Functional programming, Seminar No. 7

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

Introduction

On the previous seminar, we

- got acquainted with the notion of a monad as a uniform interface for pipelines of actions

Introduction

On the previous seminar, we

- got acquainted with the notion of a monad as a uniform interface for pipelines of actions

Today we

- study such monads as `IO`, `Reader`, `Writer`, and `State`

The Input/Output Monad

The problem of purity

- The purity of functions is rather a problem than advantage if we deal with input and output
- If input/output functions were pure, then they would yield the same value at the same point, since their behaviour is fully determined by the referential transparency principle.

The problem of purity

- Input/output functions are clearly impure. Here are some example of such functions

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

- In fact, these functions have types:

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

The problem of purity

- Input/output functions are clearly impure. Here are some example of such functions

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

- In fact, these functions have types:

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

- The philosophical question: what is `RealWorld`?

The approximate definition of `IO`

- The value of the `IO a` type is such value that can perform input/output actions
- The approximate Haskell implementation:

```haskell
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 the `RealWorld` and we cannot use the same `RealWorld` twice!

The IO as a Monad

- The rough Monad instance

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

- A side effect of every action occurs only onces
- The order of side effects is strictly determined

The basic console input/output function

- Input:

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

- Output:

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

- Input/output:

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

The example of `IO`

```haskell
main :: IO ()
main = do
  putStrLn "Hello, what is your name?"
  name <- getLine
  putStrLn $ "Hi, " ++ name
  putStrLn $
    "Now leave me alone, " ++ name ++
    ", I'm tired of you"
```

The `getLine` function closely

Let us take a look at the approximate `getLine` implementation:

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

The `putStr` functions closely

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

A more sophisticated version:

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

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

The `putStr` functions closely

A more sophisticated version of a more sophisticated version:

```
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
```

The Reader Monad

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

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

instance Monad (Reader r) where
  return = error "This is your homework"
  x >>= k = error "This is your homework"
```

- `return` yields an argument ingoring a given environment
- `(>>=)` passes a given environment to both computations
- The useful combinators:

```haskell
ask :: Reader r r
local :: (r -> r) -> Reader r a -> Reader r a
```

The Writer Monad

- The `Writer` monad with the logging features

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

instance Monoid w => Monad (Writer w) where
  return = error "This is your homework"
  (>>=)  = error "This is your homework"
```

- 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
```

The State Monad

- The `State` monad is a monad for a mutable state processing

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

instance Monad (State s) where
  return = error "This is your homework"
  (>>=)  = error "This is your homework"
```

- The useful combinators:

```haskell
get :: State s s
put :: s -> State s ()
```