CSE 331

Procedural Abstraction

Kevin Zatloukal

Administrivia

- HW3 released
  - start early and ask questions when you get stuck
  - remember that your code must pass our tests to get points

- Signup form for creation of a GitLab repo
  - useful to back up the work on your machine
  - repo only visible to you and the staff (as we require)

- Fixed up the Type Erasure slides from last week
  - revisit if you are interested

Structural Induction

Example 5: Reversing a List

```
func rev(nil)           :=  nil
     rev(cons(x, L))    := concat(rev(L), cons(x, nil))    for any x : ℤ and
                                                              any L : List
```

- This correctly reverses a list but is slow
  - concat takes $\Theta(n)$ time, where n is length of L
  - n calls to concat takes $\Theta(n^2)$ time

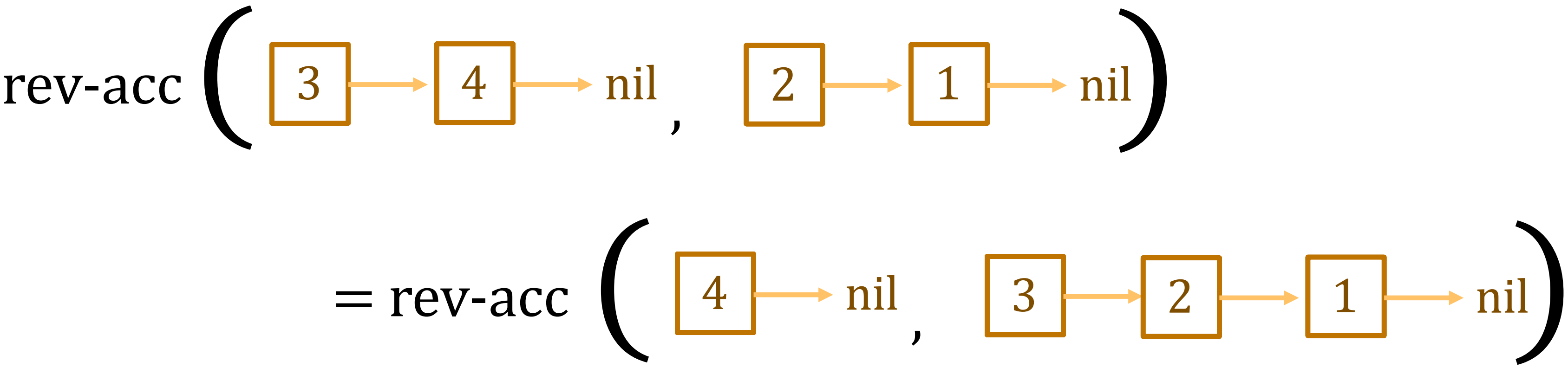
- Can we do this faster?
  - yes, but we need a helper function

Example 5: Reversing a List

func rev(nil) := nil
rev(cons(x, L)) := concat(rev(L), cons(x, nil)) for any x : ℤ and
any L : List

- **Helper function** rev-acc(S, R) **for any** S, R : List

func rev-acc(nil, R) := R for any R : List
rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R)) for any x : ℤ and
any L, R : List



rev-acc(3 → 4 → nil , 2 → 1 → nil)

= rev-acc (4 → nil , 3 → 2 → 1 → nil)

Example 5: Reversing a List

```
func rev-acc(nil, R)            :=  R                          for any R : List
     rev-acc(cons(x, L), R)     :=  rev-acc(L, cons(x, R))     for any x : ℤ and
                                                                  any L, R : List
```

- **Can prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
- **Can prove that** $\text{concat}(L, \text{nil}) = L$
  - **structural induction like prior examples**

- **Prove that** $\text{rev}(S) = \text{rev-acc}(S, \text{nil})$

rev-acc(S, nil)	= concat(rev(S), nil)	**Lemma 1**
	= rev(S)	**Lemma 2**

Procedural Abstraction

Reasoning about Function Calls

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

- Can replace $f(..)$ by its definition

$2\ f(10) = 2\ (2 \cdot 10 + 1)$ **def of** f

- Need to make sure the argument is non-negative

$f(n - 10)$ with $n : \mathbb{N}$

need to be sure that $n \geq 10$ for this to be allowed

  - if functions have conditions on arguments,
    we need to check that those conditions do hold

Reasoning about Function Calls

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

- **Can replace $f(..)$ by its definition and explain condition**

$2\,f(n - 10) = 2\,(2 \cdot (n - 10) + 1)$ **def of f (since $n \geq 10$)**

---

func $f(x) := 2n + 1$ **if** $x \geq 0$ for any $x : \mathbb{Z}$

$f(x) := 0$ **if** $x < 0$ for any $x : \mathbb{Z}$

- **Can replace $f(..)$ by its definition and explain condition**

$2\,f(n - 10) = 2\,(2 \cdot (n - 10) + 1)$ **def of f (since $n - 10 \geq 0$)**

Concrete vs Abstract

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

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

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

- In code, details are often hidden ("*abstracted away*")
  - we often want to purposefully hide the definition
  - gives us room to change it later

```
// n must be natural. Returns some natural number.
function f(n: number): number { .. }
```

Concrete vs Abstract

- In code, details are often hidden ("*abstracted away*")
  - we often want to purposefully hide the definition
  - hides complicated details

```
// Returns the same list but reversed, i.e.
//   rev(nil) := nil
//   rev(cons(x, L)) := concat(rev(L), cons(x, nil))
function rev(L: List): List {
  return rev_acc(L, nil);  // faster way
}
```

Level 1

- "`return concat(rev(L), cons(x, nil))`" would be level 0
- 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

Course Goals

To teach you to the skills necessary to write programs at the level of a professional software engineer

Specifically, we will teach the skills to write code that is

- correct
- easy to understand
- easy to change
- modular

Hiding details makes it easier to understand, leaves room for change, and lets people split up the work.

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
 */
function f(a: number, b: string): number
```

  - 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))
 */
function rev(L: List): List {
  return rev_acc(L, nil);  // faster
}
```

  - **English descriptions are typical for most code**

    professionals are very 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))
 */
function last(L: List): number
```

  - 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))
 */
function last(L: List): number {
  if (L === 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))
 */
function last(L: List): number {
  if (L === nil) throw new Error(“Bad client! Bad!”)
```

  - **code is the same, but the spec is <u>different</u>**

    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…
 */
function prefix(n: number, 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…
 */
function prefix(n: number, 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**

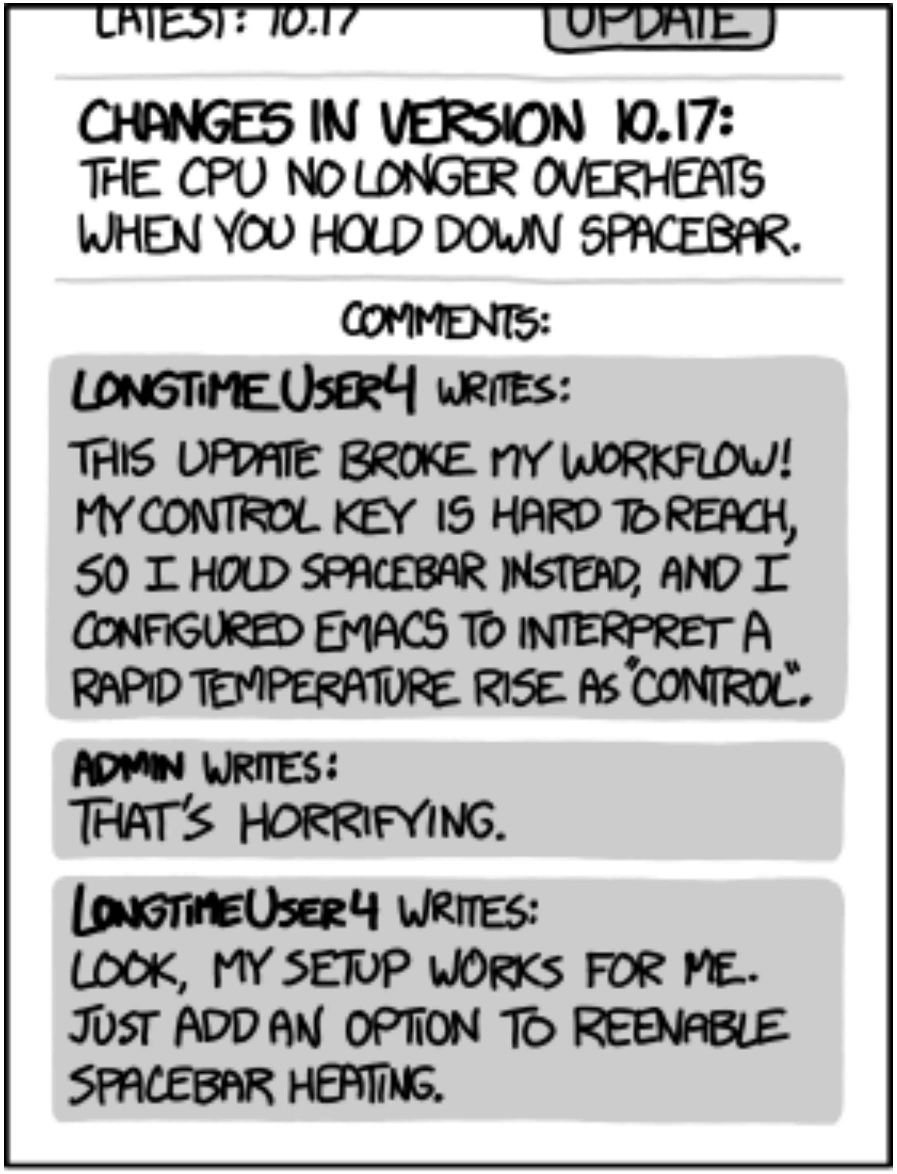
  - knowledge about code details tends to "leak"

    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



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:

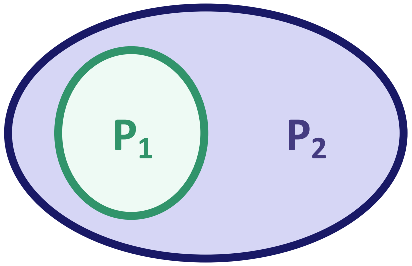
conditions included in `@return` (and `@throws`)

Correctness (satisfying the spec):

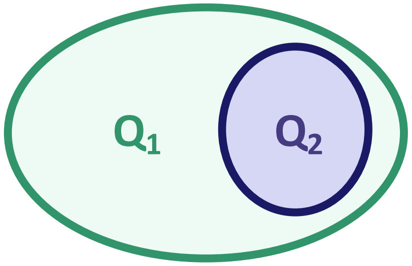
for every input satisfying the precondition,
the output will satisfy the postcondition

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

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
`@param` `@requires`	same or more allowed inputs	same or fewer allowed inputs
`@return` `@throws`	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
function indexOf(x: number, L: list): number
```

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
function indexOf(x: number, L: list): number
```

Which is stronger?

Specification A

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

C is stronger

Specification C

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

Example 3: Weaker vs Stronger

```
// Find the index of x in the list
function indexOf(x: number, L: list): number
```

Which is stronger?

Specification B

– requires L contains the value x

– returns the *first* index where x occurs in L

Specification C

incomparable

– 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”

Structural Induction

Example 5: Helper Lemma 2

func concat(nil, R) := R for any R : List

concat(cons(x, L), R) := cons(x, concat(L, R)) for any x : $\mathbb{Z}$ and
any L, R : List

- **Prove that** concat(S, nil) = S

Base Case (nil):

concat(nil, nil) = nil **def of** concat

Inductive Hypothesis: assume that concat(L, nil) = nil

Inductive Step (cons(x, L)): **prove that** concat(cons(x, L), nil) = cons(x, L)

Example 5: Helper Lemma 2

func $\text{concat}(\text{nil}, R)$	$:= R$	for any $R : \text{List}$
$\text{concat}(\text{cons}(x, L), R)$	$:= \text{cons}(x, \text{concat}(L, R))$	for any $x : \mathbb{Z}$ and any $L, R : \text{List}$

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Inductive Hypothesis: **assume that** $\text{concat}(L, \text{nil}) = \text{nil}$

Inductive Step $(\text{cons}(x, L))$:

$\text{concat}(\text{cons}(x, L), \text{nil}) =$

$= \text{cons}(x, L)$ **Ind. Hyp.**

Example 5: Helper Lemma 2

func $\text{concat}(\text{nil}, R) := R$ for any $R : \text{List}$

$\text{concat}(\text{cons}(x, L), R) := \text{cons}(x, \text{concat}(L, R))$ for any $x : \mathbb{Z}$ and any $L, R : \text{List}$

- **Prove that** $\text{concat}(S, \text{nil}) = S$

Inductive Hypothesis: assume that $\text{concat}(L, \text{nil}) = \text{nil}$

Inductive Step $(\text{cons}(x, L))$**:**

$$\begin{aligned} \text{concat}(\text{cons}(x, L), \text{nil}) &= \text{cons}(x, \text{concat}(L, \text{nil})) && \textbf{def of concat} \\ &= \text{cons}(x, L) && \textbf{Ind. Hyp.} \end{aligned}$$

Example 5: Helper Lemma 1

```
func rev-acc(nil, R)         :=  R                        for any R : List
     rev-acc(cons(x, L), R)  :=  rev-acc(L, cons(x, R))   for any x : ℤ and
                                                            any L, R : List
```

- **Prove that** rev-acc(S, R) = concat(rev(S), R)
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

rev-acc(nil, R) =

= concat(rev(nil), R) **def of** rev

```
func concat(nil, R)          :=  R
     concat(cons(x, L), R)   :=  cons(x, concat(L, R))
```

```
func rev(nil)                :=  nil
     rev(cons(x, L))         :=  concat(rev(L), cons(x, nil))
```

Example 5: Helper Lemma 1

func rev-acc(nil, R) := R   for any R : List
rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R))   for any x : $\mathbb{Z}$ and any L, R : List

- **Prove that** rev-acc(S, R) = concat(rev(S), R)
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

rev-acc(nil, R) = R   **def of** rev-acc
= concat(nil, R)   **def of** concat
= concat(rev(nil), R)   **def of** rev

func concat(nil, R) := R concat(cons(x, L), R) := cons(x, concat(L, R))	**func** rev(nil) := nil rev(cons(x, L)) := concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

func rev-acc(nil, R)	:=	R	for any R : List
rev-acc(cons(x, L), R)	:=	rev-acc(L, cons(x, R))	for any x : $\mathbb{Z}$ and any L, R : List

- **Prove that** rev-acc(S, R) = concat(rev(S), R)

Inductive Hypothesis: **assume that** rev-acc(L, R) = concat(rev(L), R) **for any** R

Inductive Step (cons(x, L)):

rev-acc(cons(x, L), R) =

= concat(rev(cons(x, L)), R) **def of** rev

func concat(nil, R) := R concat(cons(x, L), R) := cons(x, concat(L, R))	**func** rev(nil) := nil rev(cons(x, L)) := concat(rev(L), cons(x, nil))

Example 5: Helper Lemma 1

func rev-acc(nil, R) := R for any R : List
rev-acc(cons(x, L), R) := rev-acc(L, cons(x, R)) for any x : ℤ and any L, R : List

- **Prove that** rev-acc(S, R) = concat(rev(S), R)

Inductive Hypothesis: assume that rev-acc(L, R) = concat(rev(L), R) **for any** R

Inductive Step (cons(x, L))**:**

rev-acc(cons(x, L), R)	= rev-acc(L, cons(x, R))	**def of** concat
	= concat(rev(L), cons(x, R))	**Ind. Hyp.**
	= concat(rev(L), cons(x, concat(nil, R)))	**def of** concat
	= concat(rev(L), concat(cons(x, nil), R))	**def of** concat
	= concat(concat(rev(L), cons(x, nil)), R)	**Prop of** concat
	= concat(rev(cons(x, L)), R)	**def of** rev

func concat(nil, R) := R
concat(cons(x, L), R) := cons(x, concat(L, R))

func rev(nil) := nil
rev(cons(x, L)) := concat(rev(L), cons(x, nil))