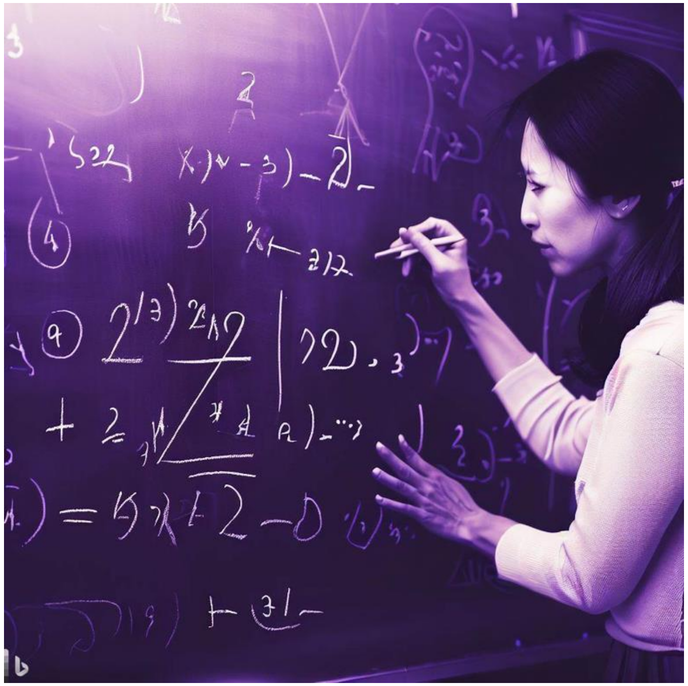


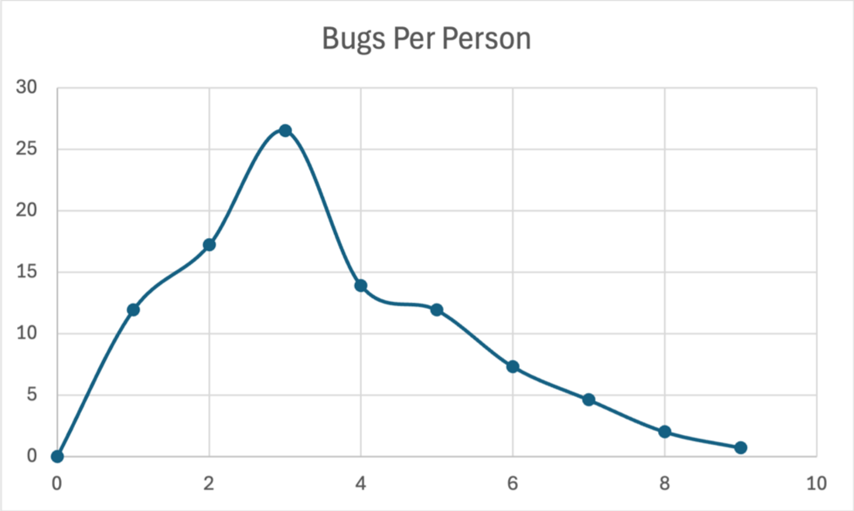
CSE 331

Reasoning

James Wilcox and Kevin Zatloukal

Summary of HW3

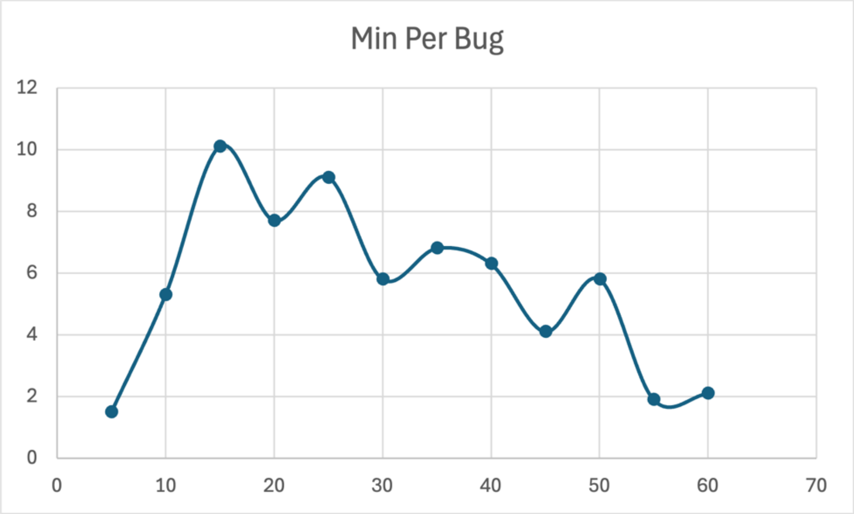
- Number of bugs logged:
  - average of 3.9 bugs per person



- Average solution was 136 lines of code (over-estimate)
  - 1 bug every 35 lines of code
  - 1 bug per 20–70 is normal even for professionals

Summary of HW3

- Time spent per bug:
  - average of 65 minutes per bug
  - 34% more than 1 hour



  - clearly a long tail to this distribution

    some bugs take a very long time to find

Summary of HW3

- **How many functions were searched**
  - **60% of bugs searched more than one function**
  - **time require for debugging**

1-2 functions	48 mins
3-4 functions	96 mins
5-6 functions	151 min

  - **on average, every extra function meant 30 more mins**

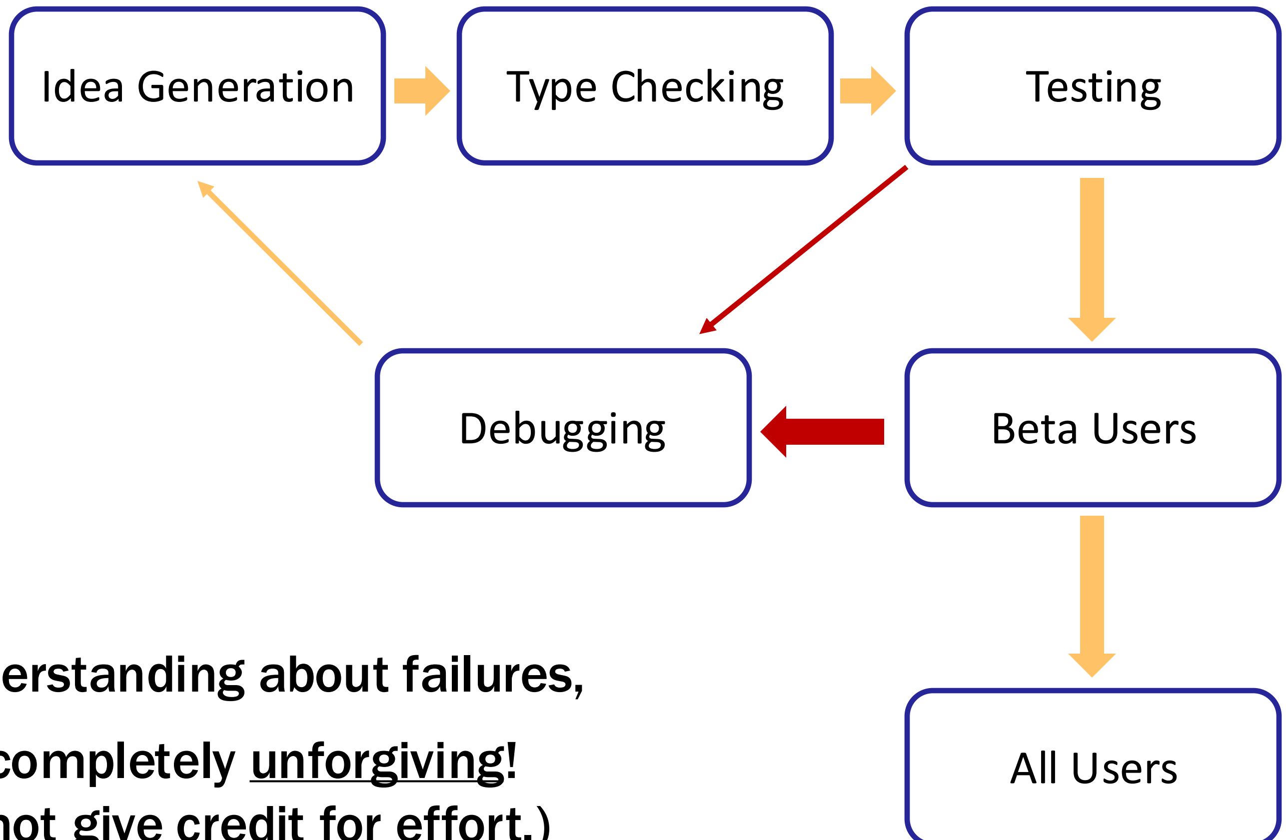
- **Shrinking the search space helps a lot**
  - **defensive programming**
  - **unit tests**
  - **run-time type checking of request/responses**

    wrong types guarantees the failure is in a different function

Software Development Process

Software Development Process

Given: a problem description (in English)



Beta users are understanding about failures,

Regular users are completely <u>unforgiving</u>!
(Regular users do not give credit for effort.)

How Much Debugging?

- Bugs typed in... 1 per 20 lines
  - we saw 11 lines in HW1
  - should get to 20 lines when more familiar with setting

- Bugs after type checking... 1 per 40 lines
  - assume 50% caught by type checker (saw 41% in HW1)
  - matches industry estimates of 20-70 lines per bug

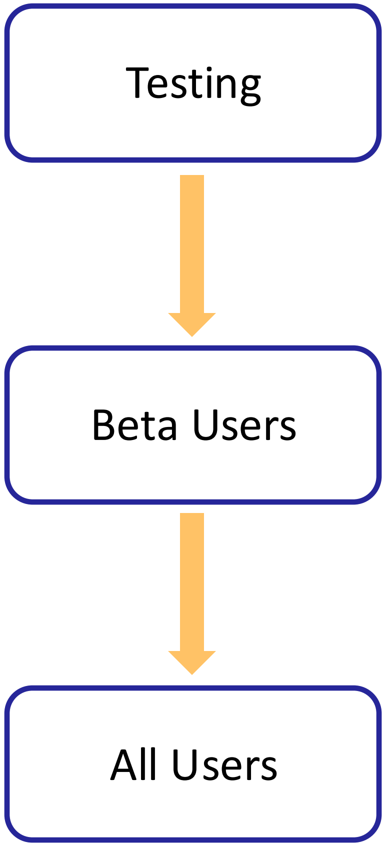
- Bugs after unit testing... 1 per 133 lines
  - assume 70% caught by unit testing

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

How Much Debugging?

- 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 are at the seams between functions
  - multiple functions need to be debugged
  - will take a long time to track down (many hours)

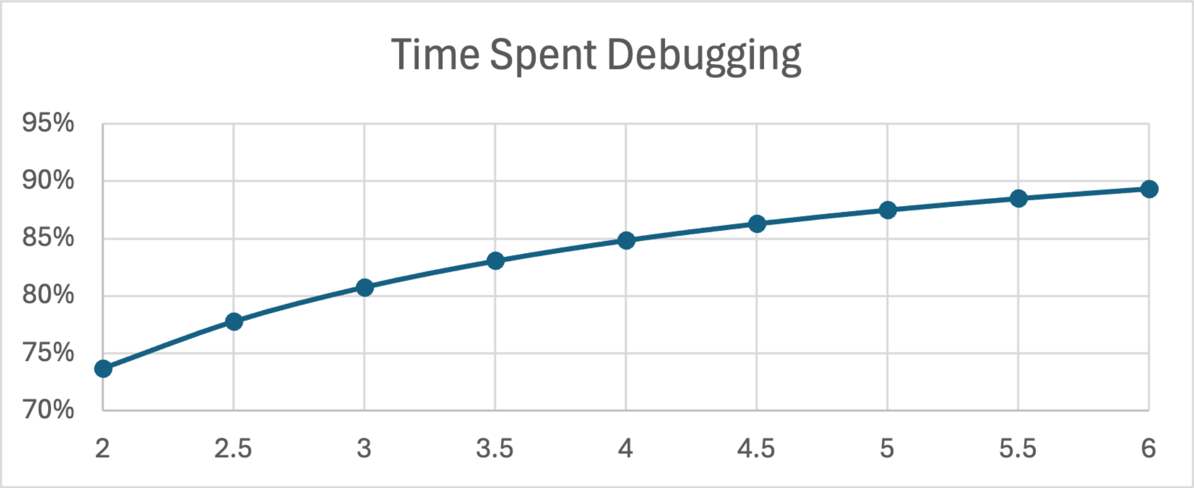
    we saw an extra 30 minutes for every additional function in HW3

    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**

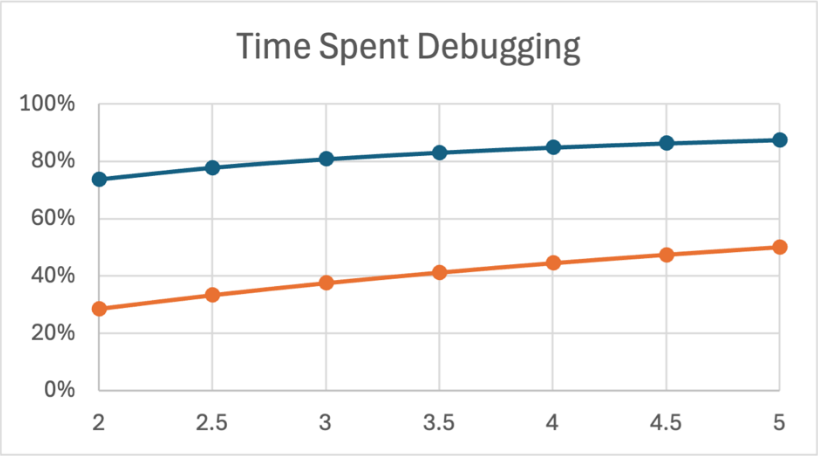
5 hours	typing & fixing type errors
5 hours	testing & fixing *unit* test failures
14h hours	debugging & fixing bugs



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
3h hours	debugging & fixing bugs



even at h=5, debugging
not the majority of time

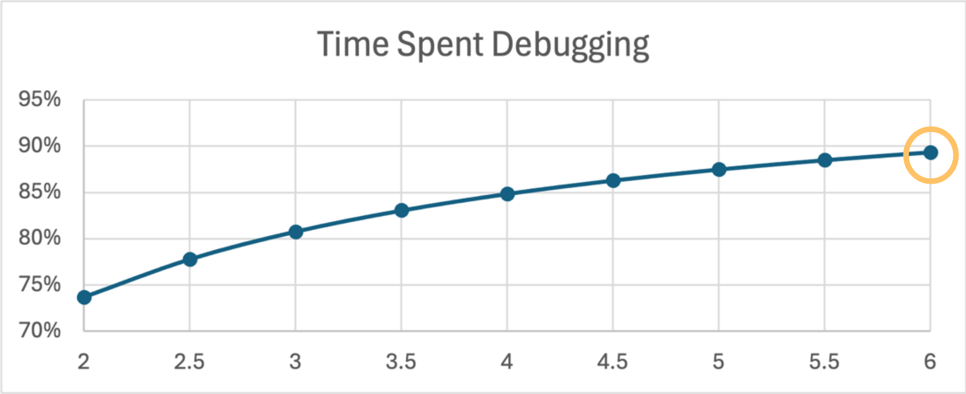
bottom programmer is
2-3 times more productive

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

“Engineers are paid to think and understand.”

— Class slogan #1

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 ("=")

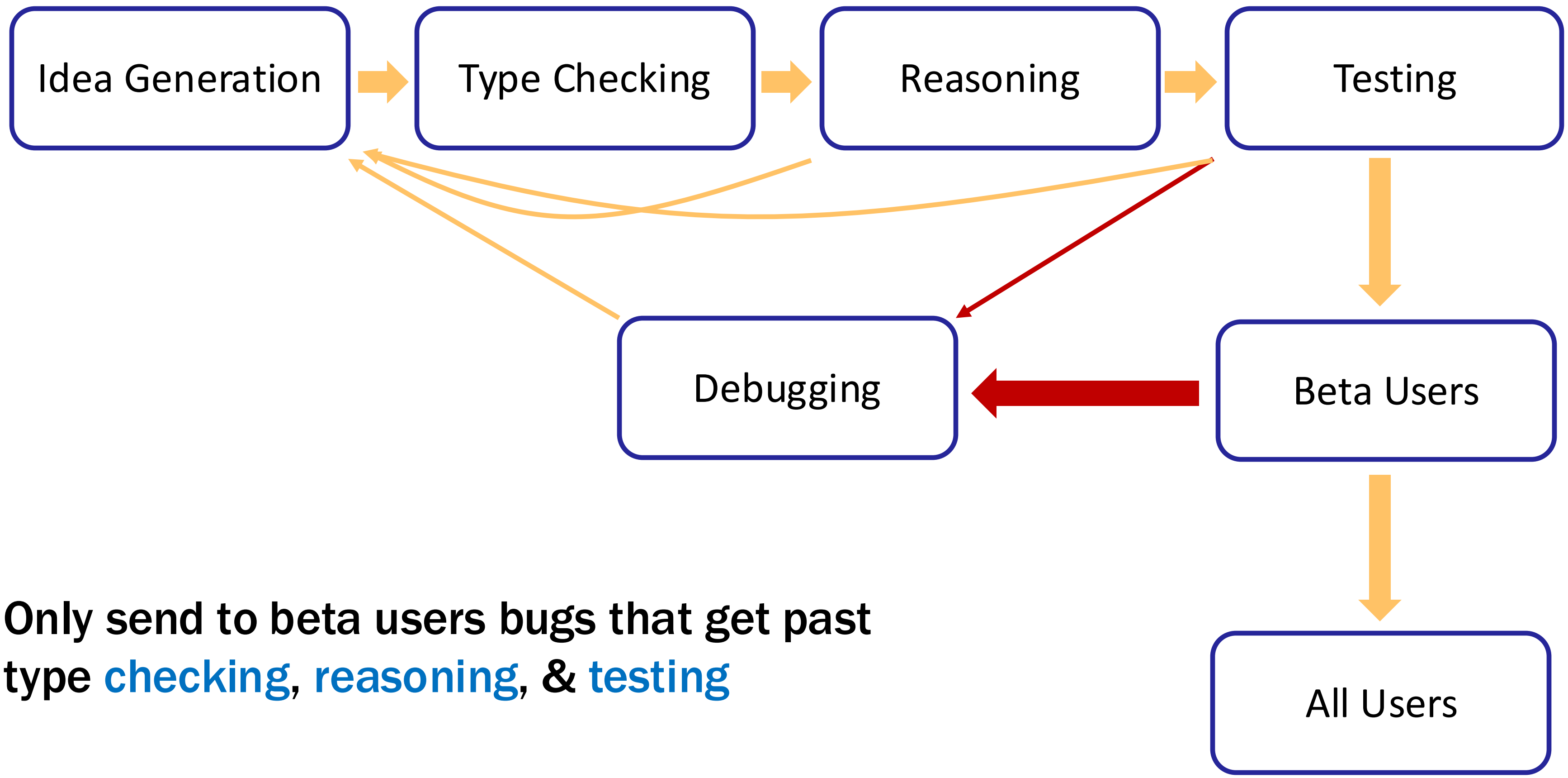
Technique	All Inputs	Fully Correct
Type Checker	✓	✗
Testing	✗	✓
Reasoning	✓	✓

- 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

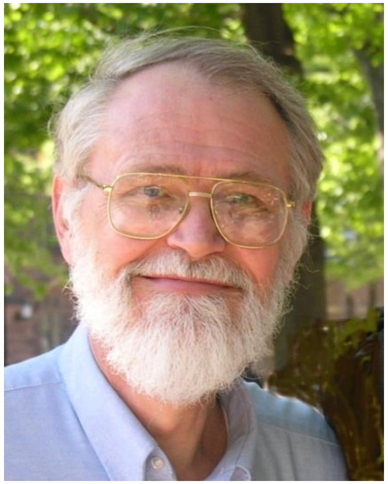
Review: Software Development Process

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!)

Unlikely to be Automated

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

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

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

    probably easier to rewrite it yourself
  - studies show far show little / no productivity improvement so far

    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

- AI is especially bad at reasoning
  - e.g., bad at learning formal properties
  - e.g., bad at catching rare cases

“These models have read every piece of code on Github, every StackOverflow question answer, every programming book, every tweet about coding, transcripts of every YouTube walkthrough and they still can’t code as well as I can in every situation.”

— Nat Friedman (former GitHub CEO)

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 steppingstone to informal reasoning (same core ideas)
    - formal reasoning still needed for the **hardest** problems

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

- 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)

Specifications in TypeScript

- 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`

Reasoning

Reasoning

- “Thinking through” what the code does on **all** 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 steppingstone to informal reasoning (same core ideas)
    - formal reasoning still needed for the **hardest** problems
- Definition of correctness comes from the specification…

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

- 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 <u>path</u> 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

- 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

```
// 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

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 <u>path</u> from top to return
- Those elements cover <u>all</u> 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
  - structural induction

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

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

$$x + z \;=\; y + z \;\le\; y + 10$$

since $x = y$

since $z \le 10$

<u>All together</u>, this tells us that $x + z \le y + 10$

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

$$
\begin{aligned}
x + z \quad &= y + z && \text{since } x = y \\
&\leq y + 10 && \text{since } z \leq 10
\end{aligned}
$$

  - 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 <u>previous</u> line

Calculation Blocks

- Chain of "=" shows first = last

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

  - proves that $a = d$
  - all 4 of these are the same number

Calculation Blocks

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

$$
\begin{array}{lll}
x + z & = y + z & \textbf{since } x = y \\
 & \leq y + 10 & \textbf{since } z \leq 10 \\
 & = y + 3 + 7 & \\
 & \leq w + 7 & \textbf{since } y + 3 \leq w
\end{array}
$$

  - **each number is equal or strictly larger that previous**

    last number is strictly larger than the first number

  - **analogous for "≥"**

Using Calculation to Prove Correctness

```
// 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

$$x + y$$

Using Calculation to Prove Correctness

```
// 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 $x + y \geq 1$

Using Calculation to Prove Correctness

```
// 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

$$x + y$$

Using Calculation to Prove Correctness

```
// 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 $x \geq 9$
	$= 1$	

Using Calculation to Prove Correctness

```
// 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

Using Calculation to Prove Correctness

```
// 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

$$x + y$$

Using Calculation to Prove Correctness

```
// 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

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

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:**

$$\mathrm{sum}(\mathrm{nil}) \quad := \quad 0$$
$$\mathrm{sum}(x :: L) \quad := \quad x + \mathrm{sum}(L)$$

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

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

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 "$L = cons(...)$"
- Must prove that $sum(L) \geq 0$

Using Definitions in Calculations

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

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

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

Using Definitions in Calculations

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

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

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

Proving Correctness with Conditionals

```
// 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: "$y \leq -1$"

$$x + y$$

Proving Correctness with Conditionals

```
// 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: "$y \leq -1$"

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

Proving Correctness with Conditionals

```
// 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: "$y \geq 0$"

$x - 1$

Proving Correctness with Conditionals

```
// 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

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

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

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

Case $m \geq 0$:

$f(m) =$

$> m$

Proof By Cases

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

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

Case $m \geq 0$:

$$\begin{aligned} f(m) &= 2m + 1 && \textbf{def of } f \textbf{ (since } m \geq 0\textbf{)} \\ &\geq m + 1 && \textbf{since } m \geq 0 \\ &> m && \textbf{since } 1 > 0 \end{aligned}$$

Proof By Cases

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

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

Case $m \geq 0$:

$$f(m) = \ldots > m$$

Case $m < 0$:

$$f(m) = 0 \qquad \textbf{def of } f \textbf{ (since } m < 0\textbf{)}$$
$$> m \qquad \textbf{since } m < 0$$

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

HW4–6

- **In HW1–3, you**
  - **learned the structure of modern applications (UIs & servers)**

    will be useful to know for just about any programming job
  - **experience what happens when bugs appear as failures**

    lots of **debugging**

- **In HW4–6, you**
  - **will learn how to ensure code is correct before you run it**
  - **experience what it is like not allow bugs to become failures**
  - **each HW is split into written and coding**

    goal is to do the **thinking** to ensure it works the first time

    (give you the opportunity to fix it up if you do make mistakes)

Recall Correctness with No Mutation

- Proving implications is the **core step** of reasoning
  - other techniques output implications for us to prove
- Core technique is "proof by calculation"
- Other techniques we will need:
  - proof by cases
  - structural induction

Structural Induction

Proof by Calculation

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

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

- For example…

Example: Repeating List Elements

- **Consider the following function:**

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

- **Produces a list where every element is repeated twice**

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

Example: Repeating List Elements

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

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

Proof by Calculation

- **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

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

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

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

- **use any known facts and definitions**

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

- 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

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 $\mathrm{P}(x :: L)$

- use known facts and definitions and Inductive Hypothesis

Why This Works

With Structural Induction, we prove two facts

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

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

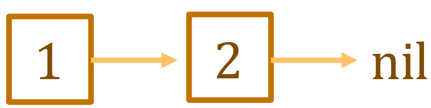
(**second assuming** $\mathrm{len}(\mathrm{echo}(\mathrm{L})) = 2\ \mathrm{len}(\mathrm{L})$)

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

Why This Works

Build up an object using constructors:

nil	**first constructor**
2 :: nil	**second constructor**
1 :: 2 :: nil	**second constructor**



nil **already exists when building** 2 :: nil

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

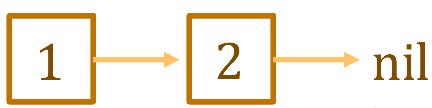
Why This Works

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

$P(\text{nil})$ $\quad\quad$ $\text{len}(\text{echo}(\text{nil})) = 2\ \text{len}(\text{nil})$

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

(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: Repeating List Elements

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

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

Base Case (nil):

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

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

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

Example: Repeating List Elements

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

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

Base Case (nil):

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

Example: Repeating List Elements

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

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

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

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

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

Example: Repeating List Elements

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

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

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

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

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

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

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

Example: Repeating List Elements

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

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

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

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

$\mathrm{len}(\mathrm{echo}(x :: L))$	$= \mathrm{len}(x :: x :: \mathrm{echo}(L))$	**def of** echo
	$= 1 + \mathrm{len}(x :: \mathrm{echo}(L))$	**def of** len
	$= 2 + \mathrm{len}(\mathrm{echo}(L))$	**def of** len
	$= 2 + 2\ \mathrm{len}(L)$	**Ind. Hyp.**
	$= 2(1 + \mathrm{len}(L))$	
	$= 2\ \mathrm{len}(x :: L)$	**def of** len

Example 2: Repeating List Elements

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

- **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** $\mathrm{sum}(\mathrm{echo}(S))$ **but code returns** $2\ \mathrm{sum}(S)$

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

Example 2: Repeating List Elements

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

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

Base Case (nil):

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

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

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

Example 2: Repeating List Elements

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

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

Base Case (nil):

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

Inductive Step (x :: L):

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

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

Example 2: Repeating List Elements

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

- **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: Repeating List Elements

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

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

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

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

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

Recall: Concatenating Two Lists

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

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

important operation abbreviated as "$+\!\!+$"

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

$$
\begin{aligned}
&\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil}) \\
&\quad = 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil}) && \textbf{def of } \text{concat} \\
&\quad = 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil}) && \textbf{def of } \text{concat} \\
&\quad = 1 :: 2 :: 3 :: 4 :: \text{nil} && \textbf{def of } \text{concat}
\end{aligned}
$$

Example 3: Length of Concatenated Lists

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

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 $\mathrm{len}(\mathrm{concat}(S, R))$ but code returns $\mathrm{len}(S) + \mathrm{len}(R)$

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

Example 3: Length of Concatenated Lists

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

- **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: Length of Concatenated Lists

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

- **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):

$$\begin{aligned} \text{len}(\text{concat}(\text{nil}, \text{R})) &= \text{len}(\text{R}) && \textbf{def of } \text{concat} \\ &= 0 + \text{len}(\text{R}) \\ &= \text{len}(\text{nil}) + \text{len}(\text{R}) && \textbf{def of } \text{len} \end{aligned}$$

Example 3: Length of Concatenated Lists

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

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

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

Need to prove that

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

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

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

Example 3: Length of Concatenated Lists

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

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

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

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

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

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

Example 3: Length of Concatenated Lists

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

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

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

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

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

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

Example: 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

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

- **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):

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

Example 4: Faster Sum

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

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

Base Case (nil):

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

Example 4: Faster Sum

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

- **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

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

- **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

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

- **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 $(x :: L)$**:**

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