CSE 331

Exceptions, Generics, & Type Erasure

Kevin Zatloukal

Administrivia

- HW2 problem 6 is now extra credit
  - but will have another problem on this in HW3, so it would be good to do this
- Added some slides on testing with HTML
  - i.e., tests of functions that return HTML
  - looks at several examples to explain the key issues
- Available on the Resources page (or directly here)

Structural Induction

Example 3: Length of Concatenated Lists

$$\begin{aligned}
&\textbf{func}\ \text{concat}(\text{nil}, R) && := R && \text{for any } R : \text{List} \\
&\quad \text{concat}(\text{cons}(x, L), R) && := \text{cons}(x, \text{concat}(L, R)) && \text{for any } x : \mathbb{Z} \text{ and} \\
& && && \quad \text{any } L, R : \text{List}
\end{aligned}$$

- **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** $\text{len}(\text{concat}(S, R))$ **but code returns** $\text{len}(S) + \text{len}(R)$

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

Base Case (nil):

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

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

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

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

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

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

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

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

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

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, 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

Example 4: Length of Reversed List

$$
\begin{aligned}
&\textbf{func } \mathrm{rev}(\mathrm{nil}) &&:= \mathrm{nil} \\
&\quad \mathrm{rev}(\mathrm{cons}(x, L)) &&:= \mathrm{concat}(\mathrm{rev}(L), \mathrm{cons}(x, \mathrm{nil})) \quad \text{for any } x : \mathbb{Z} \text{ and any } L : \mathrm{List}
\end{aligned}
$$

- **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** $\mathrm{len}(\mathrm{rev}(S))$ **but code returns** $\mathrm{len}(S)$

- **Need to prove that** $\mathrm{len}(\mathrm{rev}(S)) = \mathrm{len}(S)$ **for any** $S : \mathrm{List}$

Example 4: Length of Reversed List

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

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

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

Base Case (nil):

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

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

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

Get to assume claim holds for L**, i.e., that** $\mathrm{len}(\mathrm{rev}(L)) = \mathrm{len}(L)$

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 len(rev(L)) = len(L)

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

len(rev(cons(x, L)))

= len(cons(x, 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)):

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

Finer Points of Structural Induction

- **Structural Induction is how we reason about recursion**

- **Structure of proof follows structure of type**
  - **one implication to prove for each constructor**
  - **inductive hypothesis for each argument of same type**

- **Structure of proof also follows structure of code**
  - rev **is defined in terms of** concat
  - **proof about** $\mathrm{len}(\mathrm{rev}(\ldots))$ **used fact about** $\mathrm{len}(\mathrm{concat}(\ldots))$
  - **this is common**

Exceptions

Partial Functions in Math

Some functions do not have answers for some inputs

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

- There is no "last" element in an empty list
  - defining the type of non-empty lists is possible but not easy
- In math, we want functions to always be defined, so I had it return "undefined" in this case

Partial Functions in Code

- **When programming, we also have invalid inputs, but we want to handle them differently**

```
// L must be a non-empty list
function 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 want to handle them differently

```
// L must be a non-empty list
function 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 want to handle them differently

```
// L must be a non-empty list
function 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

Exceptions

- Syntax is the same as Java

```
throw new Error("explanation");
```

- Custom exception types can subclass `Error`
  - Java's normal base class is `Exception`
  - Java's type checker also makes sure:

    functions don't throw any exceptions they didn't declare

    callers catch any exceptions that are thrown

- Another (rare) case where Java does more than JS/TS

String Literals

- Often want to include data in error message

- Can do this using template literals

```
throw new Error(`value was negative: ${val}`);
```

  – string is between backticks (`..`) not quotes (‘..’ or “..”)

- Notice the “$” before {..}
  – HTML literals just use {..} without a “$”

Generics

Lots of Lists of Things

We have now seen lists of

- integers
- squares (`Row` in HW2)
- rows (`Quilt` in HW2)
- HTML elements (`JsxList` in HW2)

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 true in TypeScript)
- Illegal to write "`List`" without its argument

Generic Methods

We also need to update the `cons` helper function

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

function 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

Generic Methods

We also need to update the `cons` helper function

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

function 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 methods are usually *inferred*

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

Generic Types & Methods

- 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**

- They also 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 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 <u>you</u> 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)