CSE 331

Abstraction Functions & Invariants

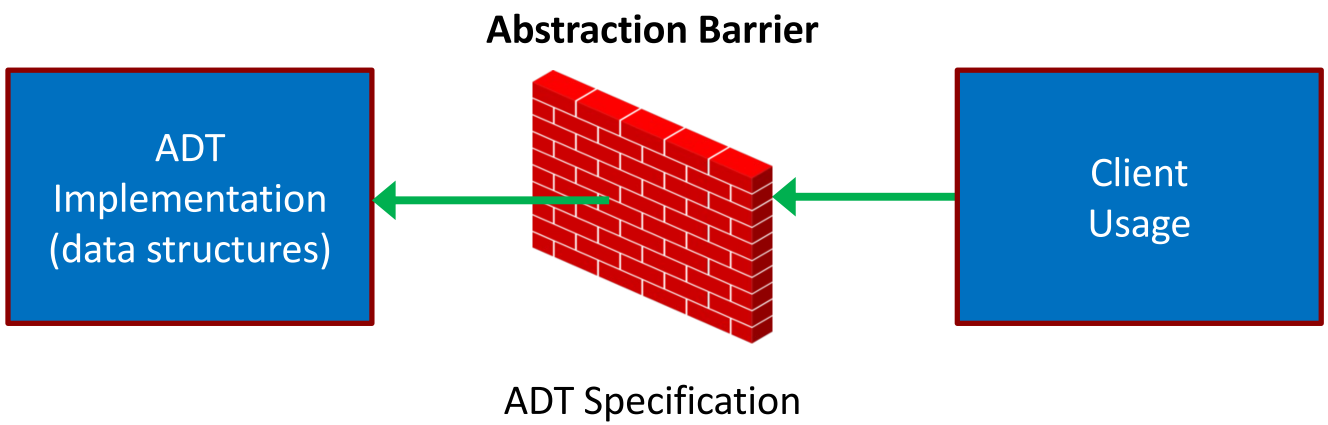
Kevin Zatloukal

Administrivia

- Bring your laptop to section tomorrow
  - we'll be doing some coding
- Section will be useful for next HW (as always)
  - practice refactoring existing code into an ADT
  - proofs about trees

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

- 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 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 FastListImpl 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**

    "this" is the record, which has 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 will always be true

defines what field values are allowed

Documenting the FastList ADT

```
class FastListImpl 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 FastListImpl 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 FastListImpl 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 obj = this.list = L

Correctness of getLast

```
class FastListImpl 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
  - free to use both RI and AF

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)

$\text{last}(\text{obj}) = x$ **by HW3 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;

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

- **Concrete state = abstract state, so easy correctness**

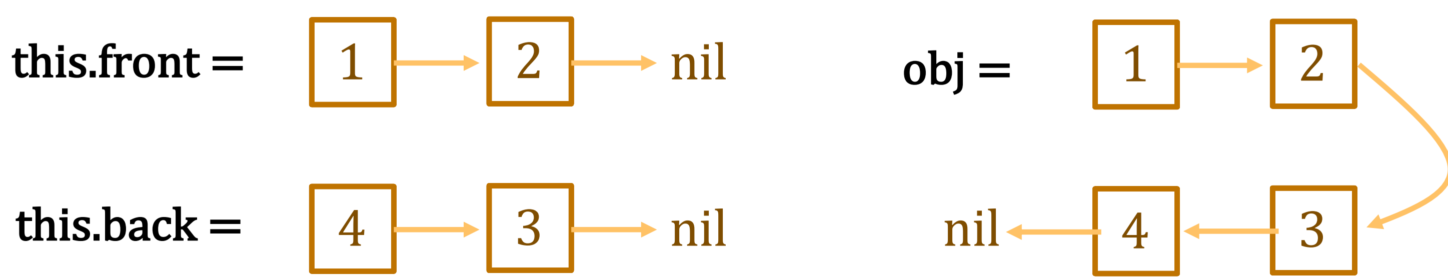
$\text{len}(\text{this.items}) = \text{len}(\text{obj})$ **by AF**

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;
  readonly back: List;
```

- 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;
  readonly back: List;
```

- 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;
  readonly back: List;
```

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

obj	= concat(nil, rev(nil))	**by AF**
	= rev(nil)	**def of** concat
	= 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;
  readonly back: List;

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

- **Will implement this later…**

Implementing a Queue with Two Lists

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

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

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

spec's return matches actual return

Implementing a Queue with Two Lists

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

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

- abstract state returned is...

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

spec's return matches actual return

Implementing a Queue with Two Lists

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

// @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 $L : \text{List}$

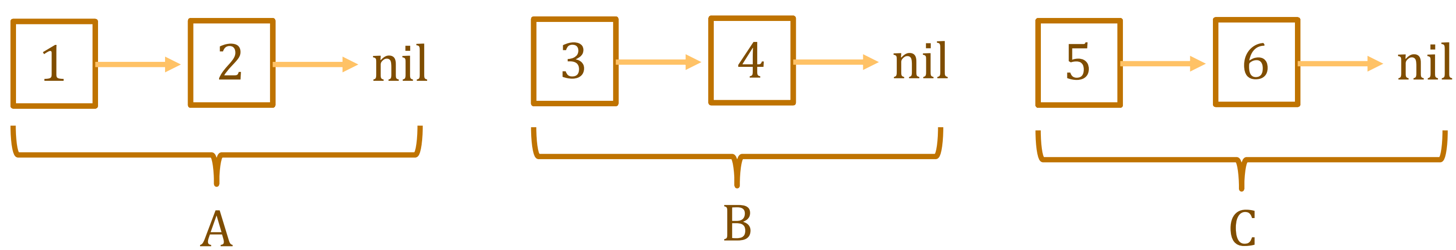
Implementing a Queue with Two Lists

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

// @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 $x : \mathbb{R}$ 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

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

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;
readonly back: List;

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

- need to check that RI holds at end of constructor
  - holds for else branch since $\text{this.back} \neq \text{nil}$

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;
readonly back: List;

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

- **holds for then branch (since** this.front = nil**)**
- **same 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;
readonly back: List;

constructor(front: NumberQueue, back: NumberQueue) {
  if (back === nil) {
    this.front = nil;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

concat(front, rev(nil))	= concat(front, nil)	**def of** rev
	= front	**Lemma 2**
	= rev(rev(front))	**because I said so**
	= concat(nil, rev(rev(front)))	**def of** concat

Set of Numbers

Set

- Recall: a set is a collection of objects
  - supported operations are "∈" and union etc.
  - we will think about them as lists (an inductive type)

```
// A list of numbers with no duplicates.
interface NumberSet {

  // @returns contains(x, obj)
  has(x: number): boolean;

  // @returns obj           if contains(x, obj)
  //          cons(x, obj)  if not contains(x, obj)
  add(x: number): NumberSet;
}
```

observer (has)

producer (add)

Set

- Recall: a set is a collection of objects
  - supported operations are “∈” and union etc.
  - we will think about them as lists (an inductive type)

```
// A list of numbers with no duplicates.
interface NumberSet { .. }

// @returns nil
function makeEmptySet(): NumberSet { .. }
```

  - an empty list contains no elements, so that is the empty set
  - use the “`add`” method to add more elements

Binary Trees

- The abstract state of our set is a list
- We can implement them however we want
  – let's use a tree!

type Tree := empty | node(x : ℤ, L : Tree, R : Tree)

func values(empty) := nil
values(node(x, L, R)) := concat(values(L), cons(x, values(R)))
for any x : ℤ and any L, R : Tree

Binary Trees

type Tree := empty | node(x : $\mathbb{Z}$, L : Tree, R : Tree)

func values(empty) := nil
values(node(x, L, R)) := concat(values(L), cons(x, values(R))) for any x : $\mathbb{Z}$ and any L, R : Tree

```
class TreeNumberSet implements NumberSet {
  // AF: obj = values(this.root)
  // RI: values(this.root) has no duplicates
  readonly root: Tree;

  // @requires values(root) has no duplicates
  // makes obj = values(root)
  constructor(root: Tree) {
    this.root = root;
  }
  …
```

Binary Trees

type Tree := empty | node(x : ℤ, L : Tree, R : Tree)

func values(empty) := nil
values(node(x, L, R)) := concat(values(L), cons(x, values(R)))
for any x : ℤ and any L, R : Tree

```
// AF: obj = values(this.root)
// RI: values(this.root) has no duplicates

// @returns contains(x, obj)        [from NumberSet]
has(x: number): boolean {
  return contains(x, values(this.root));
}
```

contains(x, values(this.root)) = contains(x, obj) **by AF**

Binary Trees

```
// AF: obj = values(this.root)
// RI: values(this.root) has no duplicates

// @returns obj           if contains(x, obj)
//          cons(x, obj) if not contains(x, obj)
add(x: number): NumberSet {
  if (contains(x, values(this.root)) {
    return this;
  } else {
    return new TreeNumberSet(node(x, empty, this.root));
  }
}
```

At the first "return", since contains(x, obj) **is true, we should return** obj**, which we do.**

contains(x, values(this.root)) = contains(x, obj) **by AF**

Binary Trees

```
// AF: obj = values(this.root)
// RI: values(this.root) has no duplicates

// @returns obj            if contains(x, obj)
//          cons(x, obj) if not contains(x, obj)
add(x: number): NumberSet {
  if (contains(x, values(this.root)) {
    return this;
  } else {
    return new TreeNumberSet(node(x, empty, this.root));
  }
}
```

At second "return", since contains(x, obj) **is false, should return** cons(x, obj).

We return an object with abstract state values(node(x, empty, this.root))

Binary Trees

func values(empty) := nil
values(node(x, L, R)) := concat(values(L), cons(x, values(R)))
for any x : ℤ and any L, R : Tree

```
// AF: obj = values(this.root)
```

values(node(x, empty, this.root))
=

= cons(x, obj)

Binary Trees

func values(empty) := nil
values(node(x, L, R)) := concat(values(L), cons(x, values(R)))
for any x : $\mathbb{Z}$ and any L, R : Tree

```
// AF: obj = values(this.root)
```

---

values(node(x, empty, this.root))
= concat(values(empty), cons(x, values(this.root))) **def of** values
= concat(nil, cons(x, values(this.root))) **def of** values
= cons(x, values(this.root)) **def of** concat
= cons(x, obj) **by AF**

Binary Trees

```
// AF: obj = values(this.root)
// RI: values(this.root) has no duplicates

// @returns obj           if contains(x, obj)
//          cons(x, obj) if not contains(x, obj)
add(x: number): NumberSet {
  if (contains(x, values(this.root)) {
    return this;
  } else {
    return new TreeNumberSet(node(x, empty, this.root));
  }
}
```

What does the tree we're building look like?

It's just a list! We'll do this properly in HW4.

Polynomials

Polynomials

- Sum of monomials (e.g., ax^n)
  - familiar mathematical object

producer

observer

```
interface Poly {
  // @returns obj + P
  add(P: Poly): Poly;
  // @returns max-exp(obj), where max-exp is
  //     max-exp(ax^n) := n
  //     max-exp(ax^n + P) := max(n, max-exp(P))
  degree(): number;
}
```

$0 = 0x^0$ has degree 0

Implementing Polynomial with a List

```
// Stores a list of monomials ordered by decreasing degree
class DegreeSortedPoly implements Poly {

  // AF: obj = poly(this.terms), where poly is
  //        poly(nil) := 0
  //        poly(cons((a, n), L)) := ax^n + poly(L)
  // RI: terms in decreasing order by degree
  readonly terms: List<[number, number]>;
```

- Terms is a list of pairs
  - the pair (a, n) represents the monomial ax^n

    the n part is the degree of the monomial

    degree of the polynomial is the maximum of these

Implementing Polynomial with a List

```
// AF: obj = poly(this.terms), where
//        poly(nil) := 0    and …
readonly terms: List<[number, number]>;

// @returns max-exp(obj), where
//     max-exp(ax^n) := n   and …
degree(): number {
  if (this.terms === nil) {
    return 0;
  } else {
    return this.terms.hd[1];
  }
}
```

max-exp(obj)	= max-exp(poly(this.terms))	**by AF**
	= max-exp(poly(nil))	**since** this.terms = nil
	= max-exp(0)	**def of** poly
	= 0	**def of** max-exp

Implementing Polynomial with a List

```
// AF: obj = poly(this.terms), where
//        poly(cons((a, n), L)) := ax^n + poly(L)
readonly terms: List<[number, number]>;

// @returns max-exp(obj), where
//     max-exp(ax^n + P) := max(n, max-exp(P))
degree(): number {
  if (this.terms === nil) {
    return 0;
  } else {
    return this.terms.hd[1];
  }
}
```

max-exp(obj)	= max-exp(poly(this.terms))	**by AF**
	= max-exp(poly(cons((a, n), L)))	**since** terms = cons(..)
	= max-exp(ax^n + poly(L))	**def of** poly
	= max(n, max-exp(poly(L)))	**def of** max-exp
	= n	??

Helper Lemma

func $\text{poly}(\text{nil}) := 0$

$\text{poly}(\text{cons}((a, n), L)) := ax^n + \text{poly}(L)$

func $\text{max-exp}(ax^n) := n$

$\text{max-exp}(ax^n + P) := \max(n, \text{max-exp}(P))$

- **Prove that** $\text{max-exp}(\text{poly}(\text{cons}((a, n), S))) = n$ **for any** $S : \text{List}$

Base Case (nil):

$\text{max-exp}(\text{poly}(\text{cons}(a, n), \text{nil}))$

$= \text{max-exp}(ax^n + \text{poly}(\text{nil}))$ **def of** poly

$= \text{max-exp}(ax^n + 0)$ **def of** max-exp

$= n$

Helper Lemma

func poly(nil) := 0
poly(cons((a, n), L)) := ax^n + poly(L)

func max-exp(ax^n) := n
max-exp(ax^n + P) := max(n, max-exp(P))

- **Prove that** max-exp(poly(cons((a, n), S))) = n **for any** S : List

Inductive Hypothesis: assume that max-exp(poly(cons((a, n), L))) = n

Inductive Step (cons((b, m), L))**:**

max-exp(poly(cons((a, n), cons((b, m), L)))
= max-exp(ax^n + poly(cons((b, m), L))) — **def of** poly
= max(n, max-exp(poly(cons((b, m), L)))) — **def of** max-exp
= max(n, m) — **Ind. Hyp.**
= n — **since** $n > m$ **(RI)**