CSE 331

Reasoning About Straight-Line Code

Katherine Murphy

Credits: Profs. Kevin Zatloukal and James Wilcox
Inductive Data Types

• Previous saw records, tuples, and unions
  – very useful but limited
    can only create types that are “small” in some sense
  – missing one more way of defining types
    arguably the most important

• One critical element is missing: recursion
  Java classes can have fields of same type, but records cannot

• Inductive data types are defined recursively
  – combine union with recursion
Inductive Data Types

• Describe a set by ways of creating its elements
  – each is a “constructor”

\[
\text{type } T := C(x : \mathbb{Z}) \mid D(x : \mathbb{Z}, y : T)
\]

  – second constructor is recursive
  – can have any number of arguments (even none)
    will leave off the parentheses when there are none

• Examples of elements

  \begin{align*}
  &C(1) \\
  &D(2, C(1)) \\
  &D(3, D(2, C(1)))
  \end{align*}

\[\text{in math, these are not function calls}\]
Inductive Data Types

• Each element is a description of how it was made
  
  \[
  \begin{align*}
  & C(1) \\
  & D(2, C(1)) \\
  & D(3, D(2, C(1)))
  \end{align*}
  \]

• Equal when they were made exactly the same way
  
  – $C(1) \neq C(2)$
  – $D(2, C(1)) \neq D(3, C(1))$
  – $D(2, C(1)) \neq D(2, C(2))$
  – $D(1, D(2, C(3))) = D(1, D(2, C(3)))$
Natural Numbers

\[
\text{type } \mathbb{N} := \text{zero } \mid \text{succ}(n : \mathbb{N})
\]

- Inductive definition of the natural numbers

\[
\begin{align*}
\text{zero} & \quad 0 \\
\text{succ(zero)} & \quad 1 \\
\text{succ(succ(zero))} & \quad 2 \\
\text{succ(succ(succ(zero)))} & \quad 3
\end{align*}
\]

The most basic set we have is defined inductively!
Even Natural Numbers

type $\mathbb{E}$ := zero | two-more(n : $\mathbb{E}$)

- Inductive definition of the even natural numbers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td>two-more(zero)</td>
<td>2</td>
</tr>
<tr>
<td>two-more(two-more(zero))</td>
<td>4</td>
</tr>
<tr>
<td>two-more(two-more(two-more(zero)))</td>
<td>6</td>
</tr>
</tbody>
</table>

much better notation
Lists

\textbf{Inductive definition of lists of integers}

\begin{align*}
\text{nil} & \approx [] \\
\text{cons}(3, \text{nil}) & \approx [3] \\
\text{cons}(2, \text{cons}(3, \text{nil})) & \approx [2, 3] \\
\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil}))) & \approx [1, 2, 3]
\end{align*}

\textcolor{orange}{array notation}
“Lists are the original data structure for functional programming, just as arrays are the original data structure of imperative programming”

Ravi Sethi
Inductive Data Types in TypeScript

• TypeScript does not natively support inductive types
  – some “functional” languages do (e.g., OCaml and ML)

• We must think of a way to cobble them together...
  – our answer is a design pattern
Design Patterns

• Introduced in the book of that name
  – written by the “Gang of Four”
    Gamma, Helm, Johnson, Vlissides
  – worked in C++ and SmallTalk

• Found that they independently developed many of the same solutions to recurring problems
  – wrote a book about them

• Many are problems with OO languages
  – authors worked in C++ and SmallTalk
  – some things are not easy to do in those languages
Type Narrowing with Records

• Use a literal field to distinguish records types
  – require the field to have one specific value
  – called a “tag” field
    cleanest way to make unions of records

```typescript
type T1 = {kind: "T1", a: bigint, b: number};
type T2 = {kind: "T2" a: bigint, b: string};

const x: T1 | T2 = ...

if (x.kind === "T1") { // legal for either type
  console.log(x.b);  // must be T1... x.b is a number
} else {
  console.log(x.b);  // must be T2... x.b is a string
}
```
Inductive Data Type Design Pattern

type T := C(x: ℤ) | D(x: $^*$, t:T)

- Implement in TypeScript as

```typescript
type T = {kind: "C", x: number}
 | {kind: "D", x: string, t: T};
```
Inductive Data Type Design Pattern

\[
\text{type } T := A \mid B \mid C(x : \mathbb{Z}) \mid D(x : \mathcal{S}^*, t : T)
\]

- **Implement in TypeScript as**

```typescript
type T = {kind: "A"} |
        {kind: "B"} |
        {kind: "C", x: bigint} |
        {kind: "D", x: string, t: T};
```
Inductive Data Types in TypeScript

type List := nil | cons(x: ℤ, L: List)

• Implemented in TypeScript as

  type List = {kind: "nil"}
  | {kind: "cons", hd: bigint, tl: List};

  – fields should also be “readonly”

How to check if a value mylist is nil?
  if (mylist.kind === "nil") {
    ...
  }

Inductive Data Types in TypeScript

- Make this look more like math notation...

```typescript
type List = {kind: "nil"}
  | {kind: "cons", hd: bigint, tl: List};

const nil: List = {kind: "nil"};

const cons = (hd: bigint, tl: List): List => {
  return {kind: "cons", hd: hd, tl: tl};
}

- use only these two functions to create Lists
  do not create the records directly
- note that we only have one instance of nil
  this is called a “singleton” (a design pattern)
Inductive Data Types in TypeScript

• Make this look more like math notation...

```typescript
const nil: List = {kind: "nil"};

const cons = (hd: bigint, tl: List): List => { .. };
```

• Can now write code like this:

```typescript
const L: List = cons(1, cons(2, nil));

if (L === nil) {
    return L;
} else {
    return cons(L.hd, R); // head of L followed by R
}
```

if someone made their own nil, then this would fail 😞
and it doesn’t typecheck
Inductive Data Types in TypeScript

• Make this look more like math notation...

```typescript
const nil: List = {kind: "nil"};

const cons = (hd: bigint, tl: List): List => {
  ..
};
```

• Still not perfect:
  – JS “===” (references to same object) does not match “=”

```typescript
cons(1, cons(2, nil)) === cons(1, cons(2, nil))  // false!
```

  – need to define an equal function for this
Inductive Data Types in TypeScript

• Objects are equal if they were built the same way

```typescript
type List = {kind: "nil"}
    | {kind: "cons", hd: bigint, tl: List};

const equal = (L: List, R: List): boolean => {
  if (L.kind === "nil") {
    return R === nil;
  } else {
    if (R.kind === "nil") {
      return false;
    } else {
      return L.hd === R.hd && equal(L.tl, R.tl);
    }
  }
};
```
Functions
Code Without Mutation

• Saw all types of code without mutation:
  – straight-line code
  – conditionals
  – recursion

• This is all that there is

• Saw TypeScript syntax for these already...
Code Without Mutation

Example function with all three types

```javascript
// n must be a non-negative integer
const f = (n: bigint): bigint => {
    if (n === 0n) {
        return 1n;
    } else {
        return 2n * f(n - 1n);
    }
};
```

What does this compute? $2^n$
Recall: Natural Numbers

\[\text{type } \mathbb{N} := \text{zero} | \text{succ}(\text{prev}: \mathbb{N})\]

- Inductive definition of the natural numbers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td>succ(zero)</td>
<td>1</td>
</tr>
<tr>
<td>succ(succ(zero))</td>
<td>2</td>
</tr>
<tr>
<td>succ(succ(succ(zero)))</td>
<td>3</td>
</tr>
</tbody>
</table>
Recall: Natural Numbers

\[
\text{type } \mathbb{N} := \text{ zero } \mid \text{ succ}(\text{ prev: } \mathbb{N})
\]

• **Potential definition in TypeScript**

```typescript
type Nat = {kind: "zero"}
| {kind: "succ", prev: Nat};

class Nat {
    kind: "zero";
}

const zero: Nat = { kind: "zero" };

const succ = (prev: Nat): Nat => {
    return {kind: "succ", prev: prev};
};
```
Induction on Natural Numbers

Could use a type that only allows natural numbers:

```javascript
const f = (n: Nat): bigint => {
  if (n.kind === "zero") {
    return 1n;
  } else {
    return 2n * f(n.prev);
  }
};
```

Cleaner definition of the function (though inefficient)

n.prev represents “n – 1”
Structural Recursion

- **Inductive types**: build new values from existing ones
  - only zero exists initially
  - build up 5 from 4 (which is built from 3 etc.)
    
      4 is the argument to the constructor of 5 = succ(4)

- **Structural recursion**: recurse on smaller parts
  - call on n recurses on n.prev
    
      n.prev is the argument to the constructor (succ) used to create n
  - guarantees no infinite loops!
    
      limit to structural recursion whenever possible

- **We will try to restrict ourselves to structural recursion**
  - for both math and TypeScript
Defining Functions in Math

• Saw math notation for defining functions, e.g.:

  \[ \text{func } f(n) := 2n + 1 \quad \text{for any } n : \mathbb{N} \]

• We need recursion to define interesting functions
  – we will primarily use structural recursion

• Inductive types fit esp. well with \textit{pattern matching}
  – every object is created using some constructor
  – match based on which constructor was used (last)
Length of a List

```
type List := nil | cons(hd: ℤ, tl: List)
```

- **Mathematical definition of length**

```
func len(nil) := 0
len(cons(x, S)) := 1 + len(S)  for any x ∈ ℤ
                        and any S ∈ List
```

- any list is either nil or cons(x, L) for some x and L
- cases are exclusive and exhaustive
Length of a List

• Mathematical definition of length

\[
\begin{align*}
\text{func } \text{len}(\text{nil}) & := 0 \\
\text{len}(\text{cons}(x, S)) & := 1 + \text{len}(S)
\end{align*}
\]
for any \( x \in \mathbb{Z} \) and any \( L \in \text{List} \)

• Translation to TypeScript

```typescript
const len = (L: List): bigint => {
    if (L.kind === "nil") {
        return 0n;
    } else {
        return 1n + len(L.tl);
    }
};
```

straight from the spec
Concatenating Two Lists

- **Mathematical definition of concat(L, R)**

\[
\begin{align*}
\text{func} \ \text{concat}(\text{nil}, \ R) & \ := \ R \quad \text{for any } R \in \text{List} \\
\text{concat}(\text{cons}(x, S), \ R) & \ := \ \text{cons}(x, \ \text{concat}(S, R)) \quad \text{for any } x \in \mathbb{Z} \text{ and any } S, R \in \text{List}
\end{align*}
\]

- concat(L, R) defined by pattern matching on L (not R)
Concatenating Two Lists

• **Mathematical definition of** \( \text{concat}(L, R) \)

\[
\begin{align*}
\text{func } \text{concat}(\text{nil}, R) & : = R \\
\text{concat}(\text{cons}(x, S), R) & : = \text{cons}(x, \text{concat}(S, R))
\end{align*}
\]
for any \( R \in \text{List} \)
for any \( x \in \mathbb{Z} \) and
any \( S, R \in \text{List} \)

• **Translation to TypeScript**

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

*straight from the spec*
Example

• See ex3 on the course website
  – Simple use of Nat in a webapp
Formalizing Specifications
Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2</td>
<td>no mutation</td>
<td>“</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>3</td>
<td>local variable mutation</td>
<td>“</td>
<td>“</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>4</td>
<td>array mutation</td>
<td>“</td>
<td>“</td>
<td>for-any facts</td>
</tr>
<tr>
<td>5</td>
<td>heap state mutation</td>
<td>“</td>
<td>“</td>
<td>rep invariants</td>
</tr>
</tbody>
</table>

“straight from spec” requires us to have a **formal** spec!
Formalizing a Specification

• Sometimes the instructions are written in English
  – English is often imprecise or ambiguous

• First step is to “formalize” the specification:
  – translate it into math with a precise meaning

• How do we tell if the specification is wrong?
  – specifications can contain bugs
  – we can only test our definition on some examples
    (formal) reasoning can only be used after we have a formal spec

• Usually best to start by looking at some examples
Definition of Sum of Values in a List

- **Sum of a List**: “add up all the values in the list”

- **Look at some examples...**

<table>
<thead>
<tr>
<th>L</th>
<th>sum(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>0</td>
</tr>
<tr>
<td>cons(3, nil)</td>
<td>3</td>
</tr>
<tr>
<td>cons(2, cons(3, nil))</td>
<td>2+3</td>
</tr>
<tr>
<td>cons(1, cons(2, cons(3, nil)))</td>
<td>1+2+3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Definition of Sum of Values in a List

• Look at some examples...

<table>
<thead>
<tr>
<th>L</th>
<th>sum(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>0</td>
</tr>
<tr>
<td>cons(3, nil)</td>
<td>3</td>
</tr>
<tr>
<td>cons(2, cons(3, nil))</td>
<td>2+3</td>
</tr>
<tr>
<td>cons(1, cons(2, cons(3, nil)))</td>
<td>1+2+3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• Mathematical definition

\[
\text{func } \text{sum}(\text{nil}) \quad :=
\]
\[
\text{sum}(\text{cons}(x, S)) \quad := \\
\quad \text{for any } x \in \mathbb{Z}
\]
\[
\text{and any } S \in \text{List}
\]
Sum of Values in a List

• Mathematical definition of sum

\[
\begin{align*}
\text{func } & \text{ sum}(\text{nil}) \ := \ 0 \\
\text{sum}(\text{cons}(x, S)) \ := \ x + \text{sum}(S) & \quad \text{for any } x \in \mathbb{Z} \\
& \quad \text{and any } S \in \text{List}
\end{align*}
\]

• Translation to TypeScript

```typescript
const sum = (L: List): bigint => {
  if (L.kind === "nil") {
    return 0n;
  } else {
    return L.hd + sum(L.tl);
  }
};
```

straight from the spec
Definition of Reversal of a List

• Look at some examples...

- \( L \)
  - nil
  - cons(3, nil)
  - cons(2, cons(3, nil))
  - cons(1, cons(2, cons(3, nil)))

- \( \text{rev}(L) \)
  - nil
  - cons(3, nil)
  - cons(3, cons(2, nil))
  - cons(3, cons(2, cons(1, nil)))

• Draw a picture?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

move 1 to end

reverse this too
Reversing A Lists

- Draw a picture?

- Mathematical definition of rev

\[
\text{func } \text{rev}(\text{nil}) := \\
\text{rev}(\text{cons}(x, S)) := \quad \text{for any } x \in \mathbb{Z} \text{ and } \text{any } S \in \text{List}
\]
Reversing A Lists

• Mathematical definition of rev

\[
\begin{align*}
\text{func } \text{rev}(\text{nil}) & : = \text{nil} \\
\text{rev}(\text{cons}(x, S)) & : = \text{concat}(\text{rev}(S), \text{cons}(x, \text{nil})) \quad \text{for any } x \in \mathbb{Z} \text{ and any } S \in \text{List}
\end{align*}
\]

• Other definitions are possible, but this is simplest

• No help from reasoning tools until later
  – only have testing and thinking about what the English means

• Always make definitions as simple as possible
Reasoning
Review: Software Development Process

Given: a problem description (in English)

You get paid for reasoning and debugging! Everything else can (and will) be automated.
• “Thinking through” what the code does on all inputs
  – neither testing nor type checking can do this

• Required in principle and in practice
  – a professional responsibility to know what your code does
  – in practice, “reasoning is not optional: either reason up front or debug and then reason”

• Can be done formally or informally
  – most professionals reason informally
    requires years of practice
  – we will teach formal reasoning
    steppingstone to informal reasoning and needed for the hardest problems
Reasoning

• In an intro class, you might be asked:

  *what does this code do on this input?*

• In this class, we are often interested in:

  *what does this code do on all inputs?*

• This is a very different question!
## Correctness Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>2 HW Quilt/Cipher</td>
<td>no mutation</td>
<td>“”</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>3 HW Weave</td>
<td>local variable mutation</td>
<td>“”</td>
<td>“”</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>4 HW Chatbot</td>
<td>array mutation</td>
<td>“”</td>
<td>“”</td>
<td>for-any facts</td>
</tr>
<tr>
<td>5 HW Squares</td>
<td>heap state mutation</td>
<td>“”</td>
<td>“”</td>
<td>rep invariants</td>
</tr>
</tbody>
</table>
Facts

• Basic inputs to reasoning are “facts”
  – things we know to be true about the variables
  – typically, “=” or “≤”

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

• At the return statement, we know these facts:
  – $n \in \mathbb{N}$
  – $m = 2n$
  – $n \in \mathbb{Z}$ and $n \geq 0$
Facts

- Basic inputs to reasoning are “facts”
  - things we know to be true about the variables
  - typically, “=” or “≤”

  ```javascript
  // n must be 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
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”

• Proving implications is necessary for checking correctness...
Checking Correctness

• Specifications include two kinds of facts
  – promised facts about the inputs (P and Q)
  – required facts about the outputs (R)

• Checking correctness is just proving implications
  – proving facts about the return values

• Two ways reasoning could be required:
  – declarative spec has facts that must hold for the return value
  – different imperative spec: must check expressions are “=”
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 implications is the core step of reasoning
  – other techniques output implications for us to prove

• The techniques we will learn are
  – proof by calculation
  – proof by cases
  – structural induction gives us two implications, each usually proven 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 \( \text{len(nil)} = 0 \)
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

\[
x + z = y + z \leq y + 10
\]

\[
\text{since } x = y \quad \text{since } z \leq 10
\]

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

\[
\begin{align*}
x + z &= y + z & \text{since } x = y \\
\leq y + 10 & \text{since } z \leq 10
\end{align*}
\]

- easier to read when split across lines
- “calculation block”, includes explanations in right column
  proof by calculation means using a calculation block
- “=” or “≤” relates that line to the \text{previous} line
Calculation Blocks

• Chain of “=” shows first = last

\[
\begin{align*}
a &= b \quad &\text{since} \ a &= b \\
&= c \quad &\text{since} \ b &= c \\
&= d \quad &\text{since} \ c &= d \\
\end{align*}
\]

– proves that \( a = d \)

– all 4 of these are the same number
Calculation Blocks

- Chain of “=” and “≤” shows first ≤ last

\[
\begin{align*}
x + z &= y + z & \text{since } x = y \\
\leq y + 10 & \quad \text{since } z \leq 10 \\
= y + 3 + 7 & \quad \text{since } y + 3 \leq w \\
\leq w + 7 & \quad \text{since } y + 3 \leq w
\end{align*}
\]

- each number is equal or strictly larger than previous

- analogous for “≥”
Using Calculation to Prove Correctness

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

• Known facts “x \geq 1” and “y \geq 1”

• Correct if the return value is a positive integer

\[
\begin{align*}
    x + y & \geq x + 1 & \text{since } y \geq 1 \\
    & \geq 1 + 1 & \text{since } x \geq 1 \\
    & = 2 \\
    & \geq 1
\end{align*}
\]

– calculation shows that \( x + y \geq 1 \)
Using Calculation to Prove Correctness

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

• Known facts “x ≥ 9” and “y ≥ -8”

• Correct if the return value is a positive integer

x + y
// 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 ≥ 9” and “y ≥ -8”

- Correct if the return value is a positive integer

\[
\begin{align*}
  x + y &\geq x + -8 & \text{since } y \geq -8 \\
  &\geq 9 - 8 & \text{since } x \geq 9 \\
  &= 1
\end{align*}
\]
Using Calculation to Prove Correctness

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

• Known facts “x ≥ 4” and “y ≥ 5”

• Correct if the return value is 10 or larger

    x + y
Using Calculation to Prove Correctness

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

• Known facts “x ≥ 4” and “y ≥ 5”

• Correct if the return value is 10 or larger

\[
x + y \geq x + 5 \quad \text{since } y \geq 5
\]
\[
\geq 4 + 5 \quad \text{since } x \geq 4
\]
\[
= 9
\]

proof doesn’t work because the code is wrong!
Using Calculation to Prove Correctness

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

• Known facts “x > 8” and “y > –9”

• Correct if the return value is a positive integer

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

since y > -9
since x > 8

proof doesn’t work because the proof is wrong

warning: avoid using “>” (or “<“) multiple times in a calculation block
Using Definitions in Calculations

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

• For example:

\[
\begin{align*}
\text{func } \text{sum}(\text{nil}) & := 0 \\
\text{sum}(\text{cons}(x, L)) & := x + \text{sum}(L) \quad \text{for any } x \in \mathbb{Z} \\
\end{align*}
\]

and any \( L \in \text{List} \)

• Can cite facts such as
  – \( \text{sum}(\text{nil}) = 0 \)
  – \( \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) = a + \text{sum}(\text{cons}(b, \text{nil})) \)

second case of definition with \( x = a \) and \( L = \text{cons}(b, \text{nil}) \)
Using Definitions in Calculations

\[
\text{func } \text{sum(nil)} := 0 \\
\text{sum(cons(x, L))} := x + \text{sum}(L) \quad \text{for any } x \in \mathbb{Z} \quad \text{and any } L \in \text{List}
\]

- **Know** “\(a \geq 0\)”, “\(b \geq 0\)”, and “\(L = \text{cons}(a, \text{cons}(b, \text{nil}))\)”

- **Prove the** “\(\text{sum}(L)\)” **is non-negative**

\[
\text{sum}(L)
\]
Using Definitions in Calculations

\[
\text{func } \text{sum}(\text{nil}) := 0 \\
\text{sum}(\text{cons}(x, L)) := x + \text{sum}(L) \quad \text{for any } x \in \mathbb{Z} \\
\text{and any } L \in \text{List}
\]

- **Know** “\(a \geq 0\)”, “\(b \geq 0\)”, and “\(L = \text{cons}(a, \text{cons}(b, \text{nil}))\)”

- **Prove the** “\(\text{sum}(L)\)” **is non-negative**

\[
\text{sum}(L) = \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) \quad \text{since } L = \text{cons}(a, \text{cons}(b, \text{nil})) \\
= a + \text{sum}(\text{cons}(b, \text{nil})) \quad \text{def of sum} \\
= a + b + \text{sum}(\text{nil}) \quad \text{def of sum} \\
= a + b \quad \text{def of sum} \\
\geq 0 + b \quad \text{since } a \geq 0 \\
\geq 0 \quad \text{since } b \geq 0
\]
Proof by Calculation
What We Get from Reasoning

• If the proof works, the code is correct
  – why reasoning is useful for finding bugs

• If the code is incorrect, the proof will not work

• If the proof does not work, the code is probably wrong
  could potentially be an issue with the proof (e.g., two “<”s)
  but that is a rare occurrence
Finding Facts at a Return Statement

• Consider this code

```javascript
// 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 ≥ 0”, “b ≥ 0”, and “L = cons(…)”
// 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 ≤ -1”

  x + y
// 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 ≤ -1”

x + y ≤ x + -1 \quad \text{since } y ≤ -1
< x + 0 \quad \text{since } -1 < 0
= x
// 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 ≥ 0”
// 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 ≥ 0”

x - 1 < x + 0 since -1 < 0
= x
Proving Correctness with Conditionals

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

• Conditionals give us extra known facts
  – get known facts from
    1. specification
    2. conditionals
    3. constant declarations
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

```javascript
// 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 : ℤ, y : ℤ, and x < y - 1`
- multiple claims to prove: `x < r` and `r < y`
  where “r” is the return value
- requires two calculation blocks
Recall: Max With an Imperative Specification

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

straight from the spec
(imperative spec)
// 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;
  }
};

- Three different facts to prove at each return

- Two known facts in each branch (return value is “r”):
  - then branch:   a ≥ b and  r = a
  - else branch:   a < b and  r = b

not straight from the spec (declarative spec)
Example Correctness with Conditionals

```javascript
// 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;  // Know a ≥ b and r = a
  } else {
    return b;
  }
};
```

- Correctness of return in “then” branch:
  - r = a holds so “r = a or r = b” holds,
  - r = a holds so “r ≥ a” holds, and

\[
\begin{align*}
r &= a \\
   &\geq b & \text{since } a \geq b
\end{align*}
\]
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;  // Know a < b and r = b
  }
};

• Correctness of return in “else” branch:
  – r = b holds so “r = a or r = b” holds,
  – r = b holds so “r ≥ b” holds, and
  – r ≥ a holds since we have r > a:
    
    r   = b
    > a     since a < b
Sum of a List

```javascript
const f = (a: bigint, b: bigint): bigint => {
    const L: List = cons(a, cons(b, nil));
    const s: bigint = sum(L);  // = a + b
    ...
};
```

- Can prove the claim in the comments by calculation

```
func sum(nil) := 0
sum(cons(x, L)) := x + sum(L)  for any x ∈ ℤ and any L ∈ List
```
Sum of a List

```javascript
const f = (a: bigint, b: bigint): bigint => {
  const L: List = cons(a, cons(b, nil));
  const s: bigint = sum(L);  // = a + b

  ...
};
```

- Can prove the claim in the comments by calculation

```
sum(L) = sum(cons(a, cons(b, nil)))
  = a + sum(cons(b, nil))
  = a + b + sum(nil)
  = a + b
```

**func** sum(nil) := 0

sum(cons(x, L)) := x + sum(L) for any \(x \in \mathbb{Z}\) and any \(L \in \text{List}\)
Sum of a List

```typescript
const f = (a: bigint, b: bigint): bigint => {
  const L: List = cons(a, cons(b, nil));
  const s: bigint = sum(L);  // = a + b
  ...
}
```

- Can prove the claim in the comments by calculation

  \[
  \text{sum(cons(a, cons(b, nil)))} = ... = a + b
  \]

- For which values of \(a\) and \(b\) does this hold?

  holds for any \(a \in \mathbb{Z}\) and \(b \in \mathbb{Z}\)
What We Have Proven

• We proved by calculation that

\[ \text{sum}(\text{cons}(a, \text{cons}(b, \text{nil}))) = a + b \]

• This holds for any \( a \in \mathbb{Z} \) and \( b \in \mathbb{Z} \)

• We have proven \textit{infinitely} many facts
  – \( \text{sum}(\text{cons}(3, \text{cons}(5, \text{nil}))) = 8 \)
  – \( \text{sum}(\text{cons}(-5, \text{cons}(2, \text{nil}))) = -3 \)
  – \( \ldots \)
  – replacing all the ‘a’s and ‘b’s with those numbers gives a calculation proving the “=” for those numbers
What We Have Proven

• We proved by calculation that

\[ \text{sum(cons(a, cons(b, nil)))} = a + b \]

for any \( a, b \in \mathbb{Z} \)

• We can use this fact for any \( a \) and \( b \) we choose
  – our proof is a “recipe” that can be used for any \( a \) and \( b \)
  – just as a function can be used with any argument values, our proof can be used with any values for the “any” variables (any values satisfying the specification)
  – use “for any ...” to make clear which things are variables

• This is called a “direct proof” of the “for any” claim
Binary Trees
Binary Trees

```plaintext
type Tree := empty | node(x : ℤ, L : Tree, R : Tree)
```

- Inductive definition of binary trees of integers

```
node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty))))
```

![Binary Tree Diagram]

1
 /   
2     3
 /     
2     4
```
Height of a Tree

type Tree := empty | node(x: \mathbb{Z}, L: Tree, R: Tree)

- Height of a tree: “maximum steps to get to a leaf”
**Height of a Tree**

*Type definition*

\[
\text{type Tree := empty | node(x: } \mathbb{Z}, \text{ L: Tree, R: Tree)}
\]

*Mathematical definition of height*

\[
\text{func height(empty)} := \\
\text{height(node(x, L, R))} := \\
\text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree}
\]
**Height of a Tree**

\[
\text{type } \text{Tree} \ := \ \text{empty} \mid \text{node}(x : \mathbb{Z}, L : \text{Tree}, R : \text{Tree})
\]

- **Mathematical definition of height**

\[
\begin{align*}
\text{func } \text{height} \left( \text{empty} \right) & \ := \ -1 \\
\text{height} \left( \text{node}(x, L, R) \right) & \ := \ 1 + \max(\text{height}(L), \text{height}(R)) \\
& \quad \text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree}
\end{align*}
\]
Using Definitions in Calculations

\[
\text{func height(} \text{empty}\text{) } := -1 \\
\text{height(} \text{node}(x, L, R)\text{) } := 1 + \max(\text{height}(L), \text{height}(R)) \\
\text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree}
\]

- **Suppose** “\(T = \text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))\)”

- **Prove that** \(\text{height}(T) = 1\)

\(\text{height}(T)\)
Using Definitions in Calculations

\[
\text{func } \text{height}(\text{empty}) := -1 \\
\text{height}(\text{node}(x, L, R)) := 1 + \max(\text{height}(L), \text{height}(R)) \\
\quad \text{for any } x \in \mathbb{Z} \text{ and any } L, R \in \text{Tree}
\]

- **Suppose** “\(T = \text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))\)”

- **Prove that** \(\text{height}(T) = 1\)

\[
\begin{align*}
\text{height}(T) &= \text{height}(\text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))) & \text{since } T = \ldots \\
&= 1 + \max(\text{height}(\text{empty}), \text{height}(\text{node}(2, \text{empty}, \text{empty}))) & \text{def of height} \\
&= 1 + \max(-1, \text{height}(\text{node}(2, \text{empty}, \text{empty}))) & \text{def of height} \\
&= 1 + \max(-1, 1 + \max(\text{height}(\text{empty}), \text{height}(\text{empty}))) & \text{def of height} \\
&= 1 + \max(-1, 1 + \max(-1, \text{height}(\text{empty}))) & \text{def of height} \\
&= 1 + \max(-1, 1 + \max(-1, -1)) & \text{def of height} \\
&= 1 + \max(-1, 1 + -1) & \text{def of max} \\
&= 1 + \max(-1, 0) & \text{def of max} \\
&= 1 + 0 \\
&= 1
\end{align*}
\]
Trees

• Trees are inductive types with a constructor that has 2+ recursive arguments

• These come up all the time...
  – no constructors with recursive arguments = “generalized enums”
  – constructor with 1 recursive arguments = “generalized lists”
  – constructor with 2+ recursive arguments = “generalized trees”

• Some prominent examples of trees:
  – HTML: used to describe UI
  – JSON: used to describe just about any data
Recall: HTML

- Nesting structure describes the tree

```html
<div>
  <p id="firstParagraph"> Some Text </p>  
  <br>
  <div>
    <p>Hello</p>
  </div>
</div>
```

![Diagram of HTML nesting structure]
Custom Tags for Modularity

- The React library lets you write “custom tags”
  - functions that return HTML

```
return (  
  <div>
    <p>Hi, Alice!</p>
    <p>Hi, Bob!</p>
  </div>);
```

can become

```
return (  
  <div>
    <SayHi name="Alice"/>
    <SayHi name="Bob"/>
  </div>);
```
Custom Tags for Modularity

- The React library lets you write “custom tags”

```javascript
return () {
  <div>
    <SayHi name="Alice"/>
    <SayHi name="Bob"/>
  </div>);
```

makes two calls to this function

```javascript
const SayHi = (props: {name: string}): JSX.Element => {
  return <p>Hi, {props.name}</p>;
};
```

- attributes are passed as a record argument (“props”)

Custom Tags for Modularity

```javascript
return (  
  <div>
    <SayHi name="Alice" lang="es"/>
    <SayHi name="Bob"/>
  </div>);

makes two calls to this function

```javascript

type SayHiProps = {name: string, lang?: string};

const SayHi = (props: SayHiProps): JSX.Element => {
  if (props.lang === "es") {
    return <p>Hola, {props.name}</p>;
  } else {
    return <p>Hi, {props.name}</p>;
  }
};
```
Custom Tags for Modularity

• The React library lets you write “custom tags”
  – attributes are passed as a record argument (“props”)

• In `render`, React will paste the parts together:

```jsx
<div>
  <SayHi name="Alice" lang="es"/>
  <SayHi name="Bob"/>
</div>
```

becomes

```jsx
<div>
  <p>Hola, Alice!</p>
  <p>Hi, Bob!</p>
</div>
```
Custom Tags for Modularity

• HTML literal syntax allows any tags

    return (  
        <div>  
            <SayHi name="Alice" lang="es"/>  
            <SayHi name="Bob"/>  
        </div>);  

    – evaluates to a tree with two nodes with tag name "SayHi"  
    – this matters when testing (comes up in HW3)

• React’s render method is what calls SayHi  
    – HTML returned is substituted where the “SayHi” tag was
React Render

• React’s render pastes strings together

```javascript
const name: string = "Fred";
return <p>Hi, {name}</p>;
```

returns a different tree than

```javascript
return <p>Hi, Fred</p>;
```

– in first tree, “p” tag has one child
– in second tree, “p” tag has two children
– render method concatenates text children into one string

• These differences matter for testing!
React Render

- React’s `render` pastes arrays into child list

```javascript
const L = [<span>Hi</span>, <span>Fred</span>];
return <p>{L}</p>;
```

returns a different tree than

```javascript
return <p><span>Hi</span><span>Fred</span></p>;
```

- in first tree, “p” tag has one child
- in second tree, “p” tag has two children
- render method turns the first into the second

- These differences matter for testing!