Входно-изходни операции в Haskell – част 1

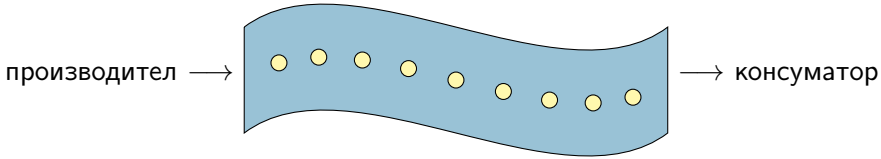
Трифон Трифонов

Функционално програмиране, 2025/26 г.

11 декември 2025 г.

Странични ефекти в Haskell

- Функциите в Haskell нямат странични ефекти
- Но входно-изходните операции по природа са странични ефекти!
- Как можем да се справим с този парадокс?
- **Идея:** Можем да си мислим за входно-изходните операции като поточна обработка на данни



Поточна обработка

Задача. Да се въведат `n` числа и да се изведе тяхното средно аритметично.
Решение: Дефинираме трансформация над стандартните вход и изход, която:

- приема `n` като параметър
- трансформира входния поток, като консумира от него `n` числа и връща списък, който ги съдържа
- пресмята средното аритметично `avg` на числата в списъка
- трансформира изходния поток, като произвежда върху него низовото представяне на числото `avg`

Трансформирането на входно-изходните потоци несъмнено е страничен ефект, но **конструирането на трансформацията** няма нужда от странични ефекти!
Функциите, които работят с вход и изход, по същество дефинират композиция на входно-изходни трансформации.

Типът IO a

Стандартният генеричен тип `IO a` задава тип на входно/изходна трансформация, резултатът от която е от тип a.
Частен случай: `IO ()` задава трансформация, която връща празен резултат.

Входни трансформации:

- `getChar :: IO Char` — връща символ, прочетен от входа
- `getLine :: IO String` — връща ред, прочетен от входа

Изходни трансформации:

- `putChar :: Char -> IO ()` — извежда символ на изхода
- `putStr :: String -> IO ()` — извежда низ на изхода
- `putStrLn :: String -> IO ()` — извежда ред на изхода

Главна функция main

- Функцията `main :: IO ()` от модула `Main` в Haskell е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме `main = putStrLn ("Въведохте: " ++ getLine)`?
- Не! `getLine :: IO String`
- Композицията на входно-изходни трансформации работи по различен начин от композицията на функции
- Низът, който връща `getLine` е „замърсен“ от входно-изходна операция
- Как да композираме трансформации?

Конструкцията do

В Haskell има специален двумерен синтаксис за композиране на трансформации:

`do { <трансформация> }`

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- <име> `<-` <трансформация>
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return` <израз>
  - празна трансформация, която връща <израз> като резултат
  - `return :: a -> IO a`
- резултатът от цялата конструкция do е резултатът от последната трансформация в композицията

```
main = do line <- getLine
          putStrLn ("Въведохте: " ++ line)
```

Локални дефиниции в do

В някакъв смисъл <- и return са обратни една на друга операции:

- <- извлича „чист“ резултат от тип a от трансформация от тип IO a
- return фиктивно „замърсява“ резултат от тип a за да стане от тип IO a
- Какъв е ефектът от <име> <- return <израз> в do конструкция?
- Създава се локалната дефиниция <име> = <израз>!
- Алтернативно, локални дефиниции могат да се създават и чрез: let <име> = <израз>
- Да не се бърка с let <име> = <израз> in <израз>!

Пример:

```
main = do putStrLn "Моля, въведете палиндром: "
          line <- getLine
          let revLine = reverse line
          if revLine == line then putStrLn "Благодаря!"
          else do putStrLn (line ++ " не е палиндром!")
                  main
```

Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: Show a => a -> String`
- `print :: Show a => a -> IO ()`
- `print = putStrLn . show`
- `read :: Read a => String -> a`
- `read "1.23"` $\longrightarrow$ Грешка!
- Haskell не може да познае типа на резултата, понеже е генеричен!
- `getInt :: IO Int`
- ```
  getInt = do line <- getLine
              return (read line)
  ```
  ```

Пример: средно аритметично на редица от числа

```
findAverage :: IO Double
findAverage = do putStr "Моля, въведете брой: "
                 n <- getInt
                 s <- readAndSum n
                 return (fromIntegral s / fromIntegral n)

readAndSum :: Int -> IO Int
readAndSum 0 = return 0
readAndSum n = do putStr "Моля, въведете число: "
                  x <- getInt
                  s <- readAndSum (n - 1)
                  return (x + s)

main = do avg <- findAverage
          putStrLn ("Средното аритметично е: " ++ show avg)
```

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
  - `type FilePath = String`
  - `data IOMode = ReadMode|WriteMode|AppendMode|ReadWriteMode`
- Има варианти на функциите за вход/изход, които работят с `Handle`
- `hGetLine`, `hGetChar`, `hPutStr`, `hPutStrLn`, `hGetContents`...
- **Пример:**

```
encrypt cypher inFile outFile =
  do h1 <- openFile inFile ReadMode
     text <- hGetContents h1
     h2 <- openFile outFile WriteMode
     hPutStr h2 (map cypher text)
```