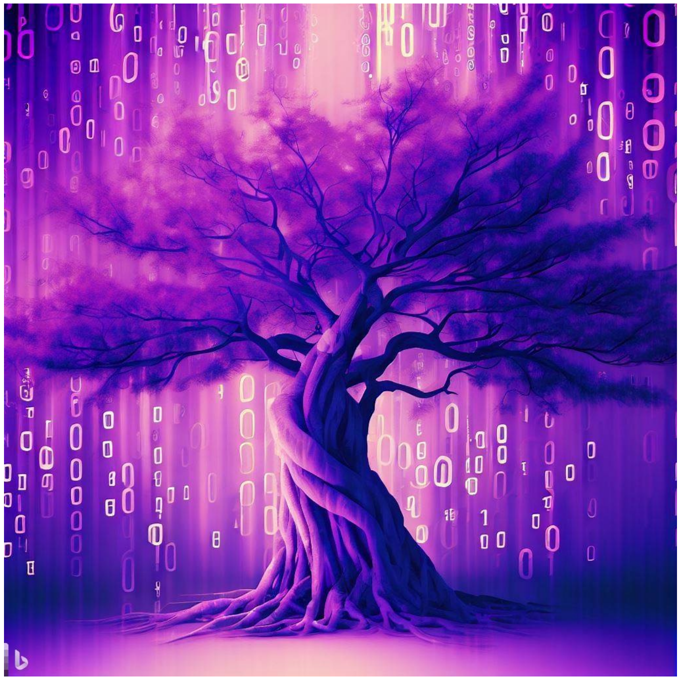
CSE 331

Data Types

Kevin Zatloukal



“binary tree” by DALL-E

Correctness

Language Features

- No need to rush out and learn all of JS / TS
- We will introduce language features along with the tools for reasoning about them
- Initially, we just need:
  - straight-line code (const / return)
  - conditionals (if)
  - recursion
- Will take couple weeks to learn to reason about them

Language Features

- Next three lectures
  1. Data types (data)
  2. Functions (code)
  3. Proofs (reasoning)

- Data is the natural place to start
  - functions operate on data, so you need data first
  - typically, the place to start when you **design** an app

    more on this later

Language Features

- Reasoning is "math"

- For Data & Code, we will define
  1. <u>Math</u> we use to think about them
  2. How to <u>model</u> a specific programming language in math

- Reasoning is language independent
- Modeling is language specific
  - e.g., how to do "string or number" in Java vs TypeScript

- I will use notation to distinguish which is which

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	no mutation	“	libraries	calculation induction
2	local variable mutation	“	“	Floyd logic
3	array / object mutation	“	“	rep invariants

amateurs

pros

Reasoning is what distinguishes professionals from amateurs

"Programming" by Trial & Error

- Beginning programmers often work by **trial & error**

    1. try something
    2. if that works, we're done!
    3. If not, go to 1

    (fine for level -1 only)

- Easy trick to catch this: take the computer away
    - good programmers can still function

      (can work on a programming problem at the beach!)
    - why interviews are without a computer

- Work toward getting it right the first time
    - carefully think through what the code is doing
    - we will work on this all quarter (starting small)

Data Types

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

Basic Data Types

Condition	Math	TypeScript	Up to Us
integer	$x \in \mathbb{Z}$	`number`	no fractional part
natural	$x \in \mathbb{N}$	`number`	non-negative
real	$x \in \mathbb{R}$	`number`	
boolean	$x \in \mathbb{B}$	`boolean`	
character	$x \in \mathbb{S}$	`string`	length 1
string	$x \in \mathbb{S}^*$	`string`	

we will often write
$x : \mathbb{Z}$ instead of $x \in \mathbb{Z}$

Basic Data Types of JavaScript

- **JavaScript includes the following types**

`number`	
`string`	
`boolean`	
`null`	
`undefined`	(another null)
`Object`	
`Array`	(special subtype of Object)

we won't use them until week 5/6

- **TypeScript also includes**

`unknown`	
`any`	(turns off type checking — do not use!)

Record Types

- JavaScript “Object” is something with “fields”

- JavaScript has special syntax for creating them

```
const p = {x: 1, y: 2};
console.log(p.x);  // prints 1
```

- The term “object” is potentially confusing
  - used for many things
  - I prefer it as shorthand for “mathematical object”

- Will refer to the math concept as a “record type”

Type Aliases

- TypeScript lets you give shorthand names for types

```
type Point = {x: number, y: number};

const p: Point = {x: 1, y: 2};
console.log(p.x);  // prints 1
```

- Always include the types when declaring variables
  - otherwise, TypeScript tries to “infer” the type, and the result is sometimes not what you expect

- In math, we will do this also

$$\textbf{type}\ \mathrm{Point} := \{x : \mathbb{N}, y : \mathbb{N}\}$$

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

Ways to Create New Types in TypeScript

- **Record Types** `{x: number, y: number}`
  - anything with *at least* fields "x" and "y"
- **Union Types** `string | number`
  - can be either one of these
- **Tuple Types** `[number, number]`
  - at runtime, this is an array of length 2
  - should really be " `readonly [number, number]` "

    likewise for "x" and "y" in the record above

Optional Values

- Records can have optional fields

```
type T = {a: number, b?: number};

const x: T = {a: 1};
```

  – type of “ `x.b` ” is “ `number | undefined` ”

- Functions can have optional arguments

```
function f(a: number, b?: number): number {
  console.log(b);
}
```

  – type of “ `b` ” is “ `number | undefined` ”

Type Narrowing

- Conditionals can change the known types

```
function f(a: number, b?: number): number {
  if (b === undefined) {
    console.log("b missing ☹");   // undefined
  } else {
    console.log(2 * b);           // number
  }
}
```

  – type checker "narrows" the type of " `b` " in each branch

Use "===" and "!=="
instead of Use "==" and "!="

Checking Types at Run Time

Condition	Code
x is undefined	`x === undefined`
x is null	`x === null`
x is a number	`typeof x === “number”`
x is an integer	**… and** `Math.floor(x) === x`
x is a string	`typeof x === “string”`
x is an object or array	`typeof x === “object”`
x is an array	`Array.isArray(x)`

Hard to check if x is a specific record type at runtime.
Much easier to let the type checker do this!

Checking Types at Run Time

- Can check if a field is present using “ `in` ”

- Allows you to distinguish between two record types:

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

const x: T1 | T2 = …;
if ("a" in x) {
  console.log(x.b);  // number
} else {
  console.log(x.b);  // string
}
```

Structural vs Nominal Typing

- TypeScript uses "**structural typing**"
  - sometimes called "duck typing"

    "if it walks like a duck and quacks like a duck, it's a duck"

```
type T1 = {a: number, b: number};
type T2 = {a: number, b: number};

const x: T1 = {a: 1, b: 2};
```

  - can pass " `x` " to a function expecting a " `T2` "!

Structural vs Nominal Typing

- Java uses "**nominal typing**"

```
class T1 { int a; int b; }
class T2 { int a; int b; }

T1 x = new T1();
```

  – **cannot** pass " `x` " to a function expecting a " `T2` "

- Libraries do not interoperate unless it was pre-planned
  – create "adapters" to work around this

    example of a design pattern used to work around language limitations

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
    another design pattern

Java Enums

- **Another design pattern built into Java:**

```
enum Color {
  RED, GREEN, BLUE
}
```

- `Color.RED` **etc. are the only 3 instances of** `Color`

- **Cannot pass a** `Color` **where** `String` **is expected**
  - **must add methods to convert between them**

Inductive Data Types

- Create new types using records, tuples, and unions
  - very useful but limited
  - can only create types that are “finite” in some sense

    if all our fields were boolean, the types would be finite sets

- One critical element is missing: **recursion**

- 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"

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

  - 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 **not** function calls

Natural Numbers

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

- **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

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

- **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

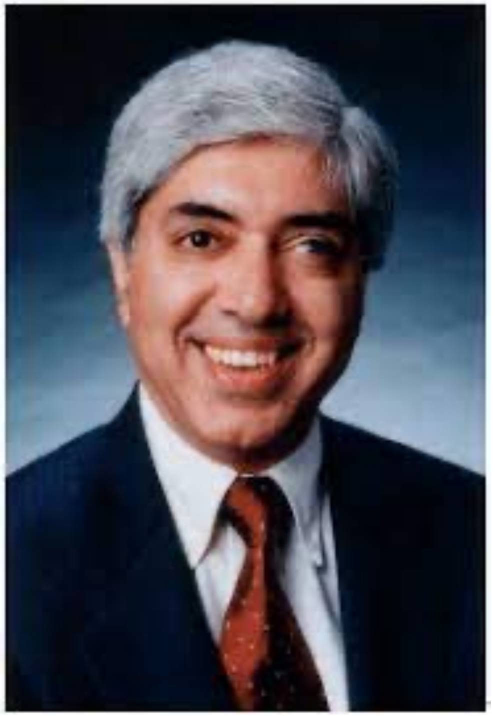
$$\textbf{type}\ \text{List} := \text{nil} \mid \text{cons}(x : \mathbb{Z},\ L : \text{List})$$

- **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 HW2+ and arrays HW5+