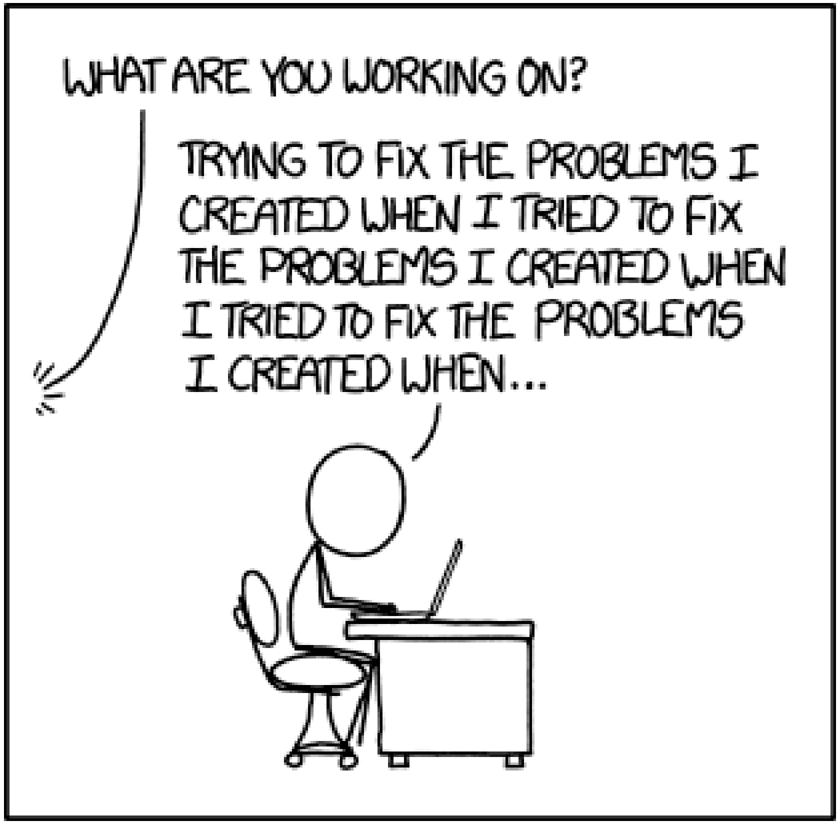
CSE 331
Summer 2025

Reasoning



xkcd #1739, ty Matt

Jaela Field

Administrivia

- HW4 is out!
  - it contains math *and* programming
  - more emphasis on correctness now!
  - Start early!
  - 6 Tasks of varying length
    ~ 1 a day is a good goal!

- Jaela OH today: 12:30 - 1:30 CSE 2/F & zoom

- Bonus lecture on software development coming this weekend!

Agenda

✓ Administrivia

- Finish Testing (finish topic 4)
  - Practice exercises
- Reasoning (start topic 5)

Reacp: Testing so far

- Ground Rules
  - Only test inputs allowed by the spec
  - Test functions individually
  - Keep test code *simple*
  - If there are < 10 inputs, test them all!

- Metrics
  - Statement coverage

    Execute every statement that is reachable by an allowed input
  - Branch coverage

    For every conditional, execute both branches (if they are reachable by an allowed input

(end of testing in Topic 4 slides)

Agenda

✓ Administrivia

✓ Finish Testing (finish topic 4)

  ✓ Practice exercises

- **Reasoning** **(start topic 5)**

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…

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

Recall: Specifications with 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`

Aside: Documentation + Testing

- We discussed clear-box testing
  - involves determining cases based on structure of code
  - can result in buggy tests due to bias!
- Alternative: Opaque-Box Testing
  - focuses solely on inputs and outputs
  - testers don't look at the code, instead test to the spec
    - still care about different input cases
  - very widely used in industry!
- Our primary approach is clear-box testing
  - rule of only testing inputs allowed by the spec is an opaque testing idea

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 "$\text{sum}(L) \geq 0$"

CSE 331 Summer 2025

Reasoning: Proof by Calculation & Cases

Jaela Field

Administrivia

- *optional* lecture on Software Development Process available on Panopto

Recall: 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

Recall: 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 <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	
heap state mutation	“”	“”	rep invariants	

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

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

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

since $x = y$ (first step: $x + z = y + z$)

since $z \leq 10$ (second step: $y + z \leq y + 10$)

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

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

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

Calculation Blocks: Inequalities

- Chain of "=" and "≤" shows first ≤ last

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

  - 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:**

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

  - **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**!

Practice #1!

```
// Inputs x and y are integers with x > 0 and y < 0
// Returns a positive integer.
const f = (x: bigint, y: bigint): bigint => {
   return x - y + 1;
};
```

- Prove that the post condition is correct
  - What is the fact to prove? $x - y + 1 \geq 1$
  - What are the known facts? $x \geq 1$ and $y \leq -1$
  - Proof:

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

Using Definitions in Calculations

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

- For example:

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

- 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 "$L = \text{cons}(\ldots)$"
- Must prove that $\text{sum}(L) \geq 0$

Using Definitions in Calculations (1/2)

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

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

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

Using Definitions in Calculations (2/2)

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

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

$$\begin{aligned} \text{sum}(L) &= \text{sum}(a :: b :: nil) && \textbf{since } L = a :: b :: nil \\ &= a + \text{sum}(b :: nil) && \textbf{def of } \text{sum} \\ &= a + b + \text{sum}(nil) && \textbf{def of } \text{sum} \\ &= a + b && \textbf{def of } \text{sum} \\ &\geq 0 + b && \textbf{since } a \geq 0 \\ &\geq 0 && \textbf{since } b \geq 0 \end{aligned}$$

Practice #2!

```
// Returns a non-empty List.
const f = (x: bigint): List<bigint> => {
    const L: List = cons(x, cons(-x, nil));
    return L;
};
```

- **Recall:** $\text{len}(\text{nil}) := 0$

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

- **Prove that the post condition is correct**
  - **What is the fact to prove?** $\text{len}(L) > 0$
  - **What are the known facts?** $L = x :: \text{-}x :: \text{nil}$
  - **Proof:**

$$
\begin{aligned}
\text{len}(L) &= \text{len}(x :: \text{-}x :: \text{nil}) && \textbf{since } L = x :: \text{-}x :: \text{nil} \\
&= 1 + \text{len}(\text{-}x :: \text{nil}) && \textbf{def of } \text{len} \\
&= 1 + 1 + \text{len}(\text{nil}) && \textbf{def of } \text{len} \\
&= 1 + 1 + 0 && \textbf{def of } \text{len} \\
&> 0
\end{aligned}
$$

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

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)

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

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

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

Case $m \geq 0$:

$f(m) =$

$> m$

Proof By Cases (2/3)

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

$\mathrm{f(m)} := 0$ **if** $\mathrm{m} < 0$

- **Prove that** $\mathrm{f(m)} > \mathrm{m}$

Case $\mathrm{m} \geq 0$:

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

Proof By Cases (3/3)

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

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

Case $m \geq 0$:

$f(m) = \ldots > m$

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.**

Recall: Pattern Matching

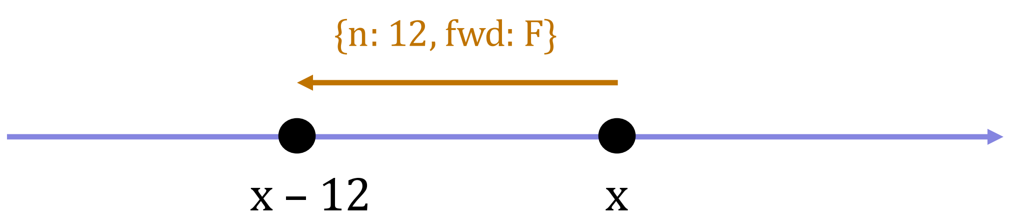
- **Define a function by an exhaustive set of patterns**

type Steps := $\{n : \mathbb{N}, fwd : \mathbb{B}\}$

change({n: n, fwd: T}) := n

change({n: n, fwd: F}) := −n

  - Steps **describes movement on the number line**
  - change(s : Steps) **says how the position changes**



  - **one of these two rules always applies**

Proof by Cases, with Records (Case T)

change({n: n, fwd: T}) := n
change({n: n, fwd: F}) := -n

- **Prove that** $|\text{change}(s)| = n$ **for any** $s = \{n: n, \text{fwd}: f\}$
  - **we need to know if** $f = T$ **or** $f = F$ **to apply the definition!**

Case $f = T$:

$\|\text{change}(\{n: n, \text{fwd}: f\})\|$	
$= \|\text{change}(\{n: n, \text{fwd}: T\})\|$	**since** $f = T$
$= \|n\|$	**def of** change
$= n$	**since** $n \geq 0$

Proof by Cases, with Records (Case F)

$\text{change}(\{n: n, \text{fwd}: T\}) \;:=\; n$

$\text{change}(\{n: n, \text{fwd}: F\}) \;:=\; -n$

- **Prove that** $|\text{change}(s)| = n$ **for any** $s = \{n: n, \text{fwd}: f\}$

Case $f = T$: $|\text{change}(\{n: n, \text{fwd}: f\})| = \ldots = n$

Case $f = F$:

$\lvert\text{change}(\{n: n, \text{fwd}: f\})\rvert$	
$= \lvert\text{change}(\{n: n, \text{fwd}: F\})\rvert$	**since** $f = F$
$= \lvert -n\rvert$	**def of** change
$= n$	**since** $n \geq 0$

Since these two cases are exhaustive, the claim 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 (mostly) focuses on proving that correct code is correct
  - Seems mean to give you incorrect code :')

    Already had our mean era in HW 1-3

  - But, problems will be <u>new</u> and <u>more challenging</u>

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

CSE 331 Summer 2025

Reasoning with Structural Induction

Jaela Field

Common Proof by Calculation Mistakes

- **Assuming claim is true**

$$2x + 1 = -(2x + 1) \qquad \text{BAD ✗}$$

$$(2x + 1)^2 = (-1)^2(2x + 1)^2 \qquad \text{square both sides}$$

$$4x^2 + 2x + 1 = 1(4x^2+2x+1) \qquad \text{foil}$$

$$0 = 0$$

- **Manipulating both sides of the equation**

Example: prove $x^2 + 1 > z$, given $x^2 = y$ and $y > z$

$$x^2 = y \qquad \text{since } x^2 = y$$

$$x^2 + 1 = y + 1 \qquad \text{add 1 to both sides}$$

$$x^2 + 1 > z \qquad \text{since } y > z$$

Common Proof by Calculation Mistakes

- Mixing $>$ and $<$
  - cannot conclude anything!

    $2 < 4$

    > 3 therefore $2 > 3$... ✗
- Applying multiple facts/defs in the same step
  - In the "real world" sometimes proof steps skip, here we want to see that you understand what applying each looks like
- Forgetting citations
  - It's okay to skip algebraic steps

Structural Induction

Proof by Calculation on Lists

- 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 <u>any length</u> 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

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

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

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

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

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

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

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

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

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

- **use any known facts and definitions**

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

- 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 $\mathrm{P}(\mathrm{S})$ 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

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

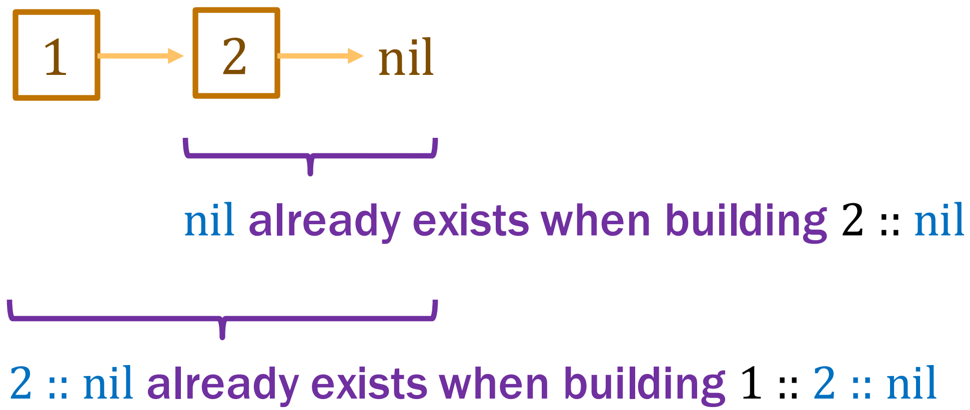
(**second assuming** $\mathrm{len(echo(L)) = 2\ len(L)}$)

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)**



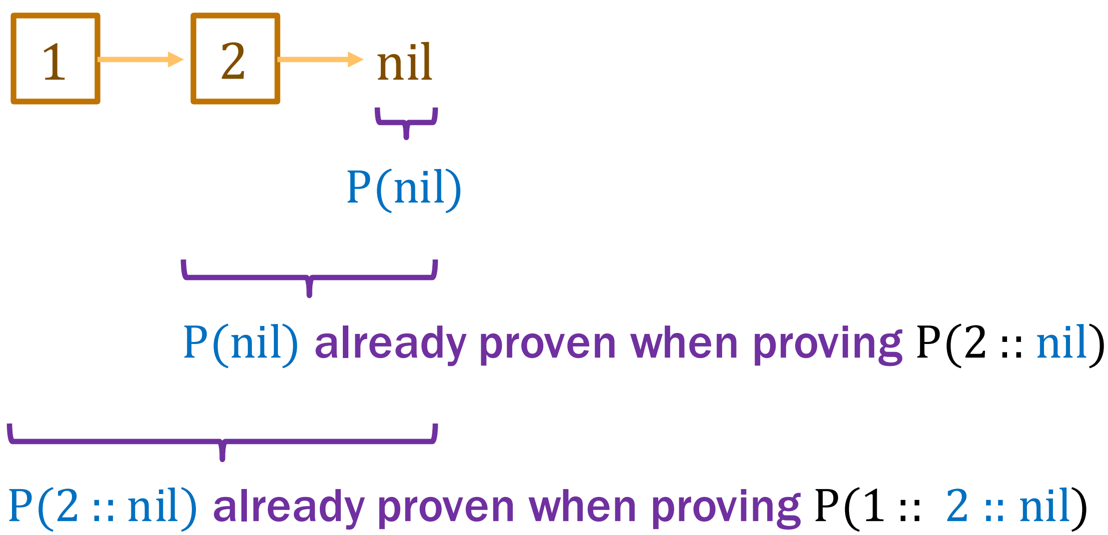
Inductive Proofs are "Built Up" in Steps

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



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)

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

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

Base Case (nil):

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

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

- **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: Echo & Len Inductive Step (2/3)

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 2: Echo & Sum

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

- 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)

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

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

Base Case (nil):

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

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

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

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):

$\text{sum}(\text{echo}(\text{nil}))$	$= \text{sum}(\text{nil})$	**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: Echo & Sum Inductive Step (1/2)

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

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

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

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

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

Example 2: Echo & Sum Inductive Step (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** $S : \text{List}$

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

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

Recall: Concatenating Two Lists

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

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

important operation abbreviated as "⧺"

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

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

Example 3: Length of Concatenated Lists

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

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)

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

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

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

Base Case (nil):

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

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

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

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

- **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: Len & Concat Inductive Step (1/3)

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

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

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

Need to prove that

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

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

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

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

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

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

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

Inductive Step (x :: L):

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

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

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

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)

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

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

- **Case** $(S = \text{nil})$: $\text{allEqual}(x :: \text{nil}) = \text{true}$ — **def of** allEqual
- **Case** $(S = y :: L)$:
  - $y :: L$ 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)

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

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

Now, what if we got a bit sloppy?

Inductive Hypothesis: assume that allEqual~~(S)~~ = true ~~**for lists** S~~

can't assume claim!

Inductive Step (x :: S):

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

Proof Strategy Advice

- 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 hypothesis be used?

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

Example 4: Faster Sum

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

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)

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

- **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):

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

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

- **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 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 $(\text{x} :: \text{L})$**:**

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

Structural Induction in General

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

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

- To prove $\mathrm{P}(\mathrm{t})$ for any t, we need to prove:
  - $\mathrm{P}(\mathrm{A})$
  - $\mathrm{P}(\mathrm{B}(\mathrm{x}))$ for any $\mathrm{x} : \mathbb{Z}$
  - $\mathrm{P}(\mathrm{C}(\mathrm{y}, \mathrm{t}))$ for any $\mathrm{y} : \mathbb{Z}$ and $\mathrm{t} : \mathrm{T}$ assuming $\mathrm{P}(\mathrm{t})$ is true
  - $\mathrm{P}(\mathrm{D}(\mathrm{z}, \mathrm{u}, \mathrm{v}))$ for any $\mathrm{z} : \mathbb{Z}$ and $\mathrm{u}, \mathrm{v} : \mathrm{T}$ assuming $\mathrm{P}(\mathrm{u})$ and $\mathrm{P}(\mathrm{v})$

- These four facts are enough to prove $\mathrm{P}(\mathrm{t})$ for any t
  - for each constructor, have proof that it produces an object satisfying P
  - generally, each inductive type has its own form of induction

Defining Cases

- Case in inductive data type = case in structural inductive proof
  - "Smallest" form of data type = Base case in proof
  - Recursive case in data type = Inductive step in proof

- To prove $\mathrm{P}(\mathrm{t})$ for any t of type T:
  - We have 2 base cases

    type $\mathrm{T} := \mathbf{A} \mid \mathbf{B}(\mathbf{x}:\mathbb{Z}) \mid \mathrm{C}(\mathrm{y}:\mathbb{Z}, \mathrm{t}:\mathrm{T}) \mid \mathrm{D}(\mathrm{z}:\mathbb{Z}, \mathrm{u}:\mathrm{T}, \mathrm{v}:\mathrm{T})$
  - and 2 recursive cases

    type $\mathrm{T} := \mathrm{A} \mid \mathrm{B}(\mathrm{x}:\mathbb{Z}) \mid \mathbf{C}(\mathbf{y}:\mathbb{Z}, \mathbf{t}:\mathbf{T}) \mid \mathbf{D}(\mathbf{z}:\mathbb{Z}, \mathbf{u}:\mathbf{T}, \mathbf{v}:\mathbf{T})$
  - Inductive proof will cover base cases in base case and recursive cases cases in inductive step

Induction Wrap up: Defining Cases

- **If math def defines a case for recursive form of with a fixed size, that is still part of inductive step!**
  - Example, from last lecture:

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

$x :: \text{nil}$ uses recursive constructor of a List, so it should be part of the inductive step:

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

Inductive Step ($x :: S$):

- **Case** ($S = \text{nil}$): $\text{allEqual}(x :: \text{nil}) = \text{true}$ — **def of** allEqual
- **Case** ($S = y :: L$): ...

we don't use the IH in every case. That's okay!

The following examples were not covered in lecture, but are useful practice, if needed!

Definition of List Reversal

- Reversal of a List: “same values but in reverse order”

- Look at some examples…

L	rev(L)	
nil	nil	
[3]	[3]	3 :: nil
[2, 3]	[3, 2]	3 :: 2 :: nil
[1, 2, 3]	[3, 2, 1]	3 :: 2 :: 1 :: nil
…	…	

Structural Recursion in List Reversal

- **Look at some examples…**

L	rev(L)
nil	nil
3 :: nil	3 :: nil
2 :: 3 :: nil	3 :: 2 :: nil
1 :: 2 :: 3 :: nil	3 :: 2 :: 1 :: nil

- **Where does** rev([2, 3]) **show up in** rev([1, 2, 3])**?**
  - **at the beginning, with** 1 :: nil ***after*** **it**

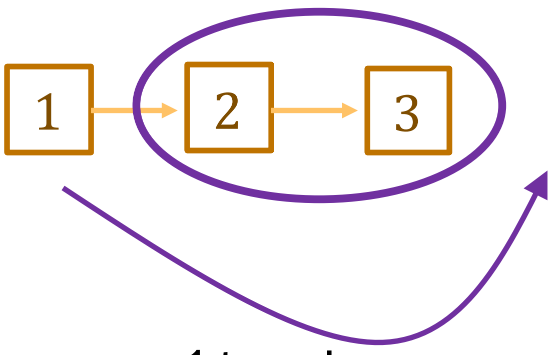
- **Where does** rev([3]) **show up in** rev([2, 3])**?**
  - **at the beginning, with** 2 :: nil ***after*** **it**

Recall: Reversing a List

- **Mathematical definition of** $\text{rev}(S)$

$$\text{rev}(\text{nil}) := \text{nil}$$
$$\text{rev}(x :: L) := \text{rev}(L) \mathbin{+\!\!+} [x]$$

  - **note that** rev **uses** concat ($\mathbin{+\!\!+}$) **as a helper function**



Definition of List Reversal: Checking Examples

$1 :: 2 :: 3 :: \text{nil}$ $\qquad\qquad$ $3 :: 2 :: 1 :: \text{nil}$

- **Mathematical definition of** $\text{rev} : \text{List} \to \text{List}$

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

- **Check that this matches examples…**

$\text{rev}(1 :: 2 :: 3 :: \text{nil})$	
$= \text{rev}(2 :: 3 :: \text{nil}) \mathbin{+\!\!+} [1]$	def of rev
$= \text{rev}(3 :: \text{nil}) \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1]$	def of rev
$= \text{rev}(\text{nil}) \mathbin{+\!\!+} [3] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1]$	def of rev
$= [\,] \mathbin{+\!\!+} [3] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1]$	def of rev
$= \ldots = [3, 2, 1]$	def of concat (many times)

Example 5: Length of Reversed List: Setup

$$\mathrm{rev}(\mathrm{nil}) := \mathrm{nil}$$
$$\mathrm{rev}(x :: L) := \mathrm{rev}(L) \mathbin{+\!\!+} [x]$$

- Suppose we have the following code:

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

  – spec returns $\mathrm{len}(\mathrm{rev}(S))$ but code returns $\mathrm{len}(S)$

- Need to prove that $\mathrm{len}(\mathrm{rev}(S)) = \mathrm{len}(S)$ for any $S : \mathrm{List}$

Example 5: Length of Reversed List (1/3)

$\text{rev}(\text{nil}) := \text{nil}$

$\text{rev}(x :: L) := \text{rev}(L) ⧺ [x]$

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

Base Case (nil):

$\text{len}(\text{rev}(\text{nil})) = \text{len}(\text{nil})$ **def of** rev

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

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

Get to assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

Example 5: Length of Reversed List (2/3)

$$\text{rev}(\text{nil}) \quad := \text{nil}$$
$$\text{rev}(x :: L) \quad := \text{rev}(L) + [x]$$

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

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

Inductive Step $(x :: L)$:

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

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

Example 5: Length of Reversed List (3/3)

$$\text{rev}(\text{nil}) \quad := \text{nil}$$
$$\text{rev}(x :: L) \quad := \text{rev}(L) +\!\!+ [x]$$

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

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

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

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

$= \text{len}(\text{rev}(L) +\!\!+ [x])$	**def of** rev
$= \text{len}(\text{rev}(L)) + \text{len}([x])$	**by Example 3**
$= \text{len}(L) + \text{len}([x])$	**Ind. Hyp.**
$= \text{len}(L) + 1 + \text{len}(\text{nil})$	**def of** len
$= \text{len}(L) + 1$	**def of** len
$= \text{len}(x :: L)$	**def of** len

Finer Points of Structural Induction

- Structural Induction is how we reason about recursion
- Reasoning also follows structure of code
  - code uses structural recursion, so reasoning uses structural induction
- Note that rev is defined in terms of concat
  - reasoning about $\mathrm{len}(\mathrm{rev}(\ldots))$ used fact about $\mathrm{len}(\mathrm{concat}(\ldots))$
  - this is common

Example 6: Reversing a List Performance

$$\text{rev}(\text{nil}) := \text{nil}$$
$$\text{rev}(x :: L) := \text{rev}(L) \mathbin{+\!\!+} [x]$$

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where n is length of L
  - n calls to concat takes $\Theta(n^2)$ time

- Can we do this faster?
  - yes, but we need a helper function

Example 6: Reversing a List, Linear Time (1/3)

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\text{rev-acc}(\text{nil}, R) := R$$

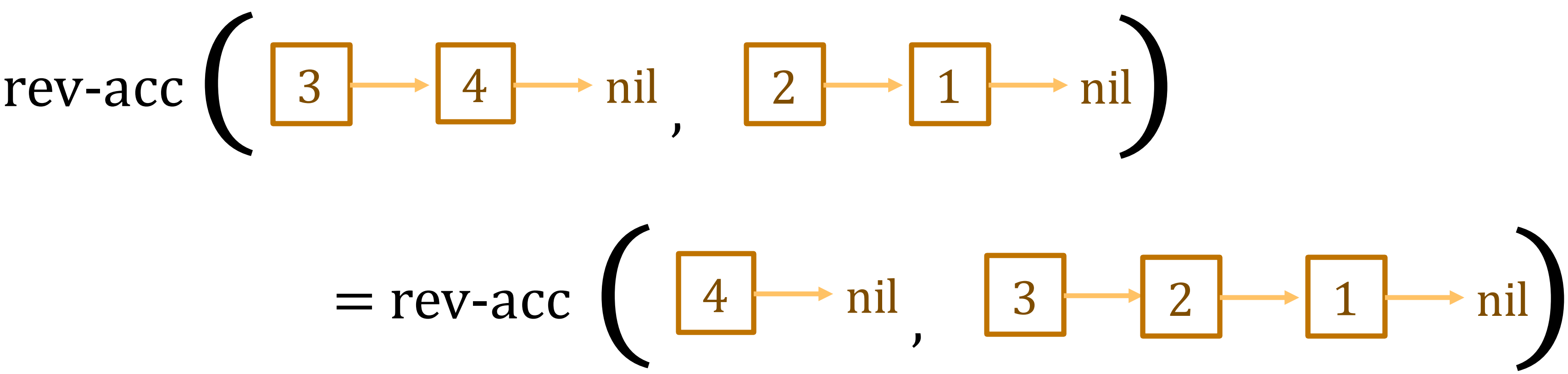
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

rev-acc

Example 6: Reversing a List, Linear Time (2/3)

- **Helper function** rev-acc(S, R) **for any** S, R : List

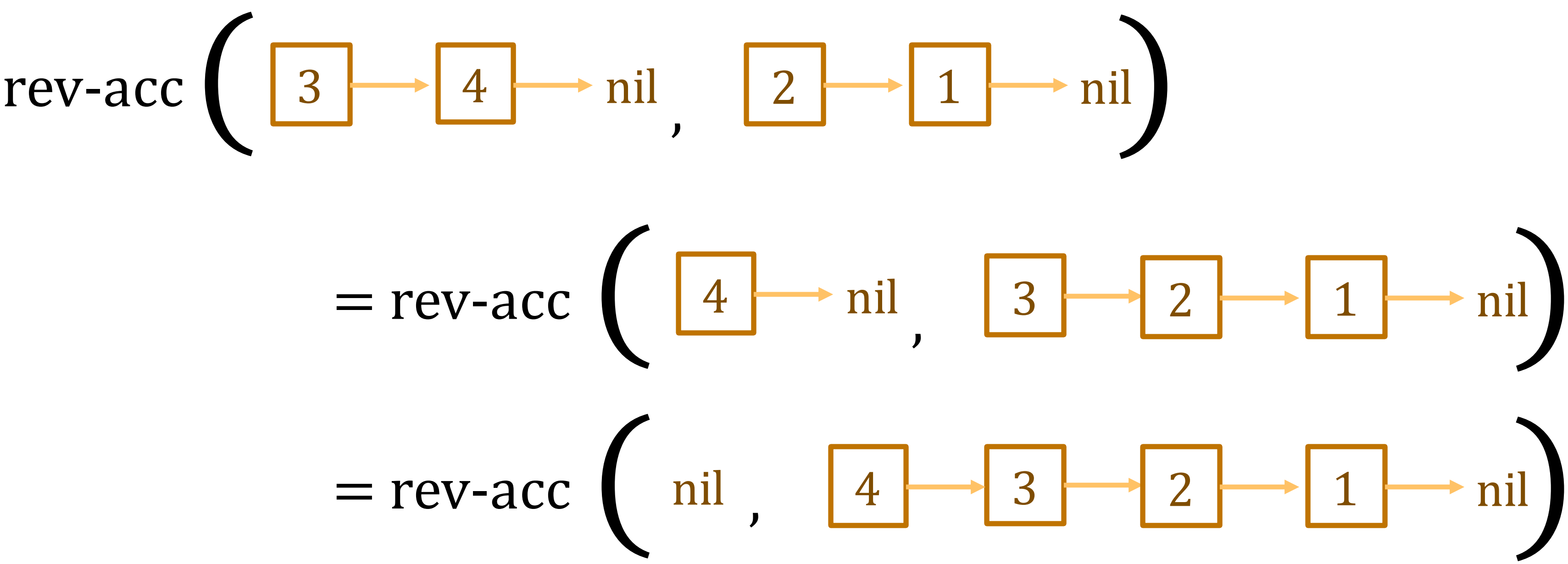
$$\begin{aligned} \text{rev-acc}(\text{nil}, R) &:= R \\ \text{rev-acc}(x :: L, R) &:= \text{rev-acc}(L, x :: R) \end{aligned}$$



Example 6: Reversing a List

- **Helper function** rev-acc(S, R) **for any** S, R : List

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$



Proving that rev-acc works, in pieces

$$\text{rev-acc}(\text{nil}, R) \quad := \quad R$$
$$\text{rev-acc}(x :: L, R) \quad := \quad \text{rev-acc}(L, x :: R)$$

- **Can prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$ **(Lemma 1)**
- **Can prove that** $\text{concat}(L, \text{nil}) = L$ **(Lemma 2)**
  - **structural induction like prior examples**
- **Prove that** $\text{rev}(S) = \text{rev-acc}(S, \text{nil})$

$$\text{rev-acc}(S, \text{nil}) = \text{concat}(\text{rev}(S), \text{nil}) \qquad \textbf{Lemma 1}$$
$$= \text{rev}(S) \qquad \textbf{Lemma 2}$$

Proving Lemma 2: Setup

$$\text{rev-acc}(\text{nil}, \text{R}) \quad := \quad \text{R}$$
$$\text{rev-acc}(\text{x} :: \text{L}, \text{R}) \quad := \quad \text{rev-acc}(\text{L}, \text{x} :: \text{R})$$

- **Prove that** $\text{concat}(\text{S}, \text{nil}) = \text{S}$

Base Case (nil):

$\text{concat}(\text{nil}, \text{nil}) \quad = \text{nil}$ **def of** concat

Inductive Hypothesis: **assume that** $\text{concat}(\text{L}, \text{nil}) = \text{nil}$

Inductive Step $(\text{cons}(\text{x}, \text{L}))$: **prove that** $\text{concat}(\text{cons}(\text{x}, \text{L}), \text{nil}) = \text{cons}(\text{x}, \text{L})$

Proving Lemma 2: Inductive Step (1/2)

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Inductive Hypothesis: **assume that** $\text{concat}(L, \text{nil}) = L$

Inductive Step $(x :: L)$:

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

$= x :: L$

Proving Lemma 2: Inductive Step (2/2)

$$\text{rev-acc}(\text{nil}, \text{R}) := \text{R}$$
$$\text{rev-acc}(\text{x} :: \text{L}, \text{R}) := \text{rev-acc}(\text{L}, \text{x} :: \text{R})$$

- **Prove that** $\text{concat}(\text{S}, \text{nil}) = \text{S}$

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

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

$\text{concat}(\text{x} :: \text{L}, \text{nil})$	$= \text{x} :: \text{concat}(\text{L}, \text{nil})$	**def of** concat
	$= \text{x} :: \text{L}$	**Ind. Hyp.**

Proving Lemma 1: Setup

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **prove by structural induction**

- **Need the following property of** $\text{concat}\ (\mathbin{+\!\!\!+})$

$$A \mathbin{+\!\!\!+} (B \mathbin{+\!\!\!+} C) = (A \mathbin{+\!\!\!+} B) \mathbin{+\!\!\!+} C$$

  - **with strings, we know that "A + (B + C) = (A + B) + C"**
  - **this says the same thing for lists with "**$\mathbin{+\!\!\!+}$**"**

Proving Lemma 1: Base Case (1/2)

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

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **prove by induction on** S **(so** R **is a variable)**

Base Case (nil):

$\text{rev-acc}(\text{nil}, R) \quad =$

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

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

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) +\!\!+ [x] \end{aligned}$$

Proving Lemma 1: Base Case (2/2)

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

- **Prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **prove by induction on** S **(so** R **is a variable)**

Base Case (nil):

$\text{rev-acc}(\text{nil}, R)$	$= R$	**def of** rev-acc
	$= \text{concat}(\text{nil}, R)$	**def of** concat
	$= \text{concat}(\text{rev}(\text{nil}), R)$	**def of** rev

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

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

Proving Lemma 1: Inductive Step (1/4)

$$\text{rev-acc}(\text{nil}, R) := R$$
$$\text{rev-acc}(x :: L, R) := \text{rev-acc}(L, x :: R)$$

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

  Inductive Hypothesis: **assume that** $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

  Inductive Step $(x :: L)$:

  $\text{rev-acc}(x :: L, R) \quad =$

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

func $\text{concat}(\text{nil}, R) := R$ $\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))$	**func** $\text{rev}(\text{nil}) := \text{nil}$ $\text{rev}(\text{cons}(x, L)) := \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))$

Proving Lemma 1: Inductive Step (2/4)

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

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

Inductive Hypothesis: **assume that** $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

Inductive Step $(x :: L)$:

$\text{rev-acc}(x :: L, R)$	$= \text{rev-acc}(L, x :: R)$	**def of** rev-acc
	$= \text{concat}(\text{rev}(L), x :: R)$	**Ind. Hyp.**
	$= (\text{rev}(L) \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} R$	??
	$= \text{concat}(\text{rev}(L) \mathbin{+\!\!+} [x], R)$	
	$= \text{concat}(\text{rev}(x :: L), R)$	**def of** rev

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

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

Proving Lemma 1: Inductive Step (3/4)

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

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

Inductive Hypothesis: assume that $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

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

$\text{rev-acc}(x :: L, R)$	$= \text{rev-acc}(L, x :: R)$	**def of** rev-acc
	$= \text{concat}(\text{rev}(L), x :: R)$	**Ind. Hyp.**
	$= \text{rev}(L) \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} R)$	
	$= (\text{rev}(L) \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} R$	**assoc. of** $\mathbin{+\!\!+}$
	$= \text{concat}(\text{rev}(L) \mathbin{+\!\!+} [x], R)$	
	$= \text{concat}(\text{rev}(x :: L), R)$	**def of** rev

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

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$

Proving Lemma 1: Inductive Step (4/4)

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

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

Inductive Hypothesis: assume that $\text{rev-acc}(L, R) = \text{concat}(\text{rev}(L), R)$ **for any** R

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

$\text{rev-acc}(x :: L, R)$	$= \text{rev-acc}(L, x :: R)$	**def of** rev-acc
	$= \text{concat}(\text{rev}(L), x :: R)$	**Ind. Hyp.**
	$= \text{rev}(L) \mathbin{+\!\!+} (x :: R)$	
	$= \text{rev}(L) \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} R)$	**def of** concat
	$= (\text{rev}(L) \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} R$	**assoc. of** $\mathbin{+\!\!+}$
	$= \text{concat}(\text{rev}(L) \mathbin{+\!\!+} [x], R)$	
	$= \text{concat}(\text{rev}(x :: L), R)$	**def of** rev

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

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} [x] \end{aligned}$$