CSE 331

Specifications

Kevin Zatloukal

Reminders

- HW1 due by 11pm tonight
- Section tomorrow starts HW2
  - HW2 itself released Thursday night
- Summary of math notation on website
- Small amount of testing material on Friday

Last Time: Correctness Levels of Difficulty

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

Math Notation

Last Time: 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)	non-standard names
$x \in \mathbb{S}$	x is a character	
$x \in \mathbb{S}^*$	x is a string	

Last Time: 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 | number`
  - can be either one of these

- Can also include literal values in the union!

```
const x: 1 | 2 | 3 =  …;

// know that x is either 1, 2, or 3
```

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 $\{x{:}\ 3, y{:}\ 5\} = \{y{:}\ 5, x{:}\ 3\}$
  - field names matter, not order
  - (also, “=“ means same values)

Record Types in TypeScript

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

- Retrieve a part by name:

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

  - can also use a type alias

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

Optional Fields in TypeScript

- Records can have optional fields

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

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

  - type of “ `t.y` ” is “ `number | undefined` ”

- Functions can have optional arguments

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

  - type of “ `b` ” is “ `number | 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 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

Tuple Types in TypeScript

- Tuple Types `[number, number]`

- At runtime, actually an array of length 2
  - could retrieve the second part using "`t[1]`" syntax

    easy to make mistakes here!
  - but would prefer to match the math more closely

    331 coding conventions require this!

- How would we do this in math?
  - we must give names to the parts to refer to them
  - (aside: this is how function arguments work too)

Retrieving Part of a Tuple

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

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

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

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

“:=” means a definition

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

- Tuple Types `[number, number]`

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

required style for 331

Readonly Values

- TypeScript can ensure values aren’t modified
  - extremely useful! (mutation makes everything harder)

- Tuple types should always be readonly

```
type NumberPair = readonly [number, number];
```

- Individual fields of records should be marked readonly

```
type NumberPair = {readonly x: number,
                   readonly y: number};
```

Simple Functions in Math

- Simplest function definitions are single expressions

- Will write them in math like this:

$$\textbf{func}\ \mathrm{double}(n : \mathbb{N}) := 2n$$

$$\textbf{func}\ \mathrm{dist}(p : \{x: \mathbb{R}, y: \mathbb{R}\}) := (p.x^2 + p.y^2)^{1/2}$$

  - any normal math allowed in the expression

Simple Functions in Math

- Can define short-hand for types in math also

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

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

- Can put the argument type on the right instead

$$\textbf{func}\ \text{dist}(p) := (p.x^2 + p.y^2)^{1/2} \qquad \text{for any } p : \text{Point}$$

  - needs to be described somewhere (we're not too picky)
  - will need this in some cases coming shortly…

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

$$\begin{array}{lll} \textbf{func} & \mathrm{abs}(x : \mathbb{R}) := x & \text{if } x \geq 0 \\ & \mathrm{abs}(x : \mathbb{R}) := -x & \text{if } x < 0 \end{array}$$

  - 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"

```
func  double(0)     := 0
      double(n+1)   := double(n) + 2          for any n : ℕ
```

  - first case matches only 0
  - second case matches $1, 2, 3, \ldots$

    if $m \geq 1$, then $m = n + 1$ for some n : ℕ

- Simplifies the math in multiple ways…

Pattern Matching

- Pattern matching definition

```
func  double(0)    := 0
      double(n+1)  := double(n) + 2        for any n : ℕ
```

  is simpler than using side conditions

```
func    double(n)    := 0                   if n = 0     for any n : ℕ
        double(n)    := double(n-1) + 2     if n > 0     for any n : ℕ
```

  - e.g., need to explain why double(n-1) is legal

    easy in this case, but it gets harder
  - (also makes the reasoning easier, as we will see later...)

- 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:

```
func  not(T) := F
      not(F) := T
```

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

Pattern Matching on Records

- Can pattern match on individual fields of a record

```
type  Steps := {n : ℕ, fwd : 𝔹}

func  change({n: n, fwd: T}) := n          for any n : ℕ
      change({n: n, fwd: F}) := −n         for any n : ℕ
```

  - clear that the rules are exclusive and exhaustive

Pattern Matching in TypeScript

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

- We have to 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 level 0 but
easy to make mistakes

Pattern Matching in TypeScript

func double(0) := 0

double(n+1) := double(n) + 2 for any n : $\mathbb{N}$

- Also need to be careful with natural numbers

Level 0

```
const double = (m: number) => {
  if (m === 0) {
    return 0;
  } else {
    return double(m - 1) + 2;
  }
};
```

spec says double(m)
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

Pattern Matching in TypeScript

func double(0) := 0

double(n+1) := double(n) + 2 for any n : $\mathbb{N}$

- This implementation returns the same thing:

```
const double = (m: number) => {
  return 2 * m;
};
```

Level 1

  - but that’s not what the spec says!
  - requires reasoning tools to check that this is correct

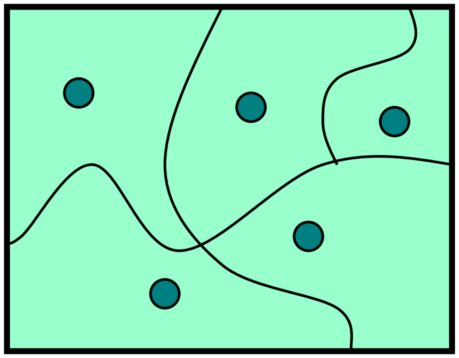
    (will come in HW3+…)

Correctness Levels

Level	Description	Testing	Tools	Reasoning
-1	small # of inputs	exhaustive		
0	straight from spec	heuristics	type checking	code reviews
1	?			
2	?			
3	?			

Testing

Key Problem



- Key question is what cases to test
  - at level -1, we can test all of them
  - at level 0+, we cannot
- Split the allowed inputs into **subdomains**
  - for inputs in one subdomain, code “does the same thing”
- Hope: code is entirely right or wrong for subdomain
  - one example in the subdomain will tell us if there is a bug
  - (note: this is not *always* true… see sec02 and HW2)
- Plan: Look at the code. See when it “does the same thing”

Testing Straight-Line Code

Straight-line Code looks like

```
return 2 * (n-1) + 1;
```

Or, more generally, like this

```
const m = n - 1;
return 2 * m + 1;
```

- Any number of constant values allowed
  - often makes the code easier to read, but no different
- Inputs where it executes the same straight-line code are "***doing the same thing***"

Testing Straight-Line Code

Rule: Same straight-line code is one subdomain

Straight-line Code looks like

```
return 2 * (n-1) + 1;
```

Or, more generally, like this

```
const m = n - 1;
return 2 * m + 1;
```

Testing Subdomains

Rule: at least two test cases per subdomain

(assuming subdomain contains at least two inputs)

- My main worry is copy-and-paste issues
  - copy “return 1;” and forget to change it later
  - if the test we pick happens to want 1, we’ll never notice
- Still doesn’t guarantee the code is right! (see HW2)
- More is obviously also okay
  - not a contest to write the fewest tests

Testing Conditionals

Conditionals look like this

```
if (n > 0) {
  return 2 * (n-1) + 1;
} else {
  return 0;
}
```

Two branches (“then” and “else”)

- in this case, both branches are straight-line code

Testing Conditionals

Rule: branches are in separate subdomains

- Would be **negligent** not to test both branches
- If both are straight-line code, then 4 tests
- With if/else if/else, we'd need 6 tests
    - 3 branches x 2 per straight-line block = 6 cases

Testing Conditionals

Conditionals look like this (with `n` an integer)

```
if (n > 0) {
  return 2 * (n-1) + 1;
} else {
  return 0;
}
```

- Boundary cases are 0 and 1
  - cases for “then” block could be 1 and 10 (say)
  - cases for “else” block could be 0 and -1 (say)