# CSE 331

## Inductive Data & Recursion

Kevin Zatloukal

# Recall: Basic Data Types

- In math, the basic data types are "sets"
  - sets are collections of objects called **elements**
  - write $x \in S$ to say that "x" is an element of set "S", and $x \notin S$ to say that it is not.

- Examples:

| | |
|---|---|
| $x \in \mathbb{Z}$ | x is an integer |
| $x \in \mathbb{N}$ | x is a non-negative integer (natural) |
| $x \in \mathbb{R}$ | x is a real number |
| $x \in \mathbb{B}$ | x is T or F (boolean) |
| $x \in \mathbb{S}$ | x is a character |
| $x \in \mathbb{S}^*$ | x is a string |

non-standard names

# Recall: Ways to Create New Types In Math

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
- **Union Types** $\mathbb{S}^* \cup \mathbb{N}$
  - contains every object in either (or both) of those sets
- **Tuple Types** $\mathbb{N} \times \mathbb{N}$
  - pair of two numbers
  - can do tuples of 3, 4, or more elements also

# Recall: TypeScript type system

- TypeScript supports records, union, tuples
  - supports real, boolean, and string

    does not have integer, natural, or character types
  - supports finite subsets of strings as unions of literal types

- Union types supported via type “narrowing”
  - “if” statements can check types at run time
  - TypeScript updates its type information for each branch

- Java and TypeScript are fundamentally different
  - nominal vs structural typing

# Inductive Data

# Inductive Data Types

- Missing one more way of defining types
  - arguably the most important
- 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"

$$\textbf{type}\ \mathrm{T} := \ \mathrm{A}(\mathrm{x} : \mathbb{Z}) \ | \ \mathrm{B}(\mathrm{x} : \mathbb{Z},\ \mathrm{y} : \mathrm{T})$$

  - second constructor is recursive
  - can have any number of arguments (even none)

    will leave off the parentheses when there are none

- Examples of elements

$\mathrm{A}(1)$
$\mathrm{B}(2, \mathrm{A}(1))$
$\mathrm{B}(3, \mathrm{B}(2, \mathrm{A}(1)))$

in math, these are **<u>not</u>** function calls

# Natural Numbers

$$\textbf{type } \mathbb{N} := \text{zero} \mid \text{succ}(\text{n} : \mathbb{N})$$

- **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

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

- **Inductive definition of the even natural numbers**

| | | |
|---|---|---|
| zero | 0 | |
| two-more(zero) | 2 | much better notation |
| two-more(two-more(zero)) | 4 | |
| two-more(two-more(two-more(zero))) | 6 | |

# Lists

$$\textbf{type}\ \mathrm{List} := \mathrm{nil} \mid \mathrm{cons}(x : \mathbb{Z},\ L : \mathrm{List})$$

- **Inductive definition of lists of integers**

$$\begin{aligned}
&\mathrm{nil} && \approx [\,] \\
&\mathrm{cons}(3, \mathrm{nil}) && \approx [3] \\
&\mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil})) && \approx [2, 3] \\
&\mathrm{cons}(1, \mathrm{cons}(2, \mathrm{cons}(3, \mathrm{nil}))) && \approx [1, 2, 3]
\end{aligned}$$

array notation

# Inductive Data Types in TypeScript

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

- We will cobble them together as follows…

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

  - union of a literal type and a record type
  - the “kind” field is technically not necessary

    can already distinguish string from record

    useful in other cases to distinguish different constructors

# Inductive Data Type Design Pattern

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

- Implement in TypeScript as

```
type T = "A"
       | "B"
       | {kind: "C", x: number}
       | {kind: "D", x: number, t: T};
```

- Another design pattern
  - work around the limitations of TypeScript (no inductive types)
  - everything above should also be "readonly"

# Inductive Data Types in TypeScript

- **Make this look more like math notation…**

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

const nil: List = “nil”;

function cons(hd: number, tl: List) {
  return {kind: ”cons”, hd: hd, tl: tl};
}
```

# Inductive Data Types in TypeScript

- Make this look more like math notation…

```
const nil: List = "nil";

function cons(hd: number, tl: List)
```

- Can now write code like this:

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

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

# Inductive Data Types in TypeScript

- Make this look more like math notation…

```
const nil: List = "nil";

function cons(hd: number, tl: List)
```

- Still not perfect:
  - JS "===" (references to same object) does not match "="

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

  - would need to define an `equal` function for this

# Inductive Data Types in TypeScript

- Objects are equal if they were built the same way

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

function equal(L: List, R: List): boolean {
  if (L === nil) {
    return R === nil;
  } else {
    if (R === nil) {
      return false;
    } else {
      return equal(L.tl, R.tl) && L.hd === R.hd;
    }
  }
}
```

# 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

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

What does this compute? $2^n$

# Recall: Natural Numbers

$$\textbf{type } \mathbb{N} := \text{ zero } \mid \text{ succ(prev: } \mathbb{N})$$

- **Inductive definition of the natural numbers**

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

# Recall: Natural Numbers

$$\textbf{type } \mathbb{N} := \text{zero} \mid \text{succ}(\text{prev}: \mathbb{N})$$

- **Definition in TypeScript**

```
type Nat = “zero” | {kind: “succ”, prev: Nat};

const zero: Nat = “zero”;

function succ(prev: Nat) {
  return {kind: ”succ”, prev: prev};
}
```

# Induction on Natural Numbers

Could use a type that only allows natural numbers:

```
function f(n: Nat): number {
  if (n === zero) {
    return 1;
  } else {
    return 2 * f(n.prev);
  }
}
```

n.prev represents "n – 1"

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 recurses on n.prev

    n.prev is the <u>argument</u> 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

- As with data, we have both math and code functions
  - our math notation looks like this:

$$\textbf{func}\ f(n) := 2n + 1 \qquad\qquad \text{for any } n : \mathbb{N}$$

- Reasoning is done with math
  - tools are language independent

- We need recursion to define interesting functions
  - we will primarily use structural recursion
  - we will show this by example

# Length of a List

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

- Mathematical definition of length

| | | | |
|---|---|---|---|
| **func** len(nil) | := 0 | | |
| len(cons(x, L)) | := 1 + len(L) | for any x ∈ $\mathbb{Z}$ and any L ∈ List | |

  – any list is either nil or cons(x, L) for some x and L
  – one of these two rules always applies
  – an example of "*pattern matching*"

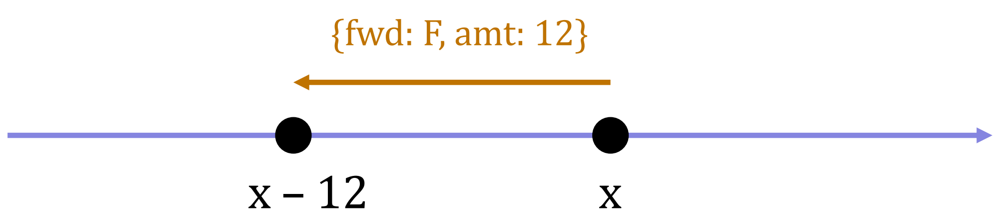
# More Pattern Matching

- Define a function by an exhaustive set of patterns

**type** Move := $\{\text{fwd} : \mathbb{B}, \text{amt} : \mathbb{N}\}$

| | | |
|---|---|---|
| **func** change({fwd: T, amt: n}) | := n | for any n : $\mathbb{N}$ |
| change({fwd: F, amt: n}) | := -n | for any n : $\mathbb{N}$ |

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




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

  every Move either has forward as T or F

# Length of a List

- **Mathematical definition of length**

**func** len(nil) := 0
len(cons(x, L)) := 1 + len(L) for any $x \in \mathbb{Z}$ and any $L \in$ List

- **Translation to TypeScript**

```
function len(L: List): number {
  if (L === nil) {
    return 0;
  } else {
    return 1 + len(L.tl);
  }
}
```

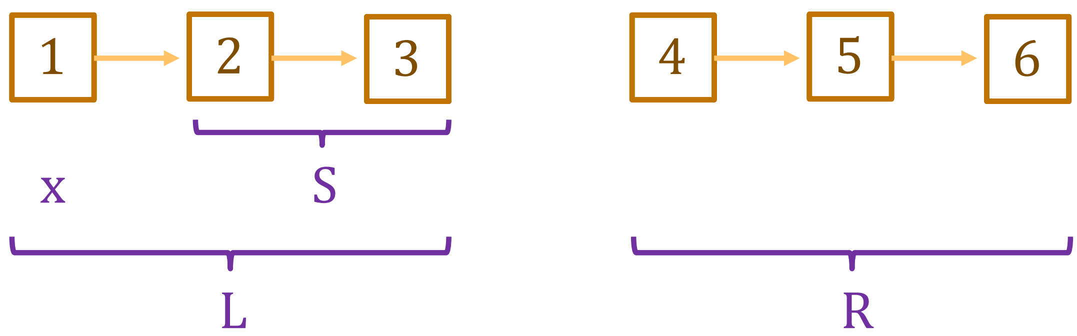
<u>Level 0</u>
straight from the spec

# Concatenating Two Lists

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

| | | | |
|---|---|---|---|
| **func** concat(nil, R) | := | R | for any $R \in \text{List}$ |
| $\text{concat}(\text{cons}(x, S), R)$ | := | $\text{cons}(x, \text{concat}(S, R))$ | for any $x \in \mathbb{Z}$ and any $L, R \in \text{List}$ |

  – $\text{concat}(\text{L}, \text{R})$ **defined by pattern matching on** $\text{L}$ **(not** $\text{R}$**)**



# Concatenating Two Lists

- **Mathematical definition of concat**

$$
\begin{array}{lll}
\textbf{func}\ \mathrm{concat}(\mathrm{nil}, R) & := R & \text{for any } R \in \mathrm{List} \\
\quad\ \mathrm{concat}(\mathrm{cons}(x, L), R) & := \mathrm{cons}(x, \mathrm{concat}(L, R)) & \text{for any } x \in \mathbb{Z} \text{ and} \\
 & & \quad \text{any } L, R \in \mathrm{List}
\end{array}
$$

- **Translation to TypeScript**

```
function concat(L: List, R: List): List {
  if (L === nil) {
    return R;
  } else {
    return cons(L.hd, concat(L.tl, R));
  }
}
```

Level 0

# 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…**

| L | sum(L) |
|---|---|
| nil | 0 |
| cons(1, nil) | 1 |
| cons(1, cons(2, nil)) | 1+2 |
| cons(1, cons(2, cons(3, nil))) | 1+2+3 |
| … | … |

# Definition of Sum of Values in a List

- **Look at some examples…**

| L | sum(L) |
|---|---|
| nil | 0 |
| cons(1, nil) | 1 |
| cons(1, cons(2, nil)) | 1+2 |
| cons(1, cons(2, cons(3, nil))) | 1+2+3 |
| … | … |

- **Mathematical definition**

**func** sum(nil) :=

sum(cons(x, L)) := for any $x \in \mathbb{Z}$ and any $L \in$ List

# Sum of Values in a List

- **Mathematical definition of sum**

$$
\begin{aligned}
\textbf{func } & \mathrm{sum}(\mathrm{nil}) && := 0 \\
& \mathrm{sum}(\mathrm{cons}(x, L)) && := x + \mathrm{sum}(L) && \text{for any } x \in \mathbb{Z} \\
& && && \text{and any } L \in \mathrm{List}
\end{aligned}
$$

- **Translation to TypeScript**

```
function sum(L: List): number {
  if (L === nil) {
    return 0;
  } else {
    return L.hd + sum(L.tl);
  }
}
```

Level 0

# Definition of Reversal of a List

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

- **Look at some examples…**

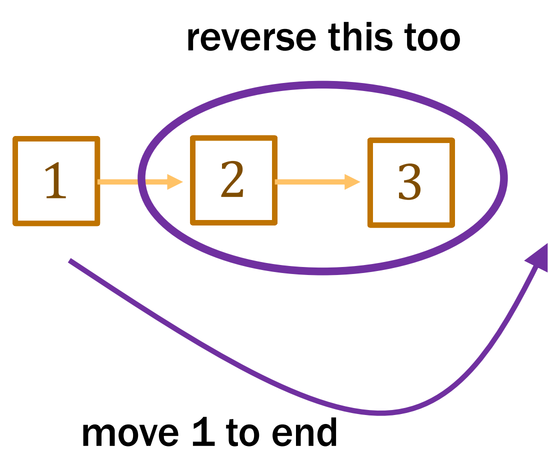
| L | rev(L) |
| --- | --- |
| nil | nil |
| cons(1, nil) | cons(1, nil) |
| cons(1, cons(2, nil)) | cons(2, cons(1, nil)) |
| cons(1, cons(2, cons(3, nil))) | cons(3, cons(2, cons(1, nil))) |
| … | … |

# Definition of Reversal of a List

- **Look at some examples…**

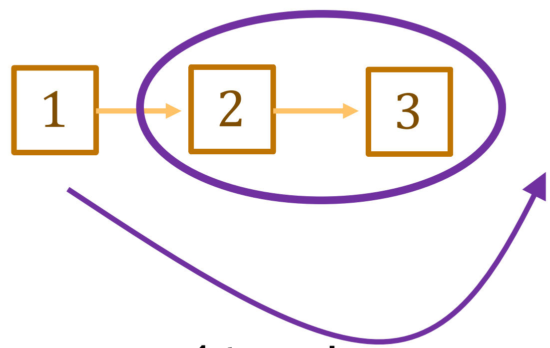
| L | rev(L) |
|---|---|
| nil | nil |
| cons(1, nil) | cons(1, nil) |
| cons(1, cons(2, nil)) | cons(2, cons(1, nil)) |
| cons(1, cons(2, cons(3, nil))) | cons(3, cons(2, cons(1, nil))) |

- **Draw a picture?**



# Reversing A Lists

- Draw a picture?




- Mathematical definition of rev

**func** rev(nil) :=

rev(cons(x, L)) :=    for any $x \in \mathbb{Z}$ and any $L \in$ List

# Reversing A Lists

- Mathematical definition of rev

$$
\begin{aligned}
\textbf{func}\ \ &\mathrm{rev}(\mathrm{nil}) &&:= \mathrm{nil} \\
&\mathrm{rev}(\mathrm{cons}(x, L)) &&:= \mathrm{concat}(\mathrm{rev}(L), \mathrm{cons}(x, \mathrm{nil})) \quad \text{for any } x \in \mathbb{Z} \text{ and} \\
& && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{any } L \in \mathrm{List}
\end{aligned}
$$

- 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