

CSE 331

Reasoning About Functional Code

James Wilcox and Kevin Zatloukal

Recall: Code Without Mutation

- Our math notation includes only...
 - expressions
 - conditionals (pattern matching & side conditions)
 - recursion
- This is all we need, mathematically
 - can write anything computable with just these
- Can do these in Java as well...

Java Code Without Mutation

- Code without mutation consists of...
 - straight-line code variable declarations and `return`
 - conditionals `if` statements
 - recursion
- Code that only uses these properties is "functional"
 - we will limit ourselves to functional code *initially*
- Can translate any of our math functions into this
- Data is not so easy
 - compound types need to be faked with classes

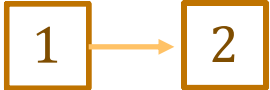
Inductive Data Types in Java

- Java does not natively support these data types
other languages support these as well as tuples and records
- Can fake them with classes, e.g.:

```
public class List {  
    final int hd;  
    final List tl;  
  
    public static final List nil = null;  
  
    public static List cons(int hd, List tl) {  
        return new List(hd, tl);  
    }  
  
    private List(int hd, List tl) {  
        this.hd = hd;  
        this.tl = tl;  
    }  
}
```

type List := nil | cons(hd : \mathbb{Z} , tl : List)

cons(1, cons(2, nil));



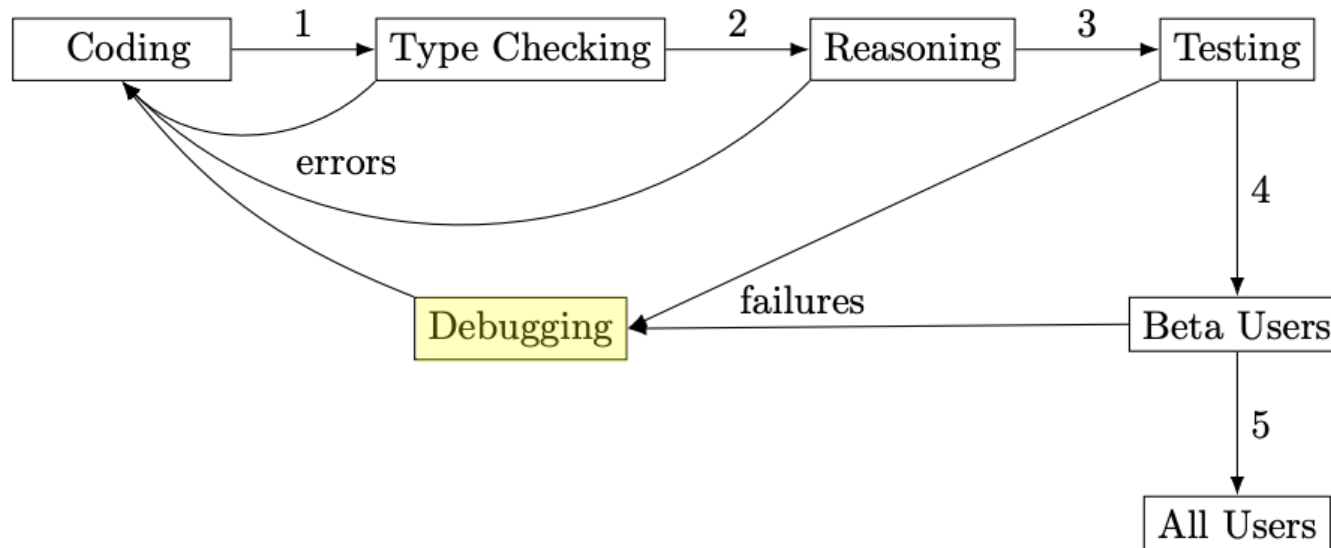
```
graph LR; n1[1] --> n2[2];
```

Recall: Software Development Process



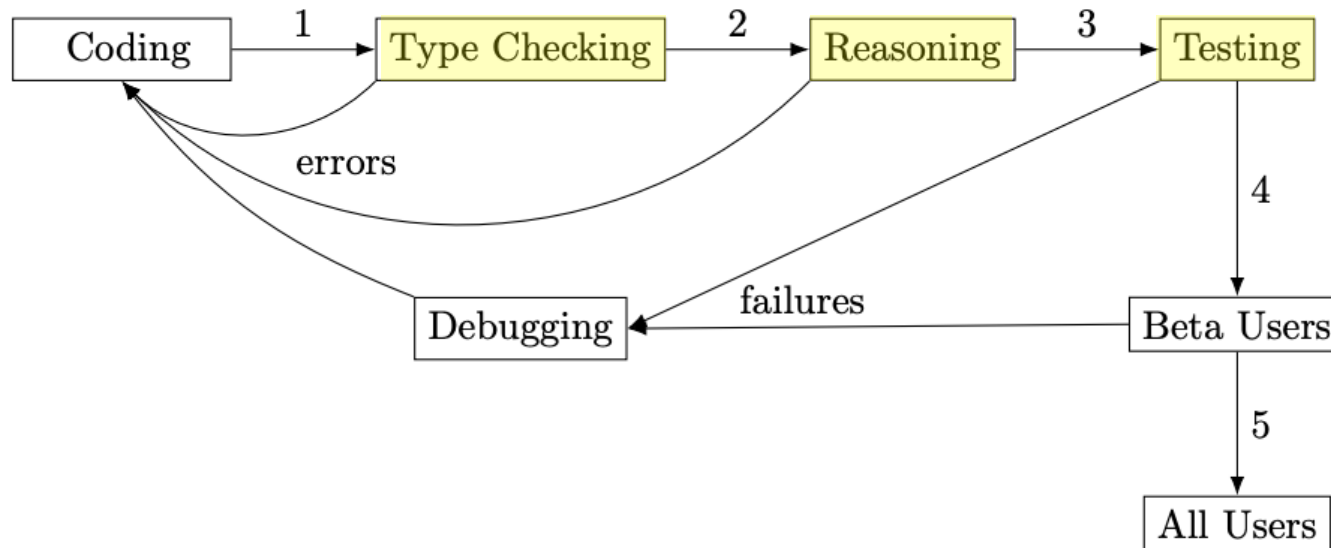
- an imperative spec comes with a plan
can translate this directly into Java code
- a declarative spec does not

Software Implementation



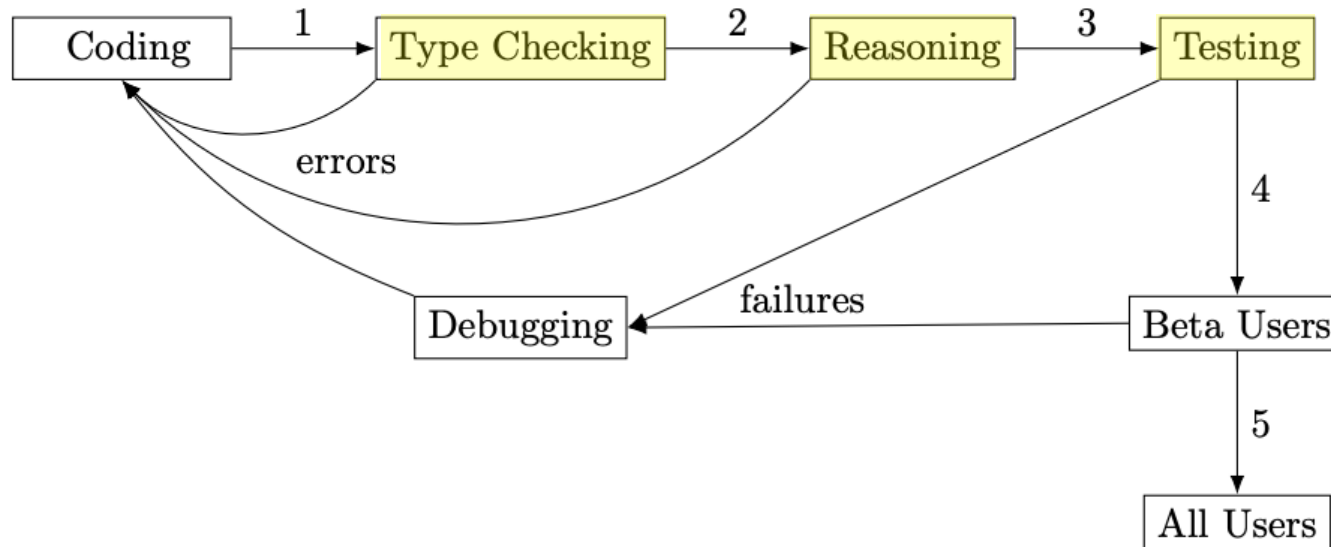
- **Debugging** is the search from failure to bug
 - harder the more code that must be searched
 - easiest possible case is a unit test failure
 - time required is *random* with a *long tail*

Software Implementation



- Three techniques used to check correctness
 - each removes $\sim 2/3^{\text{rd}}$ of the bugs present
 - but each tends to find *different* bugs
 - need all three techniques to get 99+% assurance

Software Implementation



- Three techniques used to check correctness
 - type checking is familiar (and more coming later)
 - already discussed testing
 - focus now on reasoning

Reasoning

- “Thinking through” what the code does on all inputs
 - ensuring what it does is correct in *all* cases
- Type checking does **not** do this
 - only checks that return values have the right type
 - e.g., ensures that an `int` but not that it is `1`
- Testing does **not** do this
 - only verifies a correct output on *some* inputs

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...

Correctness 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

Facts

- Starting point for reasoning is “facts” about the code
 - things we know to be true about the variables
 - these hold for all inputs (no matter what value the variable has)
 - typically, “=” or “ \leq ”

```
// @param n a natural number
int f(int n) {
    final int m = 2 * n;
    return (m + 1) * (m - 1);
};
```

find facts by reading along path
from top to return statement

- At the return statement, we know these facts:
 - $n \geq 0$
 - $m = 2n$

note: these hold for all valid inputs

Facts

- Starting point for reasoning is “facts” about the code
 - things we know to be true about the variables
 - these hold for all inputs (no matter what value the variable has)
 - typically, “=” or “ \leq ”

```
// @param n a natural number
int f(int n) {
    final int m = 2 * n;
    return (m + 1) * (m - 1);
};
```

- No need to include the fact that n is an integer ($n : \mathbb{Z}$)
 - 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 can be any integers.  
// Returns a non-negative integer.  
int f(int a, int b) {  
    final List L = cons(a, cons(b, nil));  
    if (a >= 0 && b >= 0)  
        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 “ $L = \text{cons}(\dots)$ ”
- 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**”
- **We will see how to do this shortly...**
 - will be familiar to those who have taken 311

Checking Correctness of Functional Code

- Steps for checking correctness of functional code:
 1. Collect facts at each return statement
 2. Ensure those facts **imply** each fact of the **postcondition**
- Checking correctness requires proving **implications**
 - need to prove facts about the **return** values
 - return values must satisfy the facts of the **postcondition**
- If the known facts do not imply the postcondition, then the code is **wrong**
 - some valid input does not satisfy the postcondition
 - the code will be correct in 99% of our examples,
 - but this is the reason why we do reasoning: to find mistakes

Collecting Facts

- Saw how to collect facts in code consisting of
 - "final" 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	HW2
local variable mutation	“	“	Floyd logic	HW4
heap state mutation	“	“	rep invariants	HW6
array mutation		“	for-any facts	HW8

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 $\text{len}(\text{nil}) = 0$

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

$$\begin{array}{c} x + z = y + z \leq y + 10 \\ \underbrace{\hspace{1.5cm}} \quad \underbrace{\hspace{1.5cm}} \\ \text{since } x = y \quad \text{since } z \leq 10 \end{array}$$

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

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

$x + z$	
$= y + z$	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

- 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 “ \leq ” shows first \leq last

$$x + z$$

$$= y + z$$

$$\text{since } x = y$$

$$\leq y + 10$$

$$\text{since } z \leq 10$$

$$= y + 3 + 7$$

$$\leq w + 7$$

$$\text{since } y + 3 \leq w$$

- each number is equal or strictly larger than previous
last number is strictly larger than the first number
- analogous for “ \geq ”

Using Calculation to Prove Correctness

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

- **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 with x >= 9 and y >= -8
// Returns a positive integer.
int f(int x, int y) => {
    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 with x >= 9 and y >= -8
// Returns a positive integer.
int f(int x, int y) => {
    return x + y;
};
```

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

$$\begin{array}{ll} x + y & \\ \geq x + -8 & \text{since } y \geq -8 \\ \geq 9 - 8 & \text{since } x \geq 9 \\ = 1 & \end{array}$$

Using Calculation to Prove Correctness

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

- Known facts “ $x > 8$ ” and “ $y > -9$ ”
- Correct if the return value is a positive integer

$$\begin{aligned} x + y &> x + -9 && \text{since } y > -9 \\ &> 8 - 9 && \text{since } x > 8 \\ &= -1 \end{aligned}$$

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

Using Calculation to Prove Correctness

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

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

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

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.  
int f(int a, int b) {  
    final List L = cons(a, cons(b, nil));  
    if (a >= 0 && b >= 0)  
        return sum(L);  
    ...  
}
```

find facts by reading along path
from top to return statement

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

Using Definitions in Calculations

$\text{sum}(\text{nil}) \quad := \quad 0$

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

- **Know** “ $a \geq 0$ ”, “ $b \geq 0$ ”, and “ $L = a :: b :: \text{nil}$ ”
- **Prove** that $\text{sum}(L) \geq 0$

$\text{sum}(L)$

Using Definitions in Calculations

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

- **Know “ $a \geq 0$ ”, “ $b \geq 0$ ”, and “ $L = a :: b :: \text{nil}$ ”**
- **Prove that $\text{sum}(L) \geq 0$**

$\text{sum}(L)$	
$= \text{sum}(a :: b :: \text{nil})$	since $L = a :: b :: \text{nil}$
$= a + \text{sum}(b :: \text{nil})$	def of sum
$= a + b + \text{sum}(\text{nil})$	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.  
int f(int x, int y) {  
    if (y < 0) {  
        return x + y;  
    } else {  
        return x - 1;  
    }  
};
```

- 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.  
int f(int x, int y) {  
    if (y < 0) {  
        return x + y;  
    } else {  
        return x - 1;  
    }  
};
```

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

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

Proving Correctness with Conditionals

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

- 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.  
int f(int x, int y) {  
    if (y < 0) {  
        return x + y;  
    } else {  
        return x - 1;  
    }  
};
```

- 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 each of them holds

```
// Inputs x and y are integers with  $x < y - 1$   
// Returns a number less than y and greater than x.  
int f(int x, int y) { .. };
```

- 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
int max(int a, int b) {
    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 $r = b$
 - prove an "or" holds by proving one of the two options holds

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)
 - can pick any cases we want, not just cases in the code

Example Proof By Cases

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

$$f(m) := 2m + 1 \quad \text{if } m \geq 0$$

$$f(m) := 0 \quad \text{if } m < 0$$

- 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

$$f(m) := 2m + 1 \quad \text{if } m \geq 0$$

$$f(m) := 0 \quad \text{if } m < 0$$

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

Case $m \geq 0$:

$$f(m) =$$

$$> m$$

Proof By Cases

$$f(m) := 2m + 1$$

$$f(m) := 0$$

if $m \geq 0$

if $m < 0$

- Prove that $f(m) > m$

Case $m \geq 0$:

$$f(m) = 2m + 1$$

$$= m + m + 1$$

$$\geq m + 1$$

$$> m$$

side condition requires a "since"

def of f (since $m \geq 0$)

since $m \geq 0$

since $1 > 0$

Proof By Cases

$$f(m) := 2m + 1 \quad \text{if } m \geq 0$$

$$f(m) := 0 \quad \text{if } m < 0$$

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

Case $m \geq 0$:

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

Case $m < 0$:

$$f(m) = 0$$

$$> m$$

def of f (since $m < 0$)

since $m < 0$

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

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 break down into calculations also
- 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., $\text{sum}(a :: b :: \text{nil}) \geq 0$
- Would like to prove facts about any length list L
- For example...

Example: Repeating List Elements

- Consider the following function:

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

- Produces a list where every element is repeated twice

$\text{echo}(1 :: 2 :: \text{nil})$
 $= 1 :: 1 :: \text{echo}(2 :: \text{nil})$
 $= 1 :: 1 :: 2 :: 2 :: \text{echo}(\text{nil})$
 $= 1 :: 1 :: 2 :: 2 :: \text{nil}$

def of echo
def of echo
def of echo

Example: Repeating List Elements

```
echo(nil)      := nil
echo(x :: L)   := x :: x :: echo(L)
```

- Suppose we have the following code:

```
final int m = len(S);      // S is some List
final List 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., $\text{sum}(a :: b :: \text{nil}) \geq 0$
- Would like to prove facts about any length list L
- Need more tools for this...
 - **structural recursion** *calculates* on inductive types
 - **structural induction** *reasons* about inductive typesboth tools are specific to **inductive types**

Structural Induction

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

To prove $P(S)$ holds for any list S , prove two implications

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

- use any known facts and definitions

Inductive Step: prove $P(x :: L)$

- x and L are variables
- use any known facts and definitions plus one more fact...
- make use of the fact that L is also a List

Structural Induction

To prove $P(S)$ holds for any list S , prove two implications

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

- 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 $P(x :: L)$

- use known facts and definitions and Inductive Hypothesis

Why This Works

With Structural Induction, we prove two facts

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

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

Why This Works

Build up an object using constructors:

nil

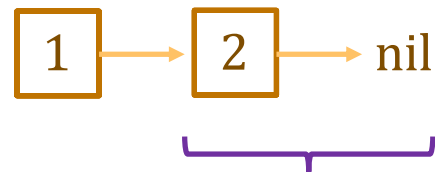
2 :: nil

1 :: 2 :: nil

first constructor

second constructor

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

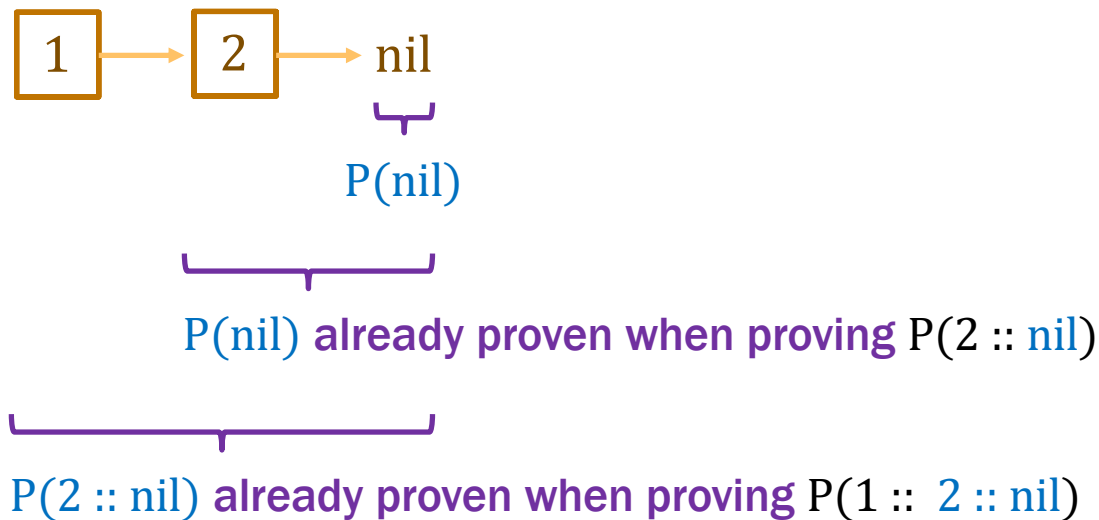
$P(\text{nil})$

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

$P(x :: L)$

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

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



Example: Repeating List Elements

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

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

Base Case (nil):

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

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

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

$\text{len}(\text{nil}) \quad := 0$

$\text{len}(x :: L) \quad := 1 + \text{len}(L)$

Example: Repeating List Elements

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

- **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: Repeating List Elements

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

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

Inductive Step $(x :: L)$:

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

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

Example: Repeating List Elements

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

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

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

Inductive Step $(x :: L)$:

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

$\text{len}(\text{nil}) \quad := 0$
 $\text{len}(x :: L) \quad := 1 + \text{len}(L)$

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

Example: Repeating List Elements

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

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

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

Inductive Step $(x :: L)$:

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

$\text{len}(\text{nil}) \quad := 0$

$\text{len}(x :: L) \quad := 1 + \text{len}(L)$

Structural Induction in General

- General case: assume P holds for constructor *arguments*

$\text{type } T := A \mid B(x : \mathbb{Z}) \mid C(y : \mathbb{Z}, t : T) \mid D(z : \mathbb{Z}, u : T, v : T)$

- To prove $P(t)$ for any t , we need to prove:
 - $P(A)$
 - $P(B(x))$ for any $x : \mathbb{Z}$
 - $P(C(y, t))$ for any $y : \mathbb{Z}$ and $t : T$ **assuming** $P(t)$ is true
 - $P(D(z, u, v))$ for any $z : \mathbb{Z}$ and $u, v : T$ **assuming** $P(u)$ and $P(v)$
- These four facts are enough to prove $P(t)$ for any t
 - for each constructor, have proof that it produces an object satisfying P

Structural Induction in General

- Each inductive type has its own form of induction
 - special way to reason about that type
- Lists are defined like this:

`type List := nil | cons(x : \mathbb{Z} , L : List)`

- To prove $P(S)$ for any list S , we need to prove:
 - $P(\text{nil})$
 - $P(x :: L)$ for any $x : \mathbb{Z}$ and $L : \text{List}$ **assuming** $P(L)$ is true

Example 2: Repeating List Elements

```
echo(nil)      := nil
echo(x :: L)   := x :: x :: echo(L)
```

- Suppose we have the following code:

```
final int y = sum(S);           // S is some List
final List 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: Repeating List Elements

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

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

Base Case (nil):

$$\begin{aligned} \text{sum}(\text{echo}(\text{nil})) &= \\ &= 2 \text{sum}(\text{nil}) \end{aligned}$$

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

Example 2: Repeating List Elements

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

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

Base Case (nil):

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

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: Repeating List Elements

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

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

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

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

$\text{sum}(\text{nil}) \quad := 0$

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

Example 2: Repeating List Elements

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

- **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

Recall: Concatenating Two Lists

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

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

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

important operation
abbreviated as "#"

- Puts all the elements of L before those of R

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

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

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

$= 1 :: 2 :: 3 :: 4 :: \text{nil}$

def of concat

def of concat

def of concat

Example 3: Length of Concatenated Lists

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

- Suppose we have the following code:

```
final int m = len(S);      // S is some List
final int 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: Length of Concatenated Lists

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

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

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

Base Case (nil):

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

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

Example 3: Length of Concatenated Lists

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

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

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

Base Case (nil):

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

def of concat

$= 0 + \text{len}(R)$

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

def of len

Example 3: Length of Concatenated Lists

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

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

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

Inductive Step $(x :: L)$:

Need to prove that

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

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

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

Example 3: Length of Concatenated Lists

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

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

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

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

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

Example 3: Length of Concatenated Lists

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

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

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

Comparing Reasoning vs Testing

```
static List concat(List S, List R) {  
    if (S == nil) {  
        return R;  
    } else {  
        return cons(S.hd, concat(S.tl, R));  
    }  
};
```

- **Testing: 3 cases**
 - loop coverage requires 0, 1, and many recursive calls
 - *does not* guarantee the code works on all inputs
- **Reasoning: 2 calculations**

Example: More Efficient Sum

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

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

memory efficient
(more on this later...)

- Suppose we have the following code:

```
final int 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: More Efficient Sum

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

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

- **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):

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

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

Example 4: More Efficient Sum

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

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

- **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: More Efficient Sum

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

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

- **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

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

Example 4: More Efficient Sum

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

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

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

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

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

Example 4: More Efficient Sum

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

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

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

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