Monads

• “Just a monoid in the category of endofunctors”

• Famous phrase in “Categories for the Working Mathematician”

  • The worst book to learn category theory from

  • The book I learned category theory from

• Fictionally attributed to Philip Wadler in *Brief, Incomplete and Mostly Wrong History of Programming Languages*

• Actually hard to really understand — we’ll do a LOT of examples, in addition to mathematical definitions and intuitive descriptions
Review of Functors and Applicatives

• Functors

\[ \text{fmap} \quad a \rightarrow b + a \rightarrow b \]

• Applicatives

\[ \text{pure} \quad a \rightarrow b + a \rightarrow a \]

\[ <*> \quad a \rightarrow b + a \rightarrow b \]

• The square denotes a type constructor (functor or applicative)

• Intuitively, the square means a box, container, structure, or context, …
Motivating Example: SafeDiv

• Given the following definition of Expr

```
data Expr = Val Int | Div Expr Expr deriving (Show)
```

• To handle the potential failure properly, define a safediv function

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv x y = Just (div x y)
```

• Write a function to evaluate an expression

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
    Nothing -> Nothing
    Just n  -> case eval y of
                Nothing -> Nothing
                Just m  -> safediv n m
```
Motivating Example: SafeDiv

- The eval function resolves the “divide by zero” issue, but in a verbose way

- Let’s try to improve it using applicatives (because Maybe is an applicative)

  ```haskell
  eval :: Expr -> Maybe Int
  eval (Val n)   = pure n
  eval (Div x y) = pure safediv <*> eval x <*> eval y
  ```

- The above code does not type check
  - Because it requires safediv to have type Int -> Int -> Int
  - But then safediv cannot indicate failure …

- The eval function does not fit the pattern captured by applicatives
Motivating Example: SafeDiv

• Observe the common pattern in the verbose eval where we need a case analysis on a Maybe value (Nothing to Nothing, Just x to sth about x)

• Let’s define a function (\(\gg\gg\)) for this pattern

\[
\text{\(\gg\gg\)} :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b
\]

\[
x \gg\gg f = \text{case } x \text{ of }
\]

\[
\text{Nothing } \rightarrow \text{Nothing}
\]

\[
\text{Just } x \rightarrow f x
\]

• Then the eval function becomes

\[
\text{eval} :: \text{Expr} \rightarrow \text{Maybe } \text{Int}
\]

\[
\text{eval } (\text{Val } n) = \text{Just } n
\]

\[
\text{eval } (\text{Div } x y) = \text{eval } x \gg\gg \text{ \(\backslash n \rightarrow\)}
\]

\[
\text{eval } y \gg\gg \text{ \(\backslash m \rightarrow\)}
\]

\[
\text{safediv } n m
\]
Monad Definition

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

return = pure

• To be a Monad, you must be an Applicative

  • (To be an Applicative, you must be a Functor)

• return is the same as pure — turns a value into a monadic value

• (>>=) or bind takes a monadic value and feeds it into a function that takes a normal value, but returns a monadic value. This ultimately returns a monadic value
Motivating Example: SafeDiv

definitions

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

\[
\begin{aligned}
\text{return} &:: a \rightarrow m a \\
(\ggg>) &:: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\end{aligned}
\]

\[
\text{return} = \text{pure}
\]

\[
(\ggg>) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b
\]

\[
\begin{aligned}
mx \ggg> f &= \text{case } mx \text{ of} \\
\quad \text{Nothing} &\rightarrow \text{Nothing} \\
\quad \text{Just } x &\rightarrow f x
\end{aligned}
\]

\[
\text{eval} :: \text{Expr} \rightarrow \text{Maybe } \text{Int}
\]

\[
\begin{aligned}
\text{eval } (\text{Val } n) &= \text{Just } n \\
\text{eval } (\text{Div } x y) &= \text{eval } x \ggg> \text{Just } n \rightarrow \\
&\quad \text{eval } y \ggg> \text{Just } m \rightarrow \\
&\quad \text{safediv } n m
\end{aligned}
\]
Motivating Example: SafeDiv

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

  return = pure

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = bind
  (eval x)
  (\n ->
    bind
      (eval y)
      (\m -> safediv n m))
Monad Laws

• Monads

\[
\text{return}
\]

\[
\text{a} \rightarrow \text{b} \rightarrow \text{b}
\]

• A type constructor is a monad if it is an instance of the Monad type class that satisfies the following laws

- (Left identity) \( \text{return } x >>= f = f x \)
- (Right identity) \( \text{mx >>= return} = \text{mx} \)
- (Associativity) \( (\text{mx >>= f}) >>= g = \text{mx >>= (x -> (f x >>= g))} \)
Monad Laws

\[ \text{a} \rightarrow \text{b} \]

Functor Stuff

Monad Stuff
eval :: Expr -> Maybe (Maybe (Maybe Int))

eval (Val n) = Just n

eval (Div x y) = fmap

    (\n ->
        fmap
            (\m -> safediv n m)
            (eval y))
    (eval x)
Maybe Monad Instance

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

```
instance Monad Maybe where
  return x = Just x
  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
```
List Monad

instance Functor [] where
  fmap f [] = []
  fmap f (h:t) = h:(fmap f t)

instance Monad [] where
  return x = [x]
  (>>=) Nothing f = []
  (>>=) (h:t) f = (f h) ++ (t >>= f)

--(>>=) l f = concat (fmap f l)
data Expr = Val Int | OneOf (Expr,Expr) | Plus (Expr,Expr)

evaluate :: Expr -> [Int]
evaluate (Val i) = return i
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) =
  evaluate e1 >>= \i1 ->
  evaluate e2 >>= \i2 ->
  return (i1 + i2)

instance Monad [] where
  return x = [x]

  (>>=) Nothing f = []
  (>>=) (h:t) f = (f h) ++ (t >>= f)
A Common Monad Pattern

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= \n -> eval y >>= \m -> safediv n m

evaluate :: Expr -> [Int]
evaluate (Val i) = return i
evaluate (OneOf (e1, e2)) = (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1, e2)) = evaluate e1 >>= (\i1 -> evaluate e2 >>= (\i2 -> return (i1 + i2)))
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = do
  n <- eval x
  m <- eval y
  safediv n m

evaluate :: Expr -> [Int]
evaluate (Val i) = return i
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) = do
  i1 <- evaluate e1
  i2 <- evaluate e2
  return (i1 + i2)
ErrJst Monad Instance

data ErrJst e a =
  Err e
  | Jst a

instance Monad ErrJst where
  return x = Jst x

  (>>>=) (Err e) _ = Err e
  (>>>=) (Jst x) f = f x
ErrJst Usage

define eval :: Expr -> ErrJst String Int

\[
\text{eval } (\text{Val } n) = \text{Jst } n \\
\text{eval } (\text{Div } x \ y) = \text{do} \\
\quad n \leftarrow \text{eval } x \\
\quad m \leftarrow \text{eval } y \\
\quad \text{case safediv } n \ m \ \text{of} \\
\quad \quad \text{Nothing } \rightarrow \text{Err } (\text{printf } \text{“Divided } %d \text{ by zero” } n) \\
\quad \quad \text{Just } v \rightarrow \text{Jst } v
\]