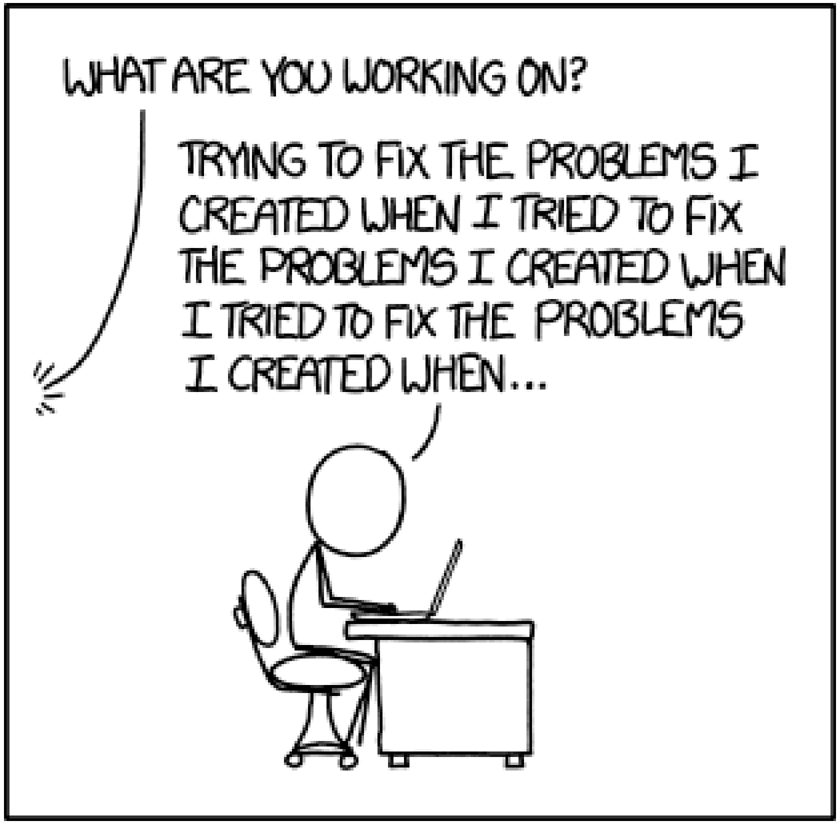
CSE 331
Spring 2025

Software Development & Reasoning



xkcd #1739

Matt Wang

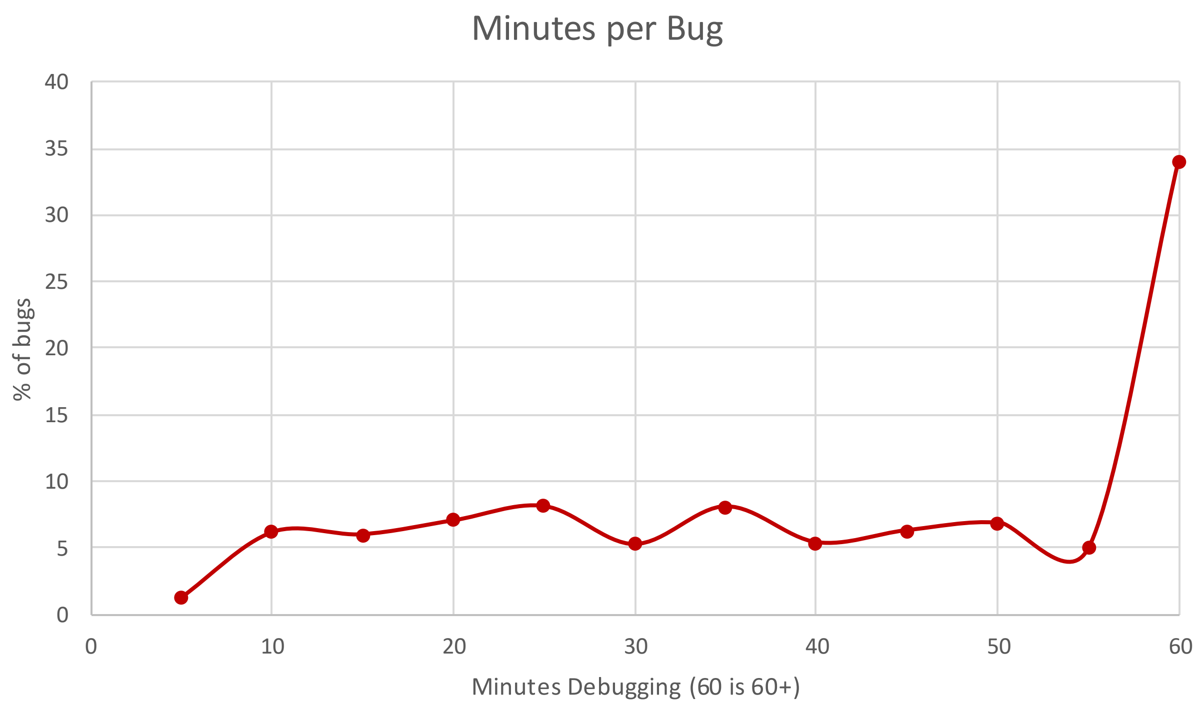
& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia

- HW4 is out!
  - it is longer & contains math *and* programming
  - <u>we are grading on correctness now</u>!
  - (it is also worth more of your grade)
- Matt has added another office hour: 11:30-12:20 on Mondays (after A lecture)

HW3 Summary: Bugs & Time per Bug

- Average solution was ~ 120 lines of code;
  ~ 1 bug per 40 lines of code



- Avg of 57 minutes per bug
- 34% more than 1 hour! Increasing “long tail” trend

HW3 Summary: Search Space of Bugs

- How many functions were searched
  - 62% of bugs searched more than one function
  - time require for debugging

1-2 functions	47 mins
3-4 functions	67 mins
5-6 functions	85 min
7+ functions	114 min

  - on average, every extra function meant ~10 more mins

- Shrinking the search space helps a lot
  - unit tests!
  - defensive programming!

    double check that preconditions are satisfied

    run-time type checking of request/responses

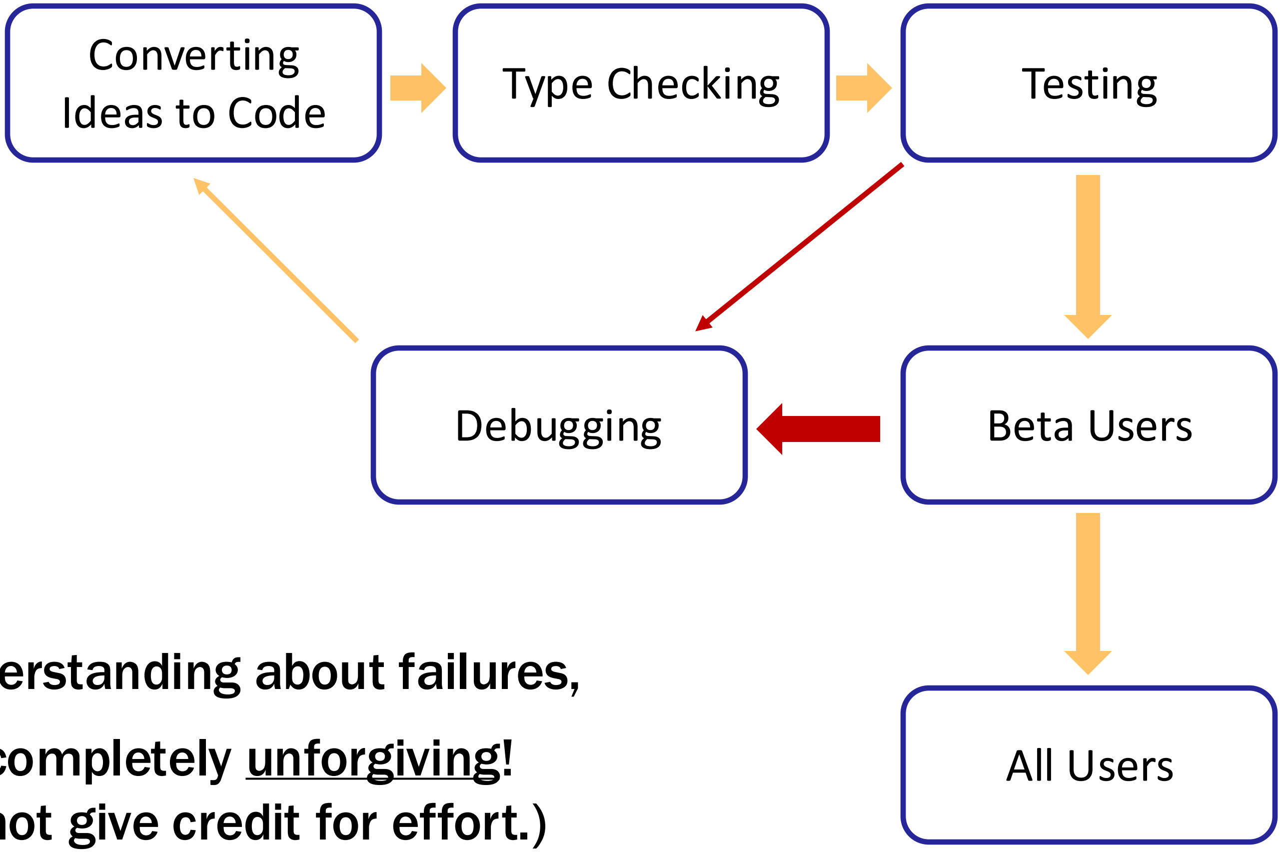
Summary of HW1–3

- HW1: type checking is important
  - found almost 50% of the bugs
- HW2: mutation is dangerous
  - cause of the most horrible kinds of debugging
- HW3: unit testing is important
  - debugging a small space for ~1/3rd of bugs
- Debugging will still happen…
  - need to get better at quickly narrowing in on the bug

Software Development Process

Software Development Process (right now)

Given: a problem description (in English)



Beta users are understanding about failures,

Regular users are completely unforgiving!
(Regular users do not give credit for effort.)

How Much Debugging? (1/2)

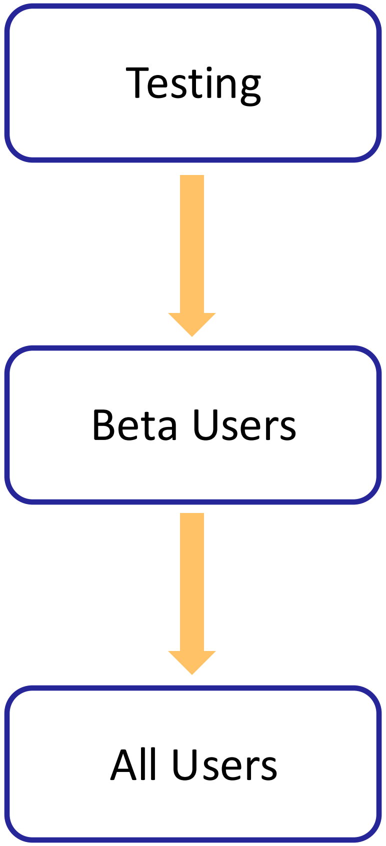
- Bugs typed in... 1 per 20 lines
  - the norm for pretty much everyone
- Bugs after type checking... 1 per 40 lines
  - assume 50% caught by type checker (saw 39% in HW1)
- Bugs after unit testing... 1 per 133 lines
  - assume 70% caught by unit testing

    optimistic: studies find about <70% are caught by unit testing
  - remaining bugs are sent to beta testers

How Much Debugging? (2/2)

- Bugs after testing... 1 per 133 lines
  - assume 70% caught by testing
  - studies find about 65% are caught by testing
- Are rest are caught by beta users?
  - not enough of them
  - millions of users will find all bugs
- Bugs after beta users... 1 per 2000 lines
  - number from Microsoft
  - anything created by humans has mistakes

    only a small number of users give 0 stars



How Many Bugs Sent to Beta Users?

- **Every 2000 lines of code**

100 bugs typed in	1 per 20 lines
– 50 bugs caught by type checker	(50%)
= 50 bugs	
– 35 bugs caught by unit testing	(70%)
= 15 bugs	

- **Need to debug 14 bugs from beta users**
  - **will still send 1 bug to regular users**

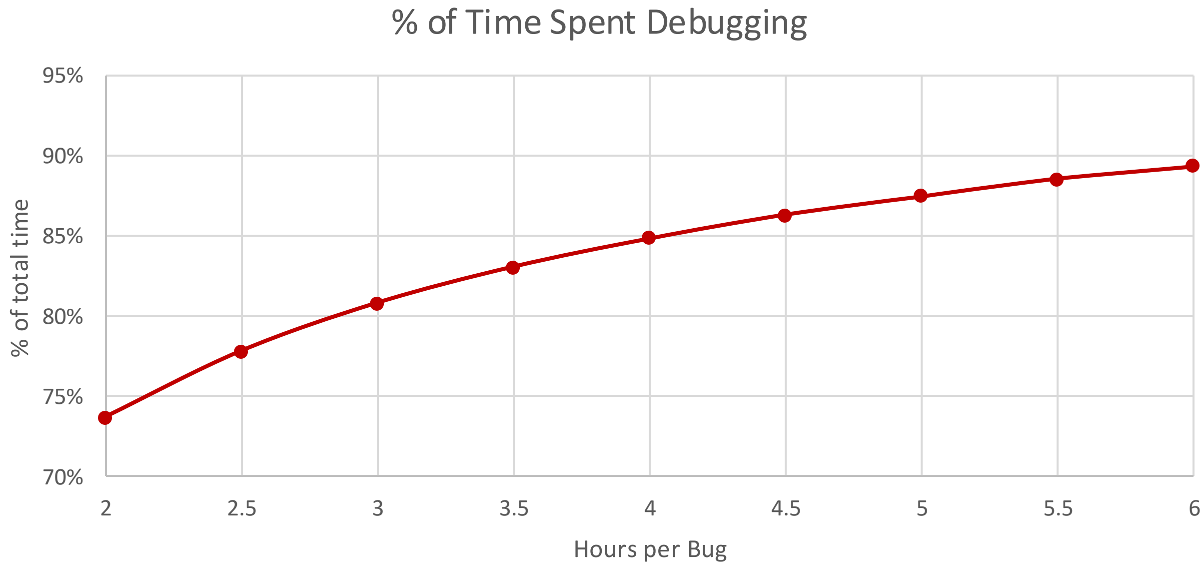
What Kind of Bugs Sent to Beta Users?

- Comes back without steps to reproduce the failure
  - only comes back with a description of the failure
    - maybe a vague (possibly incorrect) description of steps
- Only sent to beta users if it…
  - type checks
  - gets past unit tests
- Most such bugs often at the seams between functions
  - multiple functions need to be debugged
  - will take a long time to track down (many hours)
    - we saw an extra 10 minutes for every additional function in HW3
    - HW3 had 700 lines… industry programs will be 100,000 minimum

Productivity Estimate

- 2000 lines of code
  - assume a familiar setting (know how to solve problems)
  - let "h" be the number of hours to debug one such bug

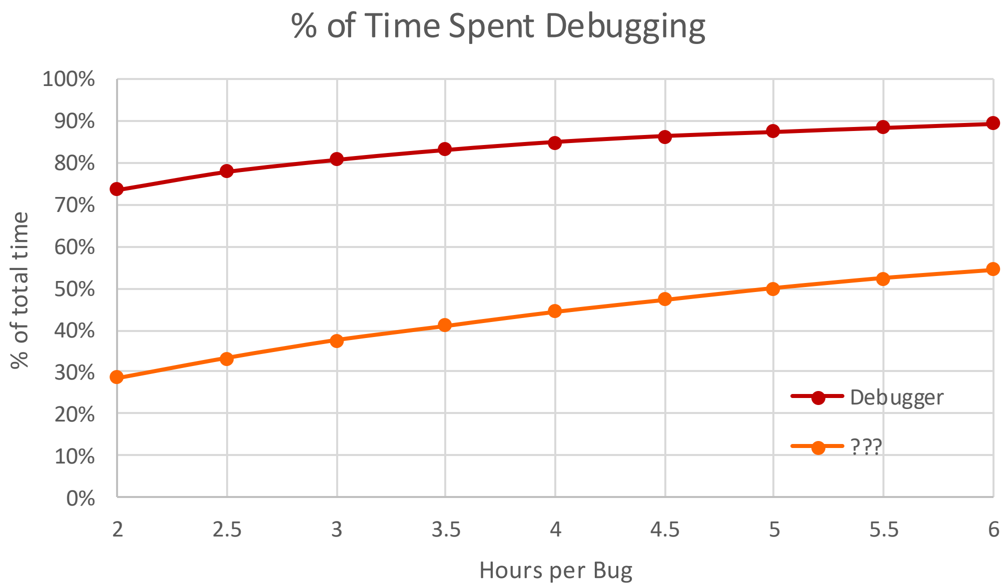
5 hours	typing & fixing type errors
5 hours	testing & fixing *unit* test failures
14 * h hours	debugging & fixing bugs



What Else Can We Do?

- **2000 lines of code**
  - **assume a familiar setting (know how to solve problems)**
  - **let "h" be the number of hours to debug one such bug**

5 hours	typing & fixing type errors
5 hours	?? **removes 11 bugs** ??
5 hours	testing & fixing *unit* test failures
3 * h hours	debugging & fixing bugs



even at h=5, debugging <u>not</u> the majority of time

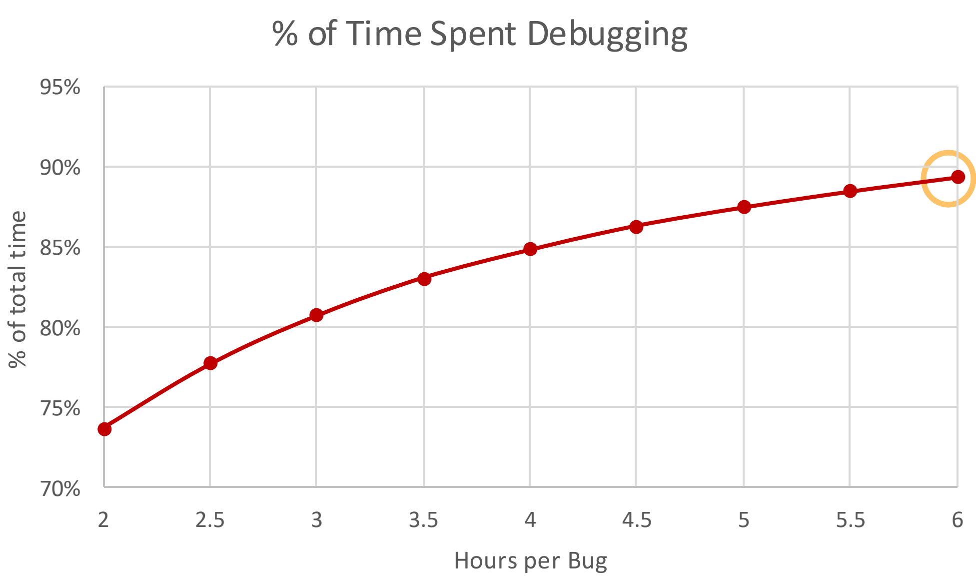
bottom programmer is 2 times more productive

How Much Room For Improvement?

- **Suppose we could…**
  - **remove all 14 bugs by the end of unit testing**
  - **in the same amount of time**

plausible since fixing unit test failures involves debugging

5 hours	typing & fixing type errors
3 hours	?? **removes 14 bugs** ??
2 hours	testing & fixing *unit* test failures



would cut 90% of time spent

would be 10x more productive

"10x developer" possible in a setting where debugging is hard but can be avoided with extra effort

Standard Techniques for Correctness

Standard practice (60+ years) uses three techniques:

- **Tools**: type checker, libraries, etc.
- **Testing**: try it on a well-chosen set of examples
- **Reasoning**: think through your code carefully
  - convince yourself it works correctly on *all inputs*
  - have another person do the same ("code review")

Comparing These Techniques

- Differ along some key dimensions
  - does it consider all allowed inputs
  - does it make sure the answer is fully correct ("=")

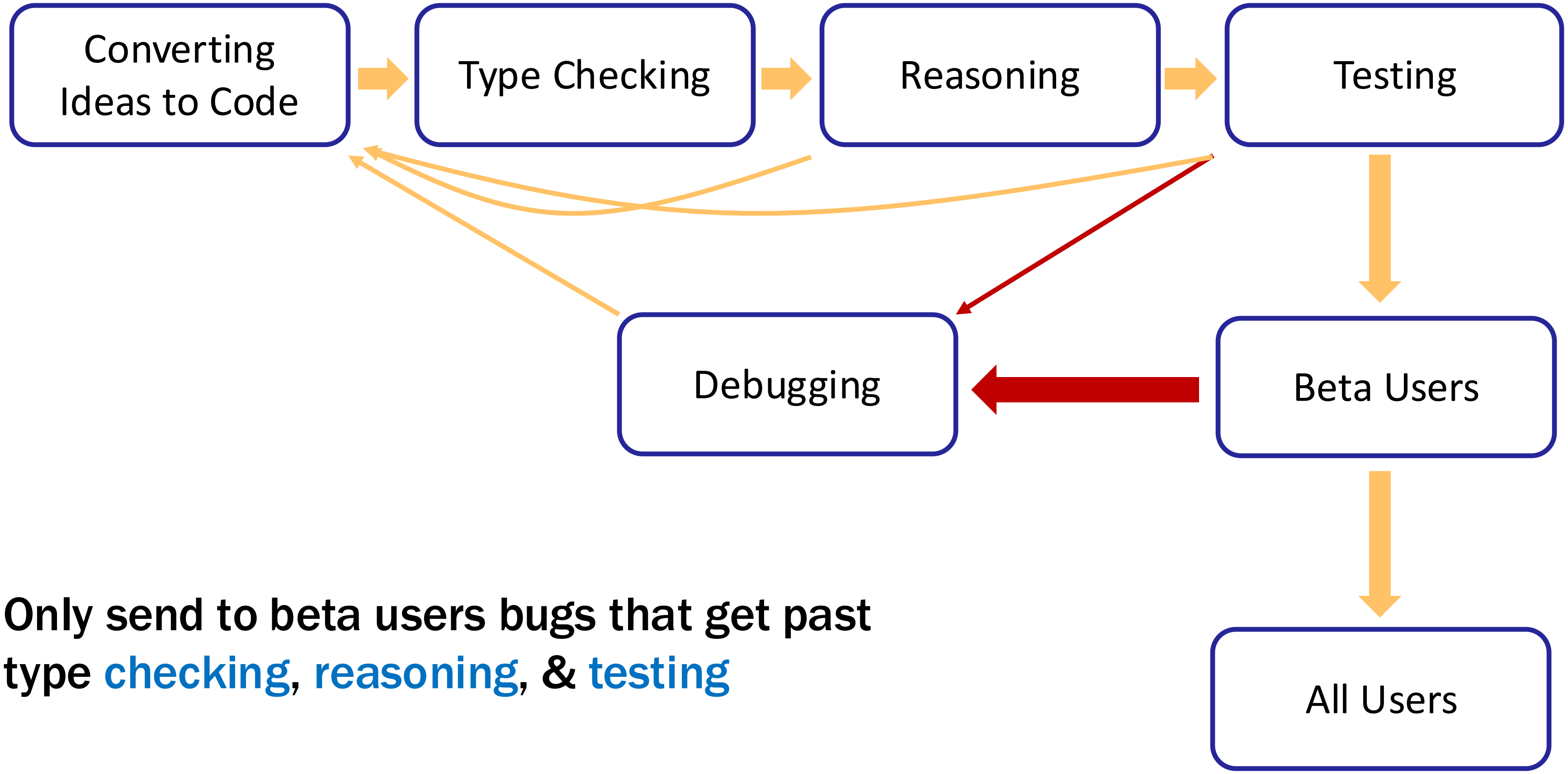
Technique	All Inputs	Fully Correct	Machine-Checkable
Type Checker	Yes	No	Yes
Testing	No	Yes	Yes
Reasoning	Yes	Yes	No (*mostly)

- Combination removes >97% of bugs
  - each tends to find different kinds of errors
  - e.g., type checker is good at typos & reasoning is not

    humans often skip right over typos when reading

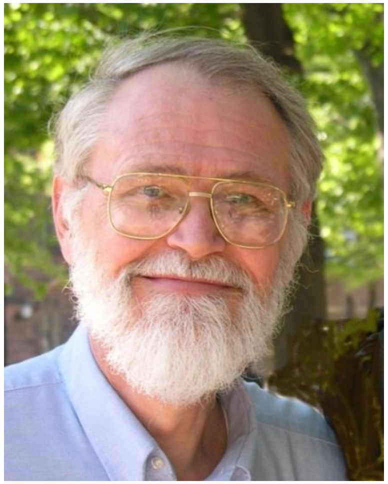
Avoiding Debugging in Software Development

Given: a problem description (in English)



Only send to beta users bugs that get past type checking, reasoning, & testing

“Debugging is twice as hard as writing the code in the first place.”



Brian Kernighan

Reasoning is Expected

- **In industry**: you will be expected to think through your code
  - standard practice is to do this *twice* ("code review")

    you think through your code then ask someone else to also

- Professionals spend most of their coding time reasoning
  - reasoning is the core skill of programming

- Interviews are tests of reasoning
  - take the computer away so you only have reasoning
  - typical coding problem has lots of cases that are easy to miss if you don't think through carefully
  - (not about knowing "the answer" to the question

    interviewers will throw out interviews that went too well!)

“Automating” Reasoning & LLMs

- Reasoning & debugging are provably impossible for a computer to solve in all cases

- Current LLM error rates are much higher than humans
  - requires an (expert) human to do a lot of debugging

    starts with reading and **understanding** all the generated code...

    probably easier to rewrite it yourself
  - studies (so far) show little productivity improvement

    if it reads your mind, it saves you typing, but that's not the limiting factor

    if it doesn't read your mind, you must still spend time understanding it

- LLMs are especially bad at reasoning
  - e.g., bad at learning formal properties
  - e.g., bad at catching rare cases

Actually Correct Automated Reasoning

- There are non-LLM (and crucially, deterministic) approaches to automated reasoning
  - "formal methods" & "formal verification"
  - SAT & SMT-based solvers (incl. model checking)
  - program synthesis
  - automated theorem proving & proof assistants
- Very promising area of research, but…
  - many require graduate-level study to use
  - many current open problems (modularity, scalability)
  - thus, not common in most software engineering fields (yet!)

Reasoning

- "Thinking through" what the code does on <u>**all**</u> inputs
  - neither testing nor type checking can do this
- Can be done formally or informally
  - most professionals reason *informally*
  - we will start with formal reasoning and move to informal
    - formal reasoning is a stepping stone to informal reasoning (same core ideas)
    - formal reasoning still needed for the **hardest** problems
- Definition of correctness comes from the specification…

Correct Requires a Specification

Specification contains two sets of facts

Precondition:

facts we are *promised* about the inputs

Postcondition:

facts we are required to *ensure* for the output

Correctness (satisfying the spec):

for every input satisfying the precondition,

the output will satisfy the postcondition

Specifications in TypeScript: JSDoc

- TypeScript, like Java, writes specs in `/** … */`

```
/**
 * High level description of what function does
 * @param a What "a" represents + any conditions
 * @param b What "b" represents + any conditions
 * @returns Detailed description of return value
 */
const f = (a: bigint, b: bigint): bigint => {..};
```

  - these are formatted as “JSDoc” comments
  - (in Java, they are JavaDoc comments)

Preconditions & Postconditions in JSDoc

- Specifications are written in the comments

```
/**
 * Returns the first n elements from the list L
 * @param n non-negative length of the prefix
 * @param L the list whose prefix should be returned
 * @requires n <= len(L)
 * @returns list S such that L = S ++ T for some T
 */
const prefix = (n: bigint, L: List): List => {..};
```

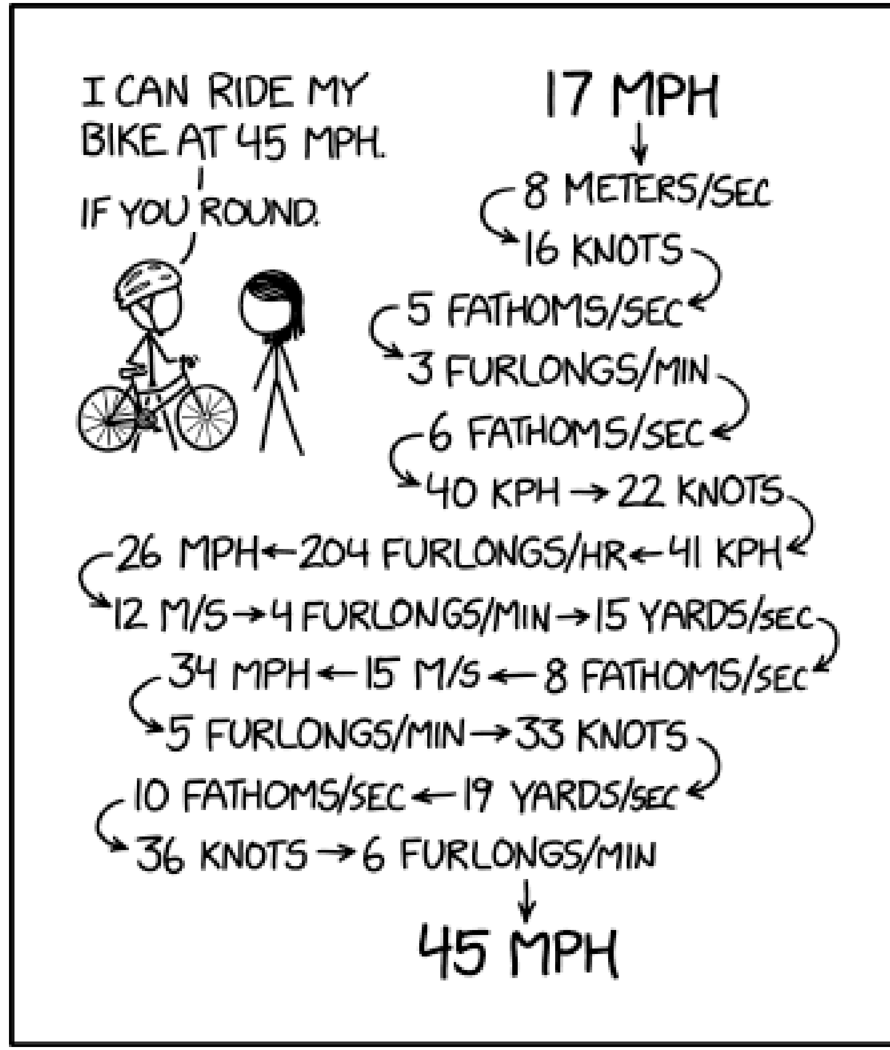
  - precondition written in `@param` and `@requires`
  - postcondition written in `@returns`

CSE 331
Spring 2025

Proof by Calculation (& Cases)

Matt Wang

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



xkcd #2585

Recall: Specification

Specification contains two sets of facts

Precondition:

facts we are *promised* about the inputs

Postcondition:

facts we are required to *ensure* for the output

Correctness (satisfying the spec):

for every input satisfying the precondition,

the output will satisfy the postcondition

Facts (1/2)

- Basic inputs to reasoning are "facts"
  - things we know to be true about the variables

    these hold for all inputs (no matter what value the variable has)
  - typically, "=" or "≤"

```
// @param n a natural number
const f = (n: bigint): bigint => {
  const m = 2n * n;
  return (m + 1n) * (m - 1n);
};
```

find facts by reading along path
from top to return statement

- At the return statement, we know these facts:
  - $n \in \mathbb{N}$ (or $n \in \mathbb{Z}$ and $n \geq 0$)
  - $m = 2n$

Facts (2/2)

- Basic inputs to reasoning are "facts"
  - things we know to be true about the variables

    these hold for all inputs (no matter what value the variable has)
  - typically, "=" or "≤"

```
// @param n a natural number
const f = (n: bigint): bigint => {
  const m = 2n * n;
  return (m + 1n) * (m - 1n);
};
```

- No need to include the fact that n is an integer ($n \in \mathbb{Z}$)
  - that is true, but the type checker takes care of that
  - no need to repeat reasoning done by the type checker

Finding Facts at a Return Statement

- Consider this code

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

find facts by reading along path
from top to return statement

facts are math statements about the code

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

Implications

- We can use the facts we know to prove more facts
  - if we can prove R using facts P and Q,
    we say that R "follows from" or "is implied by" P and Q
  - proving this fact is proving an "implication"

- Checking correctness requires proving implications
  - need to prove facts about the **return** values
  - return values must satisfy the facts of the postcondition

Collecting Facts

- Saw how to collect facts in code consisting of
  - "`const`" variable declarations
  - "`if`" statements
  - collect facts by reading along path from top to return

- Those elements cover all code without mutation
  - covers everything describable by our math notation
  - we can calculate interesting values with *recursion*

- Will need more tools to handle code with mutation…

Mutation Makes Reasoning Harder

Description	Testing	Tools	Reasoning	
no mutation	full coverage	type checker	calculation induction	HW5
local variable mutation	“”	“”	Floyd logic	HW6
array mutation	“”	“”	for-any facts	HW8
heap state mutation	“”	“”	rep invariants	HW9

Correctness with No Mutation

- Proving implications is the **core step** of reasoning
  - other techniques output implications for us to prove
- Facts are written in our math notation
  - we will use math tools to prove implications
- Core technique is "proof by calculation"
- Other techniques we will need:
  - proof by cases (today)
  - structural induction (Wednesday)

Proof by Calculation

Proof by Calculation

- Proves an implication
  - fact to be shown is an equation or inequality
- Uses known facts and definitions
  - latter includes, e.g., the fact that $\mathrm{len}(\mathrm{nil}) = 0$

Example Proof by Calculation

- Given $x = y$ and $z \leq 10$, prove that $x + z \leq y + 10$
  - show the third fact follows from the first two

- Start from the left side of the inequality to be proved

$$\underbrace{x + z = y + z}_{\text{since } x = y} \leq y + 10$$

$$y + z \underbrace{\leq y + 10}_{\text{since } z \leq 10}$$

All together, this tells us that $x + z \leq y + 10$

Example Proof by Calculation (across lines)

- **Given $x = y$ and $z \leq 10$, prove that $x + z \leq y + 10$**
  - show the third fact follows from the first two

- **Start from the left side of the inequality to be proved**

$x + z$	$= y + z$	since $x = y$
	$\leq y + 10$	since $z \leq 10$

  - easier to read when split across lines
  - "calculation block", includes explanations in right column

    proof by calculation means using a calculation block
  - "$=$" or "$\leq$" relates that line to the previous line

Calculation Blocks: Equalities

- Chain of "=" shows first = last

$$
\begin{aligned}
\mathrm{a} \quad &= \mathrm{b} \\
&= \mathrm{c} \\
&= \mathrm{d}
\end{aligned}
$$

  - proves that $\mathrm{a} = \mathrm{d}$
  - all 4 of these are the same number

Calculation Blocks: Inequalities

- Chain of "=" and "≤" shows <u>first</u> ≤ <u>last</u>

$x + z$	$= y + z$	since $x = y$
	$\leq y + 10$	since $z \leq 10$
	$= y + 3 + 7$	
	$\leq w + 7$	since $y + 3 \leq w$

  – each number is equal or strictly larger that previous

    last number is strictly larger than the first number

  – analogous for "≥"

Calculation Blocks: Mixing Inequalities Gotcha

- **Consider:**

$$
\begin{aligned}
1+1 &= 2 \\
&\geq 2 * 1 \\
&= 1 * 2 \\
&\leq 1 * 3 \\
&\geq 3
\end{aligned}
$$

  - **cannot derive meaningful conclusion from "proof"**

    each step is still true, but cannot make final conclusion

  - **rule of thumb: inequalities should only go in one direction**

Proving Code by Calculation: Example 1 (1/2)

```
// Inputs x and y are positive integers
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts “$x \geq 1$” and “$y \geq 1$”
- Correct if the return value is a positive integer

$$x + y$$

Proving Code by Calculation: Example 1 (2/2)

```
// Inputs x and y are positive integers
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 1$" and "$y \geq 1$"

- Correct if the return value is a positive integer

$$
\begin{aligned}
x + y &\geq x + 1 && \text{since } y \geq 1 \\
&\geq 1 + 1 && \text{since } x \geq 1 \\
&= 2 \\
&\geq 1
\end{aligned}
$$

– calculation shows that $x + y \geq 1$

Proving Code by Calculation: Example 2 (1/2)

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 9$" and "$y \geq -8$"
- Correct if the return value is a positive integer

$$x + y$$

Proving Code by Calculation: Example 2 (2/2)

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 9$" and "$y \geq -8$"
- Correct if the return value is a positive integer

$x + y$	$\geq x + -8$	since $y \geq -8$
	$\geq 9 - 8$	since $x \geq 9$
	$= 1$	

Proving Code by Calculation: Example 3 (1/2)

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x > 8$" and "$y > -9$"

- Correct if the return value is a positive integer

$$x + y$$

Proving Code by Calculation: Example 3 (2/2)

```
// Inputs x and y are integers with x > 8 and y > -9
// Returns a positive integer.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x > 8$" and "$y > -9$"

- Correct if the return value is a positive integer

$x + y$	$> x + -9$	since $y > -9$
	$> 8 - 9$	since $x > 8$
	$= -1$	

warning: avoid using ">" (or "<") *multiple* times in a calculation block

Proving Code by Calculation: Example 4 (1/2)

```
// Inputs x and y are integers with x > 3 and y > 4
// Returns an integer that is 10 or larger.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 4$" and "$y \geq 5$"
- Correct if the return value is 10 or larger

$$x + y$$

Proving Code by Calculation: Example 4 (2/2)

```
// Inputs x and y are integers with x > 3 and y > 4
// Returns an integer that is 10 or larger.
const f = (x: bigint, y, bigint): bigint => {
  return x + y;
};
```

- Known facts "$x \geq 4$" and "$y \geq 5$"
- Correct if the return value is 10 or larger

$$
\begin{aligned}
x + y &\geq x + 5 && \text{since } y \geq 5 \\
&\geq 4 + 5 && \text{since } x \geq 4 \\
&= 9
\end{aligned}
$$

proof doesn't work because the **code is wrong**!

Using Definitions in Calculations

- **Most useful with function calls**
  - cite the definition of the function to get the return value

- **For example:**

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

- **Can cite facts such as**
  - $\text{sum}(\text{nil}) = 0$
  - $\text{sum}(a :: b :: \text{nil}) = a + \text{sum}(b :: \text{nil})$

second case of definition with $x = a$ and $L = b :: \text{nil}$

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

find facts by reading along path
from top to return statement

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

Using Definitions in Calculations (1/2)

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

- Know “$a \geq 0$”, “$b \geq 0$”, and “$L = a :: b :: \text{nil}$”
- Prove the “$\text{sum}(L)$” is non-negative

$$\text{sum}(L)$$

Using Definitions in Calculations (2/2)

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

- **Know** "$a \geq 0$", "$b \geq 0$", **and** "$L = a :: b :: \mathrm{nil}$"
- **Prove the** "$\mathrm{sum}(L)$" **is non-negative**

$\mathrm{sum}(L)$	$= \mathrm{sum}(a :: b :: \mathrm{nil})$	**since** $L = a :: b :: \mathrm{nil}$
	$= a + \mathrm{sum}(b :: \mathrm{nil})$	**def of** sum
	$= a + b + \mathrm{sum}(\mathrm{nil})$	**def of** sum
	$= a + b$	**def of** sum
	$\geq 0 + b$	**since** $a \geq 0$
	≥ 0	**since** $b \geq 0$

Proving Correctness with Conditionals (Top)

```
// Inputs x and y are integers.
// Returns a number less than x.
const f = (x: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

- Known fact in “then” (top) branch: “$y \leq -1$”

$$
\begin{aligned}
x + y &\leq x + -1 && \text{since } y \leq -1 \\
&< x + 0 && \text{since } -1 < 0 \\
&= x
\end{aligned}
$$

Proving Correctness with Conditionals (Bottom)

```
// Inputs x and y are integers.
// Returns a number less than x.
const f = (x: bigint, y, bigint): bigint => {
  if (y < 0n) {
    return x + y;
  } else {
    return x - 1n;
  }
};
```

- Known fact in else (bottom) branch: "$y \geq 0$"

$$
\begin{aligned}
x - 1 &< x + 0 && \text{since } -1 < 0 \\
&= x
\end{aligned}
$$

Proving Correctness with Multiple Claims

- Need to check the claim from the spec at each `return`

- If spec claims multiple facts, then
  we must prove that <u>each</u> of them holds

```
// Inputs x and y are integers with x < y - 1
// Returns a number less than y and greater than x.
const f = (x: bigint, y, bigint): bigint => { .. };
```

  – multiple known facts: $x : \mathbb{Z}$, $y : \mathbb{Z}$, and $x < y - 1$
  – multiple claims to prove: $x < r$ and $r < y$
    where "r" is the return value
  – requires *two* calculation blocks

Example Correctness with Conditionals

```
// Returns r with (r=a or r=b) and r >= a and r >= b
const max = (a: bigint, b, bigint): bigint => {
  if (a >= b) {
    return a;
  } else {
    return b;
  }
};
```

declarative spec of max

- Three different facts to prove at each `return`
- Two known facts in each branch (return value is "r"):
  - then branch: $a \geq b$ and $r = a$
  - else branch: $a < b$ and $r = b$

Proof By Cases

- Sometimes necessary split a proof into cases
  - fact may be hard to prove for all values at once

- Example: can't prove it for all x at once,
  but can prove it for $x \geq 0$ and $x < 0$
  - will see an example next

- If we can prove it in those two cases, it holds for all x
  - follows since the cases are exhaustive
    (don't need to be exclusive in this case)

Example Proof By Cases

$f : \mathbb{Z} \to \mathbb{Z}$

$f(m) := 2m + 1$ **if** $m \geq 0$

$f(m) := 0$ **if** $m < 0$

- **Want to prove that** $f(m) > m$
- **Doesn't seem possible as is**
  - **can't even apply the definition of** f
  - **need to know if** $m < 0$ **or** $m \geq 0$
- **Split our analysis into these two separate cases…**

Proof By Cases (1/3)

$$f(m) := 2m + 1 \qquad \textbf{if } m \geq 0$$
$$f(m) := 0 \qquad \textbf{if } m < 0$$

- **Prove that** $f(m) > m$

Case $m \geq 0$:

$f(m) =$

$> m$

Proof By Cases (2/3)

$f(m) := 2m + 1$ **if** $m \geq 0$

$f(m) := 0$ **if** $m < 0$

- **Prove that** $f(m) > m$

Case $m \geq 0$:

$f(m) = 2m + 1$ **def of** f **(since** $m \geq 0$**)**

$\geq m + 1$ **since** $m \geq 0$

$> m$ **since** $1 > 0$

Proof By Cases (3/3)

$$f(m) := 2m + 1 \qquad \textbf{if } m \geq 0$$
$$f(m) := 0 \qquad \textbf{if } m < 0$$

- **Prove that** $f(m) > m$

Case $m \geq 0$:

$f(m) = \ldots > m$

Case $m < 0$:

$f(m) = 0$ **def of** f **(since** $m < 0$**)**

$> m$ **since** $m < 0$

Since these two cases are exhaustive, $f(m) > m$ **holds in general.**

Proofs in Class & HW versus the "Real World"

- Lecture (mostly) focuses on toy examples
  - Goal is to explain syntax & intuition (and build skill)
  - Thus, pick simple problems (that may feel "obvious")
  - Because I prep, I don't get "stuck"

- Section & HW will (mostly) focus on proving that correct code is correct
  - Seems mean to give you incorrect code :')
  - But, problems will be <u>new</u> and <u>more challenging</u>

- In real world, likely even harder examples and will *not* know correctness ahead of time

CSE 331
Spring 2025

Reasoning with Structural Induction

Matt Wang

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

JS Wacky Weekly Wednesday

```
// setTimeout: call function after n milliseconds

// prints 0 1 2
for (let i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}

// prints 3 3 3 ????
let i;
for (i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
```

Structural Induction

Proof by Calculation on Lists

- Our proofs so far have used fixed-length lists
  - **e.g.,** $\mathrm{sum}(a :: b :: \mathrm{nil}) \geq 0$

- Would like to prove facts about any length list L

- For example…

Example: Echo Function

- Consider the following function:

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

- Produces a list where every element is repeated twice

$$\begin{aligned} &\text{echo}(1 :: 2 :: \text{nil}) \\ &\quad = 1 :: 1 :: \text{echo}(2 :: \text{nil}) && \textbf{def of } \text{echo} \\ &\quad = 1 :: 1 :: 2 :: 2 :: \text{echo}(\text{nil}) && \textbf{def of } \text{echo} \\ &\quad = 1 :: 1 :: 2 :: 2 :: \text{nil} && \textbf{def of } \text{echo} \end{aligned}$$

Example: Proving Len & Echo Correct

$$\text{echo}(\text{nil}) \quad := \text{nil}$$
$$\text{echo}(x :: L) \quad := x :: x :: \text{echo}(L)$$

- Suppose we have the following code:

```
const m = len(S);         // S is some List
const R = echo(S);
…
return 2*m;   // = len(echo(S))
```

  – spec says to return $\text{len}(\text{echo}(S))$ but code returns $2\ \text{len}(S)$

- Need to prove that $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$

Matt's Proof Strategy Advice™ (1/3)

- Stuck on a proof?
  - Try splitting into cases!

Trying Proof by Cases on Len & Echo (1/2)

$\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$

Case $S = \text{nil}$**:**

$\text{len}(\text{echo}(S))$	$= \text{len}(\text{nil})$	**def of** echo (**since** $S = \text{nil}$)
	$= 0$	**def of** len
	$= 2\ \text{len}(\text{nil})$	**def of** len
	$= 2\ \text{len}(S)$	

Trying Proof by Cases on Len & Echo (2/2)

$\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$

Case $S = x :: L$ **:**

$\text{len}(\text{echo}(x :: L))$	$= \text{len}(x :: x :: \text{echo}(L))$	**def of** echo
	$= 1 + \text{len}(x :: \text{echo}(L))$	**def of** len
	$= 2 + \text{len}(\text{echo}(L))$	**def of** len

Now need to prove: $\text{len}(\text{echo}(L)) = 2\ \text{len}(L)$

Case $L = \text{nil}$**: see previous slide**

Case $L = x :: M$ **:**

$\text{len}(\text{echo}(x :: M))$	$= \text{len}(x :: x :: \text{echo}(M))$	**def of** echo
	$= 1 + \text{len}(x :: \text{echo}(M))$	**def of** len
	$= 2 + \text{len}(\text{echo}(M))$	**def of** len

Now need to prove: $\text{len}(\text{echo}(M)) = 2\ \text{len}(M)$

Proof by Cases Breaks on Inductive Data

- Our proofs so far have used fixed-length lists
  - **e.g.,** $\mathrm{sum}(a :: b :: \mathrm{nil}) \geq 0$

- Would like to prove facts about <u>any length</u> list L

- Need more tools for this…
  - structural recursion *calculates* on inductive types
  - structural induction *reasons* about structural recursion

    or more generally, to prove facts containing variables of an inductive type
  - both tools are specific to **inductive types**

Structural Induction is Two Implications

Let $\text{P}(\text{S})$ **be the claim** "$\text{len}(\text{echo}(\text{S})) = 2\ \text{len}(\text{S})$"

To prove $\text{P}(\text{S})$ **holds for** **<u>any</u>** **list** S**, prove two implications**

Base Case: prove $\text{P}(\text{nil})$

– **use any known facts and definitions**

Inductive Step: prove $\text{P}(\text{x} :: \text{L})$

– x **and** L **are variables**

– **use any known facts and definitions plus** **<u>one more fact…</u>**

– **make use of the fact that** L **is also a** List

Structural Induction: Inductive Hypothesis

To prove $\mathrm{P}(S)$ holds for any list S, prove two implications

Base Case: prove $\mathrm{P}(\mathrm{nil})$

- use any known facts and definitions

Inductive Hypothesis: assume P(L) is true

- use this in the inductive step, but not anywhere else

Inductive Step: prove $\mathrm{P}(\mathrm{x} :: \mathrm{L})$

- use known facts and definitions and Inductive Hypothesis

Why Structural Induction Works

With Structural Induction, we prove two facts

$\mathrm{P(nil)}$ $\qquad$ $\mathrm{len(echo(nil)) = 2\ len(nil)}$

$\mathrm{P(x :: L)}$ $\qquad$ $\mathrm{len(echo(x :: L)) = 2\ len(x :: L)}$

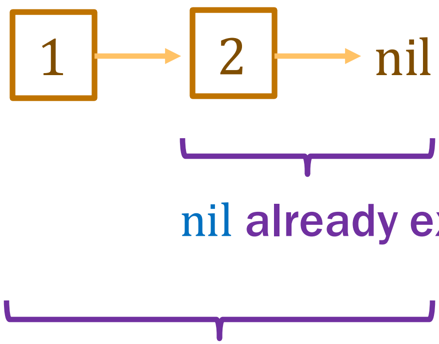
(**second assuming** $\mathrm{len(echo(L)) = 2\ len(L)}$)

Why is this enough to prove $\mathrm{P(S)}$ for any $\mathrm{S : List}$?

Inductive Data is "Built Up" in Steps

Build up an object using constructors:

nil	first constructor (nil)
2 :: nil	second constructor (cons)
1 :: 2 :: nil	second constructor (cons)



nil already exists when building 2 :: nil

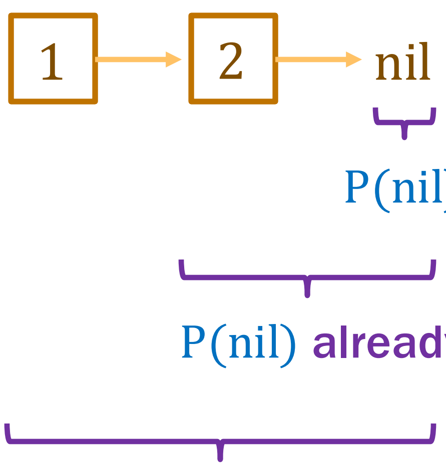
2 :: nil already exists when building 1 :: 2 :: nil

Inductive Proofs are "Built Up" in Steps

Build up a proof the same way we built up the object

$P(\text{nil})$	$\text{len}(\text{echo}(\text{nil})) = 2\ \text{len}(\text{nil})$
$P(x :: L)$	$\text{len}(\text{echo}(x :: L)) = 2\ \text{len}(x :: L)$

(**second assuming** $\text{len}(\text{echo}(L)) = 2\ \text{len}(L)$)



$P(\text{nil})$

$P(\text{nil})$ **already proven when proving** $P(2 :: \text{nil})$

$P(2 :: \text{nil})$ **already proven when proving** $P(1 :: 2 :: \text{nil})$

Example: Echo & Len Base Case (1/2)

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

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** $S : \text{List}$

Base Case (nil):

Need to prove that $\text{len}(\text{echo}(\text{nil})) = 2\ \text{len}(\text{nil})$

$\text{len}(\text{echo}(\text{nil})) \quad =$

$$\begin{aligned} \text{len}(\text{nil}) &:= 0 \\ \text{len}(x :: L) &:= 1 + \text{len}(L) \end{aligned}$$

Example: Echo & Len Base Case (2/2)

$$\text{echo}(\text{nil}) \quad := \text{nil}$$
$$\text{echo}(x :: L) \quad := x :: x :: \text{echo}(L)$$

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** $S : \text{List}$

Base Case (nil):

$\text{len}(\text{echo}(\text{nil}))$	$= \text{len}(\text{nil})$	**def of** echo
	$= 0$	**def of** len
	$= 2 \cdot 0$	
	$= 2\ \text{len}(\text{nil})$	**def of** len

Example: Echo & Len Inductive Step (1/3)

$$\text{echo}(\text{nil}) := \text{nil}$$
$$\text{echo}(\text{x} :: \text{L}) := \text{x} :: \text{x} :: \text{echo}(\text{L})$$

- **Prove that** $\text{len}(\text{echo}(\text{S})) = 2\ \text{len}(\text{S})$ **for any** $\text{S} : \text{List}$

Inductive Step $(\text{x} :: \text{L})$:

Need to prove that $\text{len}(\text{echo}(\text{x} :: \text{L})) = 2\ \text{len}(\text{x} :: \text{L})$

Get to assume claim holds for L**, i.e., that** $\text{len}(\text{echo}(\text{L})) = 2\ \text{len}(\text{L})$

Example: Echo & Len Inductive Step (2/3)

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

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{len}(\text{echo}(L)) = 2\ \text{len}(L)$

Inductive Step $(x :: L)$**:**

$$\text{len}(\text{echo}(x :: L))$$

$$= 2\ \text{len}(x :: L)$$

$$\begin{aligned} &\text{len(nil)} && := 0 \\ &\text{len}(x :: L) && := 1 + \text{len}(L) \end{aligned}$$

Example: Echo & Len Inductive Step (3/3)

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

- **Prove that** $\text{len}(\text{echo}(S)) = 2\ \text{len}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{len}(\text{echo}(\text{L})) = 2\ \text{len}(\text{L})$

Inductive Step ($\text{x} :: \text{L}$)**:**

$$\begin{aligned} \text{len}(\text{echo}(\text{x} :: \text{L})) &= \text{len}(\text{x} :: \text{x} :: \text{echo}(\text{L})) && \textbf{def of } \text{echo} \\ &= 1 + \text{len}(\text{x} :: \text{echo}(\text{L})) && \textbf{def of } \text{len} \\ &= 2 + \text{len}(\text{echo}(\text{L})) && \textbf{def of } \text{len} \\ &= 2 + 2\ \text{len}(\text{L}) && \textbf{Ind. Hyp.} \\ &= 2(1 + \text{len}(\text{L})) && \\ &= 2\ \text{len}(\text{x} :: \text{L}) && \textbf{def of } \text{len} \end{aligned}$$

Matt's Proof Strategy Advice™ (2/3)

- Stuck on a proof and…
  - the data type is *not* inductive? Try splitting into cases!
  - the data type *is* inductive? Try structural induction!

Example 2: Echo & Sum

$$\text{echo}(\text{nil}) \quad := \text{nil}$$
$$\text{echo}(x :: L) \quad := x :: x :: \text{echo}(L)$$

- Suppose we have the following code:

```
const y = sum(S);        // S is some List
const R = echo(S);
…
return 2*y;   // = sum(echo(S))
```

  – spec says to return $\text{sum}(\text{echo}(S))$ but code returns $2\ \text{sum}(S)$

- Need to prove that $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$

Example 2: Echo & Sum Base Case (1/2)

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

- **Prove that** $\text{sum}(\text{echo}(\text{S})) = 2\ \text{sum}(\text{S})$ **for any** $\text{S} : \text{List}$

Base Case (nil):

$\text{sum}(\text{echo}(\text{nil})) \quad =$

$= 2\ \text{sum}(\text{nil})$

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

Example 2: Echo & Sum Base Case (2/2)

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

- **Prove that** $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$ **for any** $S : \text{List}$

Base Case (nil):

$$\begin{aligned} \text{sum}(\text{echo}(\text{nil})) &= \text{sum}(\text{nil}) && \textbf{def of } \text{echo} \\ &= 0 && \textbf{def of } \text{sum} \\ &= 2 \cdot 0 \\ &= 2\ \text{sum}(\text{nil}) && \textbf{def of } \text{sum} \end{aligned}$$

Inductive Step $(x :: L)$:

Need to prove that $\text{sum}(\text{echo}(x :: L)) = 2\ \text{sum}(x :: L)$

Get to assume claim holds for L, **i.e., that** $\text{sum}(\text{echo}(L)) = 2\ \text{sum}(L)$

Example 2: Echo & Sum Inductive Step (1/2)

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

- **Prove that** $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{sum}(\text{echo}(L)) = 2\ \text{sum}(L)$

Inductive Step $(x :: L)$**:**

$\text{sum}(\text{echo}(x :: L)) =$

$= 2\ \text{sum}(x :: L)$

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

Example 2: Echo & Sum Inductive Step (2/2)

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

- **Prove that** $\text{sum}(\text{echo}(S)) = 2\ \text{sum}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{sum}(\text{echo}(L)) = 2\ \text{sum}(L)$

Inductive Step $(x :: L)$**:**

$\text{sum}(\text{echo}(x :: L))$	$= \text{sum}(x :: x :: \text{echo}(L))$	**def of** echo
	$= x + \text{sum}(x :: \text{echo}(L))$	**def of** sum
	$= 2x + \text{sum}(\text{echo}(L))$	**def of** sum
	$= 2x + 2\ \text{sum}(L)$	**Ind. Hyp.**
	$= 2(x + \text{sum}(L))$	
	$= 2\ \text{sum}(x :: L)$	**def of** sum

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

Recall: Concatenating Two Lists

- **Mathematical definition of** $\text{concat}(S, R)$

$$\text{concat}(\text{nil}, R) := R$$
$$\text{concat}(x :: L, R) := x :: \text{concat}(L, R)$$

important operation abbreviated as "$\mathbin{+\!\!+}$"

- **Puts all the elements of** L **before those of** R

$$\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil})$$

$= 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil})$	**def of** concat
$= 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil})$	**def of** concat
$= 1 :: 2 :: 3 :: 4 :: \text{nil}$	**def of** concat

Example 3: Length of Concatenated Lists

$$\text{concat}(\text{nil}, R) \quad := \; R$$
$$\text{concat}(x :: L, R) \quad := \; x :: \text{concat}(L, R))$$

important operation abbreviated as "⧺"

- **Suppose we have the following code:**

```
const m = len(S);        // S is some List
const n = len(R);        // R is some List
…
return m + n;   // = len(concat(S, R))
```

  – **spec returns** $\text{len}(\text{concat}(S, R))$ **but code returns** $\text{len}(S) + \text{len}(R)$

- **Need to prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Example 3: Len & Concat Base Case (1/2)

$$\text{concat}(\text{nil}, \text{R}) \quad := \ \text{R}$$
$$\text{concat}(\text{x} :: \text{L}, \text{R}) \quad := \ \text{x} :: \text{concat}(\text{L}, \text{R}))$$

- **Prove that** $\text{len}(\text{concat}(\text{S}, \text{R})) = \text{len}(\text{S}) + \text{len}(\text{R})$
  - **prove by induction on** S
  - **prove the claim for any choice of** R **(i.e.,** R **is a variable)**

Base Case (nil):

$\text{len}(\text{concat}(\text{nil}, \text{R})) =$

$= \text{len}(\text{nil}) + \text{len}(\text{R})$

Example 3: Len & Concat Base Case (2/2)

$$
\begin{aligned}
\mathrm{concat}(\mathrm{nil}, \mathrm{R}) &:= \mathrm{R} \\
\mathrm{concat}(\mathrm{x} :: \mathrm{L}, \mathrm{R}) &:= \mathrm{x} :: \mathrm{concat}(\mathrm{L}, \mathrm{R})
\end{aligned}
$$

- **Prove that** $\mathrm{len}(\mathrm{concat}(\mathrm{S}, \mathrm{R})) = \mathrm{len}(\mathrm{S}) + \mathrm{len}(\mathrm{R})$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

$\mathrm{len}(\mathrm{concat}(\mathrm{nil}, \mathrm{R}))$	$= \mathrm{len}(\mathrm{R})$	**def of** concat
	$= 0 + \mathrm{len}(\mathrm{R})$	
	$= \mathrm{len}(\mathrm{nil}) + \mathrm{len}(\mathrm{R})$	**def of** len

Example 3: Len & Concat Inductive Step (1/3)

$$\text{concat(nil, R)} := \text{R}$$
$$\text{concat(x :: L, R)} := \text{x :: concat(L, R))}$$

- **Prove that** $\text{len(concat(S, R))} = \text{len(S)} + \text{len(R)}$

Inductive Step (x :: L):

Need to prove that

$$\text{len(concat(x :: L, R))} = \text{len(x :: L)} + \text{len(R)}$$

Get to assume claim holds for L, **i.e., that**

$$\text{len(concat(L, R))} = \text{len(L)} + \text{len(R)}$$

Example 3: Len & Concat Inductive Step (2/3)

$$\text{concat}(\text{nil}, R) := R$$
$$\text{concat}(x :: L, R) := x :: \text{concat}(L, R))$$

- **Prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Inductive Hypothesis: assume that $\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$

Inductive Step $(x :: L)$**:**

$$\text{len}(\text{concat}(x :: L, R)) =$$

$$= \text{len}(x :: L) + \text{len}(R)$$

Example 3: Len & Concat Inductive Step (3/3)

$$\begin{aligned} \text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R)) \end{aligned}$$

- **Prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Inductive Hypothesis: assume that $\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$

Inductive Step $(x :: L)$**:**

$\text{len}(\text{concat}(x :: L, R))$	$= \text{len}(x :: \text{concat}(L, R))$	**def of** concat
	$= 1 + \text{len}(\text{concat}(L, R))$	**def of** len
	$= 1 + \text{len}(L) + \text{len}(R)$	**Ind. Hyp.**
	$= \text{len}(x :: L) + \text{len}(R)$	**def of** len

Matt's Proof Strategy Advice™ (3/3)

- Stuck on a proof and…
  - the data type is *not* inductive? Try splitting into cases!
  - the data type *is* inductive? Try structural induction!
- When using structural induction, consider
  - where can the inductive step be used?

    the power of structural induction!
  - which variable should be inducted on?
  - definitions can be applied in *both* directions

Comparing Reasoning vs Testing

```
const concat = (S: List, R: List): List => {
  if (S.kind === "nil") {
    return R;
  } else {
    return cons(S.hd, concat(S.tl, R));
  }
};
```

- Testing: 3 cases
  - loop coverage requires 0, 1, and many recursive calls
- Reasoning: 2 calculations

Structural Induction … Gone Wrong? (1/3)

$$
\begin{aligned}
&\text{allEqual}(\text{nil}) && := \text{true} \\
&\text{allEqual}(x :: \text{nil}) && := \text{true} \\
&\text{allEqual}(x :: y :: L) && := x = y \text{ and allEqual}(y :: L)
\end{aligned}
$$

- **Claim: this function satisfies the above spec**

```
const allEqual(S: List): boolean => {
  return true;
};
```

- **Need to prove that** $\text{allEqual}(S) = \text{true}$

Structural Induction … Gone Wrong? (2/3)

$$\begin{aligned}
\text{allEqual}(\text{nil}) &:= \text{true} \\
\text{allEqual}(x :: \text{nil}) &:= \text{true} \\
\text{allEqual}(x :: y :: L) &:= x = y \text{ and } \text{allEqual}(y :: L)
\end{aligned}$$

Base Case (nil): $\text{allEqual}(\text{nil}) = \text{true}$ — **def of** allEqual

Base Case (x :: nil): $\text{allEqual}(x :: \text{nil}) = \text{true}$ — **def of** allEqual

Now, what if we got a bit sloppy?

Inductive Hypothesis: assume that $\text{allEqual}(S) = \text{true}$ **for lists** S

Inductive Step (x :: y :: L):

$y :: L$ is a list – so, $\text{allEqual}(y :: L) = \text{true}$	**inductive hypothesis**
$x :: y :: \text{nil}$ is a list – so $\text{allEqual}(x :: y :: \text{nil}) = \text{true}$	**inductive hypothesis**
thus, $x = y$	**definition of** allEqual
$\text{allEqual}(x :: y :: L) = \text{true}$	**definition of** allEqual

Structural Induction … Gone Wrong? (3/3)

$\text{allEqual}(\text{nil}) := \text{true}$

$\text{allEqual}(x :: \text{nil}) := \text{true}$

$\text{allEqual}(x :: y :: L) := x = y \text{ and allEqual}(y :: L)$

Base Case (nil): $\text{allEqual}(\text{nil}) = \text{true}$ **def of** allEqual

Base Case (x :: nil): $\text{allEqual}(x :: \text{nil}) = \text{true}$ **def of** allEqual

Inductive Hypothesis: assume that $\text{allEqual}(L) = \text{true}$ **only applies to** L

Inductive Step (x :: y :: L):

y :: L is a list – so, $\text{allEqual}(y :: L) = \text{true}$ **not true!**

x :: y :: nil is a list – so $\text{allEqual}(x :: y :: \text{nil}) = \text{true}$ **not true!**

thus, $x = y$ **not true!**

$\text{allEqual}(x :: y :: L) = \text{true}$ **not true!**

Example 4: Faster Sum

$$\begin{aligned} \text{sum-acc}(\text{nil}, r) &:= r \\ \text{sum-acc}(x :: L, r) &:= \text{sum-acc}(L, x + r) \end{aligned}$$

linear time

- Suppose we have the following code:

```
const s = sum_acc(S, 0);        // S is some List
...
return s;  // = sum(S)
```

  – spec says to return $\text{sum}(S)$ but code returns $\text{sum-acc}(S, 0)$

- Need to prove that $\text{sum-acc}(S, 0) = \text{sum}(S)$
  – will prove, more generally, that $\text{sum-acc}(S, r) = \text{sum}(S) + r$

Example 4: Faster Sum Base Case (1/2)

$$\begin{aligned} \text{sum-acc}(\text{nil}, \text{r}) &:= \text{r} \\ \text{sum-acc}(\text{x} :: \text{L}, \text{r}) &:= \text{sum-acc}(\text{L}, \text{x} + \text{r}) \end{aligned}$$

- **Prove that** $\text{sum-acc}(\text{S}, \text{r}) = \text{sum}(\text{S}) + \text{r}$
  - **prove by induction on** S
  - **prove the claim for any choice of** r **(i.e.,** r **is a variable)**

Base Case (nil):

$$\text{sum-acc}(\text{nil}, \text{r}) =$$

$$= \text{sum}(\text{nil}) + \text{r}$$

Example 4: Faster Sum Base Case (2/2)

$$\text{sum-acc}(\text{nil}, \text{r}) \quad := \text{r}$$
$$\text{sum-acc}(\text{x} :: \text{L}, \text{r}) \quad := \text{sum-acc}(\text{L}, \text{x} + \text{r})$$

- **Prove that** $\text{sum-acc}(\text{S}, \text{r}) = \text{sum}(\text{S}) + \text{r}$
  - **prove by induction on** S
  - **prove the claim for any choice of** r **(i.e.,** r **is a variable)**

Base Case (nil):

$\text{sum-acc}(\text{nil}, \text{r})$	$= \text{r}$	**def of** sum-acc
	$= 0 + \text{r}$	
	$= \text{sum}(\text{nil}) + \text{r}$	**def of** sum

Example 4: Faster Sum Inductive Step (1/3)

$$\text{sum-acc}(\text{nil}, r) := r$$
$$\text{sum-acc}(x :: L, r) := \text{sum-acc}(L, x + r)$$

- **Prove that** $\text{sum-acc}(S, r) = \text{sum}(S) + r$

Inductive Step $(x :: L)$:

Need to prove that

$$\text{sum-acc}(x :: L, r) = \text{sum}(x :: L) + r$$

Get to assume claim holds for L, i.e., that

$$\text{sum-acc}(L, r) = \text{sum}(L) + r \qquad \textbf{holds for any } r$$

Example 4: Faster Sum Inductive Step (2/3)

$$\text{sum-acc}(\text{nil}, r) := r$$
$$\text{sum-acc}(x :: L, r) := \text{sum-acc}(L, x + r)$$

- **Prove that** $\text{sum-acc}(S, r) = \text{sum}(S) + r$

Inductive Hypothesis: assume that $\text{sum-acc}(L, r) = \text{sum}(L) + r$

Inductive Step $(x :: L)$**:**

$$\text{sum-acc}(x :: L, r) =$$

$$= \text{sum}(x :: L) + r$$

Example 4: Faster Sum Inductive Step (3/3)

$$\begin{aligned} \text{sum-acc}(\text{nil}, \text{r}) &:= \text{r} \\ \text{sum-acc}(\text{x} :: \text{L}, \text{r}) &:= \text{sum-acc}(\text{L}, \text{x} + \text{r}) \end{aligned}$$

- **Prove that** $\text{sum-acc}(\text{S}, \text{r}) = \text{sum}(\text{S}) + \text{r}$

Inductive Hypothesis: assume that $\text{sum-acc}(\text{L}, \text{r}) = \text{sum}(\text{L}) + \text{r}$

Inductive Step **(**$\text{x} :: \text{L}$**):**

$$\begin{aligned} \text{sum-acc}(\text{x} :: \text{L}, \text{r}) &= \text{sum-acc}(\text{L}, \text{x} + \text{r}) && \textbf{def of } \text{sum-acc} \\ &= \text{sum}(\text{L}) + \text{x} + \text{r} && \textbf{Ind. Hyp.} \\ &= \text{x} + \text{sum}(\text{L}) + \text{r} && \\ &= \text{sum}(\text{x} :: \text{L}) + \text{r} && \textbf{def of } \text{sum} \end{aligned}$$