CSE 331

Abstraction Functions & Invariants

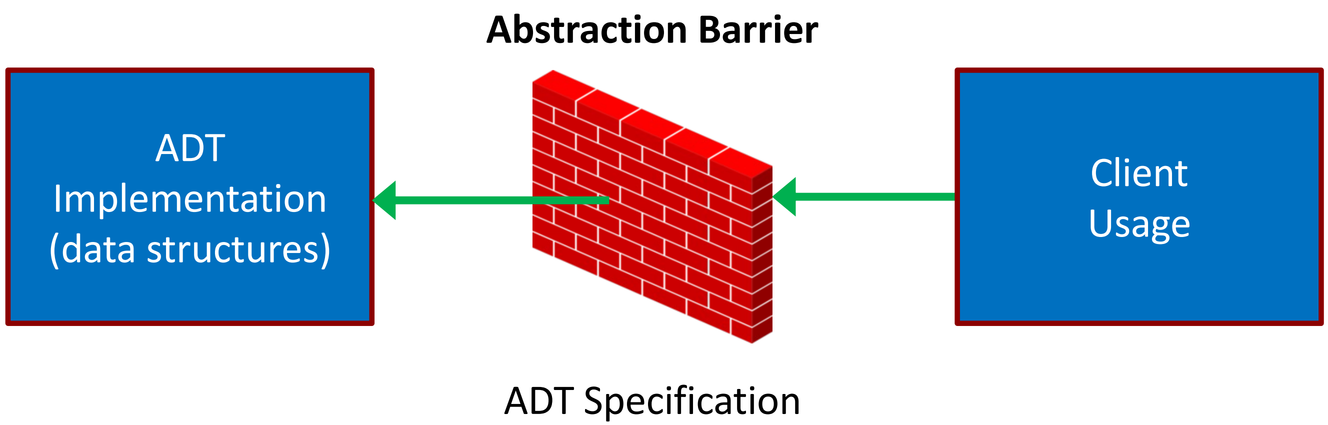
Kevin Zatloukal

Administrivia

- Bring your laptop to section tomorrow
  - we'll be doing some coding
- Do the coding setup <u>**beforehand**</u>
  - will post a message on Ed with instructions
- Section will be critical for next HW
  - practice refactoring existing code into an ADT
  - proofs about trees
- Last homework without mutation
  - today's lecture completes the full set of reasoning tools

Abstraction Barrier

- Last time, we saw *data* abstraction



  - specification is the "barrier" between the sides

    hides the details of the data structure from the client

  - ADT specification is a collection of *functions*

    reduce data abstraction to procedural abstraction

Documenting an ADT Implementation

Documenting an ADT Implementation

- Last lecture, we saw how to write an ADT spec
- Key idea is the "abstract state"
  - meaning of an object in math terms
  - how clients should think (reason) about the object
- Write specifications in terms of the abstract state
  - describe the return value in terms of "obj"
- We also need to reason about ADT implementation
  - for this, we do want to talk about fields
  - fields are hidden from clients, but visible to implementers

Documenting an ADT Implementation

- We also need to document the ADT implementation
  - for this, we need two new tools

Abstraction Function

defines what abstract state the field values currently represent

- Maps the field values to the object they represent
  - object is math, so this is a *mathematical* function

    there is no such function in the code — just a tool for reasoning
  - will usually write this as an *equation*

    $\text{obj} = \ldots$ **right-hand side uses the fields**

Documenting the FastList ADT

```
class FastLastList implements FastList {
  // AF: obj = this.list
  readonly last: number | undefined;
  readonly list: List<number>;
  …
}
```

- **Abstraction Function (AF) gives the abstract state**
  - obj **= abstract state**
  - this **= concrete state (record with fields** .last **and** .list**)**
  - **AF relates abstract state to the current concrete state**

    okay that "last" is not involved here
  - **specifications only talk about** "obj"**, not** "this"

    "this" will appear in our reasoning

Documenting an ADT Implementation

- We also need to document the ADT implementation
  - for this, we need two new tools

Abstraction Function

defines what abstract state the field values currently represent

only needs to be defined when RI is true

Representation Invariants (RI)

facts about the field values that should always be true

defines what field values are allowed

AF only needs to apply when RI is true

Documenting the FastList ADT

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

- 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
  readonly last: number | undefined;
  readonly list: List<number>;

  constructor(L: List<number>) {
    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
  readonly last: number | undefined;
  readonly list: List<number>;

  // makes obj = L
  constructor(L: List<number>) {
    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 $\mathrm{obj} = \mathrm{this.list} = \mathrm{L}$

Correctness of getLast

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @returns last(obj)
  getLast = (): number | undefined => {
    return this.last;
  };
}
```

- Use both RI and AF to check correctness

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

Correctness of ADT implementation

- 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
  - may need to use both RI and AF

ADTs: the Good and the Bad

- Provides data abstraction
  - can change data structures without breaking clients
- Comes at a cost
  - more work to specify and 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

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 {

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

  // @returns cons(x, obj)
  enqueue: (x: number) => NumberQueue;

  // @requires len(obj) > 0
  // @returns (x, Q) with obj = concat(Q, cons(x, nil))
  dequeue: () => [number, NumberQueue];
}
```

observer (size)

producer (enqueue)

producer (dequeue)

$\mathrm{last}(obj) = x$ by HW4 problem 5!

Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {
  // AF: obj = this.items
  readonly items: List<number>;
```

- Easiest implementation is concrete = abstract state
  - just store the abstract state in a field
  - (see HW5)

- 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
  readonly items: List<number>;

  // @returns len(obj)
  size = (): number => {
    return len(this.items);
  };
```

- **Correctness of `size`:**

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

nothing is Level 0 anymore

Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {
  // AF: obj = this.items
  readonly items: List<number>;

  // makes obj = items
  constructor(items: List<number>) {
    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
  readonly items: List<number>;

  // @returns cons(x, obj)
  enqueue = (x: number): NumberQueue => {
    return new ListQueue(cons(x, this.items));
  };
```

- **Correctness of `enqueue`:**

return value	= cons(x, this.items)	**spec of constructor**
	= cons(x, obj)	**AF**

Implementing a Queue with a List

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {
  // AF: obj = this.items
  readonly items: List<number>;

  // @requires len(obj) > 0
  // @returns (x, Q) with obj = concat(Q, cons(x, nil))
  dequeue = (): [number, NumberQueue] => {
    return [last(this.items),
              prefix(len(this.items) - 1, this.items)];
  };
```

- Declarative spec, so more reasoning is required!
  - also, slower than necessary ($\Theta(n)$ dequeue)
  - we'll skip correctness here and do something faster in a moment…

Summary of `ListQueue`

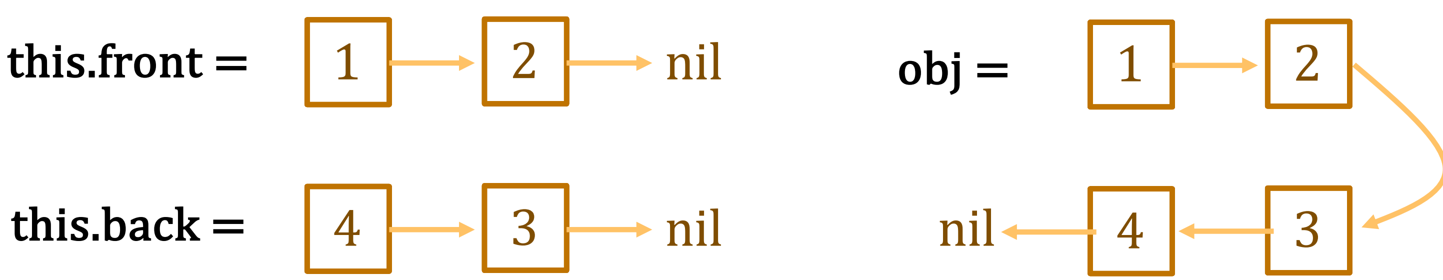
- Simplest possible implementation of ADT
  - abstract state = concrete state of one field
- Reasoning about every method is more complex
  - must apply AF to relate return value to spec's postcondition
    - code uses fields, but postcondition uses "obj"
  - this is the cost of the abstraction barrier
- Will use this approach to start HW5

Implementing a Queue with Two Lists

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

  // AF: obj = concat(this.front, rev(this.back))
  readonly front: List<number>;
  readonly back: List<number>;   // 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 = concat(this.front, rev(this.back))
  // RI: if this.back = nil, then this.front = nil
  readonly front: List<number>;
  readonly back: List<number>;
```

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

$$\text{obj} \; =$$

Implementing a Queue with Two Lists

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

  // AF: obj = concat(this.front, rev(this.back))
  // RI: if this.back = nil, then this.front = nil
  readonly front: List<number>;
  readonly back: List<number>;
```

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

obj	$= \text{concat}(\text{nil}, \text{rev}(\text{nil}))$	by AF
	$= \text{rev}(\text{nil})$	def of concat
	$= \text{nil}$	def of rev

  - 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 = concat(this.front, rev(this.back))
  // RI: if this.back = nil, then this.front = nil
  readonly front: List<number>;
  readonly back: List<number>;

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

- Will implement this later…

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @returns len(obj)
size = (): number => {
  return len(this.front) + len(this.back);
};
```

- **Correctness of `size`:**

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

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @returns len(obj)
size = (): number => {
  return len(this.front) + len(this.back);
};
```

- **Correctness of `size`:**

$$\begin{aligned} \text{len(obj)} &= \text{len(concat(this.front, rev(this.back)))} && \textbf{by AF} \\ &= \text{len(this.front)} + \text{len(rev(this.back))} && \textbf{by Example 3} \\ &= \text{len(this.front)} + \text{len(this.back)} && \textbf{by Example 4} \end{aligned}$$

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @returns cons(x, obj)
enqueue = (x: number): NumberQueue => {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

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

ret value =

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @returns cons(x, obj)
enqueue = (x: number): NumberQueue => {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

- **Correctness of `enqueue`:**

ret value	= concat(cons(x, this.front), rev(this.back))	**(constructor)**
	= cons(x, concat(this.front, rev(this.back))	**def of** concat
	= cons(x, obj)	**AF**

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @requires len(obj) > 0
// @returns (x, Q) with obj = concat(Q, cons(x, nil))
dequeue = (): [number, NumberQueue] => {
  return [this.back.hd,
          new ListPairQueue(this.front, this.back.tl)];
};
```

- as noted previously, precondition means $\text{this.back} \neq \text{nil}$
- as we know, this means $\text{this.back} = \text{cons}(x, L)$ for some $x : \mathbb{Z}$ and some $L : \text{List}$

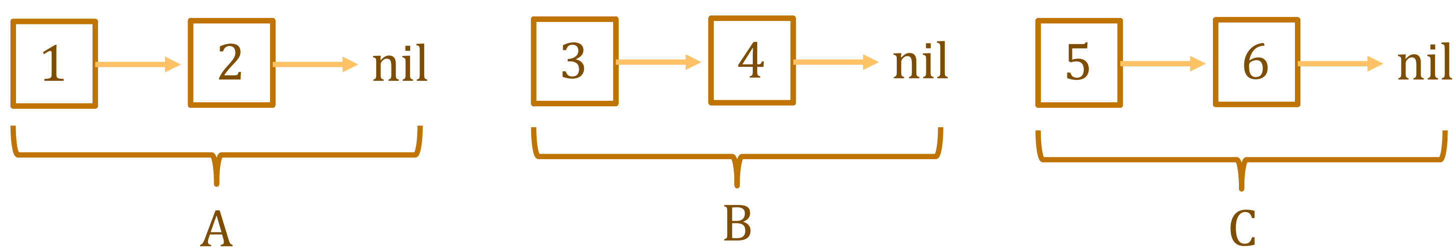
Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<number>;
readonly back: List<number>;

// @requires len(obj) > 0
// @returns (x, Q) with obj = concat(Q, cons(x, nil))
dequeue = (): [number, NumberQueue] => {
  return [this.back.hd,
          new ListPairQueue(this.front, this.back.tl)];
}
```

– **will need one other fact** ("associativity of concat")

concat(A, concat(B, C)) = concat(concat(A, B), C)   for any A, B, C : List



Implementing a Queue with Two Lists

```
// @requires len(obj) > 0
// @returns (x, Q) with obj = concat(Q, cons(x, nil))
dequeue = (): [number, NumberQueue] => {
  return [this.back.hd,
          new ListPairQueue(this.front, this.back.tl)];
}
```

– this.back = cons(x, L) for some x : ℝ and some L : List

obj =

Implementing a Queue with Two Lists

```
// @requires len(obj) > 0
// @returns (x, Q) with obj = concat(Q, cons(x, nil))
dequeue = (): [number, NumberQueue] => {
  return [this.back.hd,
          new ListPairQueue(this.front, this.back.tl)];
}
```

– this.back = cons(x, L) for some x : ℝ and some L : List

obj	= concat(this.front, rev(this.back))	**by AF**
	= concat(this.front, rev(cons(x, L)))	**since** back = …
	= concat(this.front, concat(rev(L), cons(x, nil)))	**def of** rev
	= concat(concat(this.front, rev(L)), cons(x, nil))	**assoc of** concat

x = this.back.hd and L = this.back.tl

Q = concat(this.front, rev(L))

= concat(this.front, rev(this.back.tl)) = result of constructor call

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
// RI: if this.back = nil, then this.front = nil
readonly front: List<number>;
readonly back: List<number>;

// makes obj = concat(front, rev(back))
constructor(front: List<number>, back: List<number>) {
  if (back === nil) {
    this.front = nil;
    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 = concat(this.front, rev(this.back))
// RI: if this.back = nil, then this.front = nil
readonly front: List<number>;
readonly back: List<number>;

// makes obj = concat(front, rev(back))
constructor(front: List<number>, back: List<number>) {
  if (back === nil) {
    this.front = nil;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

obj = concat(nil, rev(rev(front))) ??

obj = concat(front, rev(back))

- Need to check this creates correct abstract state

Implementing a Queue with Two Lists

```
// AF: obj = concat(this.front, rev(this.back))
// RI: if this.back = nil, then this.front = nil
readonly front: List<number>;
readonly back: List<number>;

constructor(front: List<number>, back: List<number>) {
  if (back === nil) {
    this.front = nil;
    this.back = rev(front);
  } else {
    …
  }
}
```

obj	$= \text{concat}(\text{nil}, \text{rev}(\text{rev}(\text{front})))$	**AF**
	$= \text{concat}(\text{nil}, \text{front})$	**because I said so**
	$= \text{front}$	**def of** concat
	$= \text{concat}(\text{front}, \text{nil})$	**Lemma 2**
	$= \text{concat}(\text{front}, \text{rev}(\text{nil}))$	**def of** rev
	$= \text{concat}(\text{front}, \text{rev}(\text{back}))$	**since** back = nil