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:

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

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

\[
(\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\]
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
The Box Stance

box
The Box Metaphor

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

**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
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
```

- **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

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

- **The box**
- **The things stored in the 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

```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
```

Flight control

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

*** Exception: Plane Crash

no control over runtime errors
Using Maybe for Errors

```haskell
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
```

Error handling is included in every function.

Encapsulate in >>= function

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

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

```haskell
> 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
```

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

```
> 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

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

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

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

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

The things stored in the box

<table>
<thead>
<tr>
<th>The box</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Monad m where</td>
</tr>
<tr>
<td>return :: a → m a</td>
</tr>
<tr>
<td>(&gt;&gt;=) :: m a → (a → m b) → m b</td>
</tr>
</tbody>
</table>

Put one thing in a box and close the box

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

“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)
Collection Monads

**Monadic DSLs**

**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
XXXXXXX
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'), ... ]
```

How many elements are in the result box?

---

CS 585 • Domain-Specific Languages • Spring 2018
The “do” Notation

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

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

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

\[
m loop : : Height \rightarrow Maybe \ Height \\
m loop \ h = mrise 50 \ h >>= mdrop 150 >>= mrise 100
\]

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

\[
m loop \ h = do \ u \leftarrow mrise 50 \ h \\
d \leftarrow mdrop 150 \ u \\
mrise 100 \ d
\]

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

Predict the results of the following expressions

- `return 3 >>> do [4]`
- `return 3 >>> Just 4`
- `[3] >>> Just 4`
- `Just 4 >>>= Just Nothing`
- `[] >>>= return`
- `[4] >>>= Just 4`
- `Error`
- `Just 4 >>>= return`
- `Nothing <<<= 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])`
- `[3,4] >>>= return`
- `Just 5`
- `[4,5] >>>= return
- `Just (Just 3)`
- `Error`
- `[1,1,2,1,2,3] <<<= return`
Monad Stances

(box) → (effectful) computation

(label)
The Label Stance
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

The things to be labeled

Labeled items

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

Monadic DSLs
Collecting Traces

```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

```

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

“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

Labeled items

The things to be labeled

Assign a (default) label to an 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

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) )
```

```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)
```

```haskell
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 ->
teval e' >>= (\j ->
trace (i+j)))
teval (Neg e)  = teval e >>= (\i ->
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 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
Improved Tracing

> 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

Show arguments of evaluations too

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
Monad Stances

(box) label

(effectful) computation
The Computation Stance

(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 (N i) = (i,n)
evalC n (Plus e e') = (i+j,o+1)
  where (i,m) = evalC n e
      (j,o) = evalC n 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

Monadic DSLs

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

Computation
Result of computation

Trivial computation (noop) just returning a result

"Threading":
  c >>= 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

(effectful) computation
6 Monadic DSLs

What’s a monad?

Monad stances

State monads

Monads for DSLs
The State (Transformer) Monad

State manipulated by computation

Trivial computation (noop) just returning a result

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

(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 \( d \)

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)

instance Monad (State s) where
return \( x \) = State (\( s \rightarrow (x,s) \))
State \( c \) >>= \( f \) = State (\( s \rightarrow \text{let} \ (x,s') = c \ s \text{ in} \ d \ x \ s' \))

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

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

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

Monadic Counting

\[
\text{evalC} :: \text{Counter} \rightarrow \text{Exp} \rightarrow (\text{Int, Counter})
\]

\[
\text{evalC} \ n \ (N \ i) = (i, n)
\]

\[
\text{evalC} \ n \ (\text{Plus } e \ e') = (i + j, o + 1)
\]

\[
\text{where } (i, m) = \text{evalC} \ n \ e
\]

\[
(j, o) = \text{evalC} \ m \ e'
\]

\[
\text{evalC} \ n \ (\text{Neg } e) = (-i, m + 1)
\]

\[
\text{where } (i, m) = \text{evalC} \ n \ e
\]

\[\text{runWith } 0 \$ \text{seval } $ \text{Plus } (N \ 9) \ (\text{Neg } (\text{Plus } (N \ 3) \ (N \ 4)))$ \ (2, 3)\]

State monad values are functions that need to have an effect

\[\text{seval} :: \text{Exp} \rightarrow \text{Count Int}\]

\[
\text{seval} \ (N \ i) = \text{return } i
\]

\[
\text{seval} \ (\text{Plus } e \ e') = \text{do } i \leftarrow \text{seval } e
\]

\[
j \leftarrow \text{seval } e'
\]

\[
\text{incCounter} \quad \rightarrow \quad \text{return } (i + j)
\]

\[
\text{seval} \ (\text{Neg } e) = \text{do } i \leftarrow \text{seval } e
\]

\[
\text{incCounter} = \text{return } (-i)
\]
**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

```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]
x : xs >>= f = concat (map f xs)
```

Computations that can fail

Non-deterministic 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