  - checking validity without code is proving an **implication**

    we already know how to do this!

- We often say "P is **stronger** than Q"
  - synonym for P implies Q
  - **weaker** if Q implies P

Stronger Assertions vs Specifications

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



- Stronger assertion implies the weaker one
  - stronger is a synonym for “implies”
  - weaker is a synonym for “is implied by”

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…

Mechanical Reasoning Tools

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

- **Forward** reasoning fills in postcondition

$$\{\{\ P\ \}\}\ \ \mathtt{S}\ \ \{\{\ \_\_\ \}\}$$

  - gives *strongest* postcondition making the triple valid

- **Backward** reasoning fills in precondition

$$\{\{\ \_\_\ \}\}\ \ \mathtt{S}\ \ \{\{\ Q\ \}\}$$

  - gives *weakest* precondition making the triple valid

Correctness via Forward Reasoning

- Apply forward reasoning to fill in R

```
{{ 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 to fill in R

```
{{ P }}
{{ R }}      1
   S         2
{{ Q }}
```

  - 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

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

- Can use both types of reasoning on longer code

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

S 1

$$\{\{\ R_1\ \}\}$$

2

$$\{\{\ R_2\ \}\}$$

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

- 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

Example Forward Reasoning through Assignments

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

```
x = 17;
```

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

```
y = 42;
```

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

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

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

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

Example Forward Reasoning through Assignments

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

```
x = 17;
```

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

```
y = 42;
```

$\{\{$ ____________________ $\}\}$

```
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`?**

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;
```
{{ ________________ }}

- **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`?**

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 }}
```

- **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** $y = 42$**)**

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 $z = w + x + y$ }}

- **Could write $z = w + 59$, but do not write $z > 59$!**
  - **this is true since $w > 0$**
  - **this is not the strongest postcondition**
    allows the state with $z = w = 60$, where $z = w + 59$ is false
  - **correctness check could now fail even if the code is right**

Code Example of Forward Reasoning

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

- **Let's check correctness using Floyd logic...**

Code Example of Forward Reasoning

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

- Reason forward...

Code Example of Forward Reasoning

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

- **Check implication:**

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

Code Example of Forward Reasoning

```
// @param w an integer > 0
// @returns an integer z > 59
const f = (w: number): number => {
  const x = 17;
  const y = 42;
  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: $z > 59$

Code Example of Forward Reasoning

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

- We've been doing forward reasoning all quarter!
  - forward reasoning is (only) "and" with *no mutation*
- Line-by-line facts are for "`let`" (not "`const`")

Forward Reasoning through Assignments

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

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

- **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**

Forward Reasoning through Assignments

- **Fix this by giving new names to initial values**
  - **will use "x" and "y" to refer to <u>current</u> values**
  - **can use "x_0" and "y_0" (or other subscripts) for earlier values**
    rewrite existing facts to use the names for earlier values

{{ $w = x + y$ }}
```
x = 4;
```
{{ $w = x_0 + y$ and $x = 4$ }}
```
y = 3;
```
{{ $w = x_0 + y_0$ and $x = 4$ and $y = 3$ }}

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

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 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 = ... 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

$n^2 \geq 4^2$ since $n - 3 \geq 1$ (i.e., $n \geq 4$)

$= 16$

> 10