4 Monads

"Being a nomad is okay, but you wouldn't believe the problems we have getting mail forwarded!"
4 Monads

What’s a monad?

Monad stances

State monads

Monads for DSLs
What is a Monad?

A monad is a:

- Parameterized Data Type

```
data T a = ...
```

- Injection

```
return :: a → T a
```

- Continuation

```
(>>=) :: T a → (a → T b) → T b
```

(that satisfy three laws.)

**In terms of Haskell:**

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```
A monad is NOT a thing.

A monad is a way of looking at things.

“X is a monad”:
• X can be treated as a monad
• X behaves like a monad

has different interpretations
6 Monadic DSLs

What’s a monad?
Monad metaphors
State monads
Monads for DSLs
Monad Metaphors

Context \(\Rightarrow\) Box

instance of

(Effectful) Computation

\(\cong\)

Label
The Box Metaphor

\[ \text{Context} \cong \text{Box} \]
The Box Metaphor

Put one thing in a box and close the box

"Repackaging":

\[ b >>= f \]

1. Open box \( b \) to access content (say \( x \))
2. Generate new box(es) from box content using \( f \), i.e., \( f \ x \)
3. Combine boxes into one result box

Context \( \cong \) Box

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
The “Safety Seal” Box

The box

The things stored in the box

Put one thing in a box and close the box

“Repackaging”:

\[
b \triangleright= f
\]

(1) Open box \( b \) to access content (say \( x \))
(2) Generate new box(es) from box content using \( f \), i.e. \( f \ x \)
(3) Combine boxes into one result box

Box contains only a single item

A broken seal stays broken

Creates only one box, i.e. step (3) is not needed

data Maybe a = Just a | Nothing

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

class Monad Maybe where
  return = Just
  Just x  >>= f = f x
  Nothing >>= _ = Nothing

Creates only one box, i.e.
step (3) is not needed
Safety Seal Example

```
type Height = Int

rise :: Int → Height → Height
rise x h = h+x

drop :: Int → Height → Height
drop x h | h>=x = h-x
        | otherwise = error "Plane Crash"

takeOff :: Height → Height
takeOff 0 = 50

loop :: Height → Height
loop = rise 100 . drop 150 . rise 50
```

> rise 40 . loop . rise 100 . takeOff $ 0
190

> rise 40 . loop . takeOff $ 0
*** Exception: Plane Crash

no control over runtime errors
Using Maybe for Errors

riseM :: Int → Maybe Height → Maybe Height
riseM x (Just h) = Just (h+x)
riseM _ Nothing = Nothing

dropM :: Int → Maybe Height → Maybe Height
dropM x (Just h) | h>=x = Just (h-x)
                | otherwise = Nothing
dropM _ Nothing = Nothing

takeOffM :: Maybe Height → Maybe Height
takeOffM (Just _) = Just 50
takeOffM Nothing = Nothing

loopM :: Maybe Height → Maybe Height
loopM = riseM 100 . dropM 150 . riseM 50

> riseM 40 . loopM . riseM 100 . takeOffM $ Just 0
Just 190

> riseM 40 . loopM . takeOffM $ Just 0
Nothing
Abstracting Error Handling

```haskell
mrise :: Int → Height → Maybe Height
mrise x h = Just (h+x)

mdrop :: Int → Height → Maybe Height
mdrop x h | h>=x     = Just (h-x)
            | otherwise = Nothing

mtakeOff :: Height → Maybe Height
mtakeOff _ = Just 50

mloop :: Height → Maybe Height
mloop h = mrise 50 h >>= mdrop 150 >>= mrise 100
```

```
data Maybe a = Just a | Nothing

Just x  >>= f = f x
Nothing >>= _ = Nothing
```

```
> Just 0 >>= mtakeOff >>= mrise 100 >>= mloop >>= mrise 40
Just 190

> Just 0 >>= mtakeOff >>= mloop >>= mrise 40
Nothing
```

action composition through >>=
Exercises

Explain why the difference in the ordering of the functions is needed

```
riseM 40 . loopM . riseM 100 . takeOffM $ Just 0
```

```
Just 0 >>= mtakeOff >>= mrise 100 >>= mloop >>= mrise 40
```

Define the function =<< that allows us to write the example as follows

```
mrise 40 =<< (mloop =<< (mrise 100 =<< (mtakeOff =<< Just 0)))
```
The “Collection” Box

The box

The things stored in the box

Put one thing in a box and close the box

"Repackaging":

\[ \text{data } [a] = a : [a] \mid [] \]

class Monad m where
    return :: a \rightarrow m a
    (>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b

b >>= f
1. Open box b to access content (say x)
2. Generate new box(es) from box content using f, i.e. f x
3. Combine boxes into one result box

class Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)

Create a new box for each list element (step 2)

Flatten boxes (step 3)
Collection Monads

**Histograms**

```haskell
plot :: Int → String
plot n = take n (repeat 'X') ++ "\n"

histogram :: [Int] → String
histogram xs = xs >>= plot

hist :: [Int] → IO ()
hist = putStrLn . histogram
```

```haskell
> hist [5,7,3,4]
xxxxx
xxxxxxxx
xxx
xxxx
```

**Cartesian Products**

```haskell
cart :: [a] → [b] → [(a,b)]
cart xs ys = xs >>= (\x -> map (x,) ys)
```

```haskell
> cart [2..5] ['a'..'c']
[(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c'),...]
```
The “do” Notation

\[ f :: a \to T b \]
\[ f \ x = e(x) \]

bind value(s) extracted from monad to variable

\[ m :: T a \]
\[ m >>= f \]

\[ m >>= \lambda x \to e(x) \]

\[ m \]
\[ m >>= f \]

\[ m >>= f \]
\[ \equiv \]
\[ m >> e \]

\[ mloop :: \text{Height} \to \text{Maybe Height} \]
\[ mloop h = \text{mrise} 50 h >>= \text{mdrop} 150 >>= \text{mrise} 100 \]

\[ \text{cart} :: [a] \to [b] \to [(a,b)] \]
\[ \text{cart} \ x s y s = x s >>= (\lambda x \to \text{map} \ (x,) \ y s) \]

\[ \text{mloop} h = \text{do} u \leftarrow \text{mrise} 50 h \\
\quad d \leftarrow \text{mdrop} 150 u \\
\quad \text{mrise} 100 d \]

\[ \text{cart} \ x s y s = \text{do} \ x \leftarrow x s \\
\quad y \leftarrow y s \\
\quad \text{return} \ (x,y) \]
Predict the results of the following expressions

- `return 3 >>> do [4]`
- `return 3 >>> Just 4`
- `[3] >>> Just 4`
- `Just 4 >>= Just Nothing`  
- `[] >>= return`
- `[3,4] >>= return`
- `Just 4 >>= Just . succ`
- `[3,4] >>= return . succ`
- `Just 3 >>= return . Just`
- `Just 3 >>= return . return`
- `[1..3] >>= (\x->[1..x])`
Monad Metaphors
The Label Metaphor

Label
The Label Metaphor

Assign a (default) label to an item

"Relabeling":

1 >>= f
(1) Isolate label from 1 to reveal item (say x)
(2) Generate new labeled item(s) using f, i.e. f x
(3) Combine labeled items into one labeled item

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
Collecting Traces

```
data Trace a = T String a
```

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

```
instance Monad Trace where
  return x = T "" x
  (T t x) >>= f = let T u y = f x
                  in T (t++u) y
```

Labeled items

The things to be labeled

Assign a (default) label to an item

“Relabeling”:

1. Isolate label from l to reveal item (say x)
2. Generate new labeled item(s) using f, i.e. f x
3. Combine labeled items into one labeled item

Isolated label (step 1)

Empty default label

New labeled item (step 2)

Concatenating traces (step 3)
Tracing Computations

data Trace a = T String a

instance Monad Trace where
  return x = T "" x
  (T t x) >>= f = let T u y = f x
                 in T (t++u) y

teval :: Exp -> Int
eval (N i) = i
eval (Plus e e') = eval e + eval e'
eval (Neg e) = -(eval e)

data Exp = N Int |
           | Plus Exp Exp |
           | Neg Exp

teval :: Exp -> Trace Int
teval (N i) = trace i
teval (Plus e e') = teval e >>= (\i\->
teval e' >>= (\j\->
                        trace (i+j))
teval (Neg e) = teval e >>= (\i\->
                                    trace (-i))

trace :: Show a => a -> Trace a
trace x = T (show x++";") x
Tracing Computations

\[
\begin{align*}
\text{teval} & : \text{Exp} \rightarrow \text{Trace} \text{ Int} \\
teval (N \ i) & = \text{trace } i \\
teval (\text{Plus } e \ e') & = \text{teval } e \ \gg= (\i\rightarrow \\
& \quad \text{teval } e' \ \gg= (\j\rightarrow \\
& \quad \quad \text{trace } (i+j)) \\
teval (\text{Neg } e) & = \text{teval } e \ \gg= (\i\rightarrow \\
& \quad \quad \text{trace } (-i)) \\
\end{align*}
\]

\[
\begin{align*}
\text{trace} & : \text{Show } a \Rightarrow a \rightarrow \text{Trace } a \\
\text{trace } x & = T \ (\text{show } x++";") \ x
\end{align*}
\]

> eval $\ Plus (N 9) (Neg (Plus (N 3) (N 4)))$
2

> teval $\ Plus (N 9) (Neg (Plus (N 3) (N 4)))$
T "9;3;4;7;-7;2;" 2
Exercise

Explain why the following alternative implementations are all equivalent

teval (N i) = trace i >> return i

teval (N i) = trace i >> return

teval (N i) = trace i
Improved Tracing

\[
\begin{align*}
teval &\colon \text{Exp} \to \text{Trace Int} \\
teval e@(N\ i) &\ = \ trace\ e\ i \\
teval e@(Plus\ e_1\ e_2) &\ = \ do\ i \leftarrow teval\ e_1 \\
&\quad \ j \leftarrow teval\ e_2 \\
&\quad \ trace\ e\ (i+j) \\
teval e@(Neg\ e) &\ = \ do\ i \leftarrow teval\ e \\
&\quad \ trace\ e\ (-i) \\
\end{align*}
\]

\[
\begin{align*}
teval\ e \ &= \ trace\ e\ i \\
pt \ &= \ Trace\ a \to IO\ () \\
pt\ (T\ t\ _) \ &= \ putStr\Ln\ t \\
\end{align*}
\]

> pt $ teval $ Plus (N 9) (Neg (Plus (N 3) (N 4)))
  N 9 \Rightarrow 9
  N 3 \Rightarrow 3
  N 4 \Rightarrow 4
  Plus (N 3) (N 4) \Rightarrow 7
  Neg (Plus (N 3) (N 4)) \Rightarrow -7
  Plus (N 9) (Neg (Plus (N 3) (N 4))) \Rightarrow 2

Show arguments of evaluations too
Monad Metaphors

Box ⟶ (Effectful) Computation

Label
The Computation Metaphor

(Effectful) Computation
Counting Computations

```
data Exp = N Int  
  | Plus Exp Exp 
  | Neg Exp 

eval :: Exp -> Int 
eval (N i)   = i 
eval (Plus e e') = eval e + eval e' 
eval (Neg e)  = -(eval e) 

type Counter = Int 
evalC :: Counter -> Exp -> (Int,Counter) 
evalC (N i) (N j) = (i,n) 
evalC (Plus e e') = (i+j,o+1) 
  where (i,m) = evalC n e 
  (j,o) = evalC m e' 
evalC (Neg e)  = (-i,m+1) 
  where (i,m) = evalC n e 

> eval $ Plus (N 9) (Neg (Plus (N 3) (N 4))) 
  2 
> evalC 0 $ Plus (N 9) (Neg (Plus (N 3) (N 4))) 
  (2,3)
```

State is threaded through computations
Exercise

Try to implement a Count monad and use it for re-implementing evalC.

Why does it not work?

data Count a = C Int a
instance Monad Count where
  return x = C 0 x
  C c x >>= f = ...
The Computation Metaphor

(Effectful) Computation

Trivial computation (noop) just returning a result

c \triangleright= f

"Threading":

class Monad m where
  return :: a \rightarrow m a
  (\triangleright=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b

(0) Build a computation such that if run does:

(1) Run computation c to produce a result (say x)
(2) Generate a new computation d from result using f, i.e. f x
(3) Run computation d
6 Monadic DSLs

What’s a monad?

Monad metaphors
State monads
Monads for DSLs
The State (Transformer) Monad

```
data State s a = State (s → (a, s))

class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b

instance Monad (State s) where
  return x = State (\s → (x, s))
  State c >>= f = State (\s → let (x, s') = c s
                           State d = f x in d s')
```

---

**Computation**

**Result of computation**

**Trivial computation (noop)**
just returning a result

**"Threading":**

1. Run computation `c` to produce a result (say `x`) and new state
2. Generate a new computation `d` from result using `f`, i.e., `f x`
3. Run computation `d`

**Noop computation:** return state unchanged plus result

(0) Build a computation such that if `run` does:
  (1) Run computation `c` to produce a result (say `x`) and new state
  (2) Generate a new computation `d` from result using `f`, i.e., `f x`
  (3) Run computation `d`
State Transformer Bind

```
data State s a = State (s \rightarrow (a, s))
```

```
State c >>= f = State (\s \rightarrow let (x, s') = c s
                           in d s')
```
Monadic Counting

Monads

Monadic Counting

evalC :: Counter → Exp → (Int, Counter)
evalC n (N i) = (i, n)
evalC n (Plus e e') = (i+j, o+1)
  where (i, m) = evalC n e 
  (j, o) = evalC m e'
evalC n (Neg e) = (-i, m+1)
  where (i, m) = evalC n e

> runWith 0 $ seval $
  Plus (N 9) (Neg (Plus (N 3) (N 4)))
(2, 3)

type Count a = State Counter a
incCounter :: Count ()
incCounter = State $ \c→((), c+1)

seval :: Exp → Count Int
seval (N i) = return i
seval (Plus e e') = do i ← seval e
  j ← seval e'
  incCounter
  return (i+j)
seval (Neg e) = do i ← seval e
  incCounter
  return (-i)

runWith = flip runState

State monad values are functions that need to be applied to have an effect
State Monad Operations

```
data State s a = State (s → (a,s))

runState :: State s a → s → (a,s)
runState (State f) = f

runWith :: s → State s a → (a,s)
runWith = flip runState

onState :: (s → s) → State s ()
onState f = State $ \s → ((),f s)

fromState :: (s → a) → State s a
fromState f = State $ \s → (f s,s)

readState :: State s s
readState = State $ \s → (s,s)

init :: s → State s ()
init s = State $ \_ → (() ,s)
```
Exercises

Using the State monad, define a function mineval that keeps track of the lowest intermediate result during an evaluation.

Using the State monad, redefine the function teval for tracing computations.
Monads

Unifying Monad Stances

Safety Seal Box

\[
\text{data } \text{Maybe } a = \text{Just } a \mid \text{Nothing}
\]

\[
\text{class Monad Maybe where}
\]
\[
\text{return } = \text{Just}
\]
\[
\text{Just } x \triangleright\triangleright f = f \ x
\]
\[
\text{Nothing } \triangleright\triangleright _ = \text{Nothing}
\]

Collection Box

\[
\text{data } [a] = a : [a] \mid []
\]

\[
\text{class Monad [] where}
\]
\[
\text{return } x = [x]
\]
\[
\text{xs } \triangleright\triangleright f = \text{concat (map } f \text{ xs)}
\]

Computations that can fail

Nondeterministic Computations

(Effectful) Computation

Computational Interpretation
Nondeterminism

```haskell
data [a] = a : [a] | []

class Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)

choose :: [a → a] → [a] → [a]
choose [] xs = []
choose (f:fs) xs = map f xs ++ choose fs xs

interest p x = x*(1+p/100)
spend = flip (-)
keep = interest 5
wallStreet _ = 0

> choose [wallStreet,keep] . choose [spend 400,keep] $ [1000]
[0.0,0.0,630.0,1102.5]
```
6 Monadic DSLs

What’s a monad?

Monad stances

State monads

Monads for DSLs