

Лекция 11: монада IO и взаимодействие с внешним миром

Функциональное программирование на Haskell

Алексей Романов

5 мая 2023 г.

МИЭТ

- Среди многих монад стандартной библиотеки, **IO** играет особую роль.
- Значение типа **IO** а описывает вычисление с результатом типа **a** и побочными эффектами.
- В первую очередь имеются в виду ввод-вывод, как видно из названия, но также изменяемые переменные и т.д.

Программы командной строки

- Исполняемая программа в Haskell задаётся значением `main :: IO a` (почти всегда `IO ()`).
- Это аналог `int main(int argc, char** argv)` в C, `public static void main(char[] args)` в Java и т.д.
- Соответственно, есть доступ к аргументам командной строки и возможность указать код выхода:
- `System.Environment.getArgs :: IO [String]`. Там же есть функции для чтения переменных окружения.
- `System.Exit.exitWith :: ExitCode -> IO a`.

Интерактивные программы

- Из консоли можно читать текст, вводимый пользователем:
`getline ::`

Интерактивные программы

- Из консоли можно читать текст, вводимый пользователем:

`getLine :: IO String`

- И писать в неё:

`putStr, putStrLn ::`

Интерактивные программы

- Из консоли можно читать текст, вводимый пользователем:

```
getLine :: IO String
```

- И писать в неё:

```
putStr, putStrLn :: String -> IO ()
```

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

Интерактивные программы

- Из консоли можно читать текст, вводимый пользователем:

```
getLine :: IO String
```

- И писать в неё:

```
putStr, putStrLn :: String -> IO ()
```

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

- Простой пример: читаем любую строку, выводим её длину и повторяем (а на "quit" выходим):

Интерактивные программы

- Из консоли можно читать текст, вводимый пользователем:

```
getLine :: IO String
```

- И писать в неё:

```
putStr, putStrLn :: String -> IO ()
```

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

- Простой пример: читаем любую строку, выводим её длину и повторяем (а на "quit" выходим):

```
main = do
  line <- getLine
  if line /= "quit"
    then do
      print (length line)
      main
    else
      pure ()  -- или exitSuccess
```


Работа с файлами

- Модуль `System.IO`.
- Напомню, `type FilePath = String`.
- Простейшие операции это чтение и запись в файл:

```
readFile :: FilePath -> IO String
```

```
writeFile, appendFile ::  
    FilePath -> String -> IO ()
```

- Или можно открыть файл один раз, получить `Handle` (дескриптор файла) и работать с ним:

```
openFile, openBinaryFile ::
```

```
    FilePath -> IOMode -> IO Handle
```

```
hPutStrLn :: Handle -> String -> IO ()
```

```
hGetStr...
```

Работа с файлами

- Модуль `System.IO`.
- Напомню, `type FilePath = String`.
- Простейшие операции это чтение и запись в файл:
`readFile :: FilePath -> IO String`
`writeFile, appendFile ::`
 `FilePath -> String -> IO ()`
- Или можно открыть файл один раз, получить `Handle` (дескриптор файла) и работать с ним:
`openFile, openBinaryFile ::`
 `FilePath -> IOMode -> IO Handle`
`hPutStrLn :: Handle -> String -> IO ()`
`hGetStr...`
- Функции работы с консолью сводятся к ЭТИМ:
`getLine = hGetLine stdin`

Работа с файлами (2)

- В чём проблема с кодом вроде

do

```
file <- openFile path ReadWriteMode
```

```
...
```

```
hClose file
```

Работа с файлами (2)

- В чём проблема с кодом вроде

`do`

```
file <- openFile path ReadWriteMode
```

```
...
```

```
hClose file
```

- Что, если на каком-то из шагов случится исключение? Мы остаёмся с открытым файлом.
- Если это случится много раз, программа вылетит.

Работа с файлами (2)

- В чём проблема с кодом вроде

```
do
```

```
  file <- openFile path ReadWriteMode
```

```
  ...
```

```
  hClose file
```

- Что, если на каком-то из шагов случится исключение? Мы остаёмся с открытым файлом.
- Если это случится много раз, программа вылетит.
- Чтобы закрыть файл и при исключении:
`withFile` path `ReadWriteMode` \$ \file -> do
 ...
- Это частный случай функции `bracket`.

ЛЕНИВЫЙ ВВОД-ВЫВОД

- Функция `hGetContents` не читает содержимое файла, а сразу возвращает строку, по мере доступа к которой читается файл.
- С одной стороны, это хорошо: не нужно явно указывать, сколько читать.
- Но если закрыть файл до того, как он реально прочитан, строка закончится как если бы она дошла до конца файла:

`wrong = do`

```
  fileData <- withFile "test.txt" ReadMode hGetContents  
  putStr fileData
```

- Нужно использовать данные внутри `withFile`:

```
right = withFile "test.txt" ReadMode $ \file -> do  
  fileData <- hGetContents file  
  putStr fileData
```

ByteString и Text

- Мы знаем, что `String` занимает очень много места в памяти, что ухудшает и время работы.
- Вместо него есть `Data.ByteString.ByteString`, по сути представляющий собой массив байтов, и `Data.ByteString.Lazy.ByteString` как ленивый список таких массивов.
- Многие функции для работы с ними называются так же, как для `String`, но живут в соответствующих модулях (поэтому используется `import qualified`).
- Они годятся для бинарных данных (или ASCII, с помощью `Data.ByteString.Lazy.Char8`).
- Для текста аналогичный пакет `text` и типы в модулях `Data.Text.Lazy`.

Случайные значения

- Модуль `System.Random` содержит класс

- ```
class RandomGen g where
 next :: g -> (Int, g)
 split :: g -> (g, g)
 genRange :: g -> (Int, Int)
 genRange _ = (minBound, maxBound)
```

описывающий генераторы случайных `Int`.

- Есть тип `StdGen` и `instance RandomGen StdGen`.
- Глобальный генератор живёт в `IORef`:

```
getStdGen :: IO StdGen
```

```
setStdGen :: StdGen -> IO ()
```

```
newStdGen :: IO StdGen -- применяет split к getStdGen
```

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

- Типы `next` и `getStdRandom` могут напомнить про `State` и не зря!
- Реализуем `newStdGen` и `getStdRandom`.



## Случайные значения (ответ)

- `newStdGen = do`  
    `currGen <- getStdGen`  
    `let (newGen1, newGen2) = split currGen`  
    `setStdGen newGen2`  
    `pure newGen1`

## Случайные значения (ответ)

- `newStdGen = do`  
    `currGen <- getStdGen`  
    `let (newGen1, newGen2) = split currGen`  
    `setStdGen newGen2`  
    `pure newGen1`
- `getStdRandom f = do`  
    `currGen <- getStdGen`  
    `(result, newGen) = f currGen`  
    `setStdGen newGen`  
    `pure result`

## Случайные значения (ответ)

- `newStdGen = do`  
    `currGen <- getStdGen`  
    `let (newGen1, newGen2) = split currGen`  
    `setStdGen newGen2`  
    `pure newGen1`
- `getStdRandom f = do`  
    `currGen <- getStdGen`  
    `(result, newGen) = f currGen`  
    `setStdGen newGen`  
    `pure result`
- Можем выразить одно через другое?

## Случайные значения (ответ)

- `newStdGen = do`  
    `currGen <- getStdGen`  
    `let (newGen1, newGen2) = split currGen`  
    `setStdGen newGen2`  
    `pure newGen1`
- `getStdRandom f = do`  
    `currGen <- getStdGen`  
    `(result, newGen) = f currGen`  
    `setStdGen newGen`  
    `pure result`
- Можем выразить одно через другое?  
    `newStdGen = getStdRandom split`

## Случайные значения (2)

- Ещё один класс в `System.Random` описывает типы, для которых можно получить случайные значения, если есть ГСЧ:
- `class Random a where`  
    `randomR :: RandomGen g => (a, a) -> g -> (a, g)`  
    `random :: RandomGen g => g -> (a, g)`

Первый аргумент `randomR`: диапазон, в котором берутся значения.

- Опять видим тип, похожий на `State`.
- `randomRIO :: Random a => (a, a) -> IO a`  
    `randomIO :: Random a => IO a`

используют глобальный генератор.

- Если у нас есть значение типа IO а, можно ли превратить его просто в а?

## Побег из IO

- Если у нас есть значение типа `IO a`, можно ли превратить его просто в `a`?
- На самом деле есть функция `unsafePerformIO :: IO a -> a`.
- Но само название говорит о её небезопасности.
- Конкретное использование может быть и безопасным, но для этого надо знать довольно много о внутренностях Haskell.

## Побег из IO

- Если у нас есть значение типа `IO a`, можно ли превратить его просто в `a`?
- На самом деле есть функция `unsafePerformIO :: IO a -> a`.
- Но само название говорит о её небезопасности.
- Конкретное использование может быть и безопасным, но для этого надо знать довольно много о внутренностях Haskell.
- Поэтому в рамках этого курса мы её использовать не будем, как и другие `unsafe*IO`.
- А есть ещё `accursedUnutterablePerformIO...`