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
What is a Monad?

A monad is a:

Parameterized Data Type

\[
data \ T \ a = \ldots
\]

with two operations:

Injection

\[
\text{return} :: a \rightarrow T \ a
\]

Continuation

\[
(\gg\gg) :: T \ a \rightarrow (a \rightarrow T \ b) \rightarrow T \ b
\]

In terms of Haskell:

\[
class \text{Monad} \ m \ where
\text{return} :: a \rightarrow m \ a
(\gg\gg) :: m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b
\]

(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
4 Monads

What’s a monad?
Monad metaphors
State monads
Monad Metaphors

- Context ≅ Box
- Label
- (Effectful) Computation

instance of

Monads
The Box Metaphor

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

The Box

Context

===

Box

The things stored in the box

The box

Put one thing in a box and close the box

“Repackaging”:

\[ b \gg= f \]

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

Monads
The “Safety Seal” Box

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

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

The box

The things stored in the box

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

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

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

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
```

```
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
```

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

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

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

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

```
box contains only a single item
```

```
box contains only a single item
```

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

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

```
a broken seal stays broken
```

```
a broken seal stays broken
```

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```

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

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

```
box contains only a single item
```

```
box contains only a single item
```

```
a broken seal stays broken
```

```
a broken seal stays broken
```
Safety Seal Example

type Height = Int

rise :: Int \to Height \to Height
rise x h = h+x

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

takeOff :: Height \to Height
takeOff 0 = 50

loop :: Height \to Height
loop = rise 100 . drop 150 . rise 50

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

> rise 40 . loop . takeOff $ 0
*** Exception: Plane Crash
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) = 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
Abstracting Error Handling

mrise :: \( \text{Int} \rightarrow \text{Height} \rightarrow \text{Maybe} \text{Height} \)
mrise \( x \) \( h \) = \text{Just} \( (h+x) \)

mdrop :: \( \text{Int} \rightarrow \text{Height} \rightarrow \text{Maybe} \text{Height} \)
mdrop \( x \) \( h \) | \( h \geq x \) = \text{Just} \( (h-x) \)
| otherwise = \text{Nothing}

mtakeOff :: \( \text{Height} \rightarrow \text{Maybe} \text{Height} \)
mtakeOff _ = \text{Just} \( 50 \)

mloop :: \( \text{Height} \rightarrow \text{Maybe} \text{Height} \)
mloop \( h \) = mrise \( 50 \) \( h \) >>= mdrop \( 150 \) >>= mrise \( 100 \) >>= mtakeOff >>= mloop >>= mrise \( 40 \)

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

\text{Just} \( x \) >>= f = f \( x \)
\text{Nothing} >>= _ = \text{Nothing}

action composition through >>=

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

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

Put one thing in a box and close the box

"Repackaging":

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

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

Flatten boxes (step 3)
Collection Monads

**Histories**

\[ \text{plot} :: \text{Int} \to \text{String} \]
\[ \text{plot} \ n = \text{take} \ n \left( \text{repeat} \ 'X' \right) ++ "\n" \]

\[ \text{histogram} :: [\text{Int}] \to \text{String} \]
\[ \text{histogram} \ \text{x} = \text{x} >>= \text{plot} \]

\[ \text{hist} :: [\text{Int}] \to \text{IO} () \]
\[ \text{hist} = \text{putStrLn} \ . \ \text{histogram} \]

**Cartesian Products**

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

\[ > \text{cart} \ [2..5] \ [\text{a}'..\text{c}'] \]
\[ [(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c'), \ldots] \]
The “do” Notation

\[ f :: a \rightarrow T \ b \]
\[ f \ x = e(\langle x \rangle) \]

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

\[ m \gg= \ \lambda x \rightarrow e(x) \]

\[ \text{bind value(s) extracted from monad to variable} \]

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

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

\[ \text{cart} :: [a] \rightarrow [b] \rightarrow [(a,b)] \]
\[ \text{cart} \ xs \ ys = \text{xs} \gg= (\lambda x \rightarrow \text{map } (x,) \ ys) \]

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

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])

Monads
Monad Metaphors

Box ➔ (Effectful) Computation

Label
The Label Metaphor
The Label Metaphor

Assign a (default) label to an item

“Relabeling”:

\[ l \gg= f \]

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

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

Isolated label (step 1)

Labeled items

The things to be labeled

Assign a (default) label to an item

“Relabeling”:
1 >>= 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

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)

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) )

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

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

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

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

> 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

\[
\begin{align*}
\text{teval}(N\ i) &= \text{trace} \ i \ >> \ return \ i \\
\text{teval}(N\ i) &= \text{trace} \ i \ >> \ return \\
\text{teval}(N\ i) &= \text{trace} \ i
\end{align*}
\]

\[
\begin{align*}
m \ >>= \ \lambda e. e & \equiv \ m \ >> \ e
\end{align*}
\]
Improved Tracing

```haskell
trace :: Show a => String -> a -> Trace a
trace s x = T (s ++ " ==> " ++ show x ++ "\n") x

pt :: Trace a -> IO ()
pt (T t _) = putStrLn t

trace :: Show a => String -> a -> Trace a
trace s x = T (s ++ " ==> " ++ show x ++ "\n") x

pt :: Trace a -> IO ()
pt (T t _) = putStrLn t

> pt $ teval $ Plus (N 9) (Neg (Plus (N 3) (N 4)))
N 9 ==> 9
N 3 ==> 3
N 4 ==> 4
Plus (N 3) (N 4) ==> 7
Neg (Plus (N 3) (N 4)) ==> -7
Plus (N 9) (Neg (Plus (N 3) (N 4))) ==> 2
```
Monad Metaphors

Box ⟶ (Effectful) Computation

Label
The Computation Metaphor

(Effectful) Computation
Counting Computations

\[
data \text{ Exp } = \text{ N Int} \\
| \text{ Plus Exp Exp} \\
| \text{ Neg Exp}
\]

\[
\text{ eval } :: \text{ Exp } \rightarrow \text{ Int} \\
\text{ eval } \ (\text{ N } i) \quad = \quad i \\
\text{ eval } \ (\text{ Plus } e \ e') \quad = \quad \text{ eval } e + \text{ eval } e' \\
\text{ eval } \ (\text{ Neg } e) \quad = \quad - \left( \text{ eval } e \right)
\]

\[
\text{ type } \text{ Counter } = \text{ Int} \\
\text{ evalC } :: \text{ Counter } \rightarrow \text{ Exp } \rightarrow \left( \text{ Int}, \text{ Counter} \right) \\
\text{ evalC } \ n \ (\text{ N } i) \quad = \quad \left( i, n \right) \\
\text{ evalC } \ n \ (\text{ Plus } e \ e') \quad = \quad \left( i + j, o + 1 \right) \\
\quad \quad \text{ where } \left( i, m \right) = \text{ evalC } \ n \ e \\
\quad \quad \quad \quad \quad \quad \quad \left( j, o \right) = \text{ evalC } \ m \ e' \\
\text{ evalC } \ n \ (\text{ Neg } e) \quad = \quad \left( - i, m + 1 \right) \\
\quad \quad \text{ where } \left( i, m \right) = \text{ evalC } \ n \ e
\]

\[
> \text{ eval } \ \theta \ (\text{ Plus } (\text{ N } 9) \ (\text{ Neg } \ (\text{ Plus } \ (\text{ N } 3) \ (\text{ N } 4)))) \\
2
\]

\[
> \text{ evalC } \ \theta \ (\text{ Plus } (\text{ N } 9) \ (\text{ Neg } \ (\text{ Plus } \ (\text{ N } 3) \ (\text{ N } 4)))) \\
(2,3)
\]
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

Monads

(Effectful) Computation

Trivial computation (noop) just returning a result

"Threading":
\[ c \gg= f \]

\( (\gg=) \) :: \( m a \rightarrow (a \rightarrow m b) \rightarrow m b \)

class Monad m where
  return :: a \rightarrow m a
  (\gg=) :: 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 \( d \)
4 Monads

What’s a monad?

Monad metaphors

State monads
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')
```

**data State s a = State (s -> (a, s))**

**class Monad m where**

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

**Computation**

- **Trivial computation (noop)**: just returning a result
- **"Threading"**: Run computation `c` to produce a result (say `x`) and new state. Generate a new computation `d` from result using `f`, i.e. `f x`. Run computation `d`.

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

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 manipulated by computation**

**Result of computation**

**Run**

**Generate new computation**

**Build computation**

**Run computation**
**State Transformer Bind**

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

State c

State d

\[
\text{State } c \gg\gg f = \text{State } (s \rightarrow \text{let } (x, s') = c s \\
\text{State } d = f x \\
\text{in } d s')
\]
Monadic Counting

- `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)`

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

- `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 = flip runState`

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

```haskell
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**

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

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

**Collection Box**

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

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