CSE 331

Procedural Abstraction

Kevin Zatloukal

Reminders

- HW3 was due yesterday
  - substantially more difficult, as promised

    coding takes 3x longer than you expect, even for professionals

- HW4 released last night

- Start early!
  - recommend setting aside 60-90 min each day

    usually 1 problem per day, but HW4 has 7 problems

  - stop and ask a question when you get stuck

    leaves plenty of time to wait for an answer

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**

$2\,f(10)$	$= 2\,(2 \cdot 10 + 1)$	**def of** f
	$= 2\,(21)$	
	$= 42$	

Reasoning about Function Calls

- This becomes trickier with *side conditions*

$$\textbf{func } f(x) := 2x + 1 \quad \textbf{if } x \geq 0 \quad \text{for any } x : \mathbb{Z}$$
$$f(x) := 0 \quad \textbf{if } x < 0 \quad \text{for any } x : \mathbb{Z}$$

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

$$2\ f(n - 10) = 2\ (2 \cdot (n - 10) + 1) \quad \textbf{def of } f \textbf{ (since } n - 10 \geq 0\textbf{)}$$
$$= 4n - 38$$

- 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*")

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

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

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

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

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
};
```

Level 1

  - "`if .. 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

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: 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))
 */
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): 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))
 */
const 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))
 */
const 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…
 */
const 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…
 */
const 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

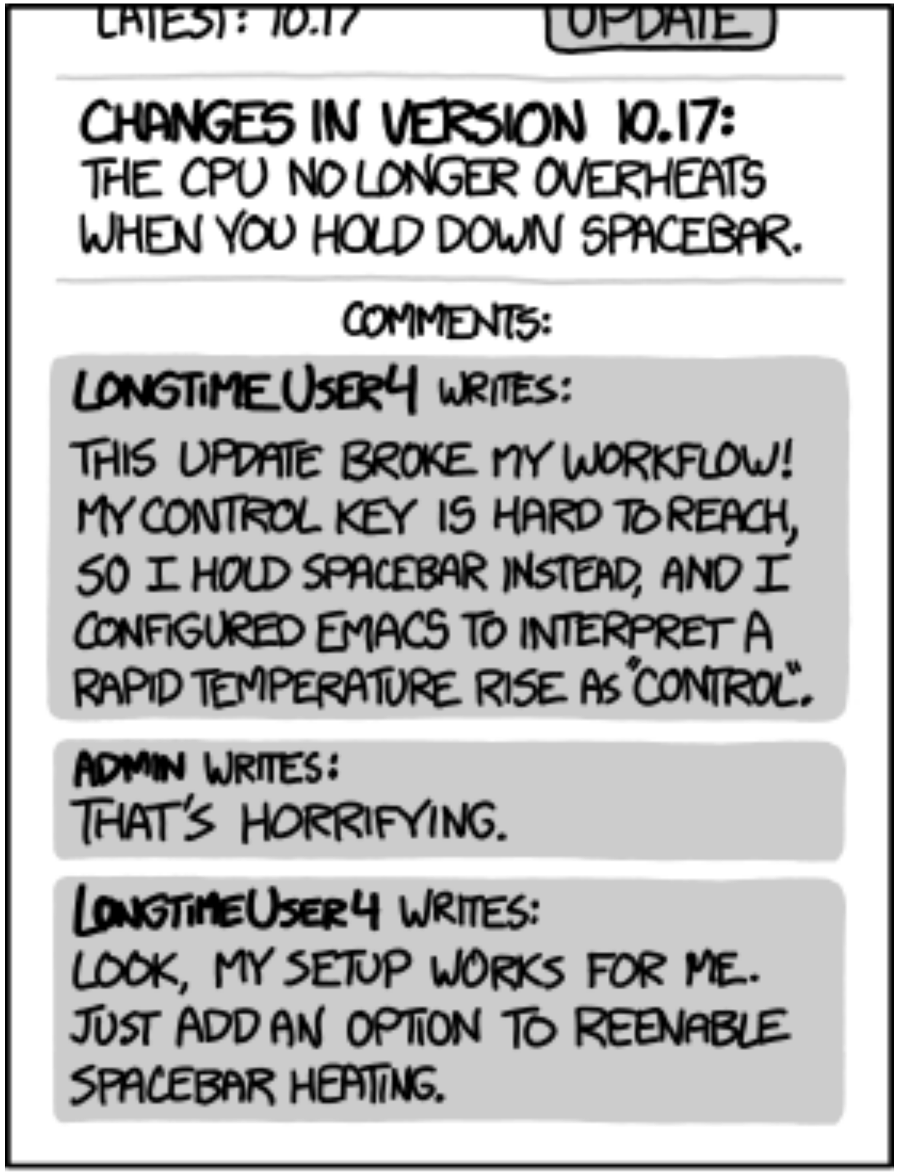
- **Correctness**
- **Changeability**
- **Modularity**
  - knowledge about code details tends to "leak"

    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:

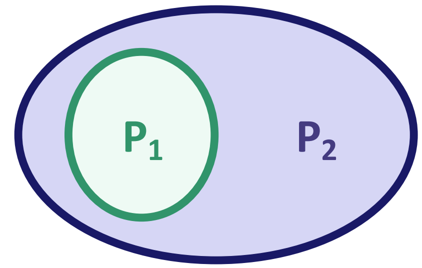
conditions included in **`@return`** (and **`@throws`**)

Correctness (satisfying the spec):

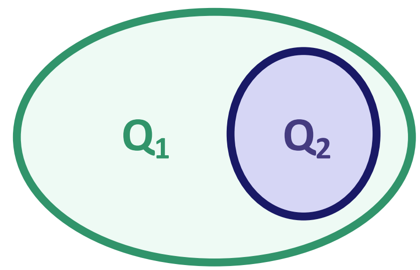
for every input satisfying the precondition,
the output will satisfy the postcondition

Weaker vs Stronger Specifications

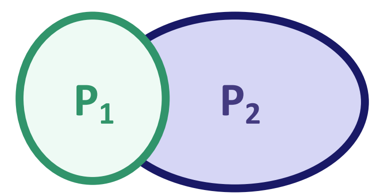
- **Definition**: specification S_2 is stronger than S_1 iff
  - precondition of S_2 is easier to satisfy than that of S_1
  - postcondition of S_2 is harder to satisfy than that of S_1 (on all inputs allowed by both)



- A **stronger** specification:
  - gives more guarantees to the client



- A **weaker** specification:
  - gives more freedom to the implementer



- 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
`@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
const 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

B is stronger

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

Example 2: Weaker vs Stronger

```
// Find the index of x in the list
const 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
const 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

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"

Structural Induction

Recall: 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

Recall: 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 : $\mathbb{Z}$ and any L, R : List

- **Can prove that** rev-acc(S, R) = concat(rev(S), R) **(Lemma 1)**
- **Can prove that** concat(L, nil) = L **(Lemma 2)**
  - structural induction like prior examples
  - (exercise!)
- **Prove that** rev(S) = rev-acc(S, nil)

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

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** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **prove by structural induction**

- **Need the following property of** concat

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

  - **with strings, we know that "A + (B + C) = (A + B) + C"**
  - **this says the same thing for lists**

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 **(so** R **is a variable)**

Base Case (nil):

rev-acc(nil, R) =

= concat(rev(nil), R)

func concat(nil, R)	:=	R	**func** rev(nil)	:=	nil
concat(cons(x, L), R)	:=	cons(x, concat(L, R))	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)
  - **prove by induction on** S **(so** 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 : 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)

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