I/O em Haskell

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```
import Data.Char

main :: IO ()
main = return ()
```

Antigamente...

- No princípio, programas eram *batch*: executados isoladamente, produziam resultado sem interação com seu ambiente de execução.

Dias atuais

- Programas atuais são interativos: respondem a comandos do usuário e a entradas provenientes de seu ambiente: rede, sensores, etc. . .
- Como representar isso em Haskell?

Transparência

- Haskell possui **transparência referencial**
- Sempre é possível substituir um termo por sua definição sem alterar seu significado.
  - Em linguagem imperativas, isso não é sempre verdade.

Exemplo

- Considere a expressão

```
reverse xs ++ xs
   where
     xs = filter p ys
```

essa expressão é igual a

```
reverse (filter p ys) ++ (filter p ys)
```

Transparência

- Separa o *significado* de uma expressão da forma como essa é executada.
- Compilador é livre para escolher a melhor forma para executar o código.
- Paralelismo torna-se muito mais simples.

Efeitos colaterais

- Interação com o "mundo-externo" não possui transparência referencial.
- Exemplo: função `getChar :: IO Char`, lê um caractere da entrada padrão.

Efeitos colaterais

- A função `getChar` é uma função impura pois não retorna o mesmo resultado a cada execução. Logo, a seguinte expressão não é verdadeira:

```
getChar == getChar
```

Efeitos colaterais

- Exemplos de efeitos colaterais:
  - Impressão no console.
  - Leitura / escrita de arquivos.
  - Comunicação com BDs e rede.
  - Atribuição.

Efeitos colaterais

- Efeitos colaterais podem alterar o "mundo-externo"
  - Como não podemos "duplicar" o ambiente de execução, não podemos substituir termos por sua definição.
- Problemas com código que possui efeitos colaterais.
  - Dificuldade para testar.
  - Dificuldade para paralelização e distribuição.

Efeitos colaterais

- Porém, a grande maioria de programas precisa de efeitos colaterais.
  - Em linguagens imperativas não há controle sobre como e onde efeitos podem ocorrer.
- Como podemos melhorar isso?
  - Impondo que efeitos colaterais só podem acontecer em certas partes de seu código.

Efeitos colaterais

- Como isolar efeitos colaterais?
  - Isolamos efeitos colaterais confinando-os em funções que usam / retornam valores do tipo do efeito.
- Para entrada e saída em Haskell, o tipo é `IO a`.

Modelando IO

- Funções de tipo IO podem:
  - Realizar uma operação de IO.
  - Retornar um resultado.
- O tipo IO deve ser dependente de uma representação do mundo "externo".

Modelando IO

- Uma possível representação de IO.

```
type IO a = World -> (a, World)
```

`World` é o tipo que representa o mundo externo.

Modelando IO

- O tipo IO recebe um valor `World` e retona um resultado de tipo `a` e um valor possivelmente modificado de `World`.
- Modificação: reflete a realização de um efeito de IO no valor World de entrada.

```
type IO a = World -> (a, World)
```

Primitivas de IO

- `getChar :: IO Char`: retorna o caractere lido da entrada padrão.
- `putChar :: Char -> IO ()`: imprime o caractere passado como argumento na entrada padrão.
  - Retorna `()`, que corresponde ao tipo `void` em linguagens imperativas.

Primitivas de IO

- Problema: Temos primitivas, mas como combiná-las?
- Precisamos de um operador que sequencia operações de IO.
  - Note que a ordem de execução de IO é importante.

Combinando IO

- Precisamos de uma função para combinar expressões de IO.
- Essa função já é definida da biblioteca de Haskell e é chamada de `bind`:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
m >>= f = \w -> let (v,w') = m w
                in f v w'
```

Echo!

```
echo :: IO ()
echo = getChar >>= \ c -> putChar c
```

getUpper

- Construir uma uma função que retorna um caractere digitado como uma letra maiúscula.

```
getUpper :: IO Char
getUpper = getChar >>= \ c -> toUpper c
```

getUpper

- Problema:

```
Couldn't match expected type IO Char
            with actual type Char
```

Qual o motivo desse erro?

getUpper

- Erro de tipo na expressão: `\ c -> toUpper c`.
  - Tipo esperado: `Char -> IO Char`
  - Tipo encontrado: `Char -> Char`
- Como converter um valor de tipo `Char` em `IO Char`?

Primitiva de IO

- Precisamos de mais uma primitiva para lidar com IO.

```
return :: a -> IO a
return x = \ w -> (x, w)
```

- Atenção: `return` não muda o fluxo de execução!

getUpper

- Corrigindo.

```
getUpper :: IO Char
getUpper
  = getChar >>= \ c -> return (toUpper c)
```

getLine

- Como definir uma função que lê uma string da entrada?
- Definindo o tipo:

```
getLine :: IO String
```

getLine

- Nota: Não podemos fazer casamento de padrão em valores do tipo `IO Char`.
- Devemos usar o operador >>= e então fazer o casamento sobre o caractere passado como parâmetro para a "continuação"

getLine

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

- Código com (»=) é ilegível...
- Alguma alternativa?

do-Notation

- Felizmente há uma notação para lidar com IO em Haskell.
- Agrupamos código de IO em blocos iniciados com a palavra reservada do
  - Operador <- dá um nome a um resultado de uma operação.

getLine

- Usando do-notation

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

do-Notation

- Equivalência entre do e »=:

```
do x1 <- a1
   x2 <- a2
   ...
   expr
```

é equivalente a

```
a1 >>= (\x1 -> a2 >>= (\x2 -> ... expr))
```

putStr

- Função para imprimir uma string no console.
  - Regra: Sempre que possível não use o (»=).

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

putStrLn

- Gera uma quebra de linha após a impressão de uma string.

```
putStrLn :: String -> IO ()
putStrLn s
   = do
       putStr s
       putChar '\n'
```

Resumo de IO

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b

getChar :: IO Char
putChar :: Char -> IO Char

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

Adivinhe o número

- Implementar um pequeno jogo para adivinhar um número.
- Interação feita via console.

Adivinhe o número

```
Escolha  número entre 1 and 100.
O número é 75? (g = maior, l = menor, c = correto)
l
O número é 62? (g = maior, l = menor, c = correto)
g
O número é 68? (g = maior, l = menor, c = correto)
l
O número é  65? (g = maior, l = menor, c = correto)
c
Acertou!
```

Adivinhe o número

- Algoritmo para adivinhar: busca binária.

```
guess :: Int -> Int -> IO ()
guess l u
  = do
      let m = (l + u) `div` 2
      putStr ("O número é " ++ show m ++ "?")
      putStrLn "(g = maior, l = menor, c = correto)"
      k <- readGuess
      case k of
        Less -> guess l (m - 1)
        Greater -> guess (m + 1) u
        Equal -> putStrLn "Acertou!"
```

First-class IO

- Tipos IO são cidadãos de 1a classe.
- Valores de tipo IO podem ser retornados como resultado, passados como argumentos e armazenados em estruturas de dados.

First-class IO

```
foo :: [IO ()]
foo = map (\ n -> putStrLn ("Hello " ++ s))
          ["João", "Maria"]
```

- Note que ao chamarmos `foo` no interpretador, obtemos uma mensagem de erro.

First-class IO

- Valores de tipo IO somente são executados em blocos `do`, dentro de outros valores de tipo IO.
- Como `foo` possui tipo `[IO ()]`, esse não é diretamente executável como uma operação de IO.

First-class IO

- Podemos definir uma função para sequenciar ações em uma lista.
- Definindo o tipo

```
sequence_ :: [IO a] -> IO ()
```

First-class IO

- Enumerando os casos

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = _
sequence_ (a : as) = _
```

First-class IO

- Definindo o caso base

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (a : as) = _
```

First-class IO

- Definindo o caso recursivo

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (a : as)
   = do
       a
       sequence_ as
```

Exemplo

- Implemente a função

```
sequence :: [IO a] -> IO [a]
```

que retorna a lista de resultados obtidos ao executar uma lista de operações de IO.

Solução

```haskell
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a : as)
   = do
       x <- a
       xs <- sequence as
       return (x : xs)
```

Arquivos

- Funções e tipos para manipular arquivos

```
type FilePath = String

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

- Note que o IO é feito por *lazy evaluation*, isto é, sob demanda.

Função main

- O início da execução de um programa Haskell se dá pela função

```
main :: IO ()
main = ...
```

que deve estar contida em um arquivo chamado Main.hs ou Main.lhs

Jogo da forca

- Outro exemplo, jogo da forca com base de palavras armazenadas em um arquivo de configuração.
- Arquivo Hangman.hs

Exercício

- Implemente um programa executável que produza estatísticas sobre um arquivo fornecido como entrada. As estatísticas produzidas devem ser o número de linhas e de palavras presentes no arquivo.