Functional programming, Seminar No. 6

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ
Higher School of Economics
The Department of Computer Science

Intro

On the previous seminar

- we worked with computational contexts using such type classes as `Functor`, `Foldable`, and `Traversable`

Intro

On the previous seminar

- we worked with computational contexts using such type classes as `Functor`, `Foldable`, and `Traversable`

Today we

- dive into monads deeply

Monads

Motivation

We are going to extend pure functions `a -> b`, we would like to extend them to computations with effects:

- A computation may fail: `a -> Maybe b`
- A computation yields more than one result: `a -> [b]`
- A computation either succeeds or yields an error: `a -> Either e b`
- A computation with log: `a -> (s, b)`
- A computation with reading from an external environment `a -> (e -> b)`
- A computation with a mutable state: `a -> (State s) b`
- An input/output computation: `a -> IO b`
- A generalised version of such a function is a Kleisli function that has type `a -> m b`

Motivation

If one needs to provide a uniform interface to deal with Kreisli functions, then this interface should satisfy the following two requirements.

1. One needs to have an opportunity inject a pure value into the computational context
2. Kleisli maps should composable:

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

3. In general, we cannot extract `a` from `m a`

The definition of the `Monad` class

Let us take a look the full definition of the `Monad` class

```haskell
class Applicative m => Monad m where
  -- | Sequentially compose two actions,
  -- | passing any value produced
  -- | by the first as an argument to the second.
  (>>=) :: m a -> (a -> m b) -> m b

  -- | Sequentially compose two actions,
  -- |discarding any value produced by the first
  (>>) :: m a -> m b -> m b
  m >> k = m >>= \_ -> k

  -- | Inject a value into the monadic type.
  return :: a -> m a
  return = pure
```

The definition of the `Monad` class

The `Monad` class has the equivalent definition, the following one:

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

The definition of the `Monad` class

The `Monad` class has the equivalent definition, the following one:

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

Moreover, such a definition is much more closer to the original mathematical definition of a Monad.

The `return` function

One may make any pure function a Kleisli one:

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

cosM :: (Monad m, Floating b) => b -> m b
cosM = toKleisli cos
```

It is clear that $\cos \pi = -1$, but `cosM pi` has the type `(Monad m, Floating b) => m b` and we have several variants:

```
> piM :: Maybe Double
Just (-1.0)
> piM :: [Double]
[-1.0]
> piM :: IO (Double)
-1.0
> piM :: Either String Double
Right (-1.0)
```

The monadic bind operator

Take a look at the monadic type signature closely:

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

If we erase all `m` we get a type of this reverse application operator:

```
(&) :: a -> (a -> b) -> b
x & f = f x
```

The monadic bind operator

One may observe this analogy, here are the direct `fmap`, `(<*>)` and flipped monadic bind:

```
fmap :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
flip (>>=) :: Monad m => (a -> m b) -> m a -> m b
```

Flipped `fmap`, flipped `(<*>)` and `(>>=)`:

```
flip fmap :: Functor f => f a -> (a -> b) -> f b
flip (<*>) :: Applicative f => f a -> f (a -> b) -> f b
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

The very first monad. The `Identity` type

Let us define the following new type

```haskell
{-# LANGUAGE DeriveFunctor #-}

newtype Identity a = Identity { runIdentity :: a }
  deriving (Show, Functor)

instance Applicative Identity where
  pure = Identity
  Identity f <*> Identity x = Identity (f x)

instance Monad Identity where
  Identity x >>= k = k x
```

This is the trivial example of a monad.

Playing with the `Identity` monad

Let us consider a quite trivial example of a Kleisli function

```
cosId, acosId, sinM
  :: Double -> Identity Double
cosId = Identity . cos
acosId = Identity . acos
sinM = Identity . sin
```

The example of composed action within a monad:

```
> runIdentity $ cosId pi >>= acosId
-1.0
> runIdentity $ cosId pi >>= acosId
3.141592653589793
> runIdentity $ cosId (pi/2) >>= acosId >>= sinM
1.0
```

In fact, `>>=` works here similarly to `(&)`.

Some of useful monadic functions

Let us take a look at some widely used and common monadic operations:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \x -> f x >>= g

join :: Monad m => m (m a) -> m a
join x = x >>= id

forever :: Applicative f => f a -> f b
forever a = let a' = a *> a' in a'
```

Monad laws

Any monad also has the following coherence conditions:

1. The left identity law:

   ```
   return a >>= k  = k a
   ```

2. The right identity law:

   ```
   m >>= return  = m
   ```

3. The monadic bind operation is associative:

   ```
   m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
   ```

4. There is the strong connection between the notions of monad and monoid, but we drop this connection.
5. Let us illustrate these laws with the `Identity` monad

The identity laws

According to the identity laws:

```
return a >>= k = k a
m >>= return = m
```

the `return` function is a sort of a neutral element:

```
> runIdentity $ cosId (pi / 4)
0.7071067811865476
> runIdentity $ return (pi / 4) >>= cosId
0.7071067811865476
> runIdentity $ cosId (pi / 4) >>= return
0.7071067811865476
```

The associativity law

The third monad law:

```
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

claims that the monadic bind is associative as follows:

```
> runIdentity $ cosId (pi/2) >>= acosId >>= sinM
1.0
> runIdentity $ cosId (pi/2) >>= (\x -> acosId x >>= sinM)
1.0
```

The associativity law

Let us take a look at these pipelines that are equivalent to each other:

```
go = cosId (pi/2) >>=
     acosId       >>=
     sinM

go2 = cosId (pi/2) >>= (\x ->
      acosId x     >>= (\y ->
      sinM y       >>= \z ->
      return z))
```

Monads and pseudoimperative programming

```
go2 = cosId (pi/2) >>= (\x ->
      acosId x     >>= (\y ->
      sinM y       >>= \z ->
      return z))
```

```
go2 = cosId (pi/2) >>= (\x ->
      acosId x     >>= (\y ->
      sinM y       >>= \z ->
      return (x, y, z)))
```

Wow, we invited imperative programming!

Monads and pseudoimperative programming

We may ignore one of results, if we are not interested in it:

```
go2 = let alpha = pi/2 in
      cosId alpha  >>= (\x ->
      acosId x     >>= (\y ->
      sinM y       >>
      return (alpha, x, y)))
```

do-Notation

In Haskell, one has a quite useful syntax sugar to write code within a monad in the imperative fashion.

do-expression

```
do { e1; e2 }
```

Unsugared version

```
e1 >> e2
```

do-expression

```
do { p <- e1; e2 }
```

Unsugared version

```
e1 >>= \p -> e2
```

do-expression

```
do { let v = e1; e2 }
```

Unsugared version

```
let v = e1 in do e2
```

do-Notation. Example

The example above:

```
go2 = let alpha = pi/2 in
      cosId alpha  >>= (\x ->
      acosId x     >>= (\y ->
      sinM y       >>
      return (alpha, x, y)))
```

might be written as follows using the do-notation sugar:

```
go2 = do
  let alpha = pi/2
  x <- cosId alpha
  y <- acosId x
  z <- sinM y
  return (alpha, x, y)
```

do-Notation. Example

Let us consider an example of a monadic function:

```
prodM :: Monad m => (a -> m b) -> (c -> m d)
  -> m (a, c) -> m (b, d)
prodM f g mp =
  mp >>= \(a,b) -> f a >>= \c -> g b >>= \d ->
  return (c, d)
```

The function above might be implemented as follows via the do-notation sugar

```
prodM :: Monad m => (a -> m b) -> (c -> m d)
  -> m (a, c) -> m (b, d)
prodM f g mp = do
  (a, b) <- mp
  c <- f a
  d <- g b
  return (c, d)
```

The Maybe monad

The Maybe monad

The `Maybe` data type is one of the simplest non-trivial monads. This monad represents an the simplest error effect. An error is represented with the `Nothing` flag

```
instance Monad Maybe where
  return = Just

  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x

  (Just _) >> a = a
  Nothing >> _ = Nothing
```

The Maybe monad. Example

```haskell
type Author = String
type Book = String
type Library = [(Author, Book)]

books :: [Book]
books = ["Faust", "Alice in Wonderland", "The Idiot"]

authors :: [Author]
authors = ["Goethe", "Carroll", "Dostoevsky"]

library :: Library
library = zip authors books
```

The Maybe monad. Example

```haskell
library' :: Library
library' = ("Dostoevsky", "Demons") :
  ("Dostoevsky", "White Nights") : library

getBook :: Author -> Library -> Maybe Book
getBook author library = lookup author library

getSecondbook, getLastBook :: Author -> Maybe Book
getFirstbook author = do
  let lib' = filter (\p -> fst p == author) library'
  book <- getBook author lib'
  return book

getLastBook author = do
  let lib' = filter (\p -> fst p == author) library'
  book <- getBook author (reverse lib')
  return book
```

The list monad

The list instance

- The `Monad` instance is the following one:

```
instance Monad [] where
  return x = [x]
  xs >>= k = concat (map k xs)
```

- The function `k :: a -> [b]` maps to the function that has type `[a] -> [b]`. We apply `k` to every element of `[a]` and concat all results.

List compeherension once more

The following functions are equivalent:

```haskell
cartesianProduct :: [a] -> [b] -> [(a, b)]
cartesianProduct xs ys =
  xs >>= \x -> ys >>= \y -> return (x, y)

cartesianProduct' :: [a] -> [b] -> [(a, b)]
cartesianProduct' xs ys = do
  x <- xs
  y <- ys
  return (x, y)

cartesianProduct'' :: [a] -> [b] -> [(a, b)]
cartesianProduct'' xs ys = [(x, y) | x <- xs, y <- ys]
```