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

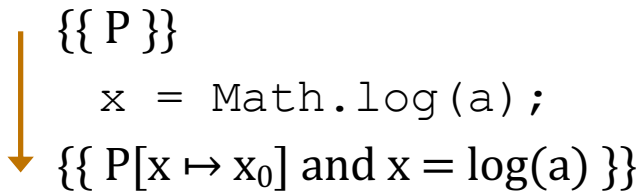
$\{\{ Q[x \mapsto \sin(a)] \}\}$

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

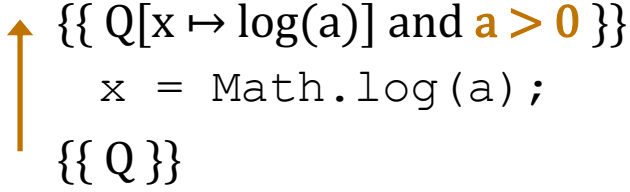
$\{\{ Q \}\}$

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

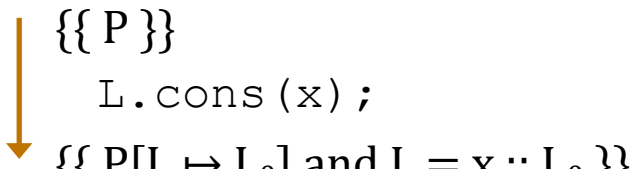
$\{\{ Q[L \mapsto x :: L] \}\}$

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

$\{\{ Q \}\}$

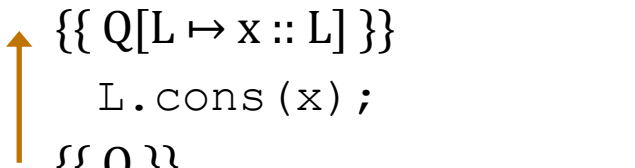
Reasoning about ADT Calls

- 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

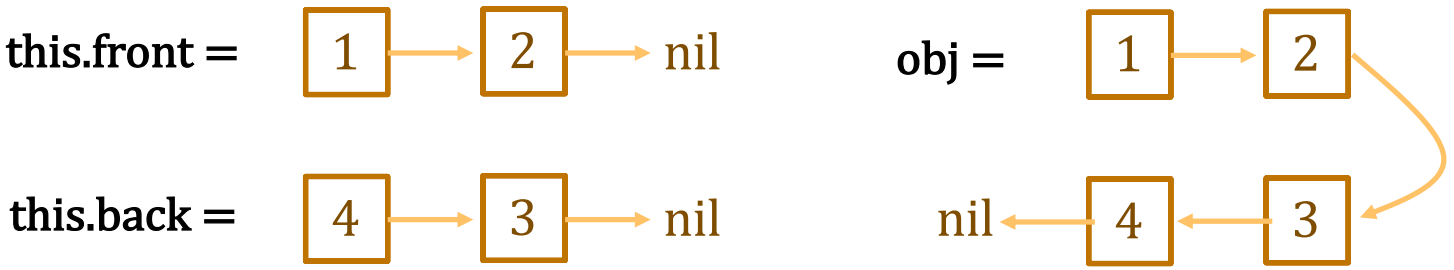
    code uses fields, but postcondition uses "obj"
  - this is the cost of the abstraction barrier

Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  private List front;
  private List back;   // in reverse order
```

- Back part stored in reverse order
  - head of front is the first element
  - head of back is the *last* element



Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  // RI: if this.back = nil, then this.front = nil
  private List front;
  private List back;
```

- If back is nil, then the queue is *empty*
  - if back = nil, then front = nil (by RI) and thus

    obj =

Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  // RI: if this.back = nil, then this.front = nil
  private List front;
  private List back;
```

- **If back is nil, then the queue is *empty***
  - **if** back = nil, **then** front = nil **(by RI) and thus**

$$\begin{aligned} \text{obj} &= \text{nil} +\!\!+ \text{rev}(\text{nil}) && \textbf{by AF} \\ &= \text{rev}(\text{nil}) && \textbf{def of } \text{concat} \\ &= \text{nil} && \textbf{def of } \text{rev} \end{aligned}$$

  - **if the queue is not empty, then back is not nil**

    (**311 alert**: this is the contrapositive)

Implementing a Queue with Two Lists

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

  // AF: obj = this.front ++ rev(this.back)
  // RI: if this.back = nil, then this.front = nil
  private List front;
  private List back;

  // makes obj = front ++ rev(back)
  ListPairQueue(List front, List back) {
    …
  }
```

- Will implement this later…

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
private List front;
private List back;

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

- **Correctness of `size`:**

$$\text{len(obj)} =$$

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
private List front;
private List back;

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

- **Correctness of `size`:**

$$\begin{aligned} \text{len(obj)} &= \text{len(this.front} +\!\!+ \text{rev(this.back))} && \text{by AF} \\ &= \text{len(this.front)} + \text{len(rev(this.back))} && \text{by Example 3} \\ &= \text{len(this.front)} + \text{len(this.back)} && \text{by another induction} \end{aligned}$$

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
private List front;
private List back;

// @returns [x] ++ obj
public NumberQueue enqueue(int x) {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

- **Correctness of `enqueue`:**

ret value =

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
private List front;
private List back;

// @returns [x] ++ obj
public NumberQueue enqueue(int x) {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

- **Correctness of** `enqueue`**:**

ret value = (x :: this.front) ⧺ rev(this.back)	**spec of constructor**
= x :: (this.front ⧺ rev(this.back))	**def of** concat
= x :: obj	**AF**
= [] ⧺ (x :: obj)	**def of** concat
= [x] ⧺ obj	**def of** concat

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
private List front;
private List back;

// @requires len(obj) > 0
// @returns (x, Q) with obj = Q ++ [x]
public DequeueParts dequeue() {
  return new DequeueParts(this.back.hd,
          new ListPairQueue(this.front, this.back.tl));
};
```

- as noted previously, precondition means this.back ≠ nil
- as we know, this means this.back = x :: L
  where x = this.back.hd and some L = this.back.tl

Implementing a Queue with Two Lists

```
// @requires len(obj) > 0
// @returns (x, Q) with obj = Q ++ [x]
public DequeueParts dequeue() {
  return new DequeueParts(this.back.hd,
          new ListPairQueue(this.front, this.back.tl));
};
```

- this.back = x :: L, where x = this.back.hd and L = this.back.tl

obj =

...

...

= (this.front ++ rev(this.back.tl)) ++ [this.back.hd]

Implementing a Queue with Two Lists

```
// @requires len(obj) > 0
// @returns (x, Q) with obj = Q ++ [x]
public DequeueParts dequeue() {
  return new DequeueParts(this.back.hd,
          new ListPairQueue(this.front, this.back.tl));
};
```

– this.back = x :: L, where x = this.back.hd and L = this.back.tl

obj	= this.front # rev(this.back)	**by AF**
	= this.front # rev(this.back.hd :: this.back.tl)	**since** back = x :: L
	= this.front # (rev(this.back.tl) # [this.back.hd])	**def of** rev
	= (this.front # rev(this.back.tl)) # [this.back.hd]	

Well-Known Facts About List Concatenation

- The List notes mention these facts

Identity $L +\!\!+ \text{nil} = L$ ($\text{nil} +\!\!+ L = L$ by definition)

Associativity $(L +\!\!+ R) +\!\!+ S = L +\!\!+ (R +\!\!+ S)$

- We will use those going forward

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
private List front;
private List back;

// makes obj = front ++ rev(back)
ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

holds since this.front = nil

holds since this.back ≠ nil

- Need to check that RI holds at end of constructor

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
private List front;
private List back;

// makes obj = front ++ rev(back)
ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

$\text{obj} = \text{nil} +\!\!+ \text{rev}(\text{rev}(\text{front}))$??

$\text{obj} = \text{front} +\!\!+ \text{rev}(\text{back})$

- Need to check this creates correct abstract state

Implementing a Queue with Two Lists

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
private List front;
private List back;

ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    …
  }
}
```

obj = nil ⧺ rev(rev(front))	
= nil ⧺ front	**AF**
= front	**because I said so**
= front ⧺ nil	**def of** concat
= front ⧺ rev(nil)	**def of** rev
= front ⧺ rev(back)	**since** back = nil

Correct ADT Implementation (Mutable Case)

Mutable ADTs

- **Previously:**
  - **state was immutable**
  - **set in the constructor and then never changed**

    only need to confirm RI holds at the end of the constructor

    if RI holds there, then it holds forever

- **Now:**
  - **allow state to be changed by methods**

Correctness of Mutable ADTs

- Check that the constructor…
  - creates a concrete state satisfying RI
  - creates the abstract state required by the spec
- Check the correctness of each observer method…
  - check value returned is the one stated by the spec
- Check the correctness of each mutator method…
  - check abstract state produced is one stated by the spec
  - check that the RI still holds
  - make sure there are no aliases

Recall: Mutable Version of Fast List ADT

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @return last(obj)
  int getLast();

  // @return obj
  List getList();

  // @modifies obj
  // @effects obj = x :: obj_0
  void cons(int x);
}
```

mutator method

Recall: One Concrete Rep for `FastList`

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

  // makes obj = list
  MutableFastListImpl(List list) {
    this.list = list;
    this.last = last(this.list);
  }
```

- We can use the same rep for a mutable version

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
  };
```

- **Let's check correctness...**

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
  };
```

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
};
```

What is missing?

Also, need the RI to hold!

obj	$= \text{this.list}$	by AF
	$= x :: \text{this.list}_0$	since $\text{this.list} = x :: \text{this.list}_0$
	$= x :: \text{obj}_0$	by AF

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 and
             this.last = last(this.list) }}
  };
```

Also, need the RI to hold!

Does it? No!

- Postcondition is **`@returns`**, **`@effects`**, and **RI**

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

Rep Invariant now holds

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.last = last(this.list);
    {{ this.last = last(this.list) }}
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list_0) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
};
```

Rep Invariant would not hold if we switched the order

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
};
```

This version is obviously correct, but O(n).

Can we do it faster?

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
    {{ ____________________________________________ }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

O(1) version, but more complex reasoning (two branches)

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list0 and this.list0 = nil and this.last = x }}
    {{ Post: obj = x :: obj0 and this.last = last(this.list) }}
  };
```

Case "then":

last(this.list) = last(x :: this.list$_0$) **since** this.list = x :: this.list$_0$

= last(x :: nil) **since** this.list$_0$ = nil

= x **def of** last

= this.last **since** x = this.last

last(x :: nil) := x

last(x :: y :: L) := last(y :: L)

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
```

{{ this.list = x :: this.list_0 and $\text{this.list}_0 \neq$ nil and
this.last = this.last_0 and **$\text{this.last}_0 = \text{last}(\text{this.list}_0)$** }}

{{ **Post**: obj = x :: obj_0 and this.last = last(this.list) }}

from the RI
(will need this)

Case "else":

last(this.list) = last(x :: this.list_0)
= last(this.list_0)
= this.last_0
= this.last

since this.list = x :: this.list_0
def of last (**since** $\text{this.list}_0 \neq$ nil)
since this.last_0 = last(this.list_0)
since this.last = this.last_0

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

Defensive Programming

Defensive Programming

- We try to catch all bugs via
  1. Type checking
  2. Reasoning
  3. Testing

- But some will get through…

- Add extra checks to catch them
  - reduces the work of debugging
  - we call this "defensive programming"

Defensive Programming

1. Add checks for invalid inputs
   - clients should not pass them
     they violate the **precondition**
   - given enough clients, some will pass invalid inputs

- EJ 49: check parameters for validity

- Check these if's not too expensive
  - would only skip if it would cause asymptotic slowdown
  - e.g., binary search cannot check the array is sorted

Defensive Programming

2. Check that the RI holds at the end of mutators
   - mutation could produce an invalid state
   - would cause painful debugging

     code doesn't crash then

     failure doesn't occur until the **next** method call (or later)
   - we call this method "`checkRep`" in 331

- Worth checking even if it is expensive
  - add a flag to enable them when testing

Defensive Programming

3. Check that the RI holds at the start of mutators
   - wait, why?
   - that's not even possible... is it?

- Can happen with rep exposure
  - mutation through an alias that breaks the RI
  - could be worse

    they could mutate it in a way that doesn't break the RI

    it's likely still a bug because the abstract state was wrongly changed

Using Mutable ADTs

Recall: Mutable Version of Fast List ADT

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @return last(obj)
  int getLast();

  // @returns first(obj), where
  //     first(nil)    := 0
  //     first(x :: L) := x
  int getFirst();

  // @return obj
  List getList();

  // @modifies obj
  // @effects obj = x :: obj_0
  void cons(int x);
}
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    i++;
  }
};
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R_0 }}
    int m = L.getFirst();
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R_0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
    int m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R0 }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R0 }}
    i++;
  }
};
```

$R = (m+i) :: R_1$

$= (m+i) :: (m+i-1) :: \ldots :: (m+1) :: R_0$ since $R_1 = \ldots$

Using the Mutable List ADT

```
void g(MutableFastList L, MutableFastList R, int k)
```

- We have proven this code correct, but…

"Beware of bugs in the above code;
I have only proved it correct, not tried it."

Donald Knuth, 1977

- We should also try it…

Using the Mutable List ADT

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
void g(MutableFastList L, MutableFastList R, int k)
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, R, 3)
System.out.println(R);
```

- **What list should this print?**

5 :: 4 :: 3 :: 2 :: 1 :: nil

Using the Mutable List ADT

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
void g(MutableFastList L, MutableFastList R, int k)
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, R, 3)
System.out.println(R);
```

- **Instead, it prints** $8 :: 5 :: 3 :: 2 :: 1 :: nil$ **! How?!?**

L and R are aliases to the same MutableFastList

Reasoning with Aliases

- **Aliasing breaks reasoning!**
  - **there was nothing wrong with our math**
  - **our math did not correctly describe the program**

    modeling programs with aliasing is basically impossible

Another Mutable Queue ADT

- Another mutable version with different methods

```
// Mutable array that only supports adding to the front
// and removing from the end.
interface MutableNumberQueue {

  // @returns obj
  List<Integer> elements();

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  void enqueue(int x);

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  int dequeue();
}
```

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
  }
```

We should check this...

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Is this really correct?

No way to know for sure at the next method call!

obj = rev(this.front) # this.back — by AF

= rev([]) # this.back — since this.front = []

= [] # this.back — def of rev

= this.back = vals — since this.back = vals

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = new ArrayList<>(vals);
  }
```

- Must make a copy of the array!
  - then, we have the only reference to it (no aliases)

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    ArrayList<Integer> result = new ArrayList<>();
    result.addAll(this.front);
    Collections.reverse(result);
    result.addAll(this.back);
    return result;
  };
```

This is O(n)...

We can optimize it if front = [].

$$\text{rev}([]) \mathbin{+\!\!+} \text{this.back} = [] \mathbin{+\!\!+} \text{this.back} = \text{this.back}$$

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    if (this.front.size() == 0) {
      return this.back;   // O(1) when this.front = []
    } else {
      ArrayList<Integer> result = new ArrayList<>();
      result.addAll(this.front);
      Collections.reverse(result);
      result.addAll(this.back);
      return result;
    }
  };
```

Is this correct?

No way to say!

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    ArrayList<Integer> result = new ArrayList<>();
    result.addAll(this.front);
    Collections.reverse(result);
    result.addAll(this.back);
    return result;
  };
```

- Cannot return an alias to `this.back`
  - must make a copy in all cases

Moral of the Story for Mutable Heap State

- More mutation gave us better efficiency
  - saved memory
  - immutable version could be just as fast
- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes
  - more work to make sure we did it right
- New possibilities for exciting bugs!
  - must avoid aliasing of anything mutable

    this is "**representation exposure**"

Need for Mutable Heap State

- Saw that aliased mutable heap state is complex
  - avoid mixing aliasing and mutation
- Use coding conventions depending on context
  1. server-side data storage – mutation without aliasing
  2. client-side UI – aliasing without mutation
- In other cases, may need other conventions
  - two phase builder pattern