Programação Funcional

I/O em Haskell

Prof. Edson Alves

Faculdade UnB Gama

2021

Sumário

Código puro e código impuro

- Haskell determina uma clara separação entre código puro e código impuro
- Esta estratégia permite que código puro fique isento de efeitos colaterais
- Além de facilitar a divisão semântica do código, ela permite aos compiladores otimizar e paralelizar trechos de código automaticamente
- Como as rotinas de entrada e saída interagem com o mundo externo, todas elas produzem ou estão suscetíveis a efeitos colaterais, sendo assim, códigos impuros

Leitura e escrita de strings em console

- Haskell provê um conjunto de funções para escrita e leitura de dados a partir do console
- No que diz respeito à strings, duas funções básicas são `putStrLn` e `getLine`
- A função `putStrLn` escreve uma string no console, seguida de uma quebra de linha:

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
```

- Já a função `getLine` lê uma string do console até encontrar uma quebra de linha e a retorna, sem a quebra
- O tipo da função `getLine` é

```
ghci> :type getLine
getLine :: IO String
```

Exemplo de leitura e escrita de strings em console

```haskell
-- Lê uma string do console e a imprime, sem modificações
-- O operador $ colocar tudo o que se segue entre parêntesis
main = do
    putStrLn "Insira uma string: "
    s <- getLine
    putStrLn $ "A string inserida foi '" ++ s ++ "'"
```

Ações de entrada e saída

- O identificador **IO** nas assinaturas das funções `getLine()` e `putStrLn()` indicam que estas funções ou tem efeitos colaterais, ou podem retornar valores distintos mesmo que invocadas com os mesmos parâmetros
- Portanto, a presença de **IO** na assinatura das funções indicam código impuro
- Uma ação de entrada e saída (I/O) tem tipo **IO** *tipo*
- Uma ação pode ser declarada e armazenada, mas só pode ser executada dentro de outra ação de I/O
- `()` é a tupla vazia e indica a ausência de retorno (similar ao tipo **void** de C/C++)

Ações de entrada e saída

- Assim, a função `putStrLn()` não tem retorno: escrever a string no terminal é seu efeito colateral
- Executar uma ação do tipo `IO t` pode gerar algum tipo de I/O e, eventualmente, retornar um valor do tipo `t`
- Já a função `getLine()` retorna uma ação do tipo `IO String`
- O operator `<-` executa a ação e extrai o retorno do ação, unindo-o a variável indicada
- A função `main()`, ponto de início da execução dos programas em Haskell, tem tipo `IO ()`
- A palavra reservada `do` define uma sequência de ações de I/O
- O valor do bloco `do` é dado pelo valor da última ação executada
- A palavra reservada `let` permite armazenar o resultado de código puro em uma acão (ou bloco de ações) de I/O

Exemplo de código puro em um bloco de I/O

```haskell
import Data.Char

capitalize [] = []
capitalize (x:xs) = toUpper x : map toLower xs

formatted name = unwords $ map capitalize $ words name

-- A palavra reservada let permite chamar código puro em uma ação de I/O
main = do
    putStrLn "Insira seu nome: "
    name <- getLine
    let s = formatted name
    putStrLn $ "Bom dia, sr(a) " ++ s
```

Código Puro vs. Código Impuro

- Haskell distingue e separa os trechos de código puro e impuro
- Código puro não tem efeitos colaterais, não altera estados e tem mesmo retorno para o mesmo conjunto de parâmetros
- Código impuro não tem garantias sobre o retorno, pode interagir com o sistema, alterando seu estado, e ter efeitos colaterais
- Código impuro pode invocar código puro
- Já o contrário não é possível
- As garantias dadas por um código puro tem vantanges, como a possibilidade de paralelismo automático, por exemplo

I/O em arquivos

- A biblioteca **System.IO** fornece uma série de funções para I/O, inclusive em arquivos
- A função openFile() retorna um **Handle** para o arquivo aberto, se bem sucedida

```
ghci> :type openFile
openFile  :: FilePath -> IOMode -> IO Handle
```

- Os modos de abertura são **ReadMode**, **WriteMode**, **AppendMode** e **ReadWriteMode**
- Uma vez finalizada a manipulação do arquivo, ele deve ser fechado por meio da função hClose()
- As funções hPutStrLn() e hGetLine() funcionam como as contrapartes sem o h, com a diferença que agem sobre o arquivo indicado

Exemplo de manipulação de arquivos

```haskell
-- Processamento de arquivos no estilo imperativo
import System.IO

main :: IO ()
main = do
    inh <- openFile "input.txt" ReadMode
    outh <- openFile "output.txt" WriteMode
    mainLoop inh outh
    hClose inh
    hClose outh

mainLoop :: Handle -> Handle -> IO ()
mainLoop inh outh = do
    ineof <- hIsEOF inh
    if ineof then return ()
             else do
                line <- hGetLine inh
                hPutStrLn outh $ stripped line
                mainLoop inh outh

stripped = unwords . words
```

return

- Em Haskell, a palavra reservada `return` tem significado distinto do utilizado em outras linguagens, como C/C++
- `return` insere um valor em uma ação de I/O
- Ele corresponde ao inverso do operador `<-`
- Deste modo, ele não encerra prematuramente um bloco, como nas linguagens imperativas, e sim produz uma ação de I/O

Controlando o cursor de leitura

- Quando uma rotina de leitura é chamada em um *handle* de um arquivo, o cursor de leitura é atualizado
- Assim, a próxima leitura tem início no ponto no qual a leitura anterior terminou
- A função `hTell()` retorna a posição atual do cursor no arquivo, contada em número de *bytes*:

```
ghci> :type hTell
hTell :: Handle -> IO Integer
```

- Ao abrir o arquivo, o cursor inicia na posição 0, exceto no **AppendMode**, onde o cursor inicia no fim do arquivo
- A função `hSeek()` permite atualizar a posição do cursor:

```
ghci> :type hSeek
hTell :: Handle -> SeekMode -> Integer -> IO ()
```

- Os modos disponíveis são: **AbsoluteSeek**, **RelativeSeek** e **SeekFromEnd**

Exemplo de controle do cursor de leitura

```haskell
-- Computa o número de bytes contidos no arquivo indicado
import System.IO

main = do
    putStrLn "Insira o nome do arquivo:"
    path <- getLine
    h <- openFile path ReadMode
    size <- fileSize h
    print size
    hClose h

fileSize :: Handle -> IO Integer
fileSize h = do
    hSeek h SeekFromEnd 0
    size <- hTell h
    return size
```

Entrada e saída padrão do sistema

- Em Haskell há três *handles* pré-definidos, associados a entrada e a saída-padrão do sistema
- O *handle* `stdin` corresponde à entrada padrão, que usualmente está associada ao teclado
- O *handle* `stdout` corresponde à saída padrão, em geral associada ao terminal
- Por fim o *handle* `stderr` corresponde à saída de erro padrão, que pode ser o próprio terminal ou ser direcionada ao um arquivo específico
- As funções de I/O sem o prefixo `h` são definidas em termos destes `handles`:

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

Removendo e renomeando arquivos

- A biblioteca **System.Directory** oferece funções que permitem manipular os arquivos diretamente
- A função removeFile recebe como argumento o caminho para um arquivo e o remove

```
ghci> :type removeFile
removeFile :: FilePath -> IO ()
```

- Já a função renameFile recebe dois parâmetros: o caminho do arquivo a ser renomeado e o caminho do seu novo nome/destino

```
ghci> :type renameFile
renameFile :: FilePath -> FilePath -> IO ()
```

- Se o arquivo de destino já existir, ele será removido para que então o arquivo original assuma seu caminho/nome

Exemplo de manipulação de nome de arquivo

```haskell
import Data.Char
import Data.List
import System.Directory

-- map ($ x) [f1, f2, ..., fN] computa [f1 x, f2 x, ..., fN x]
normalize infile = ys ++ ext
    where
        (name, ext) = span (\x -> x /= '.') infile
        xs = map (\c -> if isSpace c then '_' else c) name
        p = (\c -> any $ map ($ c) [isAlphaNum, (=='_'), (=='/')])
        ys = filter p xs

main = do
    infile <- getLine
    let outfile = normalize infile
    renameFile infile outfile
```

Lazy I/O

- Haskell também oferece funções para utilizar o *lazy evaluation* no contexto de I/O
- A função `hGetContents` recebe um *handle* e retorna uma string com o todo conteúdo a partir do ponto onde está localizado o cursor de leitura:

```
ghci> :type hGetContents
hGetContents :: Handle -> IO String
```

- Contudo, o acesso a este conteúdo é feito por demanda: a string não contém, necessariamente, todo o conteúdo de uma só vez em memória
- Este acesso é feito de forma transparente ao usuário que, para fins de código, terá uma string que pode ser utilizada em código puro
- Assim é possível escrever código com I/O mais próximos da abordagem funcional, sem a necessidade de laços para processamento de blocos de informações, como é feito nas linguagens imperativas

Exemplo de I/O no estilo funcional

```haskell
-- Processamento de arquivos no estilo funcional
import System.IO

main :: IO ()
main = do
    inh <- openFile "input.txt" ReadMode
    outh <- openFile "output.txt" WriteMode

    input <- hGetContents inh
    hPutStrLn outh $ stripped input

    hClose inh
    hClose outh

stripped = unlines . map (unwords . words) . lines
```

Lazy I/O

- O idioma de abrir um arquivo, ler seu conteúdo, processar este conteúdo, escrever o resultado em um arquivo de saída e fechar os *handles* é tão comum que Haskell oferece duas funções que abstraem este processo
- A primeira delas é a função `readFile`, que recebe o caminho do arquivo a ser aberto e retorna uma string que pode acessar o conteúdo por meio de *lazy evaluation*

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

- A segunda é a função `writeFile`, que recebe o caminho do arquivo de saída e o conteúdo a ser escrito:

```
ghci> :type writeFile
writeFile :: FilePath -> String -> IO ()
```

- Ambas funções fazem parte da biblioteca **`System.IO`**
- Estas funções mantém o *handle* dos arquivos internamente, dispensando a chamada da função `hClose`

Exemplo de I/O no estilo funcional sem handles

```haskell
import System.IO

main :: IO ()
main = do
    input <- readFile "input.txt"
    writeFile "output.txt" $ stripped input

stripped = unlines . map (unwords . words) . lines
```

Interações

- Quando os arquivos a serem processados são a entrada e a saída padrão, o idioma abstraído pelas funções `readFile` e `writeFile` pode ser condensado em uma única função
- A função `interact` tem como parâmetro uma função que recebe e retorna strings:

```
ghci> :type interact
interact :: (String -> String) -> IO ()
```

- Todo o conteúdo da entrada padrão é passado como parâmetro desta função, e o retorno dela é escrito na saída padrão
- Esta função também faz parte da biblioteca **`System.IO`**
- Quando utilizada em conjunto com filtros, esta função permite escrever programas concisos, com poucas linhas, que processam arquivos de diferentes maneiras

Exemplo de interação com filtro

```haskell
-- Elimina os comentários do arquivo
import Data.Char
import Data.List

main = interact (unlines . f . lines) where
    -- Este comentário também será filtradao
    f = filter (not . isPrefixOf "--" . dropWhile isSpace)
```

Referências

1. Hoogle. System.IO, acesso em 06/08/2020.
2. **SHALOM**, Elad. *A Review of Programming Paradigms Througout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
3. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Relly.