CSE 331

Mutable ADTs

James Wilcox and Kevin Zatloukal

Specifying & Using Mutable ADTs

Recall: Immutable Map

- An "association list" also called a "map"

```
// List of (key, value) pairs
interface Map<K, V> {

  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;

  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;

  // @returns set-value(x, v, obj)
  setValue(x: K, v: V): Map<K, V>;
}
```

observer

observer

producer

Using the Immutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  const r = M1.getValue("one");
  return r;
};
```

- Let's check that this code is correct...

Using the Immutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

```
// @requires contains-key(x, obj)
// @returns get-value(x, obj)
getValue(x: K): V;
```

Using the Immutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

```
// @returns set-value(x, v, obj)
setValue(x: K, v: V): Map<K, V>;
```

Using the Immutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  {{ get-value("one", set-value("one", 2, M)) > 0 }}
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

Using the Immutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  {{ get-value("one", set-value("one", 2, M)) > 0 }}
  const M1 = M.setValue("one", 2);
  {{ get-value("one", M_1) > 0 }}
  const r = M1.getValue("one");
  {{ Post: r > 0 }}
  return r;
};
```

get-value("one", set-value("one", 2, M))
  = get-value("one", ("one", 2) :: M)   **def of** set-value
  = 2   **def of** get-value
  > 0

set-value(x, v, L) := (x, v) :: L

get-value(x, (y, v) :: M) := v   if $x = y$
get-value(x, (y, v) :: M) := get-value(x, M)   if $x \neq y$

Recall: Immutable Map

- An "association list" also called a "map"

```
// List of (key, value) pairs
interface Map<K, V> {

  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;

  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;

  // @returns set-value(x, v, obj)
  setValue(x: K, v: V): Map<K, V>;
}
```

observer

observer

producer

Recall: Mutable Map

- An "association list" also called a "map"

observer

observer

mutator

```
// List of (key, value) pairs
interface Map<K, V> {

  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;

  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;

  // @modifies obj
  // @effects obj = set-value(x, v, obj)
  setValue(x: K, v: V): void;
}
```

We still need the immutable math functions (e.g., set-value) to *define* a mutable ADT

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  return r;
};
```

- Let's check that this code is correct...
  - try this forward this time...

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  return r;
};
```

```
// @modifies obj
// @effects obj = set-value(x, v, obj)
setValue(x: K, v: V): void;
```

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  {{ M = set-value("one", 2, M0) }}
  const r = M.getValue("one");
  return r;
};
```

Notice that two versions (M_0 vs M_1)
show up in the *reasoning*
even though our code has one version!

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
```

{{ $M = \text{set-value}(\text{"one"}, 2, M_0)$ }}

```
  const r = M.getValue("one");
```

{{ $M = \text{set-value}(\text{"one"}, 2, M_0)$ and $r = \text{get-value}(\text{"one"}, M)$ }}

```
  return r;
};
```

Using the Mutable Map

```
// @returns a positive number
const f = (M: Map<string, number>): number => {
  M.setValue("one", 2);
  const r = M.getValue("one");
  {{ M = set-value("one", 2, M₀) and r = get-value("one", M) }}
  {{ Post: r > 0 }}
  return r;
};
```

$r = \text{get-value}(\text{"one"}, M)$

$= \text{get-value}(\text{"one"}, \text{set-value}(\text{"one"}, 2, M_0))$	**since** $M = \text{set-value}(\text{"one"}, 2, M_0)$
$= \text{get-value}(\text{"one"}, (\text{"one"}, 2) :: M_0)$	**def of** set-value
$= 2$	**def of** get-value
> 0	

$\text{set-value}(x, v, L) := (x, v) :: L$

$\text{get-value}(x, (y, v) :: M) := v \quad \text{if } x = y$

$\text{get-value}(x, (y, v) :: M) := \text{get-value}(x, M) \quad \text{if } x \neq y$

Implementing Mutable ADTs

ADTs

- Main place we have heap state is in an ADT
- 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

ADTs

- Main place we have heap state is in an ADT

- New Power:
  - allow state to be changed by methods

- New Responsibilities:
  - more complex specifications

    add `@effects` and `@modifies`
  - must check the RI holds after any method that mutates

    often a good idea to write code to check this at runtime
  - more responsibilities we will meet later…

Recall: List ADT with a Fast `getLast`

```
// Represents an (immutable) list of numbers.
interface FastList {

  // @returns x :: obj
  cons: (x: bigint) => FastList;

  // @returns last(obj)
  getLast: () => bigint|undefined;

  // @returns obj
  toList: () => List<bigint>;
};

const makeFastList = (): FastList => {
  return new FastListImpl(nil);
};
```

producer method

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  cons: (x: bigint) => void;
  …
```

mutator method

- Method `cons` changes the list, putting `x` in front
  - now returns `void`
  - mutation explained in `@modifies` and `@effects`

    abstract state is the old abstract state with x put in front

Recall: One Concrete Rep for `FastList`

```
class FastListImpl implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  readonly last: bigint | undefined;
  readonly list: List<bigint>;

  constructor(list: List<bigint>) {
    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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    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** $\text{this.list} = \text{cons}(x, \text{this.list}_0)$
	$= x :: \text{obj}_0$	**by AF**

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list₀ }}
    {{ Post: obj = x :: obj₀ and
          this.last = last(this.list) }}
  };
```

Also, need the RI to hold!

Does it? No!

- Postcondition is `@returns`, `@effects`, and <u>RI</u>

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list₀ and this.last = last(this.list) }}
    {{ Post: obj = x :: obj₀ 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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    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
  last: bigint | undefined;
  list: List<bigint>;

  // @modifies obj
  // @effects obj = x :: obj_0
  cons = (x: bigint): void => {
    if (this.list === nil)
      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 {

  cons = (x: bigint): void => {
    if (this.list === nil)
      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":

$\text{last}(\text{this.list}) = \text{last}(x :: \text{this.list}_0)$ — **since** $\text{this.list} = x :: \text{this.list}_0$

$= \text{last}(x :: \text{nil})$ — **since** $\text{this.list}_0 = \text{nil}$

$= x$ — **def of** last

$= \text{this.last}$ — **since** $x = \text{this.last}$

$$\text{last}(x :: \text{nil}) := x$$
$$\text{last}(x :: y :: L) := \text{last}(y :: L)$$

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  cons = (x: bigint): void => {
    if (this.list === nil)
      this.last = x;
    this.list = cons(x, this.list);
```

{{ this.list = x :: this.list$_0$ and this.list$_0$ ≠ nil and
this.last = this.last$_0$ and **this.last$_0$ = last(this.last$_0$)** }}

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

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

More Using Mutable ADTs

Mutable List ADT with a Fast `getLast`

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

  // @modifies obj
  // @effects obj = x :: obj_0
  cons: (x: bigint) => void;

  // @returns first(obj), where
  //       first(nil)      := 0
  //       first(x :: L)  := x
  getFirst: () => bigint|undefined;

  // @returns last(obj), where …
  getLast: () => bigint|undefined;

};
```

Using the Immutable Map

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

Using the Immutable Map

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

Using the Immutable Map

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const g = (L: MutableFastList, k: bigint,
           R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R₀ }}
    const 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 Immutable Map

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const g = (L: MutableFastList, k: bigint,
           R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const 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 Immutable Map

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const g = (L: MutableFastList, k: bigint,
           R: MutableFastList): void => {
  let i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    const 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 Immutable Map

```
// @requires L /= nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
const g = (L: MutableFastList, k: bigint,
           R: MutableFastList): void => {
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  for (let i = 1; i <= k; i++) {
    const m = L.getFirst();
    R.cons(m + i);
```

{{ $R = (m+i) :: R_1$ and $R_1 = (m+i-1) :: \ldots :: (m+1) :: R_0$ and $m = \text{first}(L)$ }}

{{ $R = (m+i) :: \ldots :: (m+1) :: R_0$ }}

```
    i++;
  }
};
```

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

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

Using the Immutable Map

```
const g = (L: MutableFastList, k: bigint,
           R: MutableFastList): void
```

- 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 Immutable Map

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

- Try out the code:

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, 3, R)
console.log(R);
```

- What list should this print?

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

Using the Immutable Map

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

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, R, 3)
console.log(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 describing the program

    modeling programs with aliasing is basically impossible

- Isn't this just a weird, special case?
  - just double check that $\mathrm{L} \neq \mathrm{R}$

- How about a more practical example?

Demo: New HW8 Features

Reasoning with Aliasing

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

    modeling programs with aliasing is basically impossible

- Aliasing is rampant in applications!
  - **no way** to easily check that there are no aliases
  - cannot reason about or debug individual functions

- Only option is to prevent unexpected aliasing…

Aliasing and Mutation Don't Mix

- Aliasing breaks reasoning!
  - Root cause: mutating aliased data
  - Fix: allow mutation XOR aliasing (i.e., not both)
- Option 1: data is immutable
  - program can't tell if data is aliased
- Option 2: data is not aliased
  - local reasoning principles work great
  - new responsibility: no aliases of *my* data
- See also: Rust enforces this rule with type checker

Implementing *More* Mutable ADTs

Recall: Immutable Queue ADT

- 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 {

  // @returns len(obj)
  size: () => bigint;

  // @returns [x] ++ obj
  enqueue: (x: bigint) => NumberQueue;

  // @requires len(obj) > 0
  // @returns (x, Q) with obj = Q ++ [x]
  dequeue: ()=> [bigint, NumberQueue];
}
```

observer

producer

producer

Mutable Queue ADT

- Mutable versions has mutators instead of producers

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

  // @returns obj
  elements(): bigint[];

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

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

observer

mutator

mutator

Recall: 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
  readonly front: List;
  readonly back: List;

  // makes obj = concat(front, rev(back))
  constructor(front: List, back: List) {
    …
  }
```

- Queue was in two parts, front and back
  - back stored in reverse order
  - full list was $\text{this.front} \mathbin{+\!\!\!+} \text{rev}(\text{this.back})$

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    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
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    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
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Is this really correct?

No way to say!

obj	$= \text{rev}(\text{this.front}) \mathbin{+\!\!+} \text{this.back}$	by AF
	$= \text{rev}([]) \mathbin{+\!\!+} \text{this.back}$	since this.front = []
	$= [] \mathbin{+\!\!+} \text{this.back}$	def of rev
	$= \text{this.back} = \text{vals}$	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
  front: bigint[];
  back: bigint[];

  // makes obj = vals
  constructor(vals: bigint[]) {
    this.front = [];
    this.back = vals.slice(0, vals.length);
  }
```

- 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
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    let revFront: bigint[] =
      this.front.slice(0, this.front.length);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

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
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    if (this.front.length === 0) {
      return this.back;   // O(1) when this.front = []
    } else {
      let revFront: bigint[] =
        this.front.slice(0, this.front.length);
      revFront.reverse();
      return revFront.concat(this.back);
    }
  };
```

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
  front: bigint[];
  back: bigint[];

  // @returns obj
  elements = (): bigint[] => {
    let revFront: bigint[] = this.front.slice(0);
    revFront.reverse();
    return revFront.concat(this.back);
  };
```

- 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

    we call this "**representation exposure**"

Need for Mutable Heap State

- Saw that mutable heap state is complex
  - better to avoid when possible

- Cannot be avoided in some cases
  1. server-side data storage
  2. client-side UI

- In both cases, we try to constrain its use
  - including coding conventions to keep ourselves sane