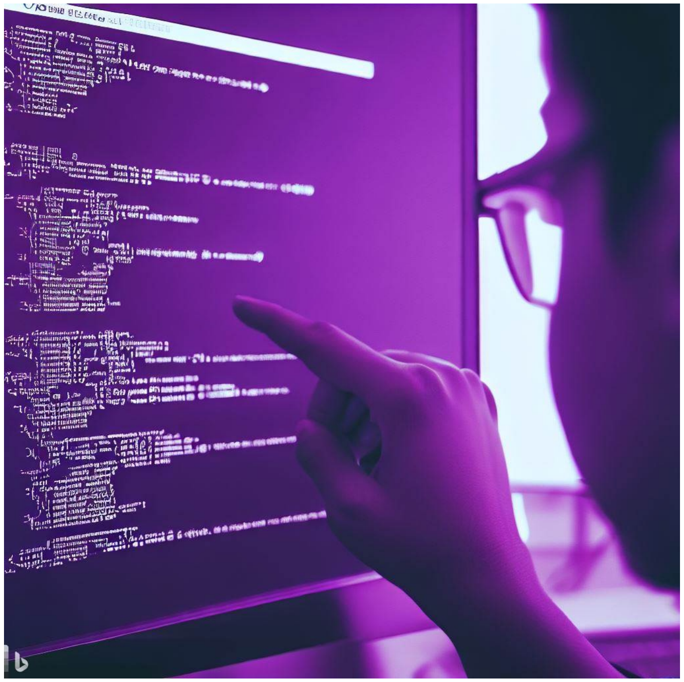
CSE 331

Floyd Logic

Kevin Zatloukal



Administrivia

- Reasoning problems are harder in HW4
  - more sophisticated data types (trees) need more reasoning
- True in general for programming
  - the harder the problem, the more work is "on paper"

    (more time thinking, less time typing)
- Start early on HW4
  - ask questions if you get stuck

Reasoning So Far

- Code so far made up of three elements
  - straight-line code
  - conditionals
  - recursion
- Have seen how to reasoning about each

Reasoning About Straight-Line Code

```
// @param n an integer with n >= 1
// @returns an integer m with m >= 0
function f(n: number): number {
  const a = n + 1;
  const b = n - 1;
  return a * b;
}
```

- Prove an implication
  - show $ab \geq 0$ follows from $n \geq 1$ and $a = n + 1$ and $b = n - 1$
  - proof is a calculation

Reasoning About Straight-Line Code

```
// @param n an integer with n >= 1
// @returns an integer m with m >= 0
function f(n: number): number {
  const a = n + 1;
  const b = n - 1;
  return a * b;
}
```

- Prove an implication

ab	$= (n + 1)b$	since $a = n + 1$
	$= (n + 1)(n - 1)$	since $b = n - 1$
	$= n^2 - 1$	
	$\geq 1 - 1$	since $n \geq 1$
	$= 0$	

Reasoning About Conditionals

```
// @returns an integer m with m >= a, m >= b, and …
function max(a: number, b: number): number {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
}
```

- Prove implications for each return statement
  - then return: show $a \geq a$ and $a \geq b$ follow from $a \geq b$
  - else return: show $b \geq a$ and $b \geq b$ follow from $a < b$
  - proofs are calculations

Reasoning About Recursion

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

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

- **Need to prove that** $\mathrm{square}(n) = n^2$ **for any** $n : \mathbb{N}$

Reasoning About Recursion

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

- **Prove that** $\mathrm{square}(n) = n^2$ **for any** $n : \mathbb{N}$
- **Structural induction requires proving two implications**
  - **base case: prove** $\mathrm{square}(0) = 0^2$
  - **inductive step: prove** $\mathrm{square}(n+1) = (n+1)^2$

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

$$\begin{aligned}
&\textbf{type}\ \mathbb{N} := 0 \\
&\qquad\quad \mid\ n+1 \qquad \text{for any } n : \mathbb{N}
\end{aligned}$$

Reasoning About Recursion

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

- **Inductive step**

$$
\begin{aligned}
\text{square}(n+1) &= \text{square}(n) + 2(n+1) - 1 && \textbf{def of } \text{square} \\
&= n^2 + 2(n+1) - 1 && \textbf{Ind. Hyp.} \\
&= n^2 + 2n + 1 \\
&= (n+1)^2
\end{aligned}
$$

Reasoning So Far

- Code so far made up of three elements
  - straight-line code
  - conditionals
  - 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 had to be extremely efficient
  - loops, mutation all over the place
  - very hard to write correctly, so they did *very little*

- Software “eats the world”
  - much larger programs
  - correctness gets even harder

- Trade computer efficiency for programmer efficiency
  - e.g., React / angular UI tries to be *functional*
  - e.g., operating systems restrict use of *heap data*
  - e.g., web workers use message passing, not locks

Brief History of Software

- Computers used to be very slow
- Software had to be extremely efficient
- Software gets much larger

- Trade computer efficiency for programmer efficiency
  - e.g., React / angular programs being more *functional*
  - e.g., operating systems restrict use of *heap data*

- **Anti-pattern**: focusing too much on efficiency
  - programmers are overconfident about 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

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	“	“	?

Mutation of Local Variables

```
// @param n an integer with n >= 1
function g(n: number): number {
  ...
                          n ≥ 1?   Yes
  n = n - 1;
                          n ≥ 1?   No!
  ...
}
```

- Facts no longer hold throughout the function
- When we state a fact, we have to say where it holds

Mutation of Local Variables

```
// @param n an integer with n >= 1
function g(n: number): number {
  {{ n ≥ 1 }}
  …
  {{ n ≥ 1 }}
  n = n - 1;
  {{ n ≥ 0 }}
  …
}
```

- 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

Mutation of Local Variables

```
// @param n an integer with n >= 1
function g(n: number): number {
  {{ n ≥ 1 }}
  …
  {{ n ≥ 1 }}
  n = n - 1;
  {{ n ≥ 0 }}
  …
}
```

- There are mechanical tools for deriving these
  - precondition is true at the top of the function (by definition)
  - "forward reasoning" says how this changes as we move down
  - "backward reasoning" says how facts change as we move up

Mutation of Local Variables

```
// @param n an integer with n >= 1
function g(n: number): number {
  {{ n ≥ 1 }}
  …
  {{ n ≥ 1 }}
  n = n - 1;
  {{ n ≥ 0 }}
  …
}
```

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

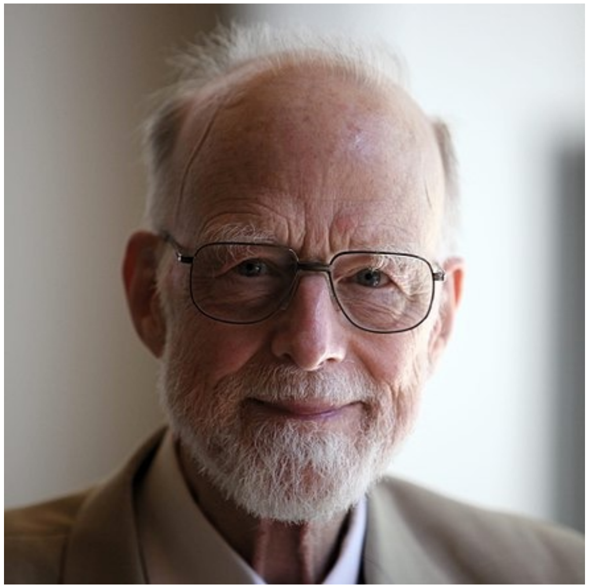
Floyd Logic

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

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

  – 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

Hoare Triples with No Code

- Code could be empty:

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

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

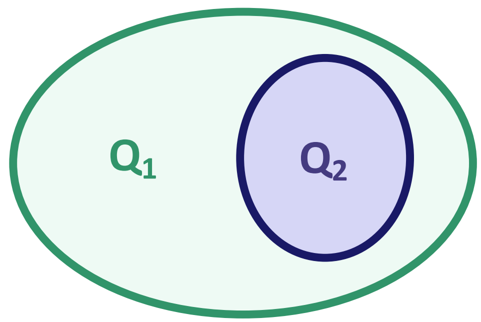
- When is such a triple valid?
  - valid = Q follows from P
  - checking validity without code is proving an **implication**

    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

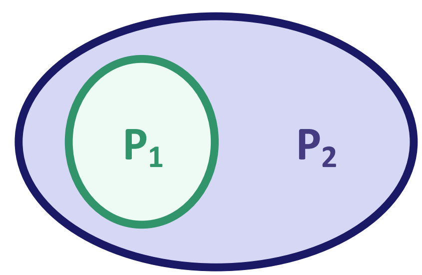
Stronger Assertions vs Specifications

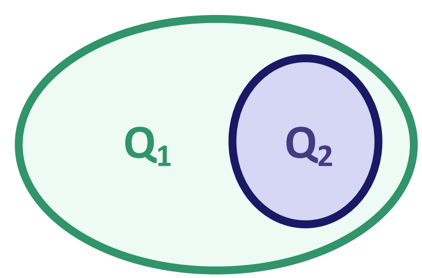
- **Assertion** is stronger iff it holds in a subset of states



- **Specification** is stronger iff
  - postcondition is stronger
  - precondition is weaker

    allows more inputs





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

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

  - 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 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 }}

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

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
  - we will use do this frequently!

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 for all three
  - 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;
```

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

```
y = 42;
```

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

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

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

- **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;
```

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

```
y = 42;
```

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

```
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 $z > 59$!**
  - **this is true since $w > 0$**
  - **this is <u>not</u> 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**

Example Backward Reasoning with Assignments

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

- **What must be true before** $\mathtt{z = w + x + y}$ **so** $\mathtt{z} < 0$ **?**
  - 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**
  - **all we did was <u>substitute</u>** "$w + x + y$" **for** z **in** "$z < 0$"

- **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**
  - **all we did was** <u>**substitute**</u> "42" **for** y **in** "$w + x + y < 0$"

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

Example Backward Reasoning with Assignments

$\{\{ w + 59 < 0 \}\}$

```
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>** "17" **for** x **in** "$w + x + 42 < 0$"

Backward Reasoning through Assignments

- For assignments, backward reasoning is substitution

$$\{\{\ Q[x \mapsto y]\ \}\}$$
```
x = y;
```
$$\{\{\ Q\ \}\}$$

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

- Mechanically simpler than forward reasoning
  - let's see why…

Forward Reasoning through Assignments

- Forward reasoning is trickier
  - previously just added "and $x = y$" to known facts
  - this is correct if x is **not** used in the other facts
  - gets harder if the changed variable appears in other facts…

Forward Reasoning through Assignments

- **Forward reasoning is trickier**
  - previously assumed changed variable was not in assertion
  - gets harder if the changed variable is included:

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

```
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**
  - suppose we start with $w = 10$, $x = 6$, and $y = 4$
  - see that $w = 10 = 6 + 4 = x + y$ is true at the start
  - but $w = 10 \neq 7 = 4 + 3 = x + y$ at the end

Forward Reasoning through Assignments

- **Forward reasoning is trickier**
  - **previously assumed changed variable was not in assertion**
  - **gets harder if the changed variable is included:**

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

- **The precondition** $w = x + y$ **is about** ***initial values***
  - **not necessarily true once the variables are changed**

    (this is why mutation is hard!)

Forward Reasoning through Assignments

- Fix this by giving new names to initial 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 = 4;
```

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

```
y = 3;
```

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

- 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, 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)