Programação Funcional
12ª Aula — Programas interativos

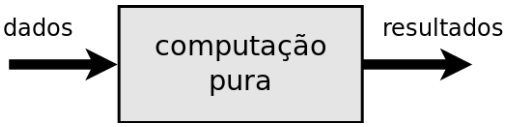
Sandra Alves
DCC/FCUP

2018/19

1 Programas interativos

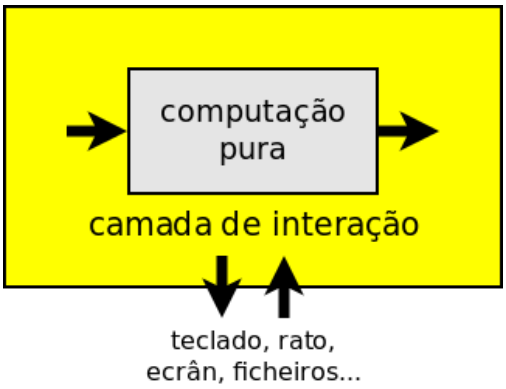
Motivação

Até agora apenas escrevemos programas que efetuam *computação pura*, i.e., transformações funcionais entre valores.



Vamos agora ver como escrever *programas interativos* i.e. que interagem com o mundo exterior:

- lêm informação do teclado, ficheiros, etc.;
- escrevem no terminal ou em ficheiros;
- ...



O tipo monádico IO

- Em Haskell, programas que produzem efeitos-colaterais (input/output, excepcções, estado, etc...), podem ser tratados usando a noção matemática de *monad*.
- Os monads representam computações: se `M` é um monad, então `M a` representa uma computação que produz um resultado do tipo `a`.

- O tipo monádico `IO` está definido no Haskell para lidar com computações que produzem input/output.
- O tipo `IO a`, representa o tipo das acções que retornam um valor do tipo a.
- O monad `IO` permite tratar de uma forma puramente funcional as acções de input/output.

Ações de I/O

Introduzimos um novo tipo `IO ()` para ações que, *se forem executadas*, fazem entrada/saída de dados.

Exemplos:

```
putChar 'A' :: IO ()                                    -- imprime um 'A'
putChar 'B' :: IO ()                                    -- imprime um 'B'

putChar :: Char -> IO ()                                -- imprimir um carater
```

Encadear ações

Podemos combinar duas ações de I/O usando o operador de *sequenciação*:

```
(>>) :: IO () -> IO () -> IO ()
```

Exemplos:

```
(putChar 'A' >> putChar 'B') :: IO ()                   -- imprimir "AB"
(putChar 'B' >> putChar 'A') :: IO ()                   -- imprimir "BA"
```

Note que `>>` é associativo mas não é comutativo!

Em alternativa podemos usando a notação-*do*:

```
  putChar 'A' >> putChar 'B' >> putChar 'C'
=
  do {putChar 'A'; putChar 'B'; putChar 'C'}
```

Podemos omitir os sinais de pontuação usando a indentação:

```
do putChar 'A'
   putChar 'B'
   putChar 'C'
```

Execução

Para efetuar as ações de I/O definimos um valor `main` no módulo `Main`.

```
module Main where

main = do putChar 'A'
          putChar 'B'
```

Compilar e executar:

```
$ ghc Main.hs -o prog
$ ./prog
AB$
```

Também podemos efetuar ações IO diretamente no `hugs/ghci`:

```
Prelude> putChar 'A' >> putChar 'B'
ABPrelude>
```

Definir novas ações

Vamos agora definir novas ações de I/O combinando ações mais simples.

Exemplo: definir `putStr` usando `putChar` recursivamente.

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

Como completar?

Ação vazia

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

`return ()` é a *ação vazia*: *se for efetuada*, não faz nada.

Mais geralmente

`IO` *a* é o tipo de ações que, *se forem executadas*, fazem entrada/saída de dados e *devolvem um valor de tipo a*.

Exemplos:

```
putChar 'A' :: IO ()                    -- escrever um 'A'; resultado vazio

getChar :: IO Char                      -- ler um caracter; resultado Char
```

Ações IO pré-definidas

```
getChar :: IO Char                      -- ler um caracter
getLine :: IO String                    -- ler uma linha
getContents :: IO String                -- ler toda a entrada padrão

putChar :: Char -> IO ()                -- escrever um carater
putStr :: String -> IO ()               -- escrever uma linha de texto
putStrLn :: String -> IO ()             -- idem com mudança de linha

print :: Show a => a -> IO ()           -- imprimir um valor

return :: a -> IO a                     -- ação vazia
```

Combinando leitura e escrita

Usamos `<-` para obter valores retornados por uma ação I/O.

Exemplo: ler e imprimir caracteres até obter um fim-de-linha.

```
main :: IO ()
main = do x<-getChar
          putChar x
          if x=='\n' then return () else main
```

Outro exemplo:

```
boasvindas :: IO ()
boasvindas = do putStr "Como te chamas? "
                nome <- getLine
                putStr ("Bem-vindo, " ++ nome ++ "!\n")
```

Valores de retorno

Podemos usar `return` para definir valores de retorno de ações.

```
boasvindas :: IO String
boasvindas
   = do putStr "Como te chamas? "
        nome <- getLine
        putStr ("Bem-vindo, " ++ nome ++ "!\n")
        return nome
```

Outro exemplo: definir `getLine` usando `getChar`.

```
getLine :: IO String
getLine = do x<-getChar
             if x=='\n' then
                return []
              else
                do xs<-getLine
                   return (x:xs)
```

Jogo *Hi-Lo*

Exemplo maior: um jogo de perguntas-respostas.

- o computador escolhe um número secreto entre 1 e 100;
- o jogador vai fazer tentativas de advinhar;
- para cada tentativa o computador diz se é *alto* ou *baixo*;
- a pontuação final é o número de tentativas.

```
Tentativa? 50
Demasiado alto!
Tentativa? 25
Demasiado baixo!
Tentativa? 35
Demasiado alto!
Tentativa? 30
Demasiado baixo!
Tentativa? 32
Acertou em 5 tentativas.
```

Vamos decompor em duas partes:

main escolhe o número secreto e inicia o jogo;

jogo função recursiva que efetua a sequência perguntas-respostas.

Programa

```
module Main where

import Data.Char(isDigit)
import System.Random(randomRIO)

main = do x <- randomRIO (1,100) -- escolher número aletório
          n <- jogo 1 x          -- começar o jogo
          putStrLn ("Acertou em " ++ show n ++
                    " tentativas")

jogo :: Int -> Int -> IO Int
jogo n x                                          -- n: tentativas, x: número secreto
  = do { putStr "Tentativa? "
       ; str <- getLine
       ; if all isDigit str then
           let y = read str in
           if y>x then
             do putStrLn "Demasiado alto!"; jogo (n+1) x
           else if y<x then
             do putStrLn "Demasiado baixo!"; jogo (n+1) x
                else return n
         else do putStrLn "Tentativa inválida!"; jogo n x
       }
```

Ações são valores

As ações IO são *valores de primeira classe*:

- podem ser argumentos ou resultados de funções;
- podem passados em listas ou tuplos;
- ...

Isto permite muita flexibilidade ao combinar ações.

Exemplo: uma função para efetuar uma lista de ações por ordem.

```
seqn :: [IO a] -> IO ()
seqn []     = return ()
seqn (m:ms) = m >> seqn ms
```

Exemplos de uso:

```
> seqn [putStrLn s | s<-["ola", "mundo"]]
ola
mundo
```

```
> seqn [print i | i<-[1..5]]
1
2
3
4
5
```

Ler e escrever para ficheiros

- Consideramos o tipo pré-definido
  `type FilePath = String`
- A função `writeFile fich texto`:
  `writeFile :: FilePath -> String -> IO ()`
  cria (ou substitui) o ficheiro `fich` com o conteúdo `texto`
- A função `appendFile fich texto`:
  `appendFile :: FilePath -> String -> IO ()`
  adiciona o conteúdo `texto` ao ficheiro `fich`
- A função `readFile fich`:
  `readFile :: FilePath -> IO String`
  lê o conteúdo do ficheiro `fich` e retorna-o como uma string

Sumário

- Programas reais necessitam de combinar *interação* e *computação pura*
- Em Haskell fica *explícito nos tipos* quais as funções que fazem interação e quais são puras.
- A notação-`do` e o tipo `IO` é usada para:
  - ler e escrever no terminal e em ficheiros;
  - estabelecer comunicações de rede;
  - serviços do sistema operativo (ex: obter data e hora do relógio de sistema);
  - A notação-`do` pode usada para outras *computações não puramente funcionais*:
    - estado, não-determinismo, exceções, etc.