Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Entrada e Saída

(COMP0393)

Aula 18

dcomp
departamento de computação

Introdução

- Sistemas computacionais interagem com o "mundo"
- Interações podem requerer
  - Sequencialidade
  - Mudanças no "mundo"
- Problemas
  - Programas funcionais não expressam ordem de avaliação e, ainda menos, sequencialidade
  - Com avaliação preguiçosa, a ordem não é fácil predizer somente olhando no texto do programa
  - Não existem efeitos colaterais

I/O com PF pura

- Um novo tipo de valores – chamados ações

```
IO a
```

- Uma ação realiza um efeito colateral e devolve um valor
- Ações acontecem "fora do mundo funcional"
  - Para uma função, uma ação é um valor constante
  - Funções não "executam" ações mas somente criam ações compostas a partir de ações mais simples
- Ações interagem "com o mundo funcional"
  - O valor devolvido pela execução de uma ação pode ser "capturado" e passado para uma função
- É como se tivéssemos uma linguagem acima de Haskell

Exemplos de ações

- Algumas ações pré-definidas

```
getLine :: IO String
getChar :: IO Char
```

- Funções "construtoras" de ações

```
putStr :: String → IO ()
```

Assim, por exemplo, `putStr "Oi pessoal"` é uma ação.

```
putStrLn :: String → IO ()
putStrLn = putStr . (++ "\n" )
```

```
print :: Show a => a → IO ()
print = putStrLn . show
```

Compondo funções construtoras de ações, posso definir outras

`print` imprime valores de vários tipos

Compondo ações com a notação do

- A notação `do` permite
  - "construir" sequências de ações
  - capturar um valor retornado por uma ação
    - o valor capturado é passado para as próximas ações

```
echo :: IO ()
echo = do str ← getLine
          putStr str
```

- Duas ações
- O valor produzido pela primeira ação é capturado em `str` e passado para a seguinte

- Observe que
  - `do` junta uma sequência de ações para formar uma nova ação
  - `do` permite captura

    Só é possível capturar e utilizar o resultado de uma ação dentro de um `do`
  - O tipo da ação composta `do ...` é dado pelo tipo da "última ação"

Exemplos

```
put3times :: String → IO ()
put3times str = do  putStrLn str
                    putStrLn str
                    putStrLn str
```

```
putNtimes :: Int → String → IO ()
putNtimes n str = if (n<=1)
                  then putStrLn str
                  else do putStrLn str
                          putNtimes (n-1) str

put3Time = putNTimes 3
```

Exemplos

Lê linhas sem capturá-las

```
read2lines :: IO ()
read2lines = do  getLine
                 getLine
                 putStr “Duas Linhas lidas."
```

A ação return

- Permite construir uma ação que simplesmente retorna um valor

```
getInt :: IO Int
getInt = do line ← getLine
            return (read line :: Int)
```

```
add2Ints :: IO ()
add2Ints = do  n1 ← getInt
               n2 ← getInt
               putStrLn (show (n1+n2))
```

Definições locais dentro de um do

```
add2Ints :: IO ()
add2Ints = do  n1 ← getInt
               n2 ← getInt
               let sum = n1+n2
               putStrLn (show sum)
```

Iteração (Loops)

```
copy :: IO ()
copy = do str ← getLine
          putStrLn str
          copy
```

- Note a recursão
- A captura é feita pelo "`str ←`"
- "`str ←`" lembra uma atribuição ... porém:
  - `str` não é variável imperativa
  - cria-se uma nova variável ("atribuição" *only once*)
- Para interromper usa Ctrl-C senão roda indefinidamente

Iteração (Loops)

- Podemos controlar o número de vezes a executar fazendo:

```
copyN :: Integer -> IO ()
copyN n = if n<=0
          then return ()
          else do str ← getLine
                  putStrLn str
                  copyN (n-1)
```

Iteração (Loops)

- Podemos controlar a terminação do laço com uma condição nos dados; copiar até uma linha vazia ser encontrada:

```
copyEmpty :: IO ()
copyEmpty= do str ← getLine
              if str == “”
              then return ()
              else do putStrLn str
                      copyEmpty
```

Iteração (Loops)

- Podemos contar o número de linhas copiadas até encontrar uma linha vazia:

```
copyCount :: Integer -> IO ()
copyCount n = do str ← getLine
                 if str == “”
                 then putStrLn (show n ++ “ linhas copiadas.”)
                 else do putStrLn str
                         copyCount (n+1)
```

A função deve ser chamada com `copyCount 0`

Iteração e recursão

Repetição de uma ação IO enquanto uma condição for verdadeira.

```
while :: IO Bool → IO () → IO ()
while test action = do res ← test
                       if res then do  action
                                       while test action
                                  else return ()
```

Exemplo

Copia entrada na saída

```
copyInpOut :: IO ()
copyInpOut = while notEOF (do line ← getLine
                               putStrLn line )
```

A ação isEOF :: IO Bool é predefinida no módulo System.IO. Testa o final de uma entrada de dados. Note que `notEOF` é também uma ação.

```
notEOF :: IO Bool
notEOF = do  cond ← isEOF
             return (not cond)
```

Mais I/O

- No módulo System.IO
  - readFile, writeFile, appendFile
  - Tratamento de erros
    - ioError
    - Catch
  - Monads ...

Exercícios Recomendados

- Exercícios do capítulo 18 do Simon Thompson, segunda edição, ou do capítulo 8 na terceira edição.