CSE 331

Reasoning About Straight-Line Code

James Wilcox & Kevin Zatloukal
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”

  type T := C(x : ℤ) | D(x : 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

  C(1)
  D(2, C(1))
  D(3, D(2, C(1)))

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

\[
\begin{align*}
C(1) \neq C(2) \\
D(2, C(1)) \neq D(3, C(1)) \\
D(2, C(1)) \neq D(2, C(2)) \\
D(1, D(2, C(3))) = D(1, D(2, C(3)))
\end{align*}
\]
Natural Numbers

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

- **Inductive definition of the natural numbers**

  - zero \(0\)
  - succ(zero) \(1\)
  - succ(succ(zero)) \(2\)
  - succ(succ(succ(zero))) \(3\)

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

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

- Inductive definition of the even natural numbers

\[
\begin{align*}
\text{zero} & : 0 \\
\text{two-more(zero)} & : 2 \\
\text{two-more(two-more(zero))} & : 4 \\
\text{two-more(two-more(two-more(zero)))} & : 6
\end{align*}
\]

much better notation
Lists

\[
\text{type} \ \text{List} := \ \text{nil} \mid \text{cons}(x : \mathbb{Z}, \ L : \text{List})
\]

- Inductive definition of lists of integers

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

Array notation:
“Lists are the original data structure for functional programming, just as arrays are the original data structure of imperative programming”

Ravi Sethi

we will work with lists in HW3+ and arrays HW7+
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
const T1 = {kind: “T1”, a: bigint, b: number};
const 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

\texttt{type} \ T := \ C(x:\mathbb{Z}) \ | \ D(x:\mathbb{S}^*, t: T)

- Implement in TypeScript as

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

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

- 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

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

- Implemented in TypeScript as

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

- fields should also be “readonly”

How to check if a value `mylist` is nil?

```typescript
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...

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

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

• Can now write code like this:

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

// Equality function for Lists
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);
        }
    }
};
```
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

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

- 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};

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)
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 \( n \) recurse on \( n.\text{prev} \)
    \( n.\text{prev} \) 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 *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 \( x \in \mathbb{Z} \)
and any \( S \in \text{List} \)

– any list is either nil or \( \text{cons}(x, L) \) for some \( x \) and \( L \)
– cases are exclusive and exhaustive
Length of a List

- **Mathematical definition of length**

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

- **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)

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

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

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

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

- **Translation to TypeScript**

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

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Testing</th>
<th>Tools</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>small # of inputs</td>
<td>exhaustive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>straight from spec</td>
<td>heuristics</td>
<td>type checking</td>
<td>code reviews</td>
</tr>
<tr>
<td>no mutation</td>
<td>“”</td>
<td>libraries</td>
<td>calculation induction</td>
</tr>
<tr>
<td>local variable mutation</td>
<td>“”</td>
<td>“”</td>
<td>Floyd logic</td>
</tr>
<tr>
<td>array mutation</td>
<td>“”</td>
<td>“”</td>
<td>for-any facts</td>
</tr>
<tr>
<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

```plaintext
func sum(nil) :=
    sum(cons(x, S)) := for any x ∈ ℤ
    and any S ∈ List
```
Sum of Values in a List

- Mathematical definition of sum

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

- 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

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

note that this English spec is declarative!

- Look at some examples...

<table>
<thead>
<tr>
<th>L</th>
<th>rev(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>rev(L)</td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>cons(3, nil)</td>
<td>cons(3, nil)</td>
</tr>
<tr>
<td>cons(2, cons(3, nil))</td>
<td>cons(3, cons(2, nil))</td>
</tr>
<tr>
<td>cons(1, cons(2, cons(3, nil)))</td>
<td>cons(3, cons(2, cons(1, nil)))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Definition of Reversal of a List

• Look at some examples...

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

• Draw a picture?

1 2 3
move 1 to end
reverse this too
Reversing A Lists

• Draw a picture?

• Mathematical definition of rev

\[
\text{func } \text{rev}(\text{nil}) := \text{rev}(\text{cons}(x, S)) :=
\]

for any \( x \in \mathbb{Z} \) and 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 \textit{simplest}

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

• Always make definitions as \textit{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.
Reasoning

• “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
# Correctness Levels

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

 HW3  
 HW4  
 HW5  
 HW6  
 HW7  
 HW8
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:
  – $n \in \mathbb{N}$ (or $n \in \mathbb{Z}$ and $n \geq 0$)
  – $m = 2n$
Facts

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

    // 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 \( \text{len}(\text{nil}) = 0 \)
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 $x = y$  
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

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

– easier to read when split across lines
– "calculation block", includes explanations in right column
  proof by calculation means using a calculation block
– "=" or "\leq" relates that line to the previous line
**Calculation Blocks**

- **Chain of “=” shows first = last**

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

- proves that \( a = d \)
- all 4 of these are the same number
Calculation Blocks

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

\[ x + z = y + z \]  \text{since } x = y

\[ \leq y + 10 \]  \text{since } z \leq 10

\[ = y + 3 + 7 \]

\[ \leq w + 7 \]  \text{since } y + 3 \leq w

– each number is equal or strictly larger that previous
  last number is strictly larger than the first number

– analogous for “≥”
Using Calculation to Prove Correctness

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

• Known facts “x ≥ 1” and “y ≥ 1”

• Correct if the return value is a positive integer

  x + y
Using Calculation to Prove Correctness

```javascript
// 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

\[
x + y \geq x + 1 \quad \text{since } y \geq 1
\]
\[
\geq 1 + 1 \quad \text{since } x \geq 1
\]
\[
= 2
\]
\[
\geq 1
\]

- 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
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

\[
\begin{align*}
x + y & \geq x + -8 \\
& \geq 9 - 8 \\
& = 1
\end{align*}
\]
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 ≥ x + 5 since y ≥ 5
   ≥ 4 + 5 since x ≥ 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

\[
x + y > x + -9 \\
> x + -9 \\
> 8 - 9 \\
= -1
\]

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

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

- **Know** “a ≥ 0”, “b ≥ 0”, and “L = cons(a, cons(b, nil))”

- **Prove the** “sum(L)” **is non-negative**

    sum(L)
Using Definitions in Calculations

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

- **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**

\[
\begin{align*}
\text{sum}(L) & = \text{sum(cons(a, cons(b, nil)))} \\
& = a + \text{sum(cons(b, nil))} \\
& = a + b + \text{sum(nil)} \\
& = a + b \\
& \geq 0 + b \\
& \geq 0 \\
\end{align*}
\]

\text{since } L = \text{cons}(a, \text{cons}(b, \text{nil})) \\
\text{def of sum} \\
\text{def of sum} \\
\text{def of sum} \\
\text{since } a \geq 0 \\
\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

```
// 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(...)”
Proving Correctness with Conditionals

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

• Known fact in then (top) branch: “y ≤ -1”

  x + y
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”

\[
\begin{align*}
x + y &\leq x + (-1) & \text{since } y \leq -1 \\
&< x + 0 & \text{since } -1 < 0 \\
&= x
\end{align*}
\]
// 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”

\[
x - 1 < x + 0 \quad \text{since } -1 < 0
\]
\[
= x
\]
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

find facts by reading along path from top to the return statement
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;
  }
};
Example Correctness with Conditionals

// Returns \( r \) with (\( r = a \) or \( r = b \)) and \( r \geq a \) and \( r \geq b \)
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: \( a \geq b \) and \( r = a \)
  – else branch: \( a < b \) and \( r = b \)

\( \text{not straight from the spec} \)
\( \text{(declarative spec)} \)
Example Correctness with Conditionals

```typescript
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 “\( r \geq a \)” holds, and
  - \( r = a \geq b \) since \( a \geq b \)
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;
  } 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 “\( r \geq b \)” holds, and
  - \( r \geq a \) 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: number = sum(L);  // = a + b
    ...
};

• Can prove the claim in the comments by calculation

    sum(L)
```

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

const \( f = (a: \text{number}, b: \text{number}): \text{number} \Rightarrow \{
  \text{const } L: \text{List} = \text{cons}(a, \text{cons}(b, \text{nil}));
  \text{const } s: \text{number} = \text{sum}(L); \quad // = a + b
  \ldots
}\);
```javascript
const f = (a: number, b: number): number => {
    const L: List = cons(a, cons(b, nil));
    const s: number = sum(L); // = a + b
    ...
}
```

- Can prove the claim in the comments by calculation

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

- 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

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

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

- We have proven infinitely many facts
  - \( \text{sum(cons(3, cons(5, nil)))} = 8 \)
  - \( \text{sum(cons(-5, cons(2, nil)))} = -3 \)
  - ...
  - 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

\[ \text{sum(cons(a, cons(b, nil)))} = a + b \quad \text{for any } a, b \in \mathbb{Z} \]

• 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

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

- Inductive definition of binary trees of integers

\texttt{node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty))})
Height of a Tree

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

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

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

- Mathematical definition of height

\[ \textbf{func} \text{ height(empty)} := \]
\[ \text{height(node}(x, L, R)) := \]

for any \( x \in \mathbb{Z} \) and any \( L, R \in \text{Tree} \)
**Height of a Tree**

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

- **Mathematical definition of height**

\[
\text{func } \text{height}(\text{empty}) := -1 \\
\text{height}(\text{node}(x, L, R)) := 1 + \max(\text{height}(L), \text{height}(R))
\]

for any \( x \in \mathbb{Z} \) and any \( L, R \in \text{Tree} \)
Using Definitions in Calculations

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

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

- **Prove that** \(\text{height}(T) = 1\)
Using Definitions in Calculations

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

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

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

\[
\text{height}(T) = \text{height}(\text{node}(1, \text{empty}, \text{node}(2, \text{empty}, \text{empty}))) \quad \text{since } T = \ldots \\
= 1 + \max(\text{height}(\text{empty}), \text{height}(\text{node}(2, \text{empty}, \text{empty}))) \quad \text{def of height} \\
= 1 + \max(-1, \text{height}(\text{node}(2, \text{empty}, \text{empty}))) \quad \text{def of height} \\
= 1 + \max(-1, 1 + \max(\text{height}(\text{empty}), \text{height}(\text{empty}))) \quad \text{def of height} \\
= 1 + \max(-1, 1 + \max(-1, -1)) \quad \text{def of height (x 2)} \\
= 1 + \max(-1, 1 + -1) \quad \text{def of max} \\
= 1 + \max(-1, 0) \quad \text{def of max} \\
= 1 + 0 \\
= 1
\]
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