CSE 331

Inductive Data Types

Kevin Zatloukal

Administrivia

- Working on HW2 on your own

- **Understand** why we have the rules we do
  - why you need 2 tests per subdomain
  - why you need to test boundary cases
  - why you need 0–1–many recursive calls

- Starting HW3 material in lecture
  - full math notation linked under this lecture

Inductive Data Types

- Create new types using 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"

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

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

    will leave off the parentheses when there are none

- Examples of elements

$$\mathrm{C}(1)$$
$$\mathrm{D}(2, \mathrm{C}(1))$$
$$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$$

in math, these are **not** function calls

Inductive Data Types

- Each element is a description of how it was made

$$\mathrm{C}(1)$$
$$\mathrm{D}(2, \mathrm{C}(1))$$
$$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$$

- Equal when they were made *exactly* the same way
  - $\mathrm{C}(1) \neq \mathrm{C}(2)$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(3, \mathrm{C}(1))$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(2, \mathrm{C}(2))$
  - $\mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3))) = \mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3)))$

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
two-more(two-more(zero))	4
two-more(two-more(two-more(zero)))	6

much better notation

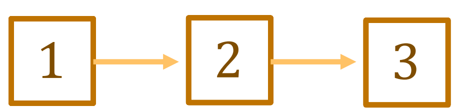
Lists

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

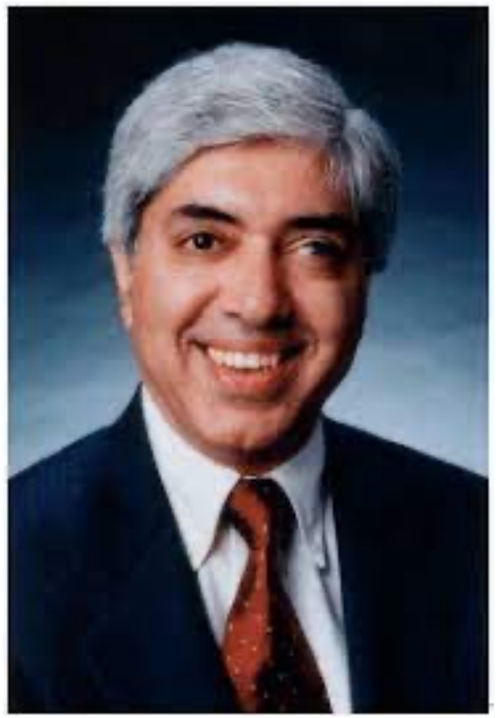
- **Inductive definition of lists of integers**

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

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 will cobble them together…

Literal Types

- A literal type includes only that literal

```
const x: "red" = "red";
const y: 1 = 1;
```

- This is useful for creating small sets

```
type Color = "red" | "green" | "blue";
const c: Color = "red";
```

- Java works around this with "enums"
  - objects that "represent" red, green, and blue

    example of a "design pattern"

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

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

const x: T1 | T2 = …;
if (x.kind === "T1") {    // legal for either type
  console.log(x.a);   // must be T1… x.a is a number
} else {
  console.log(x.b);   // must be T2… x.b is a string
}
```

Inductive Data Type Design Pattern

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

- Implement in TypeScript as

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

- A design pattern
  - work around the limitations of TypeScript (no inductive types)

- Will use a simpler representation with no arguments
  - rather than `{kind: "A"}`, we'll use just `"A"`

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};
```

- **TypeScript's narrowing still works well**
  - **if** `t !== "A"` **and** `t !== "B"`**, then** `t.kind` **makes sense**

    (and it is either `"C"` or `"D"`)

Inductive Data Types in TypeScript

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

- Becomes the following type in TypeScript

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

  - fields 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”;

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

Inductive Data Types in TypeScript

- Make this look more like math notation…

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

const cons = (hd: number, tl: List): 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";

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

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

```
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**

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

const 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
const f = (n: number): number => {
  if (n === 0) {
    return 1;
  } else {
    return 2 * f(n - 1);
  }
};
```

What does this compute? 2^n

Recall: Natural Numbers

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

- **Inductive definition of the natural numbers**

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

Recall: Natural Numbers

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

- Potential definition in TypeScript

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

const zero: Nat = “zero”;

const succ = (prev: Nat): Nat => {
  return {kind: ”succ”, prev: prev};
};
```

Induction on Natural Numbers

Could use a type that only allows natural numbers:

```
const 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 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.:

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

- 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: $\mathbb{Z}$, 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 $S \in$ 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**

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

- **Translation to TypeScript**

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

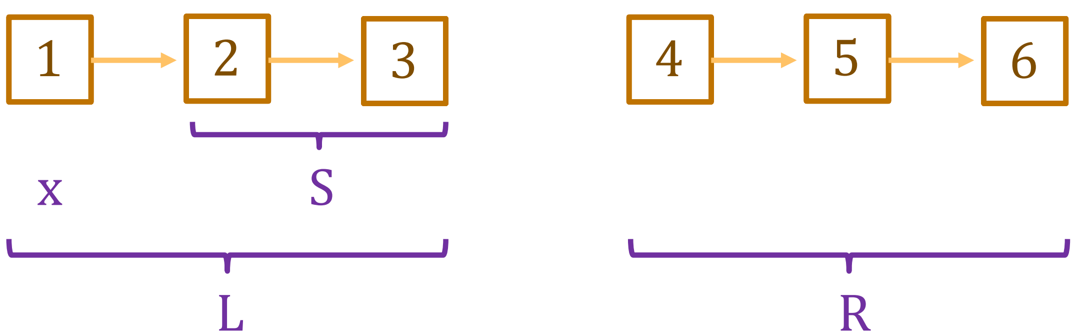
Level 0
straight from the spec

Concatenating Two Lists

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

func concat(nil, R)	:=	R	for any R ∈ List
concat(cons(x, S), R)	:=	cons(x, concat(S, R))	for any x ∈ ℤ and any S, R ∈ List

  – concat(L, R) **defined by pattern matching on** L **(not** R**)**



Concatenating Two Lists

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

func concat(nil, R)	:=	R	for any R ∈ List
concat(cons(x, S), R)	:=	cons(x, concat(S, R))	for any x ∈ ℤ and any S, R ∈ List

- **Translation to TypeScript**

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

Level 0