CSE 331

Exceptions, Generics, & Type Erasure

Kevin Zatloukal

Reminders

- “Engineers are paid to think and **understand**”
  - you should be able to understand **all** the code in HW3

- Professional programmers are required to
  - **understand** **100%** of the code they write
  - **understand** what code does on **100%** of the allowed inputs

- For Level 1+, this requires **reasoning**
  - must use reasoning to think about all allowed inputs

    not okay to give the wrong answer on even one allowed input

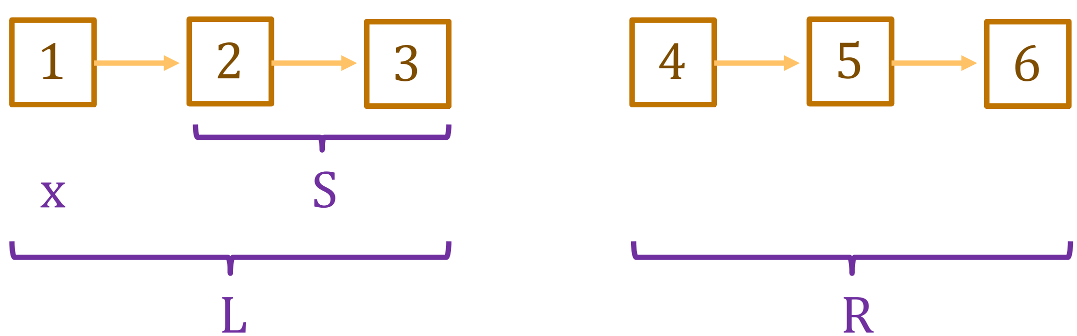
Structural Induction

Recall: Concatenating Two Lists

- **Mathematical definition of** $\text{concat}(S, R)$

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

  – $\text{concat}(S, R)$ **defined by pattern matching on** S **(not** R**)**



Example 3: Length of Concatenated Lists

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

- **Suppose we have the following code:**

```
const m: number = len(S);      // S is some List
const n: number = len(R);      // R is some List
...
return m + n;  // = len(concat(S, R))      Level 1
```

  – **spec returns** len(concat(S, R)) **but code returns** len(S) + len(R)

- **Need to prove that** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Example 3: Length of Concatenated Lists

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** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

$\text{len}(\text{concat}(\text{nil}, R)) =$

$= \text{len}(\text{nil}) + \text{len}(R)$

Example 3: Length of Concatenated Lists

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** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$
  - **prove by induction on** S
  - **prove the claim for any choice of** R (**i.e.,** R **is a variable**)

Base Case (nil):

len(concat(nil, R))	= len(R)	**def of** concat
	= 0 + len(R)	
	= len(nil) + len(R)	**def of** len

Example 3: Length of Concatenated Lists

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** $\text{len}(\text{concat}(S, R)) = \text{len}(S) + \text{len}(R)$

Inductive Step (cons(x, L)):

Need to prove that

$$\text{len}(\text{concat}(\text{cons}(x, L), R)) = \text{len}(\text{cons}(x, L)) + \text{len}(R)$$

Get to assume claim holds for L**, i.e., that**

$$\text{len}(\text{concat}(L, R)) = \text{len}(L) + \text{len}(R)$$

Example 3: Length of Concatenated Lists

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** len(concat(S, R)) = len(S) + len(R)

Inductive Hypothesis: assume that len(concat(L, R)) = len(L) + len(R)

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

len(concat(cons(x, L), R)) =

= len(cons(x, L)) + len(R)

Example 3: Length of Concatenated Lists

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** len(concat(S, R)) = len(S) + len(R)

Inductive Hypothesis: **assume that** len(concat(L, R)) = len(L) + len(R)

Inductive Step (cons(x, L)):

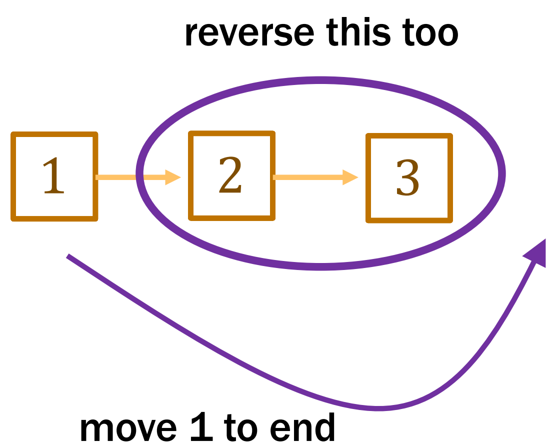
len(concat(cons(x, L), R))	= len(cons(x, concat(L, R)))	**def of** concat
	= 1 + len(concat(L, R))	**def of** len
	= 1 + len(L) + len(R)	**Ind. Hyp.**
	= len(cons(x, L)) + len(R)	**def of** len

Recall: Reversing a List

- **Mathematical definition of** $\text{rev}(S)$

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

  – **note that** rev **uses** concat **as a helper function**



Example 4: Length of Reversed List

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

- **Suppose we have the following code:**

```
const m: number = len(S);       // S is some List
const R: number = rev(S);
...
return m;  // = len(rev(S))                Level 1
```

  - **spec returns** len(rev(S)) **but code returns** len(S)

- **Need to prove that** len(rev(S)) = len(S) **for any** S : List

Example 4: Length of Reversed List

func $\text{rev}(\text{nil}) := \text{nil}$

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

- **Prove that** $\text{len}(\text{rev}(S)) = \text{len}(S)$ **for any** $S : \text{List}$

Base Case (nil):

$\text{len}(\text{rev}(\text{nil})) = \text{len}(\text{nil})$ **def of** rev

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

Need to prove that $\text{len}(\text{rev}(\text{cons}(x, L))) = \text{len}(\text{cons}(x, L))$

Get to assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

Example 4: Length of Reversed List

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

- **Prove that** $\text{len}(\text{rev}(S)) = \text{len}(S)$ **for any** S : List

Inductive Hypothesis: assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

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

$$\text{len}(\text{rev}(\text{cons}(x, L)))$$
$$=$$

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

Example 4: Length of Reversed List

func $\text{rev}(\text{nil}) \quad := \text{nil}$

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

- **Prove that** $\text{len}(\text{rev}(S)) = \text{len}(S)$ **for any** $S : \text{List}$

Inductive Hypothesis: assume that $\text{len}(\text{rev}(L)) = \text{len}(L)$

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

$$\begin{aligned}
&\text{len}(\text{rev}(\text{cons}(x, L))) \\
&\quad = \text{len}(\text{concat}(\text{rev}(L), \text{cons}(x, \text{nil}))) && \textbf{def of } \text{rev} \\
&\quad = \text{len}(\text{rev}(L)) + \text{len}(\text{cons}(x, \text{nil})) && \textbf{by Example 3} \\
&\quad = \text{len}(L) + \text{len}(\text{cons}(x, \text{nil})) && \textbf{Ind. Hyp.} \\
&\quad = \text{len}(L) + 1 + \text{len}(\text{nil}) && \textbf{def of } \text{len} \\
&\quad = \text{len}(L) + 1 && \textbf{def of } \text{len} \\
&\quad = \text{len}(\text{cons}(x, L)) && \textbf{def of } \text{len}
\end{aligned}$$

Finer Points of Structural Induction

- Structural Induction is how we reason about recursion
- Reasoning also follows structure of code
  - code uses structural recursion, so reasoning uses structural induction
- Note that rev is defined in terms of concat
  - reasoning about $\mathrm{len}(\mathrm{rev}(\ldots))$ used fact about $\mathrm{len}(\mathrm{concat}(\ldots))$
  - this is common

Exceptions

More List Functions

Functions to return the first or last element of a list

func first(nil)	:= ?	
first(cons(x, L))	:= x	for any L : List
func last(nil)	:= ?	
last(cons(x, nil))	:= x	for any $x : \mathbb{Z}$
last(cons(x, cons(y, L))	:= last(cons(y, L))	for any $x, y : \mathbb{Z}$ and any L : List

- Only makes sense for **non-empty** lists
  - there is no first or last element of an empty list
- What do we do when the input is nil?

Partial Functions in Math

Some functions do not have answers for some inputs

func first(nil) := undefined
first(cons(x, L)) := x   for any L : List

func last(nil) := undefined
last(cons(x, nil)) := x   for any x : $\mathbb{Z}$
last(cons(x, cons(y, L)) := last(cons(y, L))   for any x, y : $\mathbb{Z}$ and any L : List

- In math, we want functions to always be defined, so I had it return "undefined" in this case
  - return type is $\mathbb{Z} \cup \{\text{undefined}\}$

Partial Functions in Code

- **When programming, we also have invalid inputs, but we can handle them differently: disallow them**

```
// L must be a non-empty list
const last = (L: List): number => {
  if (L === nil) {
    throw new Error("empty list! Boooo");
  } else if (L.tl === nil) {
    return L.hd;
  } else {
    return last(L.tl);
  }
};
```

Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```
// L must be a non-empty list
const last = (L: List): number => {
  if (L === nil) {
    throw new Error("empty list! Boooo");
  …
};
```

- Specification says L will not be nil
  - we assume it is not nil when reasoning
  - do not assume it is not nil at run time

    an example of **defensive programming**

Partial Functions in Code

- When programming, we also have invalid inputs, but we can handle them differently: disallow them

```
// L must be a non-empty list
const last = (L: List): number => {
  if (L === nil) {
    throw new Error("empty list! Boooo");
  …
};
```

- In this case, we don't want to return undefined
  – better to "fail fast"…
  – debugging is easier if crash is closer to bug

Defensive Programming Rules

- Fine to disallow any inputs you don't want to handle
  - spec can say which inputs are allowed

    (the type system cannot always express this)

- Should also **check** that the inputs are valid
  - throw an exception if not
  - skip this only if the check is too expensive:

    if checking would make the function asymptotically slower, then skip it
  - after you spend 4 hours debugging a problem like this, you'll wish you had written the check

Generics

Lots of Lists of Things

We have now seen lists of

- integers
- squares (`Row` in HW3)
- rows (`Quilt` in HW3)
- HTML elements (`JsxList` in HW3)

These are all “the same” in some sense

- have nil and cons
- cons puts a new value at the front

Generic Types

We can describe this pattern with a “generic” list type

```
type List<A> = “nil”
             | {kind: ”cons”, hd: A, tl: List<A>};
```

- We can pick any type for **A**
  - TypeScript replaces all the “A”s by the type we give
  - e.g., List<number> is this type:

```
type List<number> = “nil”
  | {kind: ”cons”, hd: number, tl: List< number >};
```

Generic Types

We can describe this pattern with a "generic" list type

```
type List<A> = "nil"
             | {kind: "cons", hd: A, tl: List<A>};
```

Can now have

- `List<number>` = `List`
- `List<Square>` = `Row`
- `List<List<Square>>` = `Quilt`
- `List<JSX.Element>` = `JsxList`

Generic Types

We can describe this pattern with a "generic" list type

```
type List<A> = "nil"
             | {kind: "cons", hd: A, tl: List<A>};
```

- "**A**" is called a type parameter
- `List` is a function that takes a type as an argument and returns a new type
  - argument is the type of elements, result is list type

    (this is an *analogy* in Java, but it's literally true in TypeScript)
- Illegal to write "`List`" without its argument

Generic Functions

We also need to update the `cons` helper function

```
type List<A> = "nil"
             | {kind: "cons", hd: A, tl: List<A>};

const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: "cons", hd: x, tl: L};
};
```

- This is now a “generic function”
  - it has its own type parameter `<A,>`
  - extra comma is weird but required

    compiler thinks `<A>` is an HTML tag

Generic Functions

We also need to update the `cons` helper function

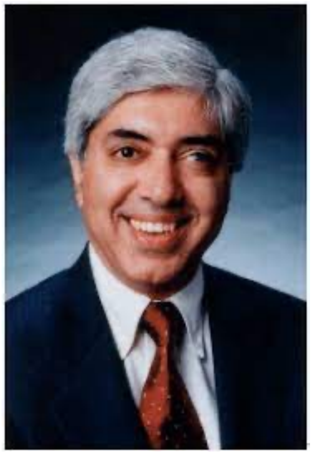
```
type List<A> = "nil"
             | {kind: "cons", hd: A, tl: List<A>};

const cons = <A,>(x: A, L: List<A>): List<A> => {
  return {kind: "cons", hd: x, tl: L};
};
```

- Parameters to generic types must be provided
- Parameters to generic functions are usually *inferred*

```
cons(1, cons(2, nil))        // has type List<number>
```

Generic Types & Functions

- We won't ask you to write generic types this quarter
- But you will need to use them
  - we will use `List<A>` in every assignment from now on
  - lists are the basic data structure of functional programming



Type Erasure

Type Checkers

- Type checkers eliminate large classes of bugs
  - e.g., cannot pass a string where an int is expected
  - critical part of ensuring **correctness**

- Sometimes give you ways to <u>opt out</u> of type checking
  - type casts says “just trust me”
  - “any” type

Run-Time Type Checking

- Java will double-check at **run-time** that you were right
  - type cast will fail with `ClassCastException`
  - however, there are cases where it **cannot** double-check

```
Integer n = (Integer) obj;                 // okay
List<Integer> L = (List<Integer>) obj;     // okay?
```

- Java can do some checks at run-time
  - can check if `obj` is an `Integer`
  - can check if `obj` is a `List<?>` (list of something)
  - **cannot** check if `obj` is a `List<Integer>`!

Run-Time Type Checking

- **Java will double-check at run-time that you were right**
  - **type cast will fail with** `ClassCastException`
  - **however, there are cases where it cannot double-check**

```
Integer n = (Integer) obj;                    // okay
List<Integer> L = (List<Integer>) obj;   // not okay
```

- **Cannot check if** `obj` **is a** `List<Integer>`
  - **all type parameters are "erased"**
  - **all** `List`**s are** `List<Object>` **at run-time**

    if it is correct, it is a `List<Object>` that happens to hold `Integer`s

Type Erasure in Java

```
if (obj instanceof List<Integer>) {      // not okay
```

- Java will give you an **error** on this line
  - it can tell if `L` is a `List`
  - it cannot tell if `L` is a `List<Integer>` (vs `List<String>`)

```
Integer n = (Integer) obj;                 // okay
List<Integer> L = (List<Integer>) obj;   // not okay
```

- Java only gives a **warning** about the second cast
  - should really be an **error**
  - programs with these warnings are unsafe

Type Erasure in TypeScript

- In TypeScript, all declared type information is erased!
  - no way to tell what type anything had in the source code

- Type casts are not double-checked at run-time
  - the only run-time type checks are ones you write

- If you use casts or “any” types, expect **pain**
  - variables will have values of types you didn’t expect
  - code will fail in bizarre ways

Handling Type Erasure

Options for avoiding painful debugging

1. Do not use (unchecked) type casts or "any" types
   - almost certainly the best option

2. Check the types yourself at run-time
   - lots of extra work
   - easy to make mistakes
   - (sometimes the only option)

More Recursion

Example 5: Reversing a List

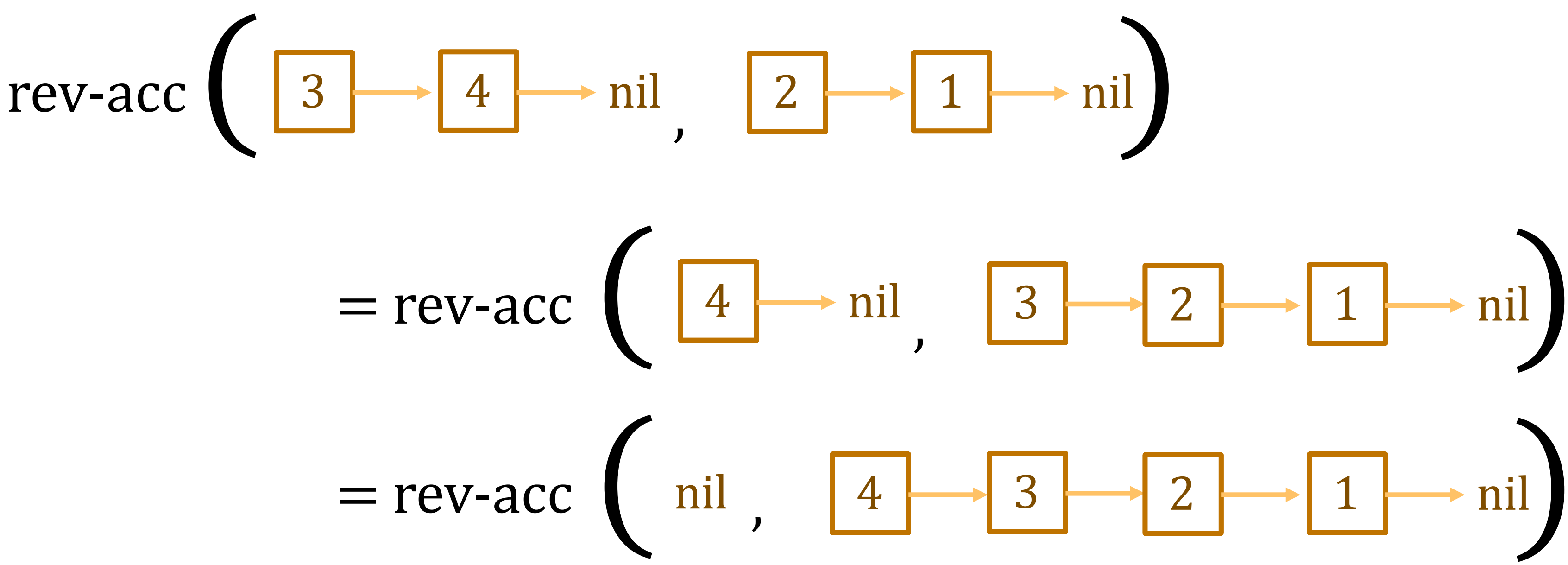
func rev(nil) := nil
rev(cons(x, L)) := concat(rev(L), cons(x, nil)) for any x : $\mathbb{Z}$ and any L : List

- 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

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



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

- **Can prove that** $\text{rev-acc}(S, R) = \text{concat}(\text{rev}(S), R)$
  - **more on this next time…**