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

Today we

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:

```
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, I'm tired of you"
```

The Reader Monad

The Writer Monad

The State Monad