



CSE 331

Procedural Abstraction

James Wilcox & Kevin Zatloukal

Procedural Abstraction

Reasoning about Function Calls

`func f(n) := 2n + 1` for any $n : \mathbb{N}$

- **When reasoning, we can replace $f(..)$ by its definition**

$$\begin{aligned} 2 f(10) &= 2 (2 \cdot 10 + 1) && \text{def of } f \\ &= 2 (21) \\ &= 42 \end{aligned}$$

Reasoning about Function Calls

- This becomes trickier with *side conditions*

<code>func f(x) := 2x + 1</code>	<code>if x ≥ 0</code>	<code>for any x : ℤ</code>
<code>f(x) := 0</code>	<code>if x < 0</code>	<code>for any x : ℤ</code>

- Need to explain why that line holds
 - suppose we know that $n \geq 10$

$2 f(n - 10)$	$= 2 (2 \cdot (n - 10) + 1)$	<code>def of f (since $n - 10 \geq 0$)</code>
	$= 4n - 38$	

- This issue does not arise with pattern matching
 - easy to see visually which line applies

Concrete vs Abstract

- In math, every definition is spelled out (“*concrete*”)

`func f(n) := 2n + 1` for any $n : \mathbb{N}$

- we know exactly what $f(n)$ is for any non-negative n

- In code, details are often hidden (“*abstracted away*”)

- gives us room to **change** the details later

```
// n must be natural. Returns some natural number.  
const f = (n: bigint): bigint => { .. };
```

- this hides what is returned
- can also hide how it is calculated...

Concrete vs Abstract

- In code, details are often hidden (“*abstracted away*”)
 - gives us room to change the details later
 - hides complication

```
// Returns the same numbers but in reverse order, i.e.
//   rev(nil) := nil
//   rev(cons(x, L)) := concat(rev(L), cons(x, nil))
const rev = (L: List): List => {
  return rev_acc(L, nil); // faster way
};
```

not straight from the spec

- straight from spec = “if ... return concat(rev(L), cons(x, nil))”
- since the answer is the same, clients don't need to know!

Procedural Abstraction

- **Hide the details of the function from the caller**
 - caller only needs to read the **specification**
 - (“procedure” means function)
- **Caller promises to pass valid inputs**
 - no promises on invalid inputs
- **Implementer then promises to return correct outputs**
 - does not matter how

Other Properties of High-Quality Code

- Professionals are expected to write **high-quality** code
- Correctness is the most important part of quality
 - users **hate** products that do not work properly
- Also includes the following
 - easy to change
 - easy to understand
 - modular

Writing Good Specifications

- TypeScript, like Java, writes specs in `/** ... */`

```
/**  
 * High level description of what function does  
 * @param a What "a" represents + any conditions  
 * @param b What "b" represents + any conditions  
 * @returns Detailed description of return value  
 */  
const f = (a: bigint, b: bigint): bigint => {..};
```

- these are formatted as “JSDoc” comments
- (in Java, they are JavaDoc comments)

Writing Good Specifications

- Descriptions can be English or formal

```
/**
 * Returns the same list but in reverse order
 * @param L The list in question
 * @returns rev(L), where rev is defined by
 *   rev(nil) := nil
 *   rev(cons(x, L)) := concat(rev(L), cons(x, nil))
 */
const rev = (L: List): List => {
  return rev_acc(L, nil); // faster
};
```

- English descriptions are typical for most code
professionals are *extremely* good at formalizing themselves

Writing Good Specifications

- Can place conditions on parameters

```
/**
 * Returns the last element in the list
 * @param L A list, which must be non-nil
 * @returns last(L), where last is defined by
 *   last(cons(x, nil)) := x
 *   last(cons(x, cons(y, L))) := last(cons(y, L))
 */
const last = (L: List): bigint => {..};
```

- clients **should not** pass in empty lists
- but they will!

Writing Good Specifications

- Can place conditions on parameters

```
/**
 * Returns the last element in the list
 * @param L A list, which must be non-nil
 * @returns last(L), where last is defined by
 *   last(cons(x, nil)) := x
 *   last(cons(x, cons(y, L))) := last(cons(y, L))
 */
const last = (L: List): bigint => {
  if (L.kind === "nil")
    throw new Error("Bad client! Bad!")
  ...
}
```

- practice **defensive programming**

Writing Good Specifications

- Can include promises to throw exceptions

```
/**
 * Returns the last element in the list
 * @param L The list in question
 * @throws Error if L is nil
 * @returns last(L), where last is defined by
 *   last(cons(x, nil)) := x
 *   last(cons(x, cons(y, L))) := last(cons(y, L))
 */
const last = (L: List): bigint => {
  if (L.kind === "nil") ...
```

- code is the same, but the spec is different

changed what behavior we **promise** (now have less freedom to change it)

Writing Good Specifications

- Can place conditions on multiple parameters

```
/**
 * Returns the first n elements from the list L
 * @param n non-negative length of the prefix
 * @param L the list whose prefix should be returned
 * @requires n <= len(L)
 * @returns prefix(n, L), where prefix is...
 */
const prefix = (n: bigint, L: List): List => {..};
```

- restrictions on one parameter can go in its @param
 - restrictions involving multiple should go in @requires
- @requires is also fine in the first case though

Writing Good Specifications

- Can include promises to throw exceptions

```
/**  
 * Returns the first n elements from the list L  
 * @param n non-negative length of the prefix  
 * @param L the list whose prefix should be returned  
 * @throws Error if n > len(L)  
 * @returns prefix(n, L), where prefix is..  
 */  
const prefix = (n: bigint, L: List): List => {..};
```

- this is also reasonable
- I prefer the @requires: promises less to the client
gives us more freedom to change it later...
might want to actually return a list in that case!

Benefits of Specifications

Clear specifications help with understandability and

- **Correctness**
 - reasoning requires clear definition of what the function does
- **Changeability**
 - implementer is free to write any code that meets spec
 - client can pass any inputs that satisfy requirements
- **Modularity**
 - people can work on different parts once specs are agreed

Benefits of Specifications

Clear specifications help with understandability and

- **Correctness**
- **Changeability**
- **Modularity**
 - **knowledge about code details tends to “leak”**
easy to do when you know how the other function works
 - **creates interdependence, trends toward “spaghetti code”**
if those details change, it could break the client
 - **requires constant work to prevent this**
may be impossible with enough clients



XKCD
1172

LATEST: 10.17

UPDATE

CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS
WHEN YOU HOLD DOWN SPACEBAR.

COMMENTS:

LONGTIMEUSER4 WRITES:

THIS UPDATE BROKE MY WORKFLOW!
MY CONTROL KEY IS HARD TO REACH,
SO I HOLD SPACEBAR INSTEAD, AND I
CONFIGURED EMACS TO INTERPRET A
RAPID TEMPERATURE RISE AS "CONTROL".

ADMIN WRITES:

THAT'S HORRIFYING.

LONGTIMEUSER4 WRITES:

LOOK, MY SETUP WORKS FOR ME.
JUST ADD AN OPTION TO REENABLE
SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Weaker vs Stronger Specifications

- **Since specs are written by us, they can have bugs!**
 - in those cases, it is necessary to change them
- **Useful terminology for comparing specs for a function**
 - spec A can be stronger or weaker than spec B (or neither)

Strengthening cannot break the clients

stronger spec accepts the original inputs (or more inputs)

stronger spec makes the original promises about outputs (or more)

Weakening cannot break the implementation

weaker spec does not allow new inputs

weaker spec does not add more promises about outputs

Weaker vs Stronger Specifications

- To be more formal, we need some terminology

Precondition:

conditions included in `@param` and `@requires`

Postcondition:

conditions included in `@return` (and `@throws`)

Correctness (satisfying the spec):

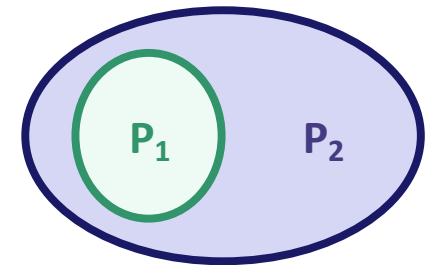
for every input satisfying the precondition,
the output will satisfy the postcondition

Weaker vs Stronger Specifications

- **Definition:** specification S_2 is stronger than S_1 iff
 - precondition of S_2 is easier to satisfy than that of S_1
 - postcondition of S_2 is harder to satisfy than that of S_1
(on all inputs allowed by both)

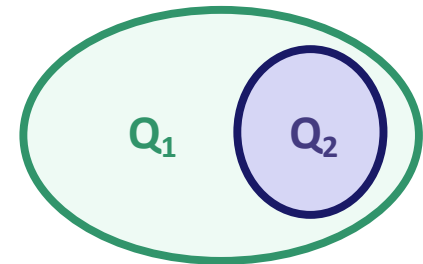
- **A stronger specification:**

- gives more guarantees to the client



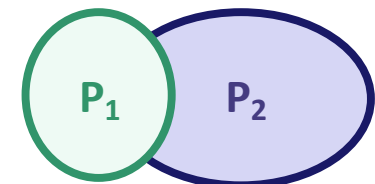
- **A weaker specification:**

- gives more freedom to the implementer



- **An incomparable specification:**

- some strengthening, some weakening



Weaker vs Stronger Specifications

- **Since specs are written by us, they can have bugs!**
 - in those cases, it is necessary to change them
- **Useful terminology for comparing specs for a function**
 - spec A can be stronger or weaker than spec B (or neither)

Category	Stronger	Weaker
<code>@param</code> <code>@requires</code>	same or more allowed inputs	same or fewer allowed inputs
<code>@return</code> <code>@throws</code>	same or more promised facts	same or fewer promised facts

(some others, but these are the main ones)

Example 1: Weaker vs Stronger

```
// Find the index of x in the list
const indexOf = (x: bigint, L: list): bigint => {..}
```

Which is stronger?

Specification A

- requires that L contains the value x
- returns an index where x occurs in L

Specification B

- requires L contains the value x
- returns the *first* index where x occurs in L

B is stronger

Example 2: Weaker vs Stronger

```
// Find the index of x in the list
const indexOf = (x: bigint, L: list): bigint => {..}
```

Which is stronger?

Specification A

- requires that L contains the value x
- returns an index where x occurs in L

Specification C

- returns an index where x occurs in L or -1 if x is not in L

C is stronger

Example 3: Weaker vs Stronger

```
// Find the index of x in the list
const indexOf = (x: bigint, L: list): bigint => {..}
```

Which is stronger?

Specification B

- requires L contains the value x
- returns the *first* index where x occurs in L

incomparable

Specification C

- returns an index where x occurs in L or -1 if x is not in L

Incomparable Specifications

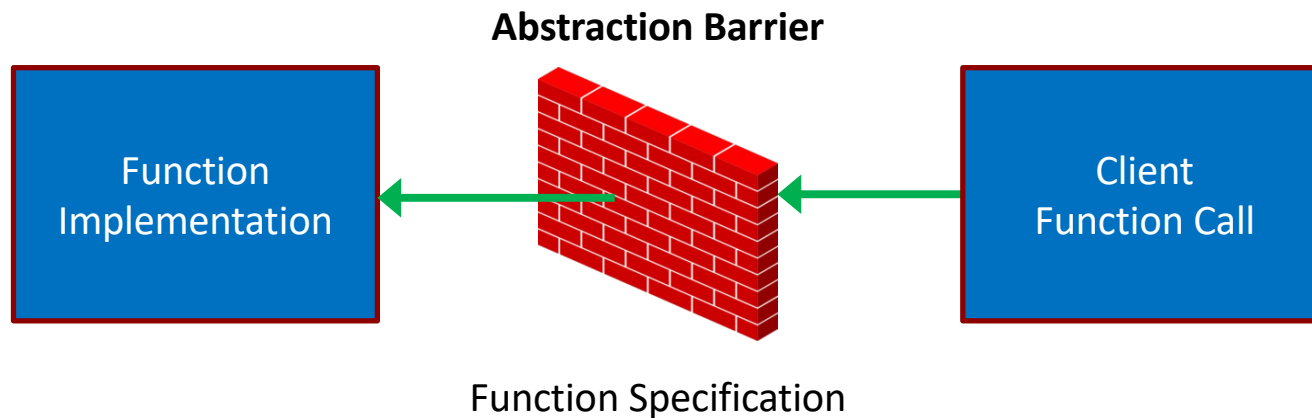
- **Not all specs are weaker or stronger**
 - most specs are “incomparable”
- **Common ways to be incomparable**
 - **weaker in some ways but stronger in others**
 - one param is strengthened (fewer inputs) but return is weakened
 - **describes different behavior**
 - one spec says to return “ $x + 1$ ” and the other says to return “ $x + 2$ ”
 - **special case: one throws and other returns on the same input**
 - throw and return are different behaviors

Which is Better?

- **Stronger does not always mean better!**
- **Weaker does not always mean better!**
- **Strength of specification trades off:**
 - usefulness to client
 - ease of simple, efficient, correct implementation
 - promotion of reuse and modularity
 - clarity of specification itself
- **“It depends”**

Abstraction Barrier

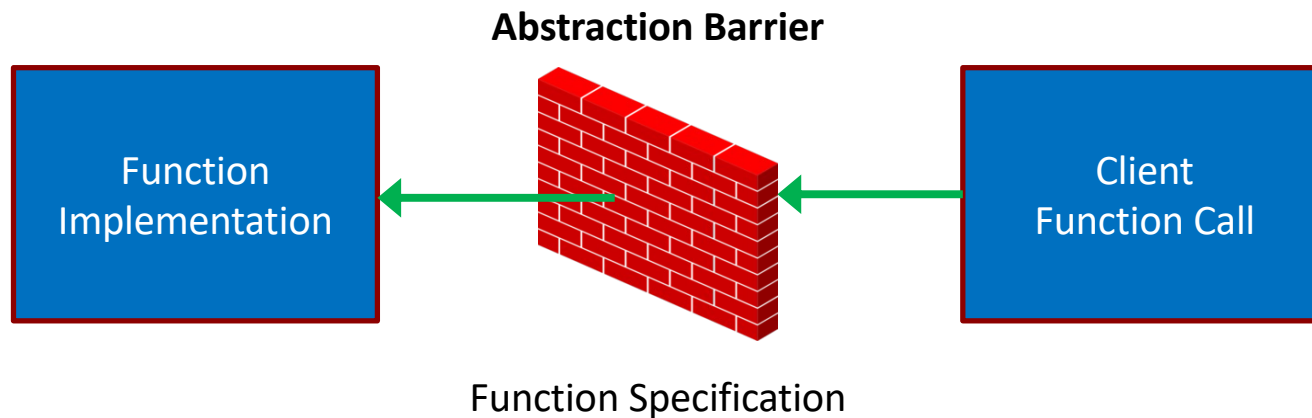
- Last time, we saw *procedural* abstraction



- specification is the “barrier” between the sides
- clients depend only on the spec
- implementer can write any code that satisfies the spec

Abstraction Barrier

- Last time, we saw *procedural* abstraction



- Specifications improve
 - understandability (client)
 - changeability (implementation)
 - modularity

correctness is impossible
without specifications

Performance Improvements

- Last time, we saw rev-acc, which is faster than rev
 - faster *algorithm* for reversing a list
 - rare to see this
- Most perf improvements change ***data structures***
 - different kind of abstraction barrier for data
- Let's see an example...

Last Element of a List

```
func last(nil)           := undefined
  last(cons(x, nil))    := x           for any x : ℤ
  last(cons(x, cons(y, L)) := last(cons(y, L)) for any x, y : ℤ and
                                          any L : List
```

- **Runs in $\Theta(n)$ time**
 - walks down to the end of the list
 - no faster way to do this on a list
- **We could cache the last element**
 - new data type just dropped:

```
type FastLastList = {list: List, last: bigint|undefined}
```

empty list has undefined last

Fast-Last List

```
type FastLastList = {list: List, last: bigint|undefined}
```

- **How do we switch to this type?**
 - change every `List` into `FastLastList`
- **Will still have functions that operate on List**
 - e.g., `len`, `sum`, `concat`, `rev`
- **Suppose `F` is a `FastLastList`**
 - instead of calling `rev(F)`, we have call `rev(F.list)`
 - cleaner to introduce a helper function

Fast-Last List

```
type FastLastList = {list: List, last: bigint|undefined}

const getLast = (F: FastLastList): bigint|undefined => {
  return F.last;
};

const toList = (F: FastLastList): List<bigint> => {
  return F.list;
};
```

- **How do we switch to this type?**
 - change every `List` into `FastLastList`
 - replace `F` with `toList(F)` where a `List` is expected
- **What happens if we need to change it again?**
 - do it all over again!

Another Fast List

- Suppose we often need the 2nd to last, 3rd to last, ... (back of the list). How can we make it faster?
 - store the list in *reverse* order!

```
type FastBackList = List<bigint>;
```

```
const getLast = (F: FastBackList): bigint|undefined => {  
  return (F.kind === "nil") ? undefined : F.hd;  
};
```

```
const getSecondToLast = (F: FastBackList): bigint|undefined => {  
  return (F.kind === "nil") ? undefined :  
    (F.tl.kind === "nil") ? undefined : F.tl.hd;  
};
```

```
const toList = (F: FastBackList): List<bigint> => {  
  return rev(F);  
};
```

Another Fast List

```
type FastBackList = List<bigint>;

const getLast = (F: FastBackList): bigint|undefined => {
  return (F.kind === "nil") ? undefined : F.hd;
};

const toList = (F: FastBackList): List<bigint> => {
  return rev(F);
};
```

- **Problems with this solution...**
 - **no type errors if someone forgets to call toList!**

```
const F: FastBackList = ...;
return concat(F, cons(1, nil)); // bad!
```

Another Fast List – Take Two

```
type FastBackList = {list: List<bigint>};

const getLast = (F: FastBackList): bigint|undefined => {
  return (F.list.kind === "nil") ? undefined : F.list.hd;
};

const toList = (F: FastBackList): List<bigint> => {
  return rev(F.list);
};
```

- **Still some problems...**
 - **no type errors if someone grabs the field directly**

```
const F: FastBackList = ...;
return concat(F.list, cons(1, nil)); // bad!
```

Another Fast List — Take Three

```
const F: FastBackList = ...;  
return concat(F.list, cons(1, nil)); // bad!
```

- **Only way to completely stop this is to hide `F.list`**
 - do not give them the data, just the functions

```
type FastList = {  
  getLast: () => bigint|undefined,  
  toList: () => List<bigint>  
};
```

- the only way to get the list is to call `F.toList()`
- seems weird... but we can make it look familiar

Another Fast List — Take Three

```
interface FastList {  
  getLast: () => bigint | undefined;  
  toList: () => List<bigint>;  
}
```

- In TypeScript, “interface” is synonym for “record type”
- You’ve seen this in Java

Java interface is a record where
field values are functions (methods)

```
interface FastList {  
  int getLast() throws EmptyList;  
  List<Integer> toList();  
}
```

- in 331, our interfaces will only include functions (methods)

Data Abstraction

Data Abstraction

- **Give clients only operations, not data**
 - operations are “public”, data is “private”
- **We call this an Abstract Data Type (ADT)**
 - invented by Barbara Liskov in the 1970s
 - fundamental concept in computer science
 - built into Java, JavaScript, etc.
 - data abstraction via procedural abstraction
- **Critical for the properties we want**
 - easier to change data structure
 - easier to understand (hides details)
 - more modular



How to Make a FastList — Attempt One

```
const makeFastList = (list: List<bigint>): FastList => {  
  const last = last(list);  
  return {  
    getLast: () => { return last; },  
    toList: () => { return list; }  
  };  
};
```

- **Values in `getLast` and `toList` fields are functions**
- **There is a cleaner way to do this**
 - will also look more familiar

How to Make a FastList

```
class FastLastList implements FastList {
  last: bigint|undefined; // should be "readonly"
  list: List<bigint>;

  constructor(list: List<bigint>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

- Can create a new record using **“new”**
 - each record has fields `list`, `last`, `getLast`, `toList`
 - bodies of functions use **“this”** to refer to the record

How to Make a FastList

```
class FastLastList implements FastList {
  last: bigint|undefined; // should be "readonly"
  list: List<bigint>;

  constructor(list: List<bigint>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

- Can create a new record using **“new”**
 - all four assignments are executed on each call to **“new”**
 - `getLast` and `toList` are always the same functions

How to Make a FastList

```
class FastLastList implements FastList {
  last: bigint|undefined; // should be "readonly"
  list: List<bigint>;

  constructor(list: List<bigint>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

- **Implements the FastList interface**
 - i.e., it has the expected `getLast` and `toList` fields
 - (okay for records to have more fields than required)

Another Way to Make a FastList

```
class FastBackList implements FastList {
  original: List<bigint>;
  reversed: List<bigint>; // in reverse order

  constructor(list: List<bigint>) {
    this.original = list;
    this.reversed = rev(list);
  }

  getLast = () => {
    return (this.reversed.kind === "nil") ?
      undefined : this.reversed.hd;
  };

  toList = () => { return this.original; }
}
```

How Do Clients Get a FastList

```
const makeFastList = (list: List<bigint>): FastList => {  
  return new FastLastList(list);  
};
```

- **Export only FastList and makeFastList**
 - completely hides the data representation from clients
- **This is called a “factory function”**
 - another **design pattern**
 - can change implementations easily in the future
becomes FastBackList with a one-line change
- **Difficult to add to the list with this interface**
 - requires three calls: toList, cons, makeFastList

Another Way To Do It

```
interface FastList {
  cons: (x: bigint) => FastList;
  getLast: () => bigint | undefined;
  toList: () => List<bigint>;
};

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

- **New method `cons` returns list with `x` in front**
 - example of a “producer” method (others are “observers”)
produces a new list for you
 - now, we only need to make an empty `FastList`
anything else can be built via `cons`

Another Way To Do It (Even Better)

```
interface FastList {
  cons: (x: bigint) => FastList;
  getLast: () => bigint|undefined;
  toList: () => List<bigint>;
};

const nilList: FastList = new FastBackList(nil);

const makeFastList = (): FastList => {
  return nilList;
};
```

- No need to create a new object using “**new**” *every time*
 - can reuse the same instance
 - only possible since these are immutable!
 - example of the “singleton” **design pattern**

Full ADT Design Pattern for 331

We will use the following **design pattern** for ADTs:

- “**interface**” used for defining ADTs
 - declares the methods available
- “**class**” used for implementing ADTs
 - defines the fields and methods
 - implements the ADT interface above
- **Factory function** used to create instances

Stick to regular functions for rest of the code!

Specifications for ADTs

Specifications for ADTs

- Run into problems when we try to write specs
 - for example, what goes after `@return`?
 - don't want to say returns the `.list` field (or reverse of that)
 - we want to hide those details from clients

```
interface FastList {  
    /**  
     * Returns the "underlying" list of items  
     * @return ??  
     */  
    toList: () => List<bigint>;  
};
```

- Need some terminology to clear up confusion

ADT Terminology

New terminology for specifying ADTs

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `{list: List, last: bigint|undefined}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- **We've had different abstract and concrete types all along!**
 - in our math, List is an inductive type (abstract)
 - in our code, List is a string or a record (concrete)

List Is Like an ADT

Inductive types also differ in abstract / concrete states:

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `{kind:"nil"} | {kind:"cons", hd:bigint, tl:List}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- Inductive types also use a **design pattern** to work in TypeScript
 - details are different than ADTs (e.g., no interfaces)

ADT Terminology

New terminology for specifying ADTs

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `{kind:"nil"} | {kind:"cons", hd:bigint, tl:List}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- Term “**object**” (or “**obj**”) will refer to abstract state
 - “object” means mathematical object
 - “obj” is the mathematical value that the record represents

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  /**
   * Returns the last element of the list (O(1) time).
   * @returns last(obj)
   */
  getLast: () => bigint | undefined;
}
```

- “obj” refers to the abstract state (the list, in this case)
 - actual state will be a record with fields `last` and `list`

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * Returns the object as a regular list of items.
   * @returns obj
   */
  toList: () => List<bigint>;
}
```

- In math, this function does nothing (“@returns obj”)– two *different* concrete representations of the same idea– details of the representations are *hidden* from clients

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * Returns a new list with x in front of this list.
   * @returns cons(x, obj)
   */
  cons = (x: bigint) => FastList;
```

- **Producer method: makes a new list for you**
 - “obj” above is a list, so `cons(x, obj)` makes sense in math

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * Returns a new list with x in front of this list.
   * @returns cons(x, obj)
   */
  cons = (x: bigint) => FastList;
```

- Specification does not talk about fields, just “obj”
 - fields are *hidden* from clients

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

Documenting the FastList ADT

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

- **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: bigint | undefined;  
    readonly list: List<bigint>;  
    ...  
}
```

- **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: bigint | undefined;  
  readonly list: List<bigint>;  
  
  constructor(L: List<bigint>) {  
    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: bigint | undefined;
  readonly list: List<bigint>;

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

Correctness of getLast

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

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

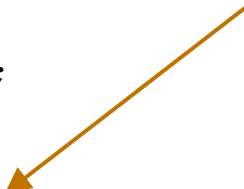
  // @requires len(obj) > 0
  // @returns (x, Q) with obj = concat(Q, cons(x, nil))
  dequeue: () => [bigint, NumberQueue];
}
```

observer

producer

producer

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<bigint>;  
}
```

- **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<bigint>;

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

- **Correctness of** `size`:

`len(this.items) = len(obj)` **by AF**

nothing is straight from the spec anymore

Implementing a Queue with a List

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

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

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

- **Correctness of enqueue:**

return value = cons(x, this.items)
 = cons(x, obj)

spec of constructor
AF

Implementing a Queue with a List

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

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

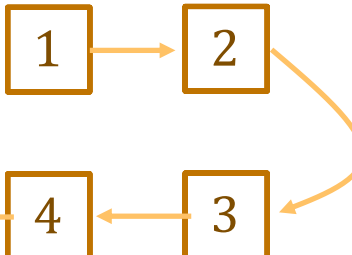
Implementing a Queue with Two Lists

```
// Implements a queue using two lists.  
class ListPairQueue implements NumberQueue {  
    // AF: obj = concat(this.front, rev(this.back))  
    readonly front: List<bigint>;  
    readonly back: List<bigint>;    // in reverse order
```

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

this.front = 

this.back = 

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<bigint>;  
    readonly back: List<bigint>;  
}
```

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

- **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<bigint>;
    readonly back: List<bigint>;

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

- Will implement this later...

Implementing a Queue with Two Lists

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

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

- **Correctness of size:**

len(obj) =

Implementing a Queue with Two Lists

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

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

- **Correctness of `size`:**

$$\begin{aligned} \text{len(obj)} &= \text{len}(\text{concat}(\text{this.front}, \text{rev}(\text{this.back}))) \\ &= \text{len}(\text{this.front}) + \text{len}(\text{rev}(\text{this.back})) \\ &= \text{len}(\text{this.front}) + \text{len}(\text{this.back}) \end{aligned}$$

by AF

by Example 3

by Example 4

Implementing a Queue with Two Lists

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

// @returns cons(x, obj)
enqueue = (x: bigint): 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<bigint>;
readonly back: List<bigint>;

// @returns cons(x, obj)
enqueue = (x: bigint): 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<bigint>;
readonly back: List<bigint>;

// @requires len(obj) > 0
// @returns (x, Q) with obj = concat(Q, cons(x, nil))
dequeue = (): [bigint, 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

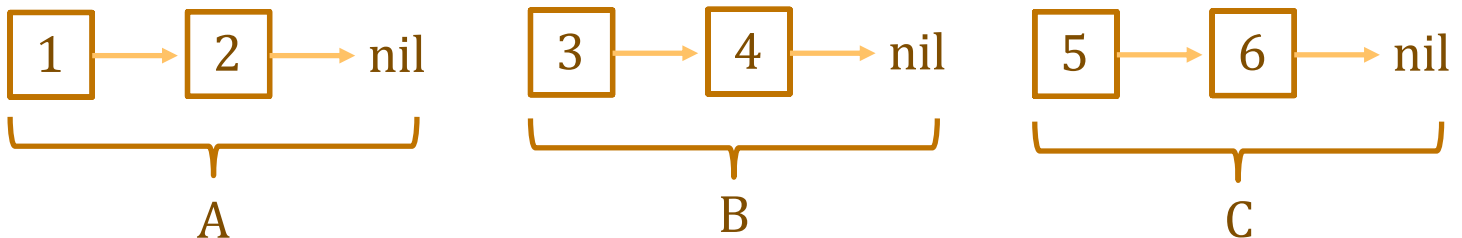
```
// AF: obj = concat(this.front, rev(this.back))
readonly front: List<bigint>;
readonly back: List<bigint>;

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

– will need one other fact

(“associativity of concat”)

$\text{concat}(A, \text{concat}(B, C)) = \text{concat}(\text{concat}(A, B), C)$ for any $A, B, C : \text{List}$



Implementing a Queue with Two Lists

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

– $\text{this.back} = \text{cons}(x, L)$ for some $x : \mathbb{R}$ and some $L : \text{List}$

obj =

Implementing a Queue with Two Lists

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

– $\text{this.back} = \text{cons}(x, L)$ for some $x : \mathbb{R}$ and some $L : \text{List}$

$\text{obj} = \text{concat}(\text{this.front}, \text{rev}(\text{this.back}))$	by AF
$= \text{concat}(\text{this.front}, \text{rev}(\text{cons}(x, L)))$	since back = ...
$= \text{concat}(\text{this.front}, \text{concat}(\text{rev}(L), \text{cons}(x, \text{nil})))$	def of rev
$= \text{concat}(\text{concat}(\text{this.front}, \text{rev}(L)), \text{cons}(x, \text{nil}))$	assoc of concat

$x = \text{this.back.hd}$ and $L = \text{this.back.tl}$

$Q = \text{concat}(\text{this.front}, \text{rev}(L))$

$= \text{concat}(\text{this.front}, \text{rev}(\text{this.back.tl})) = \text{result of constructor call}$

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<bigint>;
readonly back: List<bigint>;

// makes obj = concat(front, rev(back))
constructor(front: List<bigint>, back: List<bigint>) {
  if (back.kind === "nil") {
    this.front = nil;
    this.back = rev(front);           holds since this.front = nil
  } else {
    this.front = front;
    this.back = back;                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<bigint>;
readonly back: List<bigint>;

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

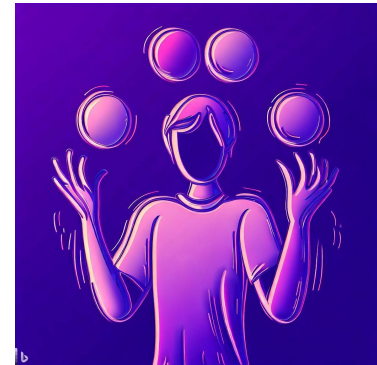
- Need to check this creates correct abstract state

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<bigint>;
readonly back: List<bigint>;

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

```
obj = concat(nil, rev(rev(front)))
    = concat(nil, front)
    = front
    = concat(front, nil)
    = concat(front, rev(nil))
    = concat(front, rev(back))
```



AF
because I said so
def of concat
Lemma 2
def of rev
since back = nil