CSE 331
Software Design & Implementation

Summer 2024

Section 2 – HW Levels: Correctness, Specifications, & Testing

Where are we?

Basics of Typescript

Basics of the browser

- Math notation & specifications
- Correctness concepts
- Testing

- Reasoning techniques (for levels 2-5)
- Abstraction
- App design / more complex React apps
- + more!

Common Bugs

- **Some common bugs to lookout for in your homework:**
  - Using the wrong boundary case (using < instead of <= in an if condition)
  - Mixing up parameter order
    - if a function takes in 2 bigints as parameters, it is easy to pass in the parameters in the wrong order
  - Typos in variable names and functions
  - Base case error
    - Your function's base case may be returning a hardcoded value (like 0) when it should return a different value
  - Copy and Paste error
    - You may have copied a function stub and forgot to change the parameters
    - Copied test cases and forgot to change the function called in them

Review – Correctness

Level	Description	Testing	Tools	Reasoning
0	small # of inputs	exhaustive		
1	straight from spec	heuristics	type checking	code reviews
2	no mutation	"	libraries	calculation induction
3	local variable mutation	"	"	Floyd logic
4	array mutation	"	"	for-any facts
5	heap state mutation	"	"	rep invariants

- Test all possible cases if reasonable, then use heuristics to approximate
- As code increases ⬆️ in complexity, formality and complexity of reasoning technique must increase ⬆️ too
- heap state mutation is the "worst case" for how difficult it is to be confident it's correct

Question 1

(a) Consider the following mathematical function defined on the integers 1, 2, 3, and 4:

$$\begin{aligned} \textbf{func}\ f(1) &:= 2 \\ f(2) &:= 3 \\ f(3) &:= 4 \\ f(4) &:= 1 \end{aligned}$$

If we implement this directly in TypeScript using a `switch` statement, what level of correctness is required?

Question 1

(b) Consider the following mathematical function defined on the inputs n and b, where n is 1, 2, 3, or 4 and b is true or false. It is defined in terms of the function f defined in part (a).

$$\textbf{func } g(n, \mathsf{T}) := f(n)$$
$$g(n, \mathsf{F}) := f(n)$$

If we implement this in TypeScript using an `if` statement (on b), what level of correctness is required?

Question 1

(c) Consider the following mathematical function defined on the inputs n and x, where n is 1, 2, 3, or 4 and x is any integer. It is defined in terms of the function f defined in part (a).

$$\textbf{func } h(n, x) := f(n) + x$$

If we implement this in TypeScript using a single `return` statement, what level of correctness is required?

Question 1

(d) Suppose that we implement the function h with the following TypeScript code. It calls f, which we will assume is implemented in TypeScript with one conditional.

```
const h = (n: bigint, x: bigint): bigint => {
  let y = f(n);
  while (x > 0n) {
    y = y + 1n;
    x = x - 1n;
  }
  return y;
}
```

What level of correctness is required now?

Review – Math Notation

Standard notations

Made up for this class

$\mathbb{N}$	all non-negative integers ("natural" numbers)
$\mathbb{Z}$	all integers
$\mathbb{R}$	all real numbers
$\mathbb{B}$	the boolean values (T and F)
$\mathbb{S}$	any character
$\mathbb{S}^*$	any sequence of characters ("strings")

- **Union**: $A \cup B$ set including everything in A *and* B
- **Tuple**: $A \times B$ all pairs (a, b) where a $\in A$ and b $\in B$
- **Record**: $\{x\colon A, y\colon B\}$ all records with fields x, y of types A, B

Review – Math Notation

- **Pattern matching**: defining function based on input cases
  - Exactly **one** rule for every valid input

  ex: $\text{func } f(0) \coloneqq 0$

  $f(\text{n+1}) \coloneqq n \qquad \text{for any n: } \mathbb{N}$

  → "n+1" is signifying that the input must be > 0 since the smallest value $\text{n: } \mathbb{N}$ would be 0

- **Side conditions**: limiting/specifying input in the right column, cleans things up, pattern matching preferred

- See the course website > Calendar > 10/4 lecture notes: "Math Notation" for more!

Question 2

func half(undefined) $:= 0$

half$(n : \mathbb{N})$ $:= n/2$ if n is even

half$(n : \mathbb{N})$ $:= -(n+1)/2$ if n is odd

(a) What is the type for the function half? (There are 2 possibilities.) Use the notation half : $A \to B$ to indicate that half takes inputs of type A and produces outputs of type B.

Question 2

half : (undefined $\cup$ N) $\rightarrow$ Z

(b) What would the declarations of this function look like in TypeScript based on the type?

Question 2

(c) What would the implementation of the body of this function look like in TypeScript?

Question 3

```
const maybeDouble = (t: {b: boolean, v: [boolean, bigint]}): bigint => {
  const [bool, num] = t.v;
  if (t.b) {
    if (bool) {
      return 2n * num;
    } else {
      return num;
    }
  } else {
    return 0n;
  }
};
```

How would you translate this into our math notation using pattern matching?

Review – Testing

- **Straight-Line Calculation**:
  - Simplest type of code. Performs calculation without any recursive calls or *if* statements
  - Need a minimum of 2 test cases (to ensure that it is not just returning a constant)

Ex: `return 2n * (x - 1n);`

Need 2 test cases

Review – Testing

- **Conditionals**:
  - Code behaves differently on inputs that fall into the “if” part vs the “else” part
  - Boundary cases: where the code switches from the “if” branch to the “else” branch
  - If branch takes 2+ inputs, need at least 2 test cases
  - If on a boundary, need to test each boundaries, and 1+ non-boundaries

Ex:

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

Need 4 test cases:

- 2 for if branch
- 2 for else branch
- 1 boundary, so one of the *if* cases should be the boundary value

Review – Testing

- **Recursion**:
  - 3 subdomains: inputs that cause 0, 1, 2+ recursive calls
    - called the “0-1-many” heuristic
  - Apply the same rules as before for each subdomain
  - For 2+ recursive calls, only need to look at the **first two** calls
  - make subdomains as specific as possible: i.e. two inputs that both make 0 recursive calls but are in separate branches are in **different** subdomains

Ex:

```
// n must be a positive integer
const f = (n: bigint): number => {
    if (n === 1n) {
        return 0;
    } else {
        return 1 + f(1n + (n - 2n) / 2n);
    }
}
```

Tests:
- 1 (0 recursive calls)
- 2, 3 (1 recursive call)
- 4, 10 (many recursive calls)

Question 4a

Consider the following implementation of a function to calculate $|x|$.

```
export const abs_value = (x: bigint): bigint => {
  if (x > 1n) {
    return x;
  } else {
    return -x;
  }
};
```

Say we included the test cases -1, -2, 2 and 3; 2 cases for each branch. All the test cases passed, but this implementation is actually incorrect. Why were those cases not enough to detect the problem? What heuristic did we forget about?

Question 4b

How many tests should we write for the following function? Why?

```
const s = (x: bigint, y: bigint): bigint => {
  if (x >= 0n) {
    if (y >= 0n) {
      return x + y;
    } else {
      return x - y;
    }
  } else {
    return y;
  }
}
```

Question 4c

How many tests should we write for the following function, defined only on the *non-negative* integers? Why? What are the tests that we should use?

```
const f = (n: bigint): bigint => {
  if (n === 0n) {
    return 0n;
  } else if (n === 1n) {
    return 1n;
  } else if (n % 2n === 1n) {  // n is > 1 and odd
    return f(n - 2n) + 1n;
  } else {                     // n is > 1 and even
    return f(n - 2n) + 3n;
  }
}
```