

# CSC324 Lecture 24

# Our last lecture...

Today:

- Return-type polymorphism
- Composing monads
- Course wrapup

# Revisiting Java's polymorphism

Here's some Java code from earlier in the term. We saw how Java can dispatch calls to the correct `bar()` method depending on the argument types that are passed to `bar()`.

```
1 class Foo {  
2     static int bar(int i, int j) {  
3         System.out.println("Adding two integers...");  
4         return i + j;  
5     }  
6     static double bar(double i, double j) {  
7         System.out.println("Adding two doubles...");  
8         return i + j;  
9     }  
10  
11     public static void main(String[] args) {  
12         bar(1,1);  
13         bar(3.14, 2.71);  
14     }
```

```
[MSFT] /tmp javac Foo.java
```

```
[MSFT] /tmp java Foo
```

```
Adding two integers...
```

```
Adding two doubles...
```

```
[MSFT] /tmp
```

# Revisiting Java's polymorphism

Now that we've learned about subtyping and covariance, this should maybe be mildly surprising.

We might expect, through subsumption, that an `int` can be treated as a `double` in the lower `bar()`, but in this case Java knows to do the precise thing.

```
1 class Foo {  
2     static int bar(int i, int j) {  
3         System.out.println("Adding two integers...");  
4         return i + j;  
5     }  
6     static double bar(double i, double j) {  
7         System.out.println("Adding two doubles...");  
8         return i + j;  
9     }  
10  
11     public static void main(String[] args) {  
12         bar(1,1);  
13         bar(3.14, 2.71);  
14     }
```

```
[MSFT] /tmp javac Foo.java
```

```
[MSFT] /tmp java Foo
```

```
Adding two integers...
```

```
Adding two doubles...
```

```
[MSFT] /tmp
```

# Revisiting Java's polymorphism

Here's a similar program, but neither `bar()` take any arguments; they only differ in their return type.

Why does Java not allow this sort of polymorphism? The typechecker is clearly sophisticated to distinguish calls when they're arguments to the function...

```
1 class Foo {
2     static int bar() {
3         System.out.println("Returning an integer...");
4         return 42;
5     }
6     static double bar() {
7         System.out.println("Returning a double...");
8         return 42.0;
9     }
10
11     public static void main(String[] args) {
12         int i = bar();
13         double d = bar();
14     }
15 }
```

[0/5306]

"Foo.java" 17L, 326C written

3,15

Top

[MSFT] /tmp javac Foo.java

Foo.java:6: error: method bar() is already defined in class Foo  
 static double bar() {  
 ^

1 error

[MSFT] /tmp

# Revisiting Java's polymorphism

The intuition:

Remove the assignments and only keep the right hand side of each.

Which `bar()` do we want to call?

```
1 class Foo {  
2     static int bar() {  
3         System.out.println("Returning an integer...");  
4         return 42;  
5     }  
6     static double bar() {  
7         System.out.println("Returning a double...");  
8         return 42.0;  
9     }  
10  
11     public static void main(String[] args) {  
12         bar();  
13         bar();  
14     }  
15 }  
16  
17
```

"Foo.java" 17L, 307C written

17,0-1

All

[MSFT] /tmp javac Foo.java

Foo.java:6: error: method bar() is already defined in class Foo

```
    static double bar() {  
                ^
```

1 error

[MSFT] /tmp

# Revisiting Java's polymorphism

You might be saying: "what's the point of calling a function if we just discard the return value?" Can the compiler just choose to not invoke any operation, since we throw away the return value?

```
1 class Foo {
2     static int bar() {
3         System.out.println("Returning an integer...");
4         return 42;
5     }
6     static double bar() {
7         System.out.println("Returning a double...");
8         return 42.0;
9     }
10
11     public static void main(String[] args) {
12         bar();
13         bar();
14     }
15 }
16
17
```

"Foo.java" 17L, 307C written

17,0-1

All

[MSFT] /tmp javac Foo.java

Foo.java:6: error: method bar() is already defined in class Foo  
 static double bar() {  
 ^

1 error

[MSFT] /tmp

# Revisiting Java's polymorphism

That only works if the function is pure! A side-effecting Java method can't be removed (elided) in that way.

```
1 class Foo {
2     static int bar() {
3         System.out.println("Returning an integer...");
4         return 42;
5     }
6     static double bar() {
7         System.out.println("Returning a double...");
8         return 42.0;
9     }
10
11     public static void main(String[] args) {
12         bar();
13         bar();
14     }
15 }
16
17
```

~

~

"Foo.java" 17L, 307C written 17,0-1 All

---

[MSFT] /tmp javac Foo.java  
Foo.java:6: error: method bar() is already defined in class Foo  
 static double bar() {  
 ^  
1 error  
[MSFT] /tmp



# Return-type polymorphism in Haskell

Because the side-effecting ambiguity isn't present in a pure language, Haskell, by contrast, allows us to implement polymorphism that is dispatched on the return type of a function.

# Return-type polymorphism in Haskell

If `Show` is a typeclass that informs a type how to be printed as a string, `Read` is a typeclass that allows a string to be parsed into some type.

What type in particular? `read` is polymorphic in its return type, so it depends on how it's used!

```
Prelude> :t read
read :: Read a => String -> a
Prelude> :t show
show :: Show a => a -> String
Prelude> █
```

# Return-type polymorphism in Haskell

If `Show` is a typeclass that informs a type how to be printed as a string, `Read` is a typeclass that allows a string to be parsed into some type.

What type in particular? `read` is polymorphic in its return type, so it depends on how it's used!

```
Prelude> :t (read "10")  
(read "10") :: Read a => a  
Prelude> :t 1 + (read "10")  
1 + (read "10") :: (Num a, Read a) => a
```

# Return-type polymorphism in Haskell

If `Show` is a typeclass that informs a type how to be printed as a string, `Read` is a typeclass that allows a string to be parsed into some type.

What type in particular? `read` is polymorphic in its return type, so it depends on how it's used!

```
Prelude> :t (read "10")
(read "10") :: Read a => a
Prelude> :t 1 + (read "10")
1 + (read "10") :: (Num a, Read a) => a
Prelude> 1 + (read "10")
11
Prelude> 1 : (read "[2,3]")
[1,2,3]
Prelude> 1.0 : (read "[2,3]")
[1.0,2.0,3.0]
Prelude> █
```

# Return-type polymorphism in Haskell

What's another example of a function we know with a polymorphic return type?

How about `return`?

```
Prelude> :t return  
return :: Monad m => a -> m a
```

# Return-type polymorphism in Haskell

What's another example of a function we know with a polymorphic return type?

How about `return`?

```
Prelude> :t return
return :: Monad m => a -> m a
Prelude> :t fmap (+1) (return 1)
fmap (+1) (return 1) :: (Monad f, Num b) => f b
Prelude> fmap (+1) (return 1) :: Maybe Int
Just 2
Prelude> fmap (+1) (return 1) :: Either a Int
Right 2
Prelude> fmap (+1) (return 1) :: IO Int
2
Prelude> █
```

# Transforming monads

We saw how a polymorphic expression can be interpreted as representing a **single** performed within **multiple** possible monads.

```
Prelude> fmap (+1) (return 1) :: Maybe Int
Just 2
Prelude> fmap (+1) (return 1) :: Either a Int
Right 2
Prelude> fmap (+1) (return 1) :: IO Int
2
```

Another question we can ask: how do we express a computation that **combines multiple** effects (for instance, performing a computation that requires both state and also IO) into a **single** monad?

Problem: Consider an  
implementation of  
`fib :: Integer -> Integer`.

```
fib :: Integer -> Integer
fib n =
    if n < 2
    then
        n
    else
        fib(n-1) + fib(n-2)
```



Consider a version of fib that,  
in addition, returns the number  
of recursive calls that were  
made.

Consider a version of fib that, in addition, returns the number of recursive calls that were made.

```
fibWithCount :: Integer -> State Integer Integer
fibWithCount n = do
  currCount <- get
  put (currCount + 1)
```

```
  if n < 2
  then
    return n
```

```
  else do
    f1 <- fibWithCount(n-1)
    f2 <- fibWithCount(n-2)
    return (f1 + f2)
```

```
*Main> runState (fibWithCount 30) 0
(832040,2692537)
```

```
*Main> 
```

Consider a version of `fib` that, in addition, returns the number of recursive calls that were made, and prints out every intermediary value of `n` and the count.

Consider a version of `fib` that, in addition, returns the number of recursive calls that were made, and prints out every intermediary value of `n` and the count.

```
fibWithCountAndPrint :: Integer -> State Integer (IO Integer)
fibWithCountAndPrint n = do
    currCount <- get
    put (currCount + 1)

    if n < 2
    then
        return (do
            print (n, currCount)
            return n)
    else do
        f1_io <- fibWithCountAndPrint(n-1)
        f2_io <- fibWithCountAndPrint(n-2)
        return (do
            print (n, currCount)
            f1 <- f1_io
            f2 <- f2_io
            return (f1 + f2))
```

Consider a version of `fib` that, in addition, returns the number of recursive calls that were made, and prints out every intermediary value of `n` and the count.

```
fibWithCountAndPrint :: Integer -> State Integer (IO Integer)
fibWithCountAndPrint n = do
  currCount <- get
  put (currCount + 1)

  if n < 2
  then
    return (do
      print (n, currCount)
      return n)
  else do
    f1_io <- fibWithCountAndPrint(n-1)
    f2_io <- fibWithCountAndPrint(n-2)
    return (do
      print (n, currCount)
      f1 <- f1_io
      f2 <- f2_io
      return (f1 + f2))
```

Operating on  
State Integer (IO Integer)

Consider a version of `fib` that, in addition, returns the number of recursive calls that were made, and prints out every intermediary value of `n` and the count.

```
fibWithCountAndPrint :: Integer -> State Integer (IO Integer)
fibWithCountAndPrint n = do
  currCount <- get
  put (currCount + 1)

  if n < 2
  then
    return (do
      print (n, currCount)
      return n)
  else do
    f1_io <- fibWithCountAndPrint(n-1)
    f2_io <- fibWithCountAndPrint(n-2)
    return (do
      print (n, currCount)
      f1 <- f1_io
      f2 <- f2_io
      return (f1 + f2))
```

Operating on  
IO Integer

Consider a version of fib that, in addition, returns the number of recursive calls that were made, and prints out every intermediary value of n and the count.

```
*Main> :set +m
*Main> let (action, count) = runState (fibWithCountAndPrint 5) 0 in
*Main|   do
*Main|     res <- action
*Main|     return res
*Main|
(5,0)
(4,1)
(3,2)
(2,3)
(1,4)
(0,5)
(1,6)
(2,7)
(1,8)
(0,9)
(3,10)
(2,11)
(1,12)
(0,13)
(1,14)
5
*Main> █
```

# Slightly tedious, slightly awkward...

We have to manually unpack the nested monad with nested dos.

It isn't immediately obvious which do corresponds to computations inside each monadic context...

```
fibWithCountAndPrint :: Integer -> State Integer (IO Integer)
fibWithCountAndPrint n = do
    currCount <- get
    put (currCount + 1)

    if n < 2
    then
        return (do
            print (n, currCount)
            return n)
    else do
        f1_io <- fibWithCountAndPrint(n-1)
        f2_io <- fibWithCountAndPrint(n-2)
        return (do
            print (n, currCount)
            f1 <- f1_io
            f2 <- f2_io
            return (f1 + f2))
```



# Removing a level of nesting

Our return type is of type `State s a`; in particular:

```
State Integer (IO Integer)
```

If we could "drop the parentheses" around the second type argument, we would have some new monad of *three* arguments, the middle of which is, itself a monad. Then maybe we wouldn't need a `do`-block within another `do`-block...

# Removing a level of nesting

Our return type is of type `State s a`; in particular:

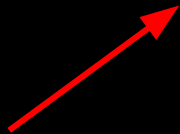
```
State Integer (IO Integer)
```

If we could "drop the parentheses" around the second type argument, we would have some new monad of *three* arguments, the middle of which is, itself a monad. Then maybe we wouldn't need a `do`-block within another `do`-block...

```
StateWithAMonad??? Integer IO Integer
```

```
Prelude> import Control.Monad.State  
Prelude Control.Monad.State> :t StateT
```

```
Prelude> import Control.Monad.State  
Prelude Control.Monad.State> :t StateT  
StateT :: (s -> m (a, s)) -> StateT s m a
```



StateT is a type of three type variables, as expected.

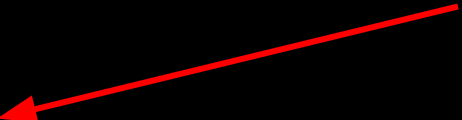
```
Prelude> import Control.Monad.State  
Prelude Control.Monad.State> :t StateT  
StateT :: (s -> m (a, s)) -> StateT s m a
```

```
Prelude> import Control.Monad.State
Prelude Control.Monad.State> :t StateT
StateT :: (s -> m (a, s)) -> StateT s m a
Prelude Control.Monad.State> :i StateT
```

```
instance Monad m => Monad (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance Functor m => Functor (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance MonadTrans (StateT s)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
```

```
Prelude> import Control.Monad.State
Prelude Control.Monad.State> :t StateT
StateT :: (s -> m (a, s)) -> StateT s m a
Prelude Control.Monad.State> :i StateT
```

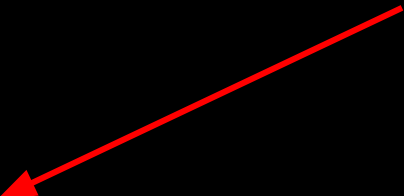
the `m` type variable  
must be a Monad (and,  
of course, a Functor)  
for `StateT` to be a  
Monad



```
instance Monad m => Monad (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance Functor m => Functor (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance MonadTrans (StateT s)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
```

```
Prelude> import Control.Monad.State
Prelude Control.Monad.State> :t StateT
StateT :: (s -> m (a, s)) -> StateT s m a
Prelude Control.Monad.State> :i StateT
```

So, StateT adds stateful computation to a base Monad m.

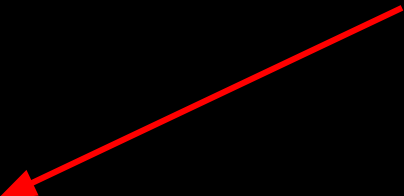


```
instance Monad m => Monad (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance Functor m => Functor (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance MonadTrans (StateT s)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
```



```
Prelude> import Control.Monad.State
Prelude Control.Monad.State> :t StateT
StateT :: (s -> m (a, s)) -> StateT s m a
Prelude Control.Monad.State> :i StateT
```

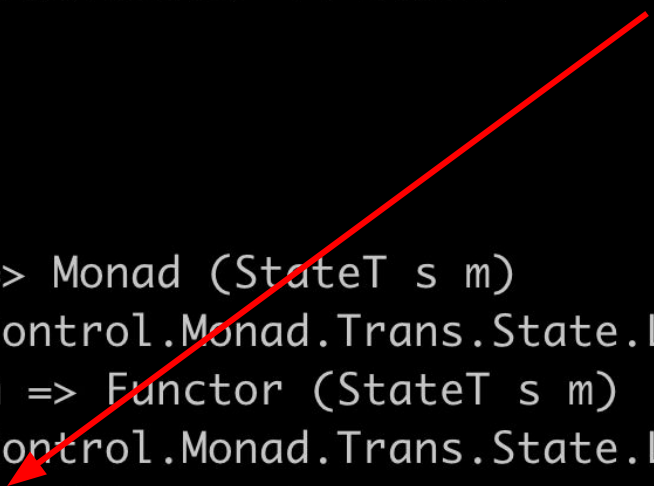
StateT allows get and put to be used from inside m's monadic context.



```
instance Monad m => Monad (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance Functor m => Functor (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance MonadTrans (StateT s)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
```

```
Prelude> import Control.Monad.State
Prelude Control.Monad.State> :t StateT
StateT :: (s -> m (a, s)) -> StateT s m a
Prelude Control.Monad.State> :i StateT
```

StateT is an instance of  
some typeclass called  
MonadTrans



```
instance Monad m => Monad (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance Functor m => Functor (StateT s m)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
instance MonadTrans (StateT s)
  -- Defined in 'Control.Monad.Trans.State.Lazy'
```

# MonadTrans is a Monad Transformer

MonadTrans contains a function lift, that takes an "inner" monad m a and "promotes" inner monad computations as part of a larger computation in the "full" monad t m a.

```
Prelude Control.Monad.State> :i MonadTrans
class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a
      -- Defined in 'Control.Monad.Trans.Class'
```

```
Prelude Control.Monad.State> :t Just 42
Just 42 :: Num a => Maybe a
Prelude Control.Monad.State> :t lift (Just 42)
lift (Just 42) :: (Num a, MonadTrans t) => t Maybe a
Prelude Control.Monad.State> :t lift (Just 42) :: StateT a Maybe Integer
lift (Just 42) :: StateT a Maybe Integer :: StateT a Maybe Integer
Prelude Control.Monad.State> :t runStateT ((lift (Just 42)) >=> (\a -> do put "state"; return a)) ""
runStateT ((lift (Just 42)) >=> (\a -> do put "state"; return a)) ""
  :: Num a => Maybe (a, [Char])
Prelude Control.Monad.State> runStateT ((lift (Just 42)) >=> (\a -> do put "state"; return a)) ""
Just (42,"state")
Prelude Control.Monad.State> █
```

```
fibWithTrans :: Integer -> StateT Integer IO Integer
fibWithTrans n = do
    currCount <- get
    put (currCount + 1)

    if n < 2
    then
        do
            lift (print (n, currCount))
            return n
    else do
        lift (print (n, currCount))
        f1 <- fibWithTrans(n-1)
        f2 <- fibWithTrans(n-2)
        return (f1 + f2)
```

# What happens in the IO...stays in the IO

We saw how we could lift a Maybe monad into a state transformer, do a stateful operation, and "drop back down" into a Maybe.

From the perspective of the final result's type, there isn't really any indication that the State monad was ever involved here.

```
*Main Control.Monad.State> runStateT ((lift (Just 42)) >>=
(\a -> do put 999; return a)) 0
Just (42,999)
*Main Control.Monad.State> █
```

# What happens in the IO...stays in the IO

We've seen that with many other monads we're able to extract values out them through pattern matching.

This is an expression, for instance, that operates on a `Maybe Int` and evaluates to an `Int`.

```
Prelude> case (Right 41) of Left x -> -1; Right x -> (x + 1)
42
Prelude> █
```

# What happens in the IO...stays in the IO

This operation is a bit like an inverse-return. (return goes from  $a \rightarrow m\ a$ , and this expression converts an  $m\ a$  to an  $a$ .)

```
Prelude> case (Right 41) of Left x -> -1; Right x -> (x + 1)
42
Prelude> █
```



# What happens in the IO...stays in the IO

IO, by contrast, is different. Once you have an IO action, there's *no way* to pull the value it produces out of the IO "data structure". In fact, the only ways to ever get the actual result from an IO action are:

- Execute the action in the REPL
- Execute the action in your `main` function

# What happens in the IO...stays in the IO

IO, by contrast, is different. Once you have an IO action, there's *no way* to pull the value it produces out of the IO "data structure". In fact, the only ways to ever get the actual result from an IO action are:

- Execute the action in the REPL
- Execute the action in your `main` function
- *...or, you could link your Haskell program with some C code and have the C code read the Haskell runtime's memory to peek at the action's value, but if you actually do this you will make me cry. Or be very impressed.*

Suffice to say that there are only two **typesafe** ways of getting a result from an IO action.

```
-- Produces the secret code, if the password is right.
getSecret :: IO (Maybe Integer)
getSecret = do
    print "Enter the password!"
    p <- getLine
    return (if p == "xyzyzy" then (Just 42) else Nothing)

printSecret :: ???
printSecret = fmap (\x -> "The secret code is: " ++ (show x)) (MaybeT getSecret)
```

```
-- Produces the secret code, if the password is right.
getSecret :: IO (Maybe Integer)
getSecret = do
    print "Enter the password!"
    p <- getLine
    return (if p == "xyzy" then (Just 42) else Nothing)

printSecret :: MaybeT IO String
printSecret = fmap (\x -> "The secret code is: " ++ (show x)) (MaybeT getSecret)
```

```
getSecret :: IO (Maybe Integer)
getSecret = do
  print "Enter the password!"
  p <- *Main> getSecret
  return "Enter the password!"
  xzyzy
  Just 42
printSecret *Main> printSecret
printSecret
```

```
<interactive>:20:1:
```

```
    No instance for (Data.Functor.Classes.Show1 IO)
      arising from a use of ‘print’
```

```
    In a stmt of an interactive GHCi command: print it
```

```
etSecret)
```

```
getSecret :: IO (Maybe Integer)
```

```
getSecret = do
```

```
  print "*Main Control.Monad.State> :t runMaybeT printSecret
```

```
  p <- get runMaybeT printSecret :: IO (Maybe String)
```

```
  return
```

```
printSecret
```

```
printSecret
```

```
getSecret)
```

```
getSecret :: IO (Maybe Integer)
```

```
getSecret = do
```

```
  print "*Main Control.Monad.State> :t runMaybeT printSecret
```

```
  p <- get runMaybeT printSecret :: IO (Maybe String)
```

```
  return *Main Control.Monad.State> runMaybeT printSecret
```

```
    "Enter the password!"
```

```
    xyzzzy
```

```
printSecret Just "The secret code is: 42"
```

```
printSecret
```

```
*Main Control.Monad.State> runMaybeT printSecret
```

```
    "Enter the password!"
```

```
    ohno
```

```
    Nothing
```

```
*Main Control.Monad.State> █
```

```
getSecret)
```

# Course wrapup

That's it for the CSC324 course material!

This material is often difficult and mind-binding, and you're learning it during a challenging time in the world. You should feel proud of yourselves.

Thank you for all your hard work this term.



# Where to go from here?

This is just the beginning of your journey of studying programming languages.

The U of T offers more PL courses to further your study:

- **CSC488:** Compilers and Interpreters
- **CSC410:** Software Testing and Verification
- **CSC465:** Formal Methods in Software Design

# Where to go from here?

In my opinion, self-study is the best way to learn advanced material and help you discover where your interests in computer science lie.

Here are some books that I've enjoyed, and that you might enjoy, too:

# Where to go from here?

If you enjoyed the Haskell portion of the course, and want to learn more about similar statically-typed languages that are often found in industry, in a rigorous but also practical way:

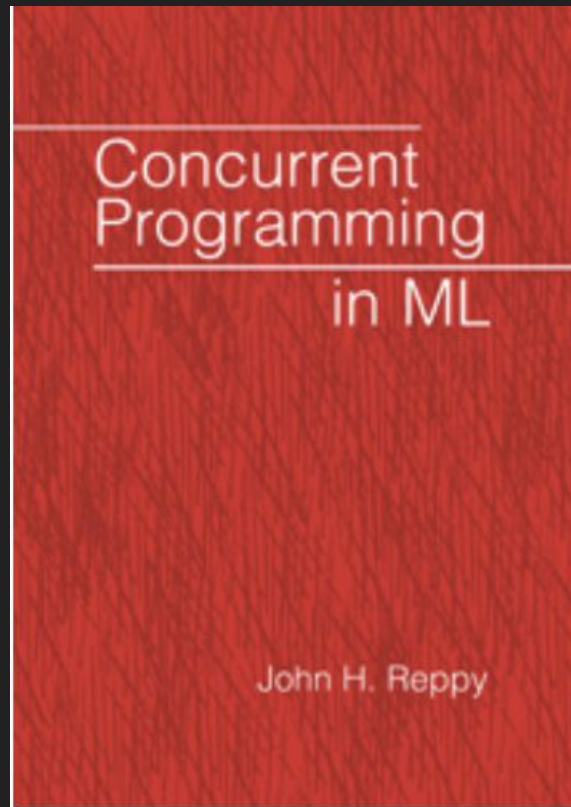
Chiusano and Bjarnson: **Functional Programming in Scala**. (Manning Press, 2014)



# Where to go from here?

If you are a straight-up low-level systems person and want to learn more about the advantages of writing concurrent and distributed systems in the functional style:

Reppy. **Concurrent Programming in ML**  
(Cambridge Press, 1999).

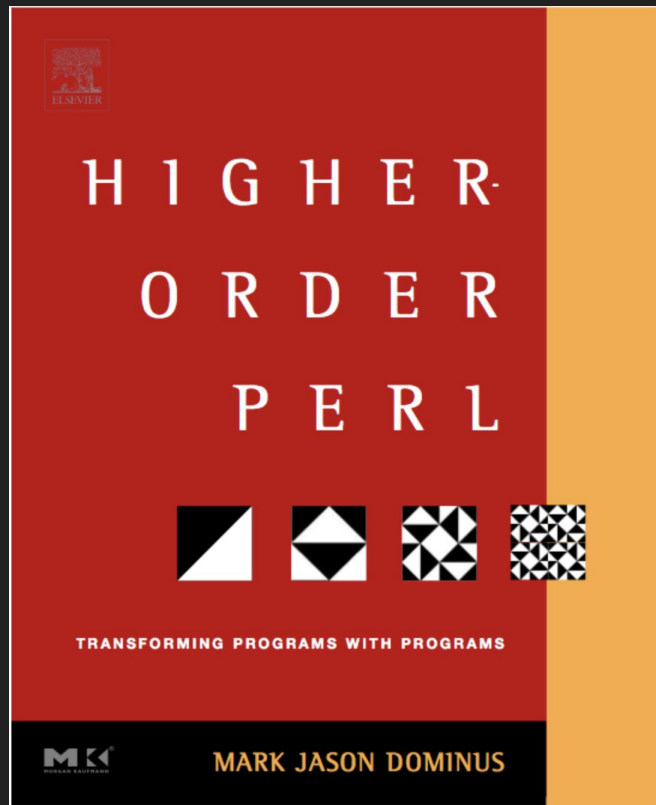


# Where to go from here?

If you enjoyed the Racket portion and want to learn more about metaprogramming; or,

If you want to learn more about programming in the FP-style but in "non-FP" languages:

[Dominus: Higher Order Perl \(Morgan Kaufman, 2005\)](#) (freely available)



# Where to go from here?

If you enjoyed the type theory part of the course and want to dig deeply into formal logic, type systems, and proof systems:

[Pierce et al. \*\*Software Foundations\*\* volumes 1-3](#)  
[\(Self-published, 2019\)](#) (Freely available)



# All the best!

Good luck with preparing for the final assessment, and I hope you and your family stay safe through the rest of the pandemic.

I'll hold office hours this and next week should you have questions leading up to taking the assessment. Otherwise, take care!

-Nathan