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 = ...

with two operations:

Injection

return :: a → T a

Continuation

(>>=) :: T a → (a → T b) → T b

In terms of Haskell:

class Monad m where

  return :: a → m a

  (>>=) :: m a → (a → m b) → m b

(that satisfy three laws.)
The Meta-Nature of Monads

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 stances
State monads
Monads for DSLs
Monad Stances

- box
- (effectful) computation
- label

instance of
instance of
The Box Stance
The Box Metaphor

The Box

Put one thing in a box and close the box

"Repackaging":

\[
\begin{align*}
\text{b} \mapsto f \\
(1) \text{Open box b to access content (say x)} \\
(2) \text{Generate new box(es) from box content using f, i.e. f x} \\
(3) \text{Combine boxes into one result box}
\end{align*}
\]

The things stored in the box

\[
\text{class Monad m where}
\]
\[
\begin{align*}
\text{return} :: a \rightarrow m a \\
(\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\end{align*}
\]
## The “Safety Seal” Box

**data Maybe a =** Just a |
| Nothing

**class Monad m where**
- **return :: a → m a**
- **(>>=) :: m a → (a → m b) → 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 Maybe where**
- **return = Just**
- **Just x >>= f = f x**
- **Nothing >>= _ = Nothing**

**Box contains only a single item**

**A broken seal stays broken**

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

```haskell
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

---

Flight control

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
```

```
data Maybe a = Just a | Nothing

data Maybe

Encapsulate in >>= function
```

```
> 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

\[
\text{riseM 40 . loopM . riseM 100 . takeOffM} \; \text{Just 0}
\]

\[
\text{Just 0 >>> mtakeOff >>> mrise 100 >>> mloop >>> mrise 40}
\]

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

\[
\text{mrise 40 =<< (mloop =<< (mrise 100 =<< (mtakeOff =<< \text{Just 0}))}
\]
The “Collection” Box

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

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

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

The things stored in the box

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

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

Flatten boxes (step 3)
Collection Monads

Histograms

\[
\begin{align*}
\text{plot} & : \text{Int} \rightarrow \text{String} \\
\text{plot} \ n & = \text{take} \ n \ (\text{repeat} \ 'X') \ + \ "\n"
\end{align*}
\]

\[
\begin{align*}
\text{histogram} & : [\text{Int}] \rightarrow \text{String} \\
\text{histogram} \ xs & = xs \ >>= \ \text{plot}
\end{align*}
\]

\[
\begin{align*}
\text{hist} & : [\text{Int}] \rightarrow \text{IO} () \\
\text{hist} & = \ \text{putStrLn} . \ \text{histogram}
\end{align*}
\]

> hist [5,7,3,4]

\[
\begin{array}{c}
\text{xxxxx} \\
\text{xxxxxxxxx} \\
\text{xxx} \\
\text{xxxx}
\end{array}
\]

How many elements are in the result box?

Cartesian Products

\[
\begin{align*}
\text{cart} & : [a] \rightarrow [b] \rightarrow [(a,b)] \\
\text{cart} \ xs \ ys & = xs \ >>= \ (\times \rightarrow \text{map} \ (\times,) \ ys)
\end{align*}
\]

> cart [2..5] ['a'..'c']

\[
\begin{array}{c}
(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c'),\ldots
\end{array}
\]
The “do” Notation

\[ f :: a \to T b \]
\[ f \ x = e \langle x \rangle \]

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

\[ \_ \to e \]
\[ m >>= \_ \to e \equiv m >> e \]

bind value(s) extracted from monad to variable

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

\[ \text{cart} :: [a] \to [b] \to [(a,b)] \]
\[ \text{cart} \ xs \ ys = \text{xs} >>= (\_ \to \text{map} \ (x,) \ \text{ys}) \]

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

\[ \text{cart} \ xs \ ys = \text{do} \ x \leftarrow \text{xs} \]
\[ \quad \text{y} \leftarrow \text{ys} \]
\[ \quad \text{return} \ (x,y) \]
Exercises

Predict the results of the following expressions

- `return 3 >>= do [4]
- return 3 >>= Just 4
- [3] >>= Just 4
- Just 4 >>= Just
- Nothing >>= Just
- [] >>= 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 Stances

(box) ⟷ (effectful) computation

(label)
The Label Stance
The Label Metaphor

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

Assign a (default) label to an item

“Relabeling”:

l >>= f

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
Monads

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

Assign a (default) label to an item

Labeled items

The things to be labeled

"Relabeling":

l >>= f
(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

```haskell
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

tracing Interpreter

  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) )

tracing Interpreter

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

Interpreter

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

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

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

> 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

\( m >>= \_ \rightarrow e \equiv m >>= e \)
Improved Tracing

\[
\begin{align*}
teval :: \text{Exp} & \rightarrow \text{Trace Int} \\
teval e@(N\ i) &= \text{trace} e \, i \\
teval e@(Plus\ e1\ e2) &= \text{do} \ i \leftarrow\ teval\ e1 \, j \leftarrow\ teval\ e2 \, \text{trace} e \,(i+j) \\
teval e@(Neg\ e) &= \text{do} \ i \leftarrow\ teval\ e \, \text{trace} e \,(-i)
\end{align*}
\]

\[
\begin{align*}
\text{trace} :: \text{Show a} & \Rightarrow \text{String} \rightarrow a \rightarrow \text{Trace a} \\
\text{trace} s\ x &= T\,(s++"\Rightarrow"++\text{show} x++\"\n")\ x \\
pt :: \text{Trace a} & \rightarrow \text{IO} () \\
pt\ (T\ t\ _) &= \text{putStrLn} t
\end{align*}
\]

> 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
Monad Stances

- Box
- Label
- (effectful) computation
The Computation Stance

(effectful) computation
Counting Computations

```haskell
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 (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  

> 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

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> 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 stances
- State monads
- Monads for DSLs
The State (Transformer) Monad

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

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

**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

- Run \( c \) to produce state and result (step 1)
- Generate new computation \( d \) (step 2)
- Run computation \( d \) (step 3)
State Transformer Bind

\[
\text{data State } s \ a = \text{State } (s \rightarrow (a,s))
\]

\[
\text{State } c \rightarrow \text{State } d
\]

\[
\begin{align*}
\text{State } c & \gg= f = \text{State } (\lambda s . \text{let } (x,s') = c s \text{ in } d s') \\
\text{State } d & = f x \\
\end{align*}
\]
Monadic Counting

```haskell
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 0 $ seval $
   Plus (N 9) (Neg (Plus (N 3) (N 4)))
(2,3)
```

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.
Unifying Monad Stances

Safety Seal Box

```haskell
data Maybe a = Just a | Nothing

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

Collection Box

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

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

Computational Interpretation

- Computations that can fail
- Nondeterministic Computations
- (effectful) computation
Nondeterminism

Data

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

Class Monad

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

Nondeterministic Computations

```haskell
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
```

Example

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