6 Monadic DSLs

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

What’s a monad?
Monad stances
State monads
Monads for DSLs
What is a Monad?

A monad is a:

**Parameterized Data Type**

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

**with two operations:**

**Injection**

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

**Continuation**

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

*(that satisfy three laws.)*

**In terms of Haskell:**

\[
\text{class Monad } m \text{ where}
\]

\[
\begin{align*}
\text{return} & :: a \rightarrow m \ a \\
(\gg=) & :: m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b
\end{align*}
\]
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
Reactions ...

Monads
6 Monadic DSLs

What’s a monad?

Monad stances

State monads

Monads for DSLs
Monad Stances

(box) instance of (effectful) computation instance of
(label)
The Box Stance
## The Box Metaphor

### Box Metaphor

**Class Monad m where**

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

**Put one thing in a box and close the box**

**“Repackaging”:**

```haskell
y >>= f
```

1. Open box `y` 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
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

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

Put one thing in a box and close the box

"Repackaging":

\[ y >>= f \]

1. Open box \( y \) 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
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
```
Using Maybe for Errors

Data

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

Encapsulate in >>= function

\[
\text{Just } x \gg= f = f x
\]
\[
\text{Nothing} \gg= _ = \text{Nothing}
\]

Error handling is included in every function

\[
riseM :: \text{Int} \rightarrow \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
riseM x \ (\text{Just } h) = \text{Just } (h+x)
\]
\[
riseM _ \ \text{Nothing} = \text{Nothing}
\]

\[
dropM :: \text{Int} \rightarrow \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
dropM x \ (\text{Just } h) \mid h=x = \text{Just } (h-x)
\]
\[
dropM x \ (\text{Just } h) \mid \text{otherwise} = \text{Nothing}
\]
\[
dropM _ \ \text{Nothing} = \text{Nothing}
\]

\[
takeOffM :: \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
takeOffM (\text{Just } _) = \text{Just } 50
\]
\[
takeOffM \ \text{Nothing} = \text{Nothing}
\]

\[
loopM :: \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
loopM = \text{riseM } 100 \ . \ \text{dropM } 150 \ . \ \text{riseM } 50
\]

> \text{riseM } 40 \ . \ \text{loopM} \ . \ \text{riseM } 100 \ . \ \text{takeOffM} \$ \ \text{Just } 0
\]
\[
\text{Just } 190
\]

> \text{riseM } 40 \ . \ \text{loopM} \ . \ \text{takeOffM} \$ \ \text{Just } 0
\]
\[
\text{Nothing}
\]
Monadic DSLs

Abstracting Error Handling

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

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

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

data Maybe a = Just a | Nothing

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

action composition through >>=
Exercises

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

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

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

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

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

data [a] = a : [a] | []

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

“Repackaging”:
y >>= f
  (1) Open box y 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)
Monadic DSLs

**Histograms**

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

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

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

> hist [5,7,3,4]
xxxxx
xxxxxxx
xxx
xxxx

How many elements are in the result box?

**Cartesian Products**

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

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

- **f :: a → T b**
  - `f x = e⟨x⟩`

- **m :: T a**
  - `m >>= f`
  - `m >>= \x→e⟨x⟩`

- **bind value(s) extracted from monad to variable**
  - `do {x ← m; e⟨x⟩}`

- **mloop**
  - `mloop :: Height → Maybe Height`
  - `mloop h = mrise 50 h >>= mdrop 150 >>= mrise 100`
  - `mloop h = do u ← mrise 50 h
d ← mdrop 150 u
mrise 100 d`

- **cart**
  - `cart :: [a] → [b] → [(a,b)]`
  - `cart xs ys = xs >>= (\x→map (x,) ys)`
  - `cart xs ys = do x ← xs
y ← ys
return (x,y)`

Monadic DSLs
Predict the results of the following expressions

- `return 3 >> do [4]`  
  - Result: `[4]`

- `return 3 >> Just 4`  
  - Result: `Just 4`

- `[3] >> Just 4`  
  - Result: `Error`

- `Just 4 >>= Just`  
  - Result: `Just 4`

- `Nothing >>= Just`  
  - Result: `Nothing`

- `[] >>= return`  
  - Result: `[]`

- `[3,4] >>= return`  
  - Result: `[3,4]`

- `Just 4 >>= Just . succ`  
  - Result: `Just 5`

- `[3,4] >>= return . succ`  
  - Result: `[4,5]`

- `Just 3 >>= return . Just`  
  - Result: `Just (Just 3)`

- `Just 3 >>= return . return`  
  - Result: `Error`

- `[] >>= return`  
  - Result: `[]`

- `return 3 >> do [4]`  
  - Result: `[4]`

- `return 3 >> Just 4`  
  - Result: `Just 4`

- `[3] >> Just 4`  
  - Result: `Error`

- `Just 4 >>= Just`  
  - Result: `Just 4`

- `Nothing >>= Just`  
  - Result: `Nothing`

- `[] >>= return`  
  - Result: `[]`

- `[3,4] >>= return`  
  - Result: `[3,4]`

- `Just 4 >>= Just . succ`  
  - Result: `Just 5`

- `[3,4] >>= return . succ`  
  - Result: `[4,5]`

- `Just 3 >>= return . Just`  
  - Result: `Just (Just 3)`

- `Just 3 >>= return . return`  
  - Result: `Error`

- `[] >>= return`  
  - Result: `[]`
Monad Stances

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

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

Labeled items

The things to be labeled

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
Collecting Traces

data Trace a = T String a

Labeled items

The things to be labeled

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

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

Empty default label

New labeled item (step 2)

Concatenating traces (step 3)
Monadic DSLs

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
```

```
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)
```

```
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

\[
\begin{align*}
teval & : \text{Exp} \rightarrow \text{Trace} \text{ Int} \\
teval (N \ i) & = \text{trace } i \\
teval (\text{Plus } e \ e') & = teval e \ >>> (\lambda i \rightarrow \\
& \hspace{1cm} teval e' \ >>> (\lambda j \rightarrow \\
& \hspace{2cm} \text{trace } (i+j) )) \\
teval (\text{Neg } e) & = teval e \ >>> (\lambda i \rightarrow \\
& \hspace{1cm} \text{trace } (-i) )
\end{align*}
\]

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

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

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

Explain why the following alternative implementations are equivalent

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

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

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

m >>= \_ -> e \equiv m >>= e
Improved Tracing

```haskell
> 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
```

```haskell
teval :: Exp -> Trace Int
teval e@(N i)     = trace e i
teval e@(Plus e1 e2) = do i ← teval e1
                          j ← teval e2
                          trace e (i+j)
teval e@(Neg e)    = do i ← teval e
                          trace e (-i)

trace :: (Show e, Show r) => e -> r -> Trace r
trace e r = T (show e++" ==> "++show r++"\n") r

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

Show arguments of evaluations too
Monad Stances

box

label

(effectful) computation
The Computation Stance

(effectful) computation
Counting Computations

**Type Definition**

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

**Function Definitions**

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

```haskell
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 
```

**Examples**

```haskell
> 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

“Threading”:

\[ c \gg= f \]

(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 \( d \)

class Monad m where
  return :: a \rightarrow m a
  (\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
6 Monadic DSLs

What’s a monad?

Monad stances

State monads

Monads for DSLs
The State (Transformer) Monad

data State s a = State (s → (a, s))

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

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

Computation

State manipulated by computation

Trivial computation (noop) just returning a result

“Threading”:

(0) Build a computation such that if run does:
RUN c to produce state and result (step 1)

(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
GENERATE new computation d (step 2)

(3) Run computation d
RUN computation d (step 3)

Result of computation

Noop computation: return state unchanged plus result

Build computation (step 0)
State Transformer Bind

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

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

Monadic Counting

\[\text{seval} :: \text{Exp} \rightarrow \text{Count}\]
\[\text{seval} (N\ i) = \text{return}\ i\]
\[\text{seval} (\text{Plus}\ e\ e') = \text{do}\ i \leftarrow \text{seval}\ e\ j \leftarrow \text{seval}\ e'\ \text{incCounter}\ \text{return}\ (i+j)\]
\[\text{seval} (\text{Neg}\ e) = \text{do}\ i \leftarrow \text{seval}\ e\ \text{incCounter}\ \text{return}\ (-i)\]

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

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

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

runWith = flip runState
Monadic DSLs

### State Monad Operations

```hs
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)
```

Computations that can fail

Nondeterministic Computations

(effectful) computation

Computational Interpretation
Nondeterminism

data [a] = a : [a] | []

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

Nondeterministic Computations

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]