CSE 331
Software Design & Implementation

Spring 2025

Section 8 – Trees and ADTs

Administrivia

- HW 8 released tonight, due Wed. May 28
    - Longer code section than recent weeks, so start doing it early and come to office hours

Proof By Calculation (Review)

- The goal of proof by calculation is to *show* that an assertion is true *given* facts that you already know
- You should ***start*** the proof with either the left or the right side of the assertion and ***end*** the proof with the other side of the assertion.
- Every symbol (=, >, <, etc.) connecting each line of the proof is the current line's relationship to the previous line in the proof (not any other lines)
- Only modify one side
- **Every** line requires justification (except for algebraic manipulations)

Proof By Calculation Bug 1

Suppose we have the facts: $x = 3$, $y = 4$, $z > 5$ and we want to use proof by calculation to prove $x^2 + y^2 < z^2$. Our proof by calculation would look like this:

$x^2 + y^2$	$< z^2$	beginning of a backwards proof
0	$< z^2 - x^2 - y^2$	
0	$< z^2 - 3^2 - y^2$	since $x = 3$
0	$< z^2 - 3^2 - 4^2$	since $y = 4$
0	$< z^2 - 25$	
25	$< z^2$	
5	$< \|z\|$	

Since $z > 5$, we know $x^2+y^2<z^2$ by above.

Manipulates both sides of the equation

Not a single chain of equalities

doesn't end with right side of the assertion (z^2)

What is wrong with this proof?

Proof by Calculation Bug 1: Explanation

The previous proof is an example of *Circular Reasoning*. We begin the proof with the conclusion manipulating both sides until we reach one of the given facts.

Just because we can prove one direction does **not** mean the other direction necessarily holds.

We must always start from what we know and end with what we want to prove.

Proof By Calculation Bug 2

Suppose we have the facts: $x = 3$, $y = 4$, $z > 5$ and we want to use proof by calculation to prove $x^2 + y^2 < z^2$. Our proof by calculation would look like this:

$$\begin{aligned} x^2 + y^2 &= 3^2 + y^2 && \text{since } x = 3 \\ &< 25 \\ &< 5^2 \\ &< z^2 \end{aligned}$$

Inequalities/equalities on lines not exclusively referring to relationship between current and previous line

This is **not** correct because while $x^2 + y^2 < z^2$, $x^2 + y^2$!< 25 it should be '='

Not every non-algebraic step has justification and some non-algebraic steps are skipped

Proof By Calculation Example Correct

Suppose we have the facts: $x = 3$, $y = 4$, $z > 5$ and we want to use proof by calculation to prove $x^2 + y^2 < z^2$. Our proof by calculation would look like this:

note that each line shows the relationship *only* to the previous line

$x^2 + y^2$	$= 3^2 + y^2$	since $x = 3$
	$= 3^2 + 4^2$	since $y = 4$
	$= 25$	
	$= 5^2$	
	$< z^2$	since $z > 5$

start with left side of assertion

end with right side of assertion

note that every line has
justification (except for algebraic manipulations)

Forward & Backward Reasoning Review

Forward Reasoning:

- After each line of code *update* variables in assertions based how they they were changed by the line of code

Backward Reasoning:

- As you work your way up the code *directly* substitute how variables are modified in the code into your assertions

General:

- Do **not** drop or simplify assertions
- Do **not** use subscripts for invertible operations (addition and subtraction are *always* invertible)

Forward Reasoning Error Example 1

```
{{ x > 1 }}
x = x + 1;
{{ x = x₀ + 1 and x > 1 }}
y = 3 * x;
{{ x = x₀ + 1 and y = 3 * x }}
z = y + 1;
{{ x = x₀ + 1 and y = 3 * x and z = (3 * x) + 1 }}
```

Drops this assertion

What's wrong with these assertions?

Uses subscripts for an invertible operation

Simplifies assertions too early

Correct Forward Reasoning Example

```
{{ x > 1 }}
x = x + 1;
{{ x - 1 > 0 }}
y = 3 * x;
{{ x - 1 > 0 and y = 3 * x }}
z = y + 1
{{ x - 1 > 0 and y = 3 * x and z = y + 1 }}
```

does not simplify assertions early

updates x for this operation rather than introducing subscripts

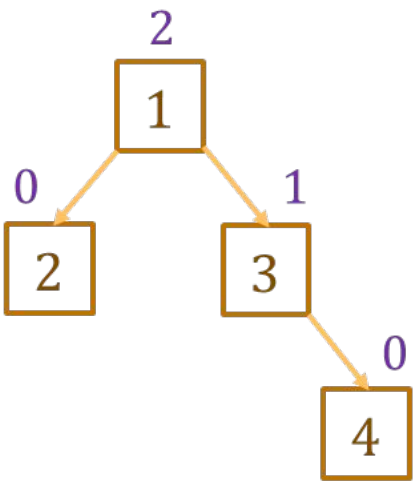
Trees

- Trees are inductive data types with a constructor that has 2+ recursive arguments

- These come up all the time…
  - no constructors with recursive arguments = "generalized enums"
  - constructor with 1 recursive arguments = "generalized lists"
  - constructor with 2+ recursive arguments = "generalized trees"

Height of a tree

- **Binary Tree:** a tree in which each node has at most 2 children
  - Not to be confused with Binary *Search* Tree, which also has the ordering property that (nodes in L) < x and (nodes in R) > x
- **type** Tree := empty | node(x: $\mathbb{Z}$, L: Tree, R: Tree)



Mathematical definition of height

$$\textbf{height : Tree} \to \mathbb{Z}$$

$$\begin{aligned} &\text{height(empty)} &&:= -1 \\ &\text{height(node(x, L, R))} &&:= 1 + \max(\text{height(L)}, \text{height(R)}) \end{aligned}$$

Tree Height Check-in

What is the height of `empty`? -1

What is the height of `node(1, empty, empty)`? 0

What is the height of

```
node(
    1,
    node(2, empty, node(4, empty, empty)),
    node(5, node(6, empty, empty), empty)
)
```

2

hint: draw it out :)

Using Definitions in Calculations (example)

$$\mathbf{height} : \mathbf{Tree} \to \mathbb{Z}$$

$$\text{height}(\text{empty}) \quad := \quad -1$$

$$\text{height}(\text{node}(x, L, R)) \quad := \quad 1 + \max(\text{height}(L), \text{height}(R))$$

Suppose "T = node(1, empty, node(2, empty, empty))"

Prove that height(T) = 1

height(T) = height(node(1, empty, node(2, empty, empty)))	since T = …
= 1 + max(height(empty), height(node(2, empty, empty)))	def of height
= 1 + max(-1, height(node(2, empty, empty)))	def of height
= 1 + max(-1, 1+ max(height(empty), height(empty)))	def of height
= 1 + max(-1, 1+ max(-1, -1))	def of height (x 2)
= 1 + max(-1, 1+ -1)	def of max
= 1 + max(-1, 0)	
= 1 + 0	def of max
= 1	

Task 1: One, Two, Tree…

The problem makes use of the following inductive type, representing a *left-leaning* binary tree

$$\begin{aligned} \textbf{type}\ \mathsf{Tree} := \ & \mathsf{empty} \\ \mid \ & \mathsf{node}(\mathsf{val} : \mathbb{Z},\ \mathsf{left} : \mathsf{Tree},\ \mathsf{right} : \mathsf{Tree}) \quad \text{with } \mathsf{height}(\mathsf{left}) \geqslant \mathsf{height}(\mathsf{right}) \end{aligned}$$

The "with" condition is an *invariant* of the node. Every node that is created must have this property, and

$$\begin{aligned} \textbf{func}\ \mathsf{height}(\mathsf{empty}) &:= -1 \\ \mathsf{height}(\mathsf{node}(x, S, T)) &:= 1 + \mathsf{height}(S) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \mathsf{Tree} \end{aligned}$$

Since height(S) >= height(T)

$$\mathsf{size} : \mathsf{Tree} \to \mathbb{N}$$

$$\begin{aligned} \mathsf{size}(\mathsf{empty}) &:= 0 \\ \mathsf{size}(\mathsf{node}(x, S, T)) &:= 1 + \mathsf{size}(S) + \mathsf{size}(T) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \mathsf{Tree} \end{aligned}$$

Prove by structural induction that, for any left-leaning tree *T* we have

$$\mathsf{size}(T) \leqslant 2^{\mathsf{height}(T)+1} - 1$$

Task 2: How do I Love Tree

a Path tells you how to get to a particular node where each step along the path (item in the list) would be a direction pointing you to keep going down the LEFT or RIGHT branch of the tree.

$$\begin{aligned} \textbf{type } \mathsf{BST} := \quad & \mathsf{empty} \\ | \quad & \mathsf{node}(x : \mathbb{Z},\ S : \mathsf{BST},\ R : \mathsf{BST}) \end{aligned}$$

$$\begin{aligned} \textbf{type } \mathsf{Dir} \quad & := \mathsf{LEFT} \ | \ \mathsf{RIGHT} \\ \textbf{type } \mathsf{Path} \quad & := \mathsf{List}\langle \mathsf{Dir} \rangle \end{aligned}$$

(a) Define a function "find(p : Path, T : BST)" that returns the node (a BST) at the path from the root of T or undefined if there is no such node.

Task 2: How do I Love Tree

(a) Define a function "find(p : Path, T : BST)" that returns the node (a BST) at the path from the root of T or undefined if there is no such node.

"undefined" sidebar

- If the end of the path cannot be reached within the tree (hit a dead-end before end of Path) → function should result in undefined
  - undefined just indicates invalid inputs
  - If an expression includes a call that results in undefined, then the *entire expression is undefined*
    - Similar to how an `Error` in code does not "return" but bubbles up to callers of the function with the error

Specifications for ADTs – Review

- New Terminology for specifying ADTs:
  - **Abstract State / Representation (Math)**
    - How clients should understand the object
    - Ex: `List(nil or cons)`
  - **Concrete State / Representation (Code)**
    - Actual fields of the record and the data stored
    - Ex: `{ list: List, last: bigint | undefined }`

- We've had different abstract and concrete types all along!
  - in our math, List is an inductive type (abstract)
  - in our code, `List` is a string or a record (concrete)

- Term "object" (or "obj") will refer to abstract state
  - "object" means mathematical object
  - "obj" is the mathematical value that the record represents

Documenting ADTs – Review

Abstract Function (AF) – defines what abstract state the field values represent

- Maps field values → the object they represent
- Output is math, this is a mathematical function

Representation Invariants (RI) – facts about the field values that must always be true

- Constructor must always make sure RI is true at runtime
- Can assume RI is true when reasoning about methods
- AF only needs to make sense when RI holds
- Must ensure that RI *always* holds

Documenting ADTs – Example

```
// A list of integers that can retrieve the last element in O(1)
export interface FastList {
/**
* Returns the object as a regular list
* @returns obj
*/
toList: () => List<bigint>
}
```

Talk about functions in terms of the abstract state (obj)

Hide the representation details (i.e. real fields) from the client

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list);
  // AF: obj = this.list;

  // @ returns last(obj)
  getLast = (): bigint | undefined => {
    return this.last;
  };
}
```

Task 3: Ready, Set, Go!

(a) Use the RI or AF to prove that the has method of the SpaceSavingSet class is correct. Note that the contains function used in the method exactly matches the definition of contains given above so you do not need to prove it.

```
// RI: noDuplicates(this.list) = true
// AF: obj = this.list
list: List<bigint>;
```

```
/**
* @returns contains(obj, value)
*/
has = (value: bigint): boolean => {
    // The contains function here exactly matches the definition
    // of contains
    return contains(this.list, value);
}
```

contains(obj, value) =

contains(this.list, value)   by AF

Task 3: Ready, Set, Go!

(b) Use the RI or AF to prove that the add method of the SpaceSavingSet class is correct.

```
// RI: noDuplicates(this.list) = true
// AF: obj = this.list
list: List<bigint>;

/**
* @returns obj where has(value) = true
* and isSetEqual(cons(value, obj_0), obj) = true
*/
add = (value: bigint): SpaceSavingSet => {
    if (this.has(value)) {


        return this;
    }
    return new SpaceSavingSet(cons(value, this.list));
}
```

Task 3: Ready, Set, Go!

(b) Use the RI or AF to prove that the add method of the SpaceSavingSet class is correct.

Case 1: has(value) = true

obj = obj_0 from the code

= this.list by AF

has(value) is true from the given condition

isSetEqual(cons(value, obj_0), obj) = true because has(value) = true

The RI holds because obj is not modified

Task 3: Ready, Set, Go!

(b) Use the RI or AF to prove that the add method of the SpaceSavingSet class is correct.

Case 2: has(value) = false

obj = cons(value, this.list) from the code

has(value) = contains(obj, value) def of has

= contains(value::this.list, value) from above

= true def of contains

isSetEqual(cons(value, obj_0), obj) = true because the new object is the old object with value at the front

The RI holds because has(value) for the old object is false and noDuplicates(obj_0) is true, so noDuplicates(obj) = noDuplicates(cons(value, this.list)) holds.

Task 3: Ready, Set, Go!

(c) Use the RI or AF to prove that the remove method of the SpaceSavingSet class is correct.

```
/**
* @requires has(value) = true
* @returns obj where has(value) = false
* and isSetEqual(obj_0, cons(value, obj)) = true
*/
remove = (value: bigint): SpaceSavingSet => {
    let removed: List<bigint> = nil;
    let rest: List<bigint> = this.list;

    // Inv: this.list = rev(removed) ++ this.rest
    // and contains(removed, value) = false
    while (rest.kind !== 'nil' && rest.hd !== value) {
        removed = cons(rest.hd, removed);
        rest = rest.tl;
    }
    rest = rest.tl;

    returns new SpaceSavingSet(concat(rev(removed), rest));
}
```

Task 3: Ready, Set, Go!

(c) Use the RI or AF to prove that the remove method of the SpaceSavingSet class is correct.

When the while loop exits, we know

- this.list = rev(removed) ++ this.rest and contains(removed, value) = false from the Invaraint

and

- rest.kind === 'nil' or rest.hd === value from the while condition

Case rest.kind === 'nil'

- Since contains(removed, value) and contains(rest, value) are false, we know that has(value) for the new object is false

Case rest.hd === value

- Since noDuplicates(this.list) = true by the RI, contains(removed, value) = false by the invariant, and contains(rest.tl, value) = false, has(value) for the new object is false.

Task 3: Let's Blow This Point

Suppose we had the following interface for a Point class that represents a point in R2 (2D space):

(a) Define the representation invariant (RI) in the form r = "..." and abstraction function (AF) in the form obj = "..." for the SimplePoint class.

```
/** Represents a point with coordinates in (x,y) space. */
interface Point {
    /** @returns the x coordinate of the point */
    getX: () => number;

    /** @returns the y coordinate of the point */
    getY: () => number;

    /**
     * Returns the distance of this point to the origin.
     * @returns Math.sqrt(obj.x*obj.x + obj.y*obj.y)
     */
    distToOrigin: () => number;
}
```

```
class SimplePoint implements Point {
    // RI: <TODO>
    // AF: <TODO>
    readonly x: number;
    readonly y: number;
    readonly r: number;

    // Creates a point with the given coordinates
    constructor(x: number, y: number) {
     this.x = x;
     this.y = y;
     this.r = Math.sqrt(x*x + y*y);
    }

    getX = (): number => this.x;
    getY = (): number => this.y;
    distToOrigin = (): number => this.r;
}
```

Task 3: Let's Blow This Point

```
/** Represents a point with coordinates in (x,y) space. */
interface Point {
    /** @returns the x coordinate of the point */
    getX: () => number;

    /** @returns the y coordinate of the point */
    getY: () => number;

    /**
     * Returns the distance of this point to the origin.
     * @returns Math.sqrt(obj.x*obj.x + obj.y*obj.y)
     */
    distToOrigin: () => number;
}
```

```
class SimplePoint implements Point {
    // RI: r = Math.sqrt(this.x * this.x + this.y * this.y)
    // AF: obj = (this.x, this.y)
    readonly x: number;
    readonly y: number;
    readonly r: number;

    // Creates a point with the given coordinates
    constructor(x: number, y: number) {
     this.x = x;
     this.y = y;
     this.r = Math.sqrt(x*x + y*y);
    }

    getX = (): number => this.x;
    getY = (): number => this.y;
    distToOrigin = (): number => this.r;
}
```

(b) Use the RI or AF to prove that the distToOrigin method of the SimplePoint class is correct.

Task 3: Let’s Blow This Point

(c) The following problem will make use of this math definition that rotates a point around the origin (x, y) by an angle θ:

$$\text{rotate} : (\text{Point},\ \mathbb{R}) \to \text{Point}$$
$$\text{rotate}((x, y), \theta) \quad = \quad (x \cdot \cos(\theta) - y \cdot \sin(\theta),\ x \cdot \sin(\theta) + y \cdot \cos(\theta))$$

Suppose we have the following implementation of the rotate method:

/** @returns rotate(obj, θ) */

```
rotate = (theta: number): Point => {
    const newX = this.x * Math.cos(theta) - this.y * Math.sin(theta);
    const newY = this.x * Math.sin(theta) + this.y * Math.cos(theta);
    return new SimplePoint(newX, newY);
}
```

RI: r = Math.sqrt(this.x * this.x + this.y * this.y)
AF: obj = (this.x, this.y)

Prove that the rotate method is correct using the RI or AF.