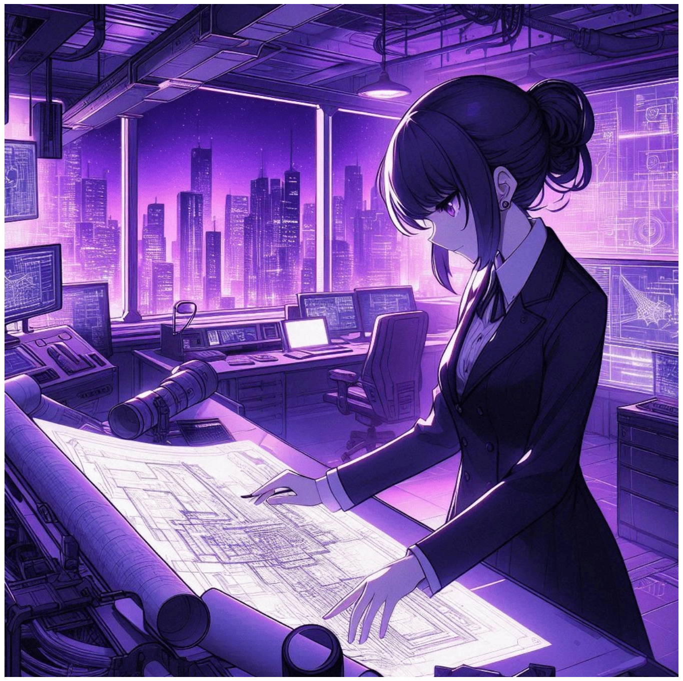


CSE 331

Specifications

Kevin Zatloukal

Specifications

Specifications

- Correctness requires a <u>definition</u> of the correct answer
- Description must be <u>precise</u>
  - can't have disagreement about what is correct
- Informal descriptions (English) are usually imprecise
  - necessary to "formalize" the English
    - turn the English into a precise *mathematical* definition
  - professionals are *very* good at this
    - usually just give English definitions
    - important skill to practice
  - we will start out completely formal to make it easier

Kinds of Specifications

- **Imperative** specification says <u>how</u> to calculate the answer
  - lays out the exact steps to perform to get the answer
- **Declarative** specification says <u>what</u> the answer looks like
  - does not say how to calculate it
  - up to us to ensure that our code satisfies the spec
- Can implement a *different* imperative specification
  - again, up to us to ensure that our code satisfies the spec

Example: Imperative Specification

- Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise
  - definition is an "if" statement

```
const abs = (x: bigint): bigint => {
  if (x >= 0n) {
    return x;
  } else {
    return -x;
  }
}
```

just translating math to TypeScript

Example: Declarative Specification

- **<u>Subtraction</u>** $(a - b)$: **return** x **such that** $b + x = a$
  - **can see that** $b + (a - b) = b + a - b = a$

```
const sub = (a : bigint, b: bigint) : bigint => {
   ??
}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Example: Declarative Specification

- Square root of x is number y such that $y^2 = x$
  - not all positive integers have integer square roots, so… let's round up
  - $(y - 1)^2 \leq x \leq y^2$

    smallest integer y such that $x \leq y^2$

```
const sqrt = (x: bigint): bigint => {

    ??

}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Example: Declarative Specification

- Absolute value $|x|$ is an integer y such that
  - $y \geq x$
  - $y \geq -x$
  - $y = x$ or $y = -x$

```
const abs = (x: bigint): bigint => {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

requires some <u>thinking</u> to make sure this code returns a number with the properties above

Example: Imperative Specification

- From HW3: Dijkstra's Algorithm

```
add a 0-step (empty) path from start to itself to active

while active is not empty:
  minPath = active.removeMin()  // shortest active path

  if minPath.end is end:
    return minPath  // shortest path from start to end!

  if minPath.end is in finished:
    continue  // longer path to minPath.end than the one we found before

  add minPath.end to finished  // just found shortest path to here!

  // add all paths that have one step added to this shortest path
  for each edge e in adjacent.get(minPath.end):
    if e.end is not in finished:
      newPath = minPath + e
      add newPath to active

return undefined  // no path from start to end :(
```

steps are described fully
(just translate to TypeScript)

"Straight From the Spec"

- If imperative, just translate math into code
  - TypeScript here, but could also be Java
  - we often call this "straight from the spec"

- if declarative (or implementing different imperative spec), then we will need new tools for checking its correctness

Recall: Kinds of Specifications

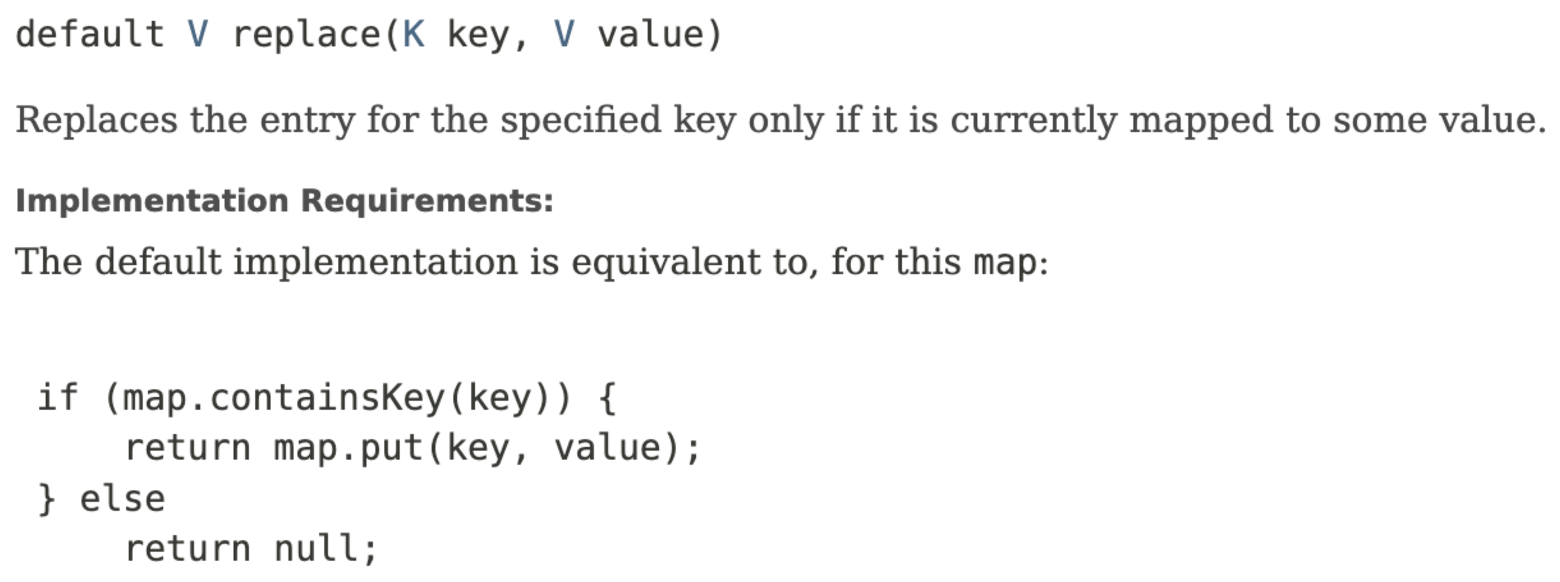
- **Imperative** specification says how to calculate the answer
  - lays out the exact steps to perform to get the answer

- **Declarative** specification says what the answer looks like
  - does not say how to calculate it
  - up to us to ensure that our code satisfies the spec

- Can implement a *different* imperative specification
  - again, up to us to ensure that our code satisfies the spec

Examples from the Java APIs

`java.util.Map` — **set of (key, value) pairs**



```
default V replace(K key, V value)
```

Replaces the entry for the specified key only if it is currently mapped to some value.

Implementation Requirements:

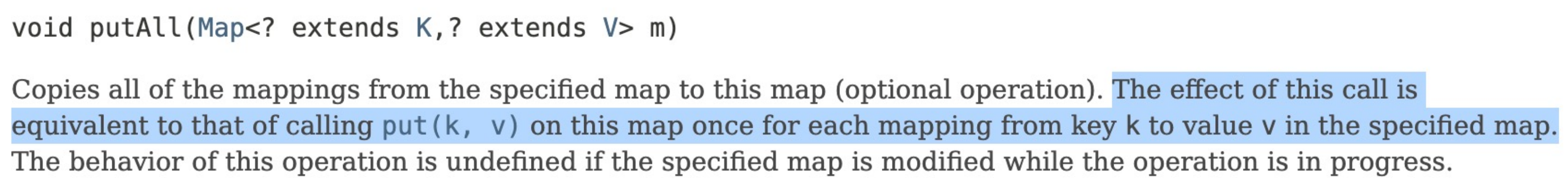
The default implementation is equivalent to, for this `map`:

```
 if (map.containsKey(key)) {
     return map.put(key, value);
 } else
     return null;
```

Imperative

Examples from the Java APIs

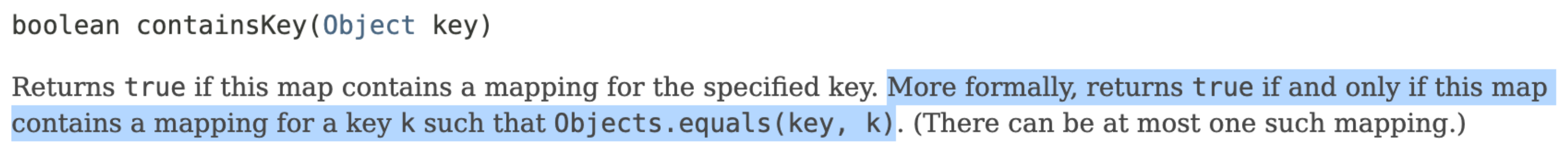
`java.util.Map` — set of (key, value) pairs



```
void putAll(Map<? extends K,? extends V> m)
```

Copies all of the mappings from the specified map to this map (optional operation). The effect of this call is equivalent to that of calling `put(k, v)` on this map once for each mapping from key `k` to value `v` in the specified map. The behavior of this operation is undefined if the specified map is modified while the operation is in progress.

Imperative



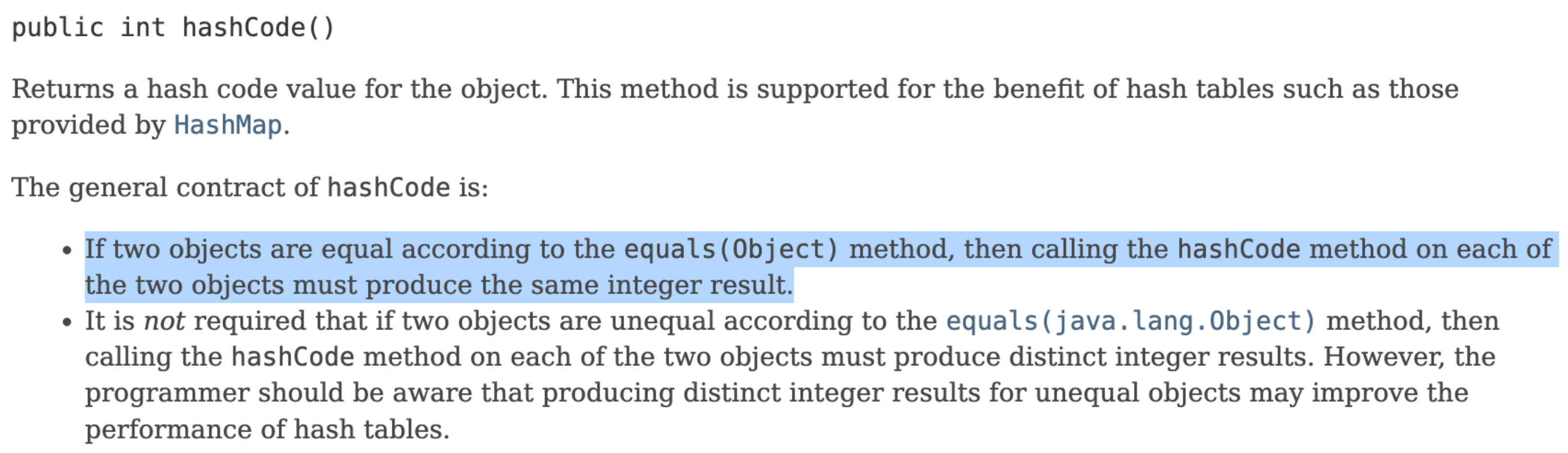
```
boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key `k` such that `Objects.equals(key, k)`. (There can be at most one such mapping.)

Declarative

Examples from the Java APIs

`java.util.Object`



```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Declarative

Next Up…

- Toolkit for writing **imperative** specifications
  - define math for data and code

    write specifications that are **language independent**

    (don't want a toolkit that only works for TypeScript)
  - describe how to translate imperative specs into TypeScript

    try to make the translations as *straightforward* as possible (fewer mistakes)
  - mention new TypeScript features when related

    critical to understand what bugs the type system caches and which it does not

- Will look at **declarative** specifications later

Math Notation

Basic Data Types in Math

- 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	non-standard names
$x \in \mathbb{S}^*$	x is a string	

Basic Data Types in TypeScript

Condition	Math	TypeScript	Up to Us
integer	$x \in \mathbb{Z}$	`bigint`	
natural	$x \in \mathbb{N}$	`bigint`	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 $x \in \mathbb{Z}$

– only subtraction on non-negative can produce negative

Ways to Create New Types In Math

- **Union Types** $\mathbb{S}^* \cup \mathbb{N}$
  - contains every object in either (or both) of those sets
  - e.g., all strings and natural numbers

- If $\mathrm{x} \in \mathbb{N} \cup \mathbb{S}^*$, then x could be a natural or string

- Two sets can contain common elements
  - in this case, the sets are disjoint

Ways to Create New Types in TypeScript

- **Union Types** `string | bigint`
  - can be either one of these

- How do we work with this code?

```
const x: string | bigint = …;

// can I call isPrime(x)?
```

- We can check the type of `x` using "`typeof`"
  - TypeScript understands these expressions
  - will "**narrow**" the type of `x` to reflect that information

Type Narrowing With “If” Statements

- **Union Types** `string | bigint`
  - can be either one of these

- **How do we work with this code?**

```
const x: string | bigint = …;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  …                          // x is a string
}
```

Type Narrowing vs Casting

```
const x: string | bigint = …;

if (typeof x === "bigint") {
  console.log(isPrime(x))    // okay! x is a bigint
} else {
  …                          // x is a string
}
```

- **Note that this does not require a type cast**
  - **TypeScript knows x is a `bigint` inside the "if" (narrowing)**

- **331: there are no type casts (won't even show syntax)**
  - **unlike Java, TypeScript casts are unchecked at runtime**
  - **seem designed to create extremely painful debugging**

Type Narrowing Gotcha

```
const f = (x: bigint): string | bigint => …;

if (typeof f(x) === “bigint”) {
  console.log(isPrime(f(x)))    // why not allowed?
}
```

- TypeScript will (properly) reject this
  - no guarantee that f(x) returns the same value both times!

Type Narrowing of Function Calls

```
const f = (x: bigint): string | bigint => …;

const y = f(x);
if (typeof y === “bigint”) {
  console.log(isPrime(y))        // this works now
}
```

- TypeScript can see that the two values are the same
- Functions that return different values for the same inputs are confusing!
  - maybe better to avoid that

Compound Types In Math

- Compound types combine multiple data types
  - multiple ways build them

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x{:}\ 3, y{:}\ 5\}$

- Note that $\{x{:}\ 3, y{:}\ 5\} = \{y{:}\ 5, x{:}\ 3\}$ in math
  - field names matter, not order
  - note that these are not "$==$" in JavaScript

      in math, “=“ means same values

      in JavaScript, "==" is reference equality

Record Types in TypeScript

- **Record Types** `{x: bigint, y: bigint}`
  - anything with *at least* fields “x” and “y”

- **Retrieve a part by name:**

```
const t: {x: bigint, y: bigint} = … ;
console.log(t.x);
```

Optional Fields in TypeScript

- Records can have optional fields

```
type T = {x: bigint, y?: bigint};

const t: T = {x: 1n};
```

  – type of “ `t.y` ” is “ `bigint | undefined` ”

- Functions can have optional arguments

```
const f = (a: bigint, b?: bigint): bigint => {
  console.log(b);
};
```

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

Compound Types In Math

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x: 3, y: 5\}$

- Tuple Types $\mathbb{N} \times \mathbb{N}$
  - pair of two natural numbers, e.g., $(5, 7)$
  - can do tuples of 3, 4, or more elements also

- Mostly equivalent alternatives
  - both let us put parts together into a larger object
  - record distinguishes parts by name
  - tuple distinguishes parts by order

Retrieving Part of a Tuple

- To refer to tuple parts, we must give them names

- Tuple Types $\mathbb{N} \times \mathbb{N}$

Let $(a, b) := t$.

":=" means a definition

Suppose we know that $t = (5, 7)$

Then, we have $a = 5$ and $b = 7$

- Tuple Types `[bigint, bigint]`

```
const t: [bigint, bigint] = …;
const [a, b] = t;
console.log(a);  // first part of t
```

Simple Functions in Math

- Simplest function definitions are single expressions
- Will write them in math like this:

$$\text{double} : \mathbb{N} \to \mathbb{N}$$

$$\text{double}(n) := 2n$$

  - first line declares the type of double function

    takes a natural number input to a natural number output

  - second line shows the calculation

    know that "n" is a natural number from the *first* line

  - will often put the type in the text before the definition, e.g.,

    The function $\text{double} : \mathbb{N} \to \mathbb{N}$ is defined by...

$$\text{double}(n) := 2n$$

Simple Functions in Math

- Another example:

$$\text{dist} : \{x\colon \mathbb{R},\ y\colon \mathbb{R}\} \to \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

  – first line tells us that "p" is a record and "$p.x$" is a real number

- Can define short-hand for types in math also

$$\textbf{type}\ \text{Point} := \{x\colon \mathbb{R},\ y\colon \mathbb{R}\}$$

$$\text{dist} : \text{Point} \to \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

Complex Functions in Math

- **Most interesting functions are not simple expressions**
  - **need to use different expressions in different cases**

- **Can use side-conditions to split into cases**

$$
\begin{aligned}
&\mathrm{abs} : \mathbb{R} \to \mathbb{R} && \\
&\mathrm{abs}(x) := x && \text{if } x \geq 0 \\
&\mathrm{abs}(x) := -x && \text{if } x < 0
\end{aligned}
$$

  - **conditions must be <u>exclusive</u> and <u>exhaustive</u>**

    we do not want to require on *order* to determine which applies
  - **there is a better way to do this in many cases…**

Pattern Matching

- Can also define functions by "pattern matching"

$$\begin{aligned} &\text{double} : \mathbb{N} \to \mathbb{N} \\ &\text{double}(0) \quad := 0 \\ &\text{double}(n+1) := \text{double}(n) + 2 \end{aligned}$$

  - first case matches only 0
  - second case matches numbers 1 more than some $n : \mathbb{N}$ …

    double(6) = double(5+1) so it matches with n = 5

    since n $\geq$ 0, we have n+1 $\geq$ 1, so it matches 1, 2, 3, …
  - pattern "$n+2$" would match 2, 3, 4, …

- Simplifies the math in multiple ways…

Pattern Matching on Natural Numbers

- **Pattern matching definition**

$$\begin{aligned} \mathrm{double}(0) &:= 0 \\ \mathrm{double}(n+1) &:= \mathrm{double}(n) + 2 \end{aligned}$$

  is simpler than using side conditions

$$\begin{aligned} \mathrm{double}(n) &:= 0 && \text{if } n = 0 \\ \mathrm{double}(n) &:= \mathrm{double}(n\text{-}1) + 2 && \text{if } n > 0 \end{aligned}$$

  – **e.g., need to explain why** $\mathrm{double}(n\text{-}1)$ **is legal**

    easy in this case, but it gets harder

- **We will prefer pattern matching whenever possible**

Pattern Matching on Booleans

- Booleans have only two legal values: T and F

- Can pattern match just by listing the values:
  - the function $\mathrm{not} : \mathbb{B} \to \mathbb{B}$ is defined as follows:

$$\begin{aligned}\mathrm{not}(\mathrm{T}) &:= \mathrm{F}\\ \mathrm{not}(\mathrm{F}) &:= \mathrm{T}\end{aligned}$$

  - negates a boolean value
  - no simpler way to define this function!

Pattern Matching on Records

- Can pattern match on individual fields of a record

$$\textbf{type}\ \ \text{Steps} := \{\text{n} : \mathbb{N}, \text{fwd} : \mathbb{B}\}$$

$$\begin{aligned}&\text{change} : \text{Steps} \to \mathbb{N}\\&\text{change}(\{\text{n: m, fwd: T}\}) := \text{m}\\&\text{change}(\{\text{n: m, fwd: F}\}) := -\text{m}\end{aligned}$$

  - clear that the rules are exclusive and exhaustive

- Can match on multiple parameters
  - e.g., $\text{change}(\{\text{n: m+5, fwd: T}\}) := 2\text{m}$
  - just make sure the rules are exclusive and exhaustive

Pattern Matching in TypeScript

- TypeScript does not provide pattern matching
  - some other languages do! (see 341)

- We must translate into "`if`"s on our own

```
type Steps = {n: number, fwd: boolean};

const change = (s: Steps) => {
  if (s.fwd) {
    return s.n;
  } else {
    return -s.n;
  }
};
```

still straight from the spec
but easy to make mistakes

Pattern Matching in TypeScript

double(0) := 0

double(n+1) := double(n) + 2

- Also need to be careful with natural numbers

```
// m is non-negative
const double = (m: bigint) => {
  if (m === 0n) {
    return 0n;
  } else {
    return double(m - 1n) + 2n;
  }
};
```

spec says double(n)
but code says double(m – 1)

  – pattern matching uses “n+1” but the code uses “m” (or “n”)

    sadly, TypeScript will not let “n+1” be the argument value

Code Without Mutation

- Saw all types of code without mutation:
  - straight-line code
  - conditionals
  - recursion
- This is all that there is!
  - can write anything computable with just these
- Saw TypeScript syntax for these already…

Code Without Mutation

Example function with all three types

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

What does this compute?

$$f(m) = 2^m$$

$f : \mathbb{N} \to \mathbb{N}$

$$\begin{aligned} f(0) &:= 1 \\ f(n+1) &:= 2 \cdot f(n) \end{aligned}$$

Inductive Data Types

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"

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

  - 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

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

- **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

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

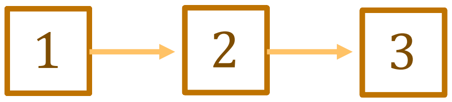
- **Inductive definition of lists of integers**

  nil

  cons(3, nil)

  cons(2, cons(3, nil))

  cons(1, cons(2, cons(3, nil)))



$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \end{tikzcd}$$

Shorthand Notation for Lists

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

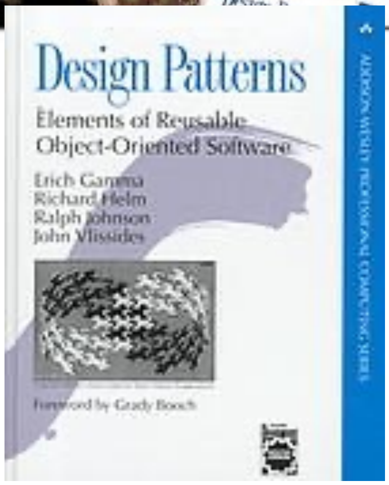
- **We will use:**
  - "x :: L" **to mean** "cons(x, L)"
  - "[1, 2, 3]" **to mean** "1 :: 2 :: 3 :: nil"
- **Examples:**

nil	nil	[]
cons(3, nil)	3 :: nil	[3]
cons(2, cons(3, nil))	2 :: 3 :: nil	[2, 3]
cons(1, cons(2, cons(3, nil)))	1 :: 2 :: 3 :: nil	[1, 2, 3]

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


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- 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 **<u>not easy</u>** 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

```
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

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

- **Implement in TypeScript as**

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

Inductive Data Type Design Pattern

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

- **Implement in TypeScript as**

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

Inductive Data Types in TypeScript

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

- **Implemented in TypeScript as**

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

- **How do I check if my list is empty?**

```
if (mylist.kind === "nil") {
  …
}
```

Inductive Data Types in TypeScript

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

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

const nil: Readonly<List> = {kind: "nil"};

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

  – **use <u>only</u> these two functions to create `List`s**

    do not create the records directly

  – **note that we only have one instance of `nil`**

    this is called a "singleton"   (there is a **design pattern** for ensuring this)

Inductive Data Types in TypeScript

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

```
const nil: Readonly<List> = {kind: "nil"};

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

- **Can now write code like this:**

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

Inductive Data Types in TypeScript

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

```
const nil: Readonly<List> = {kind: "nil"};

const cons = (hd: bigint, 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**

    will see this later…

Functions Defined on Inductive Data Types

- 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

Length of a List

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

- **Mathematical definition of list length:**

$$\text{len} : \text{List} \to \mathbb{N}$$

$$\begin{aligned} \text{len}(\text{nil}) &:= 0 \\ \text{len}(x :: L) &:= 1 + \text{len}(L) \end{aligned}$$

  - **any list is either** nil **or** $x :: L$ **for some** x **and** L
  - **cases are exclusive and exhaustive**

Length of a List

- **Mathematical definition of length**

$$\begin{aligned} \text{len(nil)} &:= 0 \\ \text{len}(x :: L) &:= 1 + \text{len}(L) \end{aligned}$$

- **Translation to TypeScript**

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

TypeScript will see that this is valid since `S.kind != "nil"`

Swapping Elements in a List

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

- **Function that swaps adjacent elements in a list:**

```
swap : List → List

swap(nil)            :=  nil
swap(x :: nil)       :=  x :: nil
swap(x :: y :: L)    :=  y :: x :: swap(L)
```

  - **any list is either** nil **or** x :: nil **or** x :: y :: L **for some** x, y **and** L
  - **cases are exclusive and exhaustive**

Swapping Elements in a List

swap(nil)	:= nil
swap(x :: nil)	:= x :: nil
swap(x :: y :: L)	:= y :: x :: swap(L)

- **Translation to TypeScript**

```
const swap = (S: List): bigint => {
  if (S.kind === "nil") {
    return nil;
  } else if (S.tl.kind === "nil") {
    return cons(S.hd, nil);
  } else {
    return cons(S.tl.hd, S.hd, swap(S.tl.tl));
  }
};
```

= S

TypeScript will see that these are valid since `S.kind != "nil"` and `S.tl.kind != "nil"`

Structural Recursion

- Examples only recurse on *parts* of the input

$$\mathrm{len}(x :: L) \ := \ 1 + \mathrm{len}(L)$$

  - call on $x :: L$ recurses on L

$$\mathrm{swap}(x :: y :: L) \ := \ y :: x :: \mathrm{swap}(L)$$

  - call on $x :: y :: L$ recurses on L
  - such cases are called "<u>structural recursion</u>"

- Guarantees no infinite recursion!
  - one argument gets *strictly smaller* on each call
  - restrict ourselves to structural recursion in math and TS

Formalizing Specifications

Formalizing a Specification

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

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

- Best to start by looking at some examples
  - try to spot a pattern
  - that usually indicates recursion

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
3 :: nil	3
2 :: 3 :: nil	2+3
1 :: 2 :: 3 :: nil	1+2+3
…	…

Definition of Sum of Values in a List

- Look at some examples…

L	sum(L)
nil	0
3 :: nil	3
2 :: 3 :: nil	2+3
1 :: 2 :: 3 :: nil	1+2+3
…	…

- Mathematical definition of sum:

```
sum(nil)      :=
sum(x :: L)   :=
```

Definition of Sum of Values in a List

L	sum(L)
1 :: 2 :: 3 :: nil	1+2+3

- **Mathematical definition of sum:**

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that this works on the examples…**

$$\begin{aligned} &\text{sum}(1 :: 2 :: 3 :: \text{nil}) \\ &\quad = 1 + \text{sum}(2 :: 3 :: \text{nil}) && \text{def of sum (2nd line)} \\ &\quad = 1 + 2 + \text{sum}(3 :: \text{nil}) && \text{def of sum (2nd line)} \\ &\quad = 1 + 2 + 3 + \text{sum}(\text{nil}) && \text{def of sum (2nd line)} \\ &\quad = 1 + 2 + 3 && \text{def of sum (1st line)} \end{aligned}$$

Sum of Values in a List

- **Mathematical definition of sum**

$$\begin{aligned}\text{sum}(\text{nil}) &:= 0\\ \text{sum}(x :: L) &:= x + \text{sum}(L)\end{aligned}$$

- **Translation to TypeScript**

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

Definition of List Equality

- Equal lists: “built with same steps”

- Look at some examples…

L	R	equal(L, R)
nil	nil	
nil	1 :: nil	
1 :: nil	nil	
1 :: nil	1 :: nil	
1 :: nil	2 :: nil	
1 :: 2 :: nil	1 :: 3 :: nil	

Definition of List Equality

L	R	equal(L, R)
nil	nil	T
nil	1 :: nil	F
1 :: nil	nil	F
1 :: nil	1 :: nil	T
1 :: nil	2 :: nil	F
1 :: 2 :: nil	1 :: 3 :: nil	F

- **Mathematical definition of** equal : (List, List) $\to \mathbb{B}$

equal(nil, nil)	:= T
equal(nil, y :: R)	:= F
equal(x :: L, nil)	:= F
equal(x :: L, y :: R)	:= (x = y) and equal(L, R)

Definition of Sum of Values in a List

L	R	equal(L, R)
1 :: 2 :: nil	1 :: 3 :: nil	F

- **Mathematical definition of** equal : (List, List) $\to \mathbb{B}$

equal(nil, nil)	:= T
equal(nil, y :: R)	:= F
equal(x :: L, nil)	:= F
equal(x :: L, y :: R)	:= (x = y) and equal(L, R)

- **Check that this works on the examples...**

equal(1 :: 2 :: nil, 1 :: 3 :: nil)	
= (1 = 2) and equal(2 :: nil, 3 :: nil)	def of equal (4th line)
= (1 = 2) and (2 = 3) and equal(nil, nil)	def of equal (4th line)
= T and T and F	def of equal (1st line)

Inductive Data Types in TypeScript

- **Translation to TypeScript**

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

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

math definition may be easier to read

Definition of List Concatenation

- Concatenate L and R: "list containing
  the elements of L followed by the elements of R"

- Look at some examples…

L	R	concat(L, R)
nil	nil	nil
nil	3 :: nil	3 :: nil
1 :: nil	3 :: nil	1 :: 3 :: nil
1 :: 2 :: nil	3 :: nil	1 :: 2 :: 3 :: nil
1 :: 2 :: nil	3 :: 4 :: nil	1 :: 2 :: 3 :: 4 :: nil
…		

Definition of List Concatenation

L	R	concat(L, R)
nil	nil	nil
nil	3 :: nil	3 :: nil
1 :: nil	3 :: nil	1 :: 3 :: nil
1 :: 2 :: nil	3 :: nil	1 :: 2 :: 3 :: nil
1 :: 2 :: nil	3 :: 4 :: nil	1 :: 2 :: 3 :: 4 :: nil

- **Mathematical definition of** concat : (List, List) → List

```
concat(nil, R)          := R
concat(x :: L, R)       := x :: concat(L, R)
```

Definition of List Concatenation

$1 :: 2 :: \text{nil} \qquad 3 :: 4 :: \text{nil} \qquad 1 :: 2 :: 3 :: 4 :: \text{nil}$

- **Mathematical definition of** $\text{concat} : (\text{List}, \text{List}) \to \text{List}$

$$\begin{aligned} \text{concat}(\text{nil}, R) &:= R \\ \text{concat}(x :: L, R) &:= x :: \text{concat}(L, R) \end{aligned}$$

- **Check that this matches examples…**

$$\begin{aligned} &\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil}) \\ &\quad = 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil}) && \text{def of concat (2nd line)} \\ &\quad = 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil}) && \text{def of concat (2nd line)} \\ &\quad = 1 :: 2 :: 3 :: 4 :: \text{nil} && \text{def of concat (1st line)} \end{aligned}$$

Definition of List Concatenation

- **Mathematical definition of** $\text{concat} : (\text{List}, \text{List}) \to \text{List}$

$$\begin{aligned} \text{concat}(\text{nil}, R) \quad &:= R \\ \text{concat}(x :: L, R) \quad &:= x :: \text{concat}(L, R) \end{aligned}$$

- **Translation to TypeScript**

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

Notes on Lists Posted on the Website

- Shorter version of everything we've discussed

- In addition:
  1. Defines a few more useful list functions
  2. Mentions important properties of concat:
     - operator notation "$+\!\!+$"
     - associativity and identity
  3. Mentions important applications of lists
     - maps *are* lists of (key, value) pairs
     - sets can be defined defined as lists

- Lists are our most important data type!

Definition of List Reversal

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

- Look at some examples...

L	rev(L)	
nil	nil	
[3]	[3]	3 :: nil
[2, 3]	[3, 2]	3 :: 2 :: nil
[1, 2, 3]	[3, 2, 1]	3 :: 2 :: 1 :: nil
...	...	

Definition of List Reversal

- Look at some examples...

L	rev(L)
nil	nil
3 :: nil	3 :: nil
2 :: 3 :: nil	3 :: 2 :: nil
1 :: 2 :: 3 :: nil	3 :: 2 :: 1 :: nil

- **Where does** $\mathrm{rev}([2, 3])$ **show up in** $\mathrm{rev}([1, 2, 3])$**?**
  - **at the beginning, with** $1 :: \mathrm{nil}$ ***after*** **it**

- **Where does** $\mathrm{rev}([3])$ **show up in** $\mathrm{rev}([2, 3])$**?**
  - **at the beginning, with** $2 :: \mathrm{nil}$ ***after*** **it**

Definition of List Reversal

$1 :: 2 :: 3 :: \text{nil}$ $\qquad\qquad$ $3 :: 2 :: 1 :: \text{nil}$

- **Mathematical definition of** $\text{rev} : \text{List} \to \text{List}$

$$\begin{aligned} \text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \mathbin{+\!\!+} (x :: \text{nil}) \end{aligned}$$

- **Check that this matches examples...**

$$\begin{aligned} &\text{rev}(1 :: 2 :: 3 :: \text{nil}) && \\ &\quad = \text{rev}(2 :: 3 :: \text{nil}) \mathbin{+\!\!+} [1] && \text{def of rev} \\ &\quad = \text{rev}(3 :: \text{nil}) \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1] && \text{def of rev} \\ &\quad = \text{rev}(\text{nil}) \mathbin{+\!\!+} [3] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1] && \text{def of rev} \\ &\quad = [\,] \mathbin{+\!\!+} [3] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [1] && \text{def of rev} \\ &\quad = \ldots = [3, 2, 1] && \text{def of concat (many times)} \end{aligned}$$

Reversing A Lists

- **Mathematical definition of** $\mathrm{rev} : \mathrm{List} \to \mathrm{List}$

$$\begin{aligned} \mathrm{rev}(\mathrm{nil}) &:= \mathrm{nil} \\ \mathrm{rev}(x :: L) &:= \mathrm{rev}(L) \mathbin{+\!\!+} (x :: \mathrm{nil}) \end{aligned}$$

- **Other definitions are possible, but this is simplest**
- **Always make definitions as simple as possible**

Formalizing a Specification

- Sometimes the instructions are written in English
  - English is often imprecise or ambiguous
- First step then 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!

    Is it obvious that equal, concat, & rev are correct? Maybe not.
- We can test our definition on some examples
  - what can we do to increase the odds we spot bugs?