# CSE 331

## Reasoning about ADTs

James Wilcox and Kevin Zatloukal

# Motivation, part 1

- Hard to be <u>sure</u> ADT implementation is correct

```
/**
 * Adds a new integer to the front of the list
 * @param n the integer to add
 * @returns n :: obj
 */
public IntStack push(int n);
```

```
(2)   // AF: obj = rev(this.list)
      // RI: this.size = this.list.length

 public IntStack push(int n) {
   int newSize = this.size + 1;

   int[] newList = new int[newSize];
   for (int i = 0; i < this.size; i++) {
     newList[i] = this.list[i];
   }
   newList[newSize-1] = n;

   return new IntStackImpl(newList, newSize);
 }
```

- Would be nice to have tools for this!

# Motivation, part 2

- Typical CSE 331 final is about an ADT:
  - write part of the specification
  - test parts of the implementation
  - prove parts of the implementation correct

    plus some things we'll see later…

- This topic gets us through the core material

# Clients of ADTs

# Reasoning about Function Calls

- Not too difficult if the function is…
  1. defined for all inputs
  2. defined imperatively
  3. no arguments are mutated

- Plan for today:
  1. simple case with none of the problems above
  2. allow some undefined inputs
  3. ADTs
  4. declarative specifications

- We will <u>not</u> consider argument mutation this Topic

# Reasoning about Function Calls

- For the simplest case only…
- Forward reasoning rule is

$\{\{ P \}\}$

```
x = Math.sin(a);
```

$\{\{ P[x \mapsto x_0] \text{ and } x = \sin(a) \}\}$

```
// @param x
// @return sin(x)
double sin(double x)
```

- Backward reasoning rule is

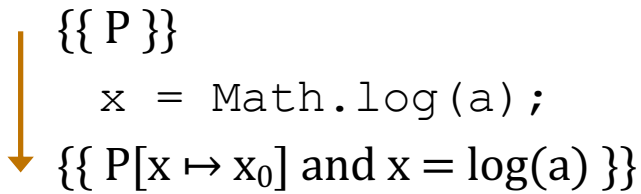
$\{\{ Q[x \mapsto \sin(a)] \}\}$

```
x = Math.sin(a);
```

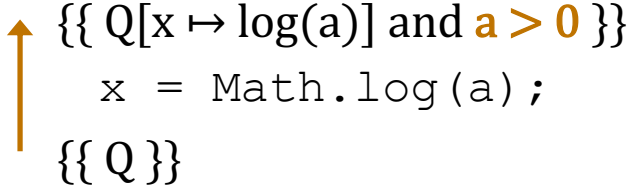
$\{\{ Q \}\}$

# Reasoning about Function Calls

- **Preconditions must be checked**
  - not valid to call the function on disallowed inputs
- **Forward reasoning rule is**




- **Backward reasoning rule is**




**Must also check** $a > 0$

```
// @param x with x > 0
// @return log(x)
double log(double x)
```

# Reasoning about Function Calls

- Applies to functions we define with imperative specs

```
// @param n a non-negative integer
// @returns square(n), where
//        square(0) := 0
//      square(n+1) := square(n) + 2n + 1
public int square(int n) {..}
```

- Reasoning is the same. E.g., forward rule is

{{ P }}

```
x = square(n);
```

{{ $P[x \mapsto x_0]$ and $x = \text{square}(n)$ }}

**Must** also check that `n` is non-negative

# Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ _____ }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

# Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

# Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

inverting operation gives $r^2 = x + 2$

c.f. to when $\mathrm{r} = r_o + 1$ and $\mathrm{P}(r_o)$

here we have $\mathrm{r} = \sqrt{r_o}$ and $r_o = x + 2$

second fact is *already* solved for $r_o$
so we can substitute right into left
instead of left into right

# Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ x ≥ 0 and r = √(x + 2) + 1 }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

$$\begin{aligned} r &= x + 2 \\ &\geq 0 + 2 \qquad \text{since } x \geq 0 \\ &\geq 0 \end{aligned}$$

this looks good

what did we forget?

# Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ ____________ }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

# Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

# Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________________ }}
    double r = x + 2;
    {{ √r + 1 = √(x + 2) + 1 and r ≥ 0 }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

# Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
```

$\{\{ x \geq 0 \}\}$

$\{\{ \sqrt{x+2} + 1 = \sqrt{x+2} + 1 \text{ and } x + 2 \geq 0 \}\}$

check this implication

```
    double r = x + 2;
```

$\{\{ \sqrt{r} + 1 = \sqrt{x+2} + 1 \text{ and } r \geq 0 \}\}$

```
    r = Math.sqrt(r);
```

$\{\{ r + 1 = \sqrt{x+2} + 1 \}\}$

```
    r = r + 1;
```

$\{\{ r = \sqrt{x+2} + 1 \}\}$

```
    return r;
}
```

$x \geq 0$ implies $x + 2 \geq 0$

# Reasoning about ADT Calls

- ADT methods are calls involving abstract states
- Forward reasoning rule is

$\{\{ P \}\}$

```
L = L.cons(x);
```

$\{\{ P[L \mapsto L_0] \text{ and } L = x :: L_0 \}\}$

```
// @return x :: obj
FastList cons(int x)
```

L is a mathematical list

- Backward reasoning rule is

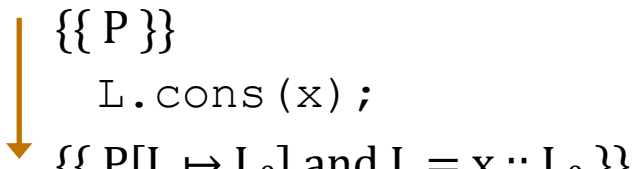
$\{\{ Q[L \mapsto x :: L] \}\}$

```
L = L.cons(x);
```

$\{\{ Q \}\}$

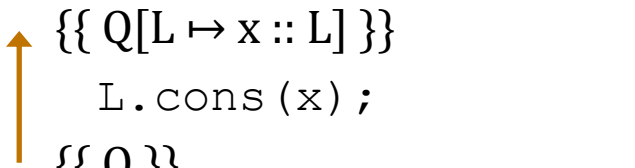
# Reasoning about ADT Calls

- Very little changes with mutators…
- Forward reasoning rule is




```
// @modifies obj
// @effects obj = x :: obj0
void cons(int x)
```

- Backward reasoning rule is




`@effects` says it is an assignment
so we get an identical result

# Example Calls with Declarative Specs

```
// @returns x such that x >= a and x >= b
public int max(int a, int b) {..}
```

- **Forward reasoning rule is**

{{ P }}
  x = **max**(a, b);
{{ P[x ↦ $x_0$] and x ≥ a and x ≥ b }}

- **Backward reasoning rule is**

{{ a > 0 }}
  x = **max**(a, b);
{{ a > 0 and 2x ≥ a + b }}

**Must check that** x ≥ a and x ≥ b **implies** 2x ≥ a + b

# Function Calls with Declarative Specs

```
// @requires P2                  -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
public int f(int a, int b) {..}
```

- **Forward reasoning rule is**

{{ P }}
  x = f(a, b);
{{ P[x ↦ $x_0$] and R }}

**Must also check that P implies $P_2$**

- **Backward reasoning rule is**

{{ $Q_1$ and $P_2$ }}
  x = f(a, b);
{{ $Q_1$ and $Q_2$ }}

**Must also check that R implies $Q_2$**

**$Q_2$ is the part of postcondition using "x"**

# Correct ADT Implementation

# Recall: Documenting the FastList ADT

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;
  …
}
```

- **Representation Invariant (RI) holds info about** this.last
  - **fields cannot have *just any* number and list of numbers**
  - **they must fit together by satisfying RI**

    last must be the last number in the list stored

# Correctness of FastList Constructor

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

  FastLastList(List L) {
    this.list = L;
    this.last = last(this.list);
  }
  …
```

- Constructor must ensure that RI holds at end
  - we can see that it does in this case
  - since we **don't mutate**, they will *always* be true

# Correctness of FastList Constructor

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

  // @effects obj = L
  FastLastList(List L) {
    this.list = L;
    this.last = last(this.list);
  }
```

- Constructor must create the requested abstract state
  - client wants obj to be the passed in list
  - we can see that $\text{obj} = \text{this.list} = \text{L}$

# Correctness of getLast

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @return last(obj)
  public int getLast() {
    return this.last;
  };
}
```

- **Use both RI and AF to check correctness**

last(obj) =

# Correctness of getLast

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @return last(obj)
  public int getLast() {
    return this.last;
  };
}
```

- **Use both RI and AF to check correctness**

| | | |
|---|---|---|
| last(obj) | = last(this.list) | **by AF** |
| | = this.last | **by RI** |

# Correctness of Immutable ADTs

- Check that the constructor…
  - creates a concrete state satisfying RI
  - creates the abstract state required by the spec
- Check the correctness of each method…
  - check value returned is the one stated by the spec
  - will need to use RI in most methods
  - will need to use AF in every method

# ADTs: the Good and the Bad

- Provides data abstraction
  - can change data structures without breaking clients
- Comes at a cost
  - more work both to specify and to check correctness
- Not everything needs to be an ADT
  - don't be like Java and make everything a class
- Prefer concrete types for most things
  - concrete types are easier to think about
  - introduce ADTs when the first *change* occurs

# Immutable Queues

# Recall: Immutable Queue

- A queue is a list that can *only* be changed two ways:
  - add elements to the front
  - remove elements from the back

```
// List that only supports adding to the front and
// removing from the end
interface NumberQueue {

  // @return len(obj)
  int size();

  // @return [x] ++ obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;
```

- Easiest implementation is concrete = abstract state
  - just store the abstract state in a field
- Still requires extra work to check correctness...
  - abstraction barrier comes with a cost

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @returns len(obj)
  public int size() {
    return len(this.items);
  };
```

- **Correctness of `size`:**

  len(this.items) =

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @returns len(obj)
  public int size() {
    return len(this.items);
  };
```

- **Correctness of `size`:**

$$\text{len}(\text{this.items}) = \text{len}(\text{obj})$$ **by AF**

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @effects obj = items
  ListQueue(List items) {
    this.items = items;
  }
```

- **Correctness of `constructor`:**

items =

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @effects obj = items
  ListQueue(List items) {
    this.items = items;
  }
```

- **Correctness of `constructor`:**

items = this.items (from code)
= obj **AF**

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @returns [x] ++ obj
  public NumberQueue enqueue(int x) {
    return new ListQueue(cons(x, this.items));
  };
```

- **Correctness of `enqueue`:**

return value =

# Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  private List items;

  // @returns [x] ++ obj
  public NumberQueue enqueue(int x) {
    return new ListQueue(cons(x, this.items));
  };
```

- **Correctness of `enqueue`:**

| | | |
|---|---|---|
| return value | = x :: this.items | **spec of constructor** |
| | = x :: obj | **AF** |
| | = [] ⧺ (x :: obj) | **def of** concat |
| | = [x] ⧺ obj | **def of** concat |

# Summary of `ListQueue`

- Simplest possible implementation of ADT
  - abstract state = concrete state of one field

- Reasoning about every method is more complex
  - must apply AF to relate return value to spec's postcondition

    code uses fields, but postcondition uses "obj"
  - this is the cost of the abstraction barrier