Week 12: Return Type Polymorphism, Monad Transformers, and Wrapping Up

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science

Return type polymorphism
Recall type classes

In Haskell, a function defined in a type class has multiple implementations. The compiler selects the right one when the argument type is inferred.

```haskell
x == y
f x >>= \y -> return [y]
```

Java's method overloading

```java
class Point {
  void move(int dx, int dy) { ... }
  int move(float dx, float dy) { ... }
  String move() { ... }
}
Point p = Point();
p.move(1, 2);
p.move(1.5, 3.5);
p.move();
```

This seems more general than type classes!
Now consider Read:

```haskell
class Read a where
    read :: String -> a
```

read is ad-hoc polymorphic in its return type.

This is known as return type polymorphism.

This isn’t allowed by Java!

```java
class Point {
    void     move(int dx, int dy) { ... }
    int      move(int dx, int dy) { ... }
    String   move(int dx, int dy) { ... }
}
```
When `move` is called in Java, the compiler chooses the right implementation based on the arguments, not its return type. Why?

Because of side effects, the return value of a function can be ignored—the compiler might never see it!

```java
Point p = Point(2, 3);
p.move(10, 20);
p.move(1, 2);
p.move(100, 0);
```
Contrast this with Haskell:

```haskell
read :: Read a => String -> a
take (read "10") [1..100]
True && (read "False")
True && head (read "[True, False]"
```

Or consider

```haskell
return :: Monad m => a -> m a
f :: Maybe String -> State Integer String -> IO String
  -> ...
f = ...
f (return "David") (return "David") (return "David")
```
But type inference isn’t magic! Consider this function:

```haskell
f :: String -> String
f input = show (read input)
```

Combining monads
We've seen two monads representing different kinds of effects: State $s$ (stateful computations), and IO (input/output).

How do we express a computation that combines these effects?

Problem: given a recursive function, count the total number of times the function is called.

But also print out every intermediate count.
**Attempt 1**

The `a` is now an `IO` action:

\[
\text{fibCountedIO} :: \text{Integer} \to \text{State} \text{State Integer} (\text{IO Integer})
\]

**Attempt 2**

Using the `StateT` monad transformer, which is a “higher-order monad”:

\[
data \text{StateT} s m a = \text{StateT} (s \to m (a, s))
\]

Haskell implements the following “monad lifting”:

- If `m` is a monad, then `StateT s m` is also a monad.
Note: state, get, and put actually work with all StateT instances.

```haskell
state :: Monad m => (s -> (a, s)) -> StateT s m a
get :: Monad m => StateT s m s
put :: Monad m => s -> StateT s m ()
```

David's fun picture
Looking back, looking ahead

I'll Miss You
So you just... learned about programming languages?

In this course, you will learn to:

1. Define, analyze, and modify syntactic features of a programming language.
2. Define and analyze semantic features of a programming language.
3. Design and implement programs that operate on other programs.
4. Write programs using the functional programming paradigm.

And along the way, expand your definition of “programming”.
Computer languages differ not so much in what they make possible, but in what they make easy.

Larry Wall (creator of Perl)

The Blub Paradox (by Paul Graham)
I’m going to use a hypothetical language called Blub, and a hypothetical Blub programmer named David.

Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.
Of course David wouldn’t program in machine language. That’s what compilers are for. And as for Cobol, he doesn’t know how anyone can get anything done with it. It doesn’t even have [Blub feature of your choice].

Languages less powerful than Blub are obviously less powerful, because they’re missing some feature that David’s used to.
But when David looks up the power continuum, what he sees are merely weird languages. They seem about equivalent in power to Blub, but with all this other hairy stuff thrown in.

Blub is good enough for him, because he thinks in Blub.
The Lambda Calculus and Functional Programming

Pure functions as the basic unit of abstraction.
(define (add1ToAll lst)
  (if (null? lst)
      (list)
      (cons ((add1 (first lst))
             (add1ToAll (rest lst)))))

(define (sqrAll lst)
  (if (null? lst)
      (list)
      (cons ((sqr (first lst))
             (sqrAll (rest lst)))))

(map f lst)
Sorting as a higher-order function

; Racket
(sort lst less-than?)

# Python
list.sort(lst, key)

// Java
Collections.sort(lst, Comparator)

Name bindings: environments and closures

How are name bindings resolved?

What do “static” and “dynamic” mean?
Remaking the programming language in your image

Two main macro libraries

- **my-class**: a dynamic, message-passing approach to OOP
- **choice**: representing non-deterministic choices
Macros push the limit of expressing what we want to compute.

Types push the limit of expressing the constraints on what we compute.

Type systems
Polymorphism as abstraction

headInts :: [Integer] -> Integer
headBools :: [Bool] -> Bool

head :: [a] -> a

mapList :: (a -> b) -> [a] -> [b]
mapVector :: (a -> b) -> Vector a -> Vector b

fmap :: Functor f => (a -> b) -> f a -> f b
**Monads as abstraction of “chainable” computations**

- Computations that use “mutable” state: `State s`
- Failing computations: `Maybe`, `Either String`
- Non-deterministic computations: `[]`
- Computations that talk to the outside world: `IO`

**Purity encoded by types**

In Haskell, everything is pure by default. A function of type `Integer -> String` is mathematically pure.

It can't return `null`; can't mutate a global variable; can't be non-deterministic; can't interact with the outside world.
And now, a few words on what comes next

The classic: writing a compiler

In the first part of CSC324, we built an interpreter: given a program, perform the computation the programmer expressed.

A compiler: given a program, generate machine code that will perform the computation the programmer expressed.

CSC488: Compilers and Interpreters
Zero-cost abstractions

How do we take the abstractions we’ve discussed in this course and implement them efficiently?

Purely Functional Data Structures by Chris Okasaki

David’s wish list: Rust

“Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

“Rust’s rich type system and ownership model guarantee memory-safety and thread-safety — and enable you to eliminate many classes of bugs at compile-time.”
Formal verification: static proofs of correctness

How can we specify a program’s behaviour, and automatically prove its correctness without running a single test?

**CSC410**: Software Testing and Verification

**CSC465**: Formal Methods in Software Design

Dependent types: enhancing the type system

\[
[1, 2, 3] :: Vector Integer 3 \\
(++) :: Vector a n -> Vector a m -> Vector a (n + m)
\]
One last picture!

Thank you for being a terrific class.

Good luck on the final exam, and happy holidays!