Monads

Monads are just monoids in the category of endofunctors.

What's the problem?
Outline

• What is a monad?
  • Box metaphor
  • Do notation, relationships, laws
  • Label metaphor
  • Computation metaphor
  • The IO monad
  • MonadPlus
Monad myths

Monads …

• are impure
• depend on laziness
• provide a “back-door” to perform side-effects
• are an imperative language inside Haskell
• require knowing abstract mathematics
• are about effects
• are about state
• are about IO

} monads are used for all of these, but it's not what they're about

http://dev.stephendiehl.com/hask/#monadic-myths
So what is a monad?

Just another abstraction over types … like Functor, Foldable, …

Specifically:

• a *parameterized data type*
• with *two operations* (that satisfy *three laws*)

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

In fact, you know a couple monads already! \([a]\) and Maybe \(a\)
Structuring effects

One of the main motivations for the monad “pattern”

What is an effect?
- failure
- exceptions
- nondeterminism
- context
- tracing
- state
- input/output
- ...

Effects in FP – lots of boilerplate
- check failure in each function
- pass context to each function
- thread state through functions
- ...

The monad pattern provides a way to write the boilerplate only once (in the Monad instance)
Monad metaphors

These are just metaphors … be wary of over applying them!
Outline

• What is a monad?

• **Box metaphor**

• Do notation, relationships, laws

• Label metaphor

• Computation metaphor

• The IO monad

• MonadPlus
Box metaphor

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

Repackaging: b >>= f
1. Open box b to access content x
2. Generate new box(es) from content using f, i.e. f x
3. Combine boxes into one result box
Maybe monad: a “possibly empty” box

Useful for managing failure

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

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

creates only one box (no step 3 needed)

empty box stays empty (we have nothing to generate new boxes)
List monad: a “collection” box

Useful for managing variation/nondeterminism

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

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

create a new box for each element (step 2)

combine boxes into one result box (step 3)
Outline

- What is a monad?
- Box metaphor
- **Do notation, relationships, laws**
- Label metaphor
- Computation metaphor
- The IO monad
- MonadPlus
Syntactic sugar: do notation

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

  (>>) :: Monad m => m a -> m b -> m b
  m >> n = m >>= \_ -> n
```

```
m >> n  
  ==>
  do { m; n }
```

```
m >>= (\x -> ... x ...)  
  ==>
  do { x <- m; ... x ... }
```

With layout:
```
do m
  n
```
```
do x <- m
  ... x ...
```
Relationship to Applicative

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

class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do f <- mf
                   a <- ma
                   return (f a)
```

Every monad is an applicative functor!
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Functor t where
  fmap :: (a -> b) -> t a -> t b

  liftM :: Monad m => (a -> b) -> m a -> m b
  liftM f m = m >>= return . f

fmap <=> liftM

Every monad is a functor!
Monad laws

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

left identity
  return a >>= f  <=>  f a

right identity
  m >>= return  <=>  m

associativity
  (m >>= f) >>= g  <=>  m >>= (\x -> f x >>= g)
Outline

• What is a monad?
• Box metaphor
• Do notation, relationships, laws
• **Label metaphor**
• Computation metaphor
• The IO monad
• MonadPlus
Label metaphor

1. Take label off of \( l \) to reveal item \( x \)
2. Generate new labeled item(s) using \( f \), i.e. \( f \ x \)
3. Combine old label and new labeled items into one labeled item

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

(Logging.hs)
Logging monad

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

data Log a = L String a

instance Monad Log where
  return x = L "" x
  L s x >>= f = let (L t y) = f x
                   in L (s ++ t) y

log :: String -> Log ()
log s = L s ()
class Applicative m => Monad m where

  return :: a -> m a

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

instance Monoid w => Monad (Writer w) where

  return x = W mempty x

  W s x >>= f = let (W t y) = f x
                in W (mappend s t) y

tell :: w -> Writer w ()
tell s = W s ()
Outline

• What is a monad?
• Box metaphor
• Do notation, relationships, laws
• Label metaphor

• **Computation metaphor**
• The IO monad
• MonadPlus
**Effectful computation metaphor**

- Trivial computation: just return result
- “Threading”:

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

**Kinds of effects that can occur**

**Threading:** \( c >>= f \)

Build a computation that:

1. Runs computation \( c \) to produce intermediate result \( x \)
2. Generates new computation \( d \) using \( f \), i.e., \( f \ x \)
3. Runs computation \( d \)

(StateCalc.hs)
Reader monad

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

data Reader r a = R (r -> a)

instance Monad (Reader r) where
  return x = R (\r -> x)
  R c >>= f = R $ \r ->
      let x   = c r
          R d = f x
      in d r

ask :: Reader r r
ask = R id
...
State monad

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

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

instance Monad (State s) where
  return x = S (\s -> (x,s))
  S c >>= f = S (\s -> let (x,t) = c s in d t)
  where
    S d = f x

put :: s -> State s ()
put s = S (\_ -> (((),s)))

get :: State s s
get = S (\s -> (s,s))

run original computation
build new computation
run new computation
Writer vs. State

```
data Writer w a = W w a

data State s a = S (s -> (a,s))
```

```
eval :: Expr -> Writer w Int
eval (Add l r) = liftM2 (+) (eval l) (eval r)
```

- eval \( l \) and eval \( r \) independently, return result and accumulated \( w \)'s

```
eval :: Expr -> State s Int
eval (Add l r) = liftM2 (+) (eval l) (eval r)
```

- eval \( l \) with \( s_0 \), then eval \( r \) with \( s_1 \), return result and \( s_2 \)
Outline

• What is a monad?
• Box metaphor
• Do notation, relationships, laws
• Label metaphor
• Computation metaphor
• The IO monad
• MonadPlus
Interacting with the “real world”

Remember, functions in Haskell are pure:
• always return same output for same inputs
• don't do anything else (no “side effects”)

So how do we do we implement this in Haskell?

```c
int confirm() {
    char c;
    printf("Are you sure? [y/n] ");
    c = getchar();
    if (c == 'y')
        return 1;
    return 0;
}
```

What we need (not pure):

```haskell
getChar :: () -> Char
putStrLn :: String -> ()
```
**IO monad, conceptually**

Idea: make the “real world” explicit

```haskell
putStrLn :: String -> RealWorld -> ((), RealWorld)
getchar  :: RealWorld -> (Char, RealWorld)
```

```haskell
data IO a = IO (RealWorld -> (a, RealWorld))
```

But this representation is hidden!

Can never get a value of type `RealWorld` ...
Can only interact with it through the IO monad

```
instance Monad IO where
  return a = ...
  io >>= f = ...
```

return value without changing real world

“thread” real world through computations
Using the IO monad

```c
int confirm() {
    printf("Are you sure? [y/n]";);
    char c = getchar();
    if (c == 'y')
        return 1;
    return 0;
}
```

```haskell
confirm :: IO Bool
confirm = do
    putStrLn "Are you sure? [y/n]"
    c <- getChar
    return (c == 'y')
```

System.IO has many more functions!
IO best practices

Once you’re in \texttt{IO} you’re stuck!

Basic principles:
- maximize IO-free code
- keep IO \textit{small} and \textit{focused}

Creating an executable: \texttt{main} is an IO action
- can still follow the principles above
- read inputs, pass to pure code, write outputs

\texttt{main :: IO ()
main = ...}

\textit{interacts w/ real world}

can call pure code, but can’t return pure values

simpler, more compositional
… advantages of FP
Final thoughts on the IO monad

Metaphors for a value of type `IO a`:

• an effectful computation where the “real world” is threaded behind the scenes
• a value representing a sequence of IO actions to be executed by the Haskell runtime system

What have we gained?

• clear separation of code that depends on the outside world (impossible to get out of IO monad)
Outline

• What is a monad?
• Box metaphor
• Do notation, relationships, laws
• Label metaphor
• Computation metaphor
• The IO monad
• Alternative and MonadPlus
class Applicative t => Alternative t where
  empty :: t a
  ( <|> ) :: t a -> t a -> t a

identity
“or”
empty and <|> form a monoid for any type t a

empty <|> x <=> x
x <|> empty <=> x
(x <|> y) <|> z <=> x <|> (y <|> z)
Alternative instances

class Applicative t => Alternative t where
  empty :: t a
  ( <|> ) :: t a -> t a -> t a

instance Alternative Maybe where
  empty = Nothing
  Just a <|> _ = Just a
  Nothing <|> mb = mb

instance Alternative [] where
  empty = []
  ( <|> ) = (++)
MonadPlus

monads that produce monoids—or—monads that support failure and choice

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
  mzero = empty
  mplus = (<>|>
```
Guards

Fail immediately if argument is False

```haskell
guard :: Alternative m => Bool -> m ()
guard True  = pure ()
guard False = empty
```

```haskell
divAll :: [Int] -> [Int] -> [Int]
divAll xs ys = do
  x <- xs
  y <- ys
  guard (y /= 0)
  return (x `div` y)
```

```haskell
>>> divAll [4,9,12] [2,0,3]
[2,1,4,3,6,4]
```