Functional programming, Seminar No. 7

Daniel Rogozin
Institute for Information Transmission Problems, RAS
Serokell OÜ
Higher School of Economics
The Department of Computer Science

Today

We will study the following monads

Input/Output

The `IO` monad, motivation

- `IO a` is a type whose values are input/output actions that produce values of `a`
- Here are first examples of `IO` functions

```
getChar :: IO Char
getLine :: IO String
```

- In fact, these functions have the following types:

```
getChar :: RealWorld -> (RealWorld, Char)
getLine :: RealWorld -> (RealWorld, String)
```

- A philosophical question: what is `RealWorld`?

The approximate definition of `IO`

- `IO` is defined approximately as follows:

```
newtype IO a = IO (RealWorld -> (RealWorld, a))
```

- According to Hoogle, "`RealWorld` is deeply magical. It is primitive... We never manipulate values of type RealWorld... it's only used in the type system"
- That is, an engineer has no access to values of `RealWorld` and we cannot use the same `RealWorld` twice!

The `IO` as a Monad

The `Monad` instance (very roughly):

```
instance Monad IO where
  return x = IO $ \w -> (w, x)
  m >>= k = IO $ \w ->
    case m w of
      (w', a) -> k a w'
```

- An effect of every action occurs only once.
- Note that the order of effects matters!

Basic console input/output functions

- Input:

```
getChar :: IO Char
getLine, getContents :: IO String
```

- Output:

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

- Input/output:

```
interact :: (String -> String) -> IO ()
```

An example of IO

```haskell
main :: IO ()
main = do
  putStrLn "Hello, what is your name?"
  name <- getLine
  putStrLn $ "Hi, " ++ name
  putStrLn $
    "Gotta go, " ++ name ++
    ", have a nice day"
```

The getLine function closely

Let us take a look at the rough version of `getLine`:

```
getLine' :: IO String
getLine' = do
  c <- getChar
  case c == '\n' of
    True -> return []
    False -> do
      cs <- getLine'
      return (c : cs)
```

The putStr function

```
putStr' :: String -> IO ()
putStr' [] = return ()
putStr' (x : xs) = putChar x >> putStr' xs
```

Using `sequence_`:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

```
putStr'' :: String -> IO ()
putStr'' = sequence_ . map putChar
```

The `putStr` function

Using `sequence_` and `mapM_`:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f

putStr''' :: String -> IO ()
putStr''' = mapM_ putChar
```

Reader, Writer, and State

Reader

The `Reader` monad allows one to read values from an environment:

```
newtype Reader r a = Reader { runReader :: (r -> a) }

instance Monad (Reader r) where
  -- return :: a -> Reader r a
  return x = Reader $ const x

  -- (>>=) :: Reader r a -> (a -> Reader r b) -> Reader r b
  Reader f >>= k = Reader $ \e -> let v = runReader m e
                                  in runReader (k v) e
```

- Here `(>>=)` passes a given environment to both computations
- Useful functions:

```
ask   :: Reader e e
asks  :: (e -> a) -> Reader e a
local :: (e -> b) -> Reader b a -> Reader e a
```

Reader. An example

```haskell
data Environment = Environment { ids  :: [Int]
                               , name :: Int -> String
                               , near :: Int -> (Int, Int) }

inEnv :: Int -> Reader Environment Bool
inEnv i = asks (elem i . ids)

anyInEnv :: (Int, Int) -> Reader Environment Bool
anyInEnv (i, j) = inEnv i ||^ inEnv j

checkNeighbours :: Int -> Reader Environment (Maybe String)
checkNeighbours i =
  asks (`near` i) >>= \pair ->
  anyInEnv pair   >>= \res  ->
  if res
  then Just <$> asks (`name` i)
  else pure Nothing
```

Writer

- The `Writer` monad for computation with logs

```
newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid w => Monad (Writer w) where
  -- return :: a -> Writer w a
  return x = Writer (x, mempty)

  -- (>>=)
  --   :: Writer r a -> (a -> Writer r b) -> Writer r b
  Writer (x,v) >>= f = let (Writer (y, v')) = f x
                       in Writer (y, v `mappend` v')
```

- The useful combinators:

```
tell :: Monoid w => w -> Writer w ()
listen :: Monoid w => Writer w a -> Writer w (w, a)
pass :: Monoid w => Writer w (a, w -> w) -> Writer w a
execWriter :: Writer w a -> w
```

Writer. An example

```haskell
binPow :: Int -> Int -> Writer String Int
binPow 0 _      = return 1
binPow n a
  | even n    = binPow (n `div` 2) a >>= \b ->
                Writer (b * b, "Square " ++ show b ++ "\n")
  | otherwise =
      binPow (n - 1) a >>= \b ->
      Writer
        (a * b, "Mul " ++ show a ++ " and " ++ show b ++ "\n")
```

State

- The `State` monad for processing of mutable states

```haskell
newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where
  -- return :: a -> State s a
  return x = State $ \s -> (x, s)

  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  State act >>= f = State $ \s ->
    let (a, s') = act s
    in runState (k a) s'
```

- Useful functions:

```haskell
get       :: State s s
put       :: s -> State s ()
modify    :: (s -> s) -> State s ()
gets      :: (s -> a) -> State s a
withState :: (s -> s) -> State s a -> State s a
```

State. An example

```haskell
type Stack = [Int]

pop :: State Stack Int
pop = State $ \(x:xs) -> (x, xs)

push :: Int -> State Stack ()
push x = State $ \xs -> ((), x:xs)

stackOps :: State Stack Int
stackOps = pop >>= \x -> push 5 >> push 10 >> return x
```