Haskell: Monads

Subhajit Roy

April 8, 2019

Notion of monads

- Separate the “impure” world from the “pure” world

Notion of monads

- Separate the "impure" world from the "pure" world
- The rules of the game:
  - The impure world cannot interact directly with the pure world
  - Pure values can enter the impure world but cannot escape it

Notion of monads

- Separate the "impure" world from the "pure" world
- The rules of the game:
  - The impure world cannot interact directly with the pure world
  - Pure values can enter the impure world but cannot escape it
- We "wrap" the result of computation and impure side effects in an monadic object

The pure world

- Primary pillars of functional programming
  - Function application
    - (f x)

The pure world

- Primary pillars of functional programming
  - Function application
    - (f x)
  - Function composition
    - ((f. g) x) or (g (f x))

The pure world

- Primary pillars of functional programming
  - Function application
    - (f x)
  - Function composition
    - ((f. g) x) or (g (f x))
  - All functions have a single argument (made possible via currying)

The pure world

- Primary pillars of functional programming
  - Function application
    - (f x)
  - Function composition
    - ((f. g) x) or (g (f x))
  - All functions have a single argument (made possible via currying)
- The above three properties support the *pipelined* view of computation

$$\boxed{\text{input} \Longrightarrow f_1 \Longrightarrow f_2 \Longrightarrow \ldots \Longrightarrow f_n \Longrightarrow \text{output}}$$

Now, the impure world...

- We would like to have the same flavor of computation in the impure world...

Now, the impure world...

- We would like to have the same flavor of computation in the impure world...
- but, also establish that the impure world remains *isolated*

Now, the impure world...

- We would like to have the same flavor of computation in the impure world...
- but, also establish that the impure world remains *isolated*
- It has the following difficulties:
    - The computations will have to be done on pure values (the monadic values are essentially boxed instances, and computations can only be on what is contained in the box)

Now, the impure world...

- We would like to have the same flavor of computation in the impure world...
- but, also establish that the impure world remains *isolated*
- It has the following difficulties:
  - The computations will have to be done on pure values (the monadic values are essentially boxed instances, and computations can only be on what is contained in the box)
  - The result of any computation in the monadic world must return a monadic value (so that we cannot escape to the pure world)

Now, the impure world...

- We would like to have the same flavor of computation in the impure world...
- but, also establish that the impure world remains *isolated*
- It has the following difficulties:
  - The computations will have to be done on pure values (the monadic values are essentially boxed instances, and computations can only be on what is contained in the box)
  - The result of any computation in the monadic world must return a monadic value (so that we cannot escape to the pure world)
- So, the type of any function operating within the monadic world must look like:
  - f: $a \rightarrow m\ a$ (where m is a monadic type)

... completing the picture

- To enable a computation pipeline as in the pure world, we would need:
  - an ability to execute a monadic function
  - bind (>>=) operation
  - $>>=\ :\ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

… completing the picture

- To enable a computation pipeline as in the pure world, we would need:
  - an ability to execute a monadic function
  - bind (>>=) operation
  - $>>=\ :\ m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$$\boxed{\text{monadic i/p} >>= f_1 >>= f_2 >>= \ldots >>= f_n >>= \text{monadic o/p}}$$

- To enable a computation pipeline as in the pure world, we would need:
  - an ability to execute a monadic function
  - bind (>>=) operation
  - $>>= : \ m\ a \to (a \to m\ b) \to m\ b$

$$\boxed{\text{monadic i/p} >>= f_1 >>= f_2 >>= \ldots >>= f_n >>= \text{monadic o/p}}$$

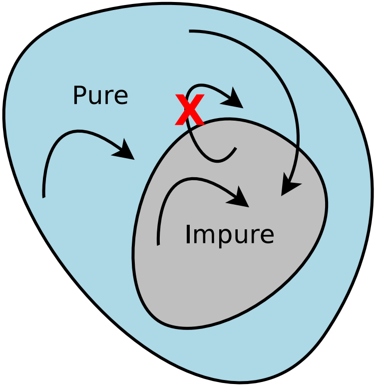
- Enable pure values to enter the monadic world
  - return : $a \ \to m\ a$

The Monad typeclass

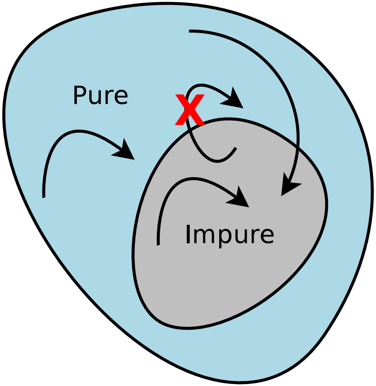
```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

- return and (>>=) are the interface methods to be defined by any monad; the other two you get for free.

Summary

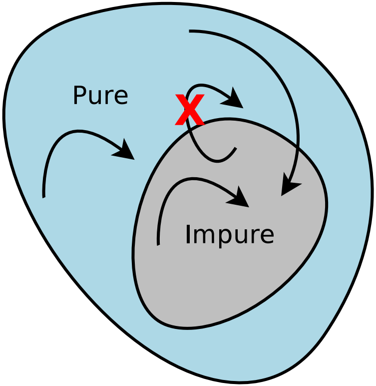


Summary



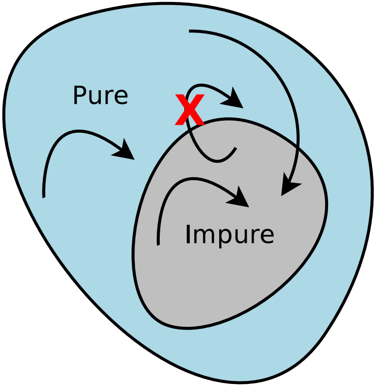
- pure → pure: function application (f a)

Summary



- pure $\rightarrow$ pure: function application (f a)
- impure $\rightarrow$ impure: bind (f >>= a)

Summary



- pure → pure: function application (f a)
- impure → impure: bind (f >>= a)
- pure → impure: return (return a)

The Maybe monad

```haskell
-- define the Maybe type
data Maybe a = Just a | Nothing

-- add Maybe type as an instance of the Monad typeclass
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
  fail _ = Nothing
```

The Maybe monad

```
-- define the Maybe type
data Maybe a = Just a | Nothing

-- add Maybe type as an instance of the Monad typeclass
instance Monad Maybe where
  return x = Just x
  y >>= f = case y of
    Just x -> f x
    Nothing -> Nothing
  fail _ = Nothing
```

What was the other option without monads?

```
data Maybe a = Just a | Nothing (* define the Maybe type *)

addMaybes :  Maybe a -> Maybe a
addMaybes x = case x of
  Just y -> Just y + 1
  Nothing -> Nothing

subMaybes :  Maybe a -> Maybe a
subMaybes x = case x of
  Just y -> Just y - 1
  Nothing -> Nothing
```

Notice the pattern...

```
f :  Maybe a -> a -> Maybe a
f x = case x of
  Just y -> f y
  Nothing -> Nothing
```

Notice the pattern...

```
f :  Maybe a -> a -> Maybe a
f x = case x of
  Just y -> f y
  Nothing -> Nothing
```

and, compare with the definition of bind:

Notice the pattern...

```
f :  Maybe a -> a -> Maybe a
f x = case x of
  Just y -> f y
  Nothing -> Nothing
```

and, compare with the definition of bind:

```
instance Monad Maybe where
  .  .  .
  x >>= f = case x of
    Just y -> f y
    Nothing -> Nothing
```

Notice the pattern...

```
f :  Maybe a -> a -> Maybe a
f x = case x of
  Just y -> f y
  Nothing -> Nothing
```

and, compare with the definition of bind:

```
instance Monad Maybe where
 .  .  .
  x >>= f = case x of
    Just y -> f y
    Nothing -> Nothing
```

That is, we simply give the pattern a shortcut: `x >>= f`

Notice the pattern...

```
addMaybes x = x >>= \x -> (x+1)
subMaybes x = x >>= \x -> (x-1)
```

Notice the pattern...

```
addMaybes x = x >>= \x -> (x+1)
subMaybes x = x >>= \x -> (x-1)
```

... and the type is **explicit** of the effect (i.e. like the function can fail)

Syntax Sugar

```
x >>= f >>= g >>= h
```

is same as:

```
do
  y1 <- x
  y2 <- (f y1)
  (h y2)
```

Syntax Sugar

```
x >>= f >>= g >>= h
```

is same as:

```
do
  y1 <- x
  y2 <- (f y1)
  (h y2)
```

Note: Unlike pure computations, monadic computations happen in the given **sequence** (to keep an consistent effect)

IO monad

```
getChar ::  IO Char
putChar ::  Char -> IO ()
getLine :: IO String
putStr :: String -> IO ()
putStr :: String -> IO ()

main ::  IO ()
main = do
  c <- getChar
  putChar c
```

Common mistakes

`let name = getLine`

versus

do

name <- getLine

Reading multiple lines

```
main = do
  line <- getLine
  if null line then
  return ()
  else do
    putStrLn (purefun line)
  main
```

printing values

```
main = do
  print 42
  print True
  print "Hello"
  print [32,14,31]
  print 3.2

print ::  Show a => a -> IO ()
```

Higher order functions

```
mapM ::  Monad m => (a -> m b) -> [a] -> m [b]
mapM print
>> [1,2,3]

mapM_ ::  Monad m => (a -> m b) -> [a] -> m ()

fmap ::  (Functor f) => (a -> b) -> f a -> f b
fmap (+1) (Just 1)
>> Just 2
```

sequence :: [IO a] -> IO [a]

Monad Laws

- Left identity
  - return x >>= f $\equiv$ f x

Monad Laws

- Left identity
  - return x >>= f ≡ f x
- Right identity
  - m >>= return ≡ m

Monad Laws

- Left identity
  - return x >>= f ≡ f x
- Right identity
  - m >>= return ≡ m
- Associativity
  - (m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)

Monad Laws

- Left identity
  - return x >>= f $\equiv$ f x
- Right identity
  - m >>= return $\equiv$ m
- Associativity
  - (m >>= f) >>= g $\equiv$ m >>= (\x -> f x >>= g)

Intuition

- Left and right identity say that entering the monadic space does not change the pure value in any way, other than wrapping it in a box.
- Associativity says that in monadic function, function composition (bind) associates the same way as in pure functions.

What is the output?

```
main = do
  x <- getLine
  let a = (read x ::  Int) in
    let y = case a of
          0 -> Nothing
          y -> Just y
    in print (do
          b <- y
          if (b == 20) then return "Hello" else return
"World")
```