

Investigação de Linguagens de Programação: Haskell

Larissa Salerno, Gabriel Fanto Stundner
Escola Politécnica — PUCRS

4 de abril de 2020

1 Introdução

Linguagens funcionais são linguagens que trabalham puramente com o uso de funções. Diferente de linguagens com outros paradigmas, linguagens funcionais trabalham com funções matemáticas e não com o uso de estados. Nessa atividade vamos apresentar um estudo feito em cima da linguagem Haskell.

A seção 2 fala um pouco sobre o paradigma funcional, que é essencial para entendermos o conceito de Haskell. A seção 3 é dedicada a explicação do Haskell, incluindo aspectos históricos, características, e depois artefatos mais técnicos como notação e implementação. Nessa mesma seção apresentamos a implementação de 2 algoritmos, um de fatorial e outro de leitura de arquivos, o qual iremos explicar minuciosamente sua construção.

2 Programação Funcional

Antes de falar sobre Haskell é importante entendermos o que veio antes do surgimento da linguagem. A programação funcional é um paradigma de programação que utiliza funções matemáticas, ou seja, evitam ao máximo trabalhar com estados e dados imutáveis. Diferente do paradigma imperativo, na programação funcional valores atribuídos a variáveis não podem ser alterados, quando no paradigma imperativo é realizado uma sequência de passos alterando estados e modificando variáveis. A seguir, será apresentado um exemplo de implementação de uma função para realizar uma soma, mostrando a diferença entre o paradigma imperativo e funcional.

Imperativo:

```
v1 = 5
v2 = 3

def soma():
    return v1 + v2
```

Funcional:

```
def soma(v1, v2):
    return v1 + v2
```

Como podemos ver, o primeiro algoritmo é imperativo, então declara as variáveis e retorna elas diretamente, sem que seja necessário passá-las por parâmetro. Já na segunda implementação funcional, a função `soma()` já espera o valor das variáveis por parâmetro. Isso mostra que no paradigma funcional não é necessário criar uma sequência de passos, como foi mostrado na primeira implementação. O paradigma funcional trabalha em cima da execução de funções. Isso também acaba as tornando mais simples e fáceis de visualizar.

3 Haskell

Haskell (nomeada em homenagem ao lógico Haskell Curry) é uma linguagem funcional criada em 1987 por um comitê de acadêmicos que sentiram a necessidade de criar uma linguagem que pudesse ser usada tanto para pesquisa, quanto para o ensino de programação funcional. A linguagem levou cerca de 10 anos para finalmente ser publicada com uma versão estável, conhecida como Haskell 98. No ano de 2003 tiveram algumas atualizações, e suas últimas modificações foram feitas em 2010. Diferente do que estamos acostumados a programar, a programação em Haskell é muito mais rápida e consome um número menor de linhas, já que ao invés de digitar uma sequência de passos, o programador apenas declara o que ele quer que aconteça no programa. A seguir, vão ser apresentadas as características da linguagem e a implementação de 2 algoritmos, um de fatorial e outro de leitura de arquivos.

3.1 Características

Haskell possui diversas características, sendo elas:

- **Lazy:** o conceito *lazy* significa que a linguagem não calcula e executa funções a não ser que seja realmente explícito no código.
- **Estática:** quer dizer que quando uma variável for declarada, ela não poderá sofrer alterações depois. **Ex:** Se uma variável *int* for declarada, a mesma não pode receber um valor *string* depois.
- **Alto nível:** Haskell é uma linguagem alto nível, ou seja, é muito mais "limpa" e fácil de entender, já que não precisa se preocupar com questões internas de processamento como, por exemplo, ponteiros em C.
- **Laços de repetição:** A linguagem não trabalha com laços como o *for*, por exemplo. A maneira que ela encontra de expressar esses laços é a partir de recursão.
- **Listas:** A utilização de listas é muito comum e quase primordial na programação em Haskell. A linguagem usa isso como uma forma de semelhar conjuntos matemáticos.
- **Imutabilidade:** Não pode haver mudança nas variáveis, uma vez que a variável recebe um valor, essa jamais poderá ser alterada.
- **Funções de alta-ordem:** Haskell pode receber funções como parâmetros de funções
- **Polimorfismo:** A linguagem permite criar funções genéricas, então essas podem ser usadas para variáveis de diversos tipos diferentes.

3.2 Implementação

Nessa subseção vamos explicar aspectos de notação e vamos apresentar e explicar dois algoritmos em Haskell, um de Fatorial e outro de Leitura de arquivos.

3.2.1 Notação em Haskell:

Comentários: Para fazermos um comentário em Haskell devemos usar dois traços seguidos, como representado abaixo:

```
-- Comentário de uma linha

--
-- Comentário de múltiplas linhas
-- neste arquivo
-- pode se colocar em cada linha um novo --
```

Definindo o tipo de uma variável: Usamos a notação `::` para definir variáveis, como no exemplo abaixo:

```
-- nomeVariavel::tipo
idade::Int
booleano::Bool
```

Funções: Quando estamos trabalhando com uma Função em Haskell devemos dizer o tipo de entrada e o tipo de saída antes de elaborar a função. No exemplo abaixo, queremos pegar um valor inteiro(*Int*) e retornar um valor inteiro também, onde isso tem que ser definido antes de implementar o que a função faz.

```
-- Função que retorna o Quadrado do Valor
quadrado::Int -> Int
quadrado x = x * x
```

Testando uma função: Para testar a função chamamos um teste criando um *main*:

```
main = print (quadrado 2)
-- Saída: 4
```

Utilização de *Guards*: *Guards* são expressões booleanas que são iniciadas pelo símbolo `"|"`, onde são opções que a função irá verificar por um teste booleano. Caso nenhuma das expressões seja *true*, deve ser elaborado o *otherwise* que sempre é chamado quando todos os outros falham, exemplo:

```
-- Verificação se o valor inicial é o menor
menor:: Int -> Int -> Int
menor a b
  | a <= b    = a
  | otherwise = b
```

3.2.2 Algoritmo Fatorial

Para explicar como funcionam recursões em Haskell, criamos um programa simples que faz o fatorial de um número.

A primeira coisa que devemos fazer é definir o caso base da função, que no caso de um fatorial é quando o fatorial de 0 é igual a 1. Para definir esse valor contante devemos chamar primeiro o nome da função e o valor base, que no nosso caso é 0, e depois colocamos o símbolo de igualdade para dizermos que quando chegar nesse caso o valor que será transmitido é 1.

```
fatorial 0 = 1
```

Depois de definirmos o valor base de um fatorial, devemos implementar como a função funciona, adicionando toda a estrutura recursiva que iremos precisar. A função do fatorial necessita de um valor de entrada, que pode ser qualquer um, então iremos definir esse argumento que irá receber um valor da compilação como `"n"`. Depois de definirmos o argumento, temos que construir o esqueleto do que nossa função deve fazer, onde nesse caso é multiplicar o valor de entrada com a chamada da função novamente, onde definimos o que deve fazer a função fatorial com seu número anterior.

```
fatorial n = n * fatorial(n-1)
```

Com essa estrutura, não importa o número colocado para rodar o fatorial, a função vai ser chamada até que o valor de entrada seja 0 e resolver todos os outros de forma crescente.

No vídeo, mostramos a criação de uma estrutura *main* que irá ter um caso de exemplo, onde queremos que o Haskell resolva o fatorial de 5 e nos entregue impresso no terminal, onde utilizamos uma função básica do Haskell chamada *Print*.

3.2.3 Algoritmo Leitura de Arquivos

Neste algoritmo quisemos explicar como funciona a utilização de funções já existentes no sistema Haskell, onde não precisamos criar funções se já existem outras que possam nos auxiliar. Para demonstrar a diferença de linguagens Imperativas e Orientadas a Objetos criamos um algoritmo que lê e escreve em um arquivo criado. As funções que utilizamos para construir o algoritmo possuem operações utilizando uma estrutura do Haskell chamada **IO**.

Estrutura IO:

A estrutura IO se comunica com o sistema operacional e possui funções específicas para tratar a entrada definida, as funções que utilizamos para criar, escrever, e ler possuem essa estrutura. Abaixo encontra-se a lista de funções usadas nesse programa e seus escopos:

- **putStr**:: String -> IO()
- **getLine**:: IO String
- **writeFile**:: String -> String -> IO()
- **readFile**:: String -> IO String
- **return**:: a -> IO a

Essas funções possuem sua própria estrutura, então não precisamos construí-las, apenas usá-las. A função **putStr** requer uma *String* que deve ser colocada dentro dos (), por isso o IO(). Depois ele irá imprimir na tela, onde a *String* colocada dentro dos () vai ser uma pergunta que o usuário deve responder e depois clicar no ENTER para finalizar a entrada.

A função **getLine** pega a entrada inserida pelo usuário depois da função **putStr**, onde a função **getLine** espera que a entrada seja uma *String*, por isso o IO *String* mostrado no escopo. A função deve enviar essa *String* para um atributo criado somente na hora de compilação, como mostraremos mais tarde.

A função **writeFile** serve para enviarmos uma linha de texto para dentro de um arquivo, o qual a função necessita de duas *Strings* para funcionar. A utilização de () é opcional, mas se desejar pode ser colocado. A primeira *String* de entrada deve ser o nome do arquivo desejado e a segunda *String* é todo o texto que deseja colocar no arquivo.

A função **readFile** serve para ler um arquivo, onde ela precisa receber como argumento a *String* do nome do arquivo que se deseja ler, assim como funciona o **getLine**, então ela pega o nome do arquivo e transmite para um atributo de momento toda a informação escrita no arquivo. Já a função **return** é utilizada para retornar qualquer valor para o usuário, somente é necessário informar ao *return* o que deve ser retornado.

Explicação do algoritmo:

Uma sequência de entradas e saídas de dados em Haskell deve-se colocar a expressão *"do"* dentro da estrutura *main* quando quiser utilizar as funções mostradas anteriormente. As funções mostradas anteriormente já estão definidas no Haskell, então devemos nos preocupar principalmente em utilizar essas funções diretamente no *main* do nosso programa, que deve ser inicializada assim:

```
main = do
```

Agora podemos utilizar todas as funções dentro dessa estrutura, onde a primeira etapa é pegar o nome do arquivo que desejamos construir e a informação que queremos colocar na linha, o qual no nosso exemplo somente coloca uma linha no arquivo e o finaliza. Se for tentarmos utilizar de novo ele irá apagar a linha já existente e reescrever com a nova informação chamada.

```
putStr("Escreva o nome do Arquivo: ")
nome <- getLine
putStr("Escreva uma linha de texto: ")
linha <- getLine
```

Utilizamos a função **getLine**, que envia a entrada do usuário no **putStr**, onde o primeiro pega o nome do arquivo e salva em um atributo momentâneo chamado *nome*, e o segundo pega a linha de *text* e salva no atributo momentâneo chamado *linha*. Agora que temos o nome do arquivo e a linha de texto, somente passamos os atributos para a função, da seguinte forma:

```
writeFile nome linha
```

Se desejamos ler do arquivo e salvar em um atributo, podemos usar a função **readFile** como já dito anteriormente, e isso fica assim:

```
lendoArquivo <- readFile nome
```

Passamos o nome do arquivo para a função **readFile** e salvamos o conteúdo de saída da função em um atributo momentâneo, e depois podemos chamar o atributo externamente de duas formas: ou utilizando o **putStr** ou usando **return**. No exemplo que fizemos foi pelo **putStr**, mas colocamos comentado da mesma maneira que utiliza-se o **return** para retornar o nome do arquivo.

```
putStr("Nome do Arquivo: " ++ nome ++ "\n"      )
putStr("Linha no Arquivo: " ++ linha ++ "\n"    )
return (nome)
return (linha)
```

Referências

- [1] Hudak, P. and Fasel, J.: "A gentle introduction to Haskell.", ACM Sigplan Notices, 1992.
- [2] Hutton, G.: "Programming in Haskell" (2nd Edition). Cambridge University Press, 2016.
- [3] HaskellWiki: <https://wiki.haskell.org/Haskell>, 2013.
- [4] Mueller, J.: "Functional Programming for Dummies (1st Edition), John Wiley and Sons, 2019.
- [5] Sebesta, R.: "Concepts of Programming Languages" (11th Edition), Pearson, 2015.
- [6] Lipovača, M.: "Learn You a Haskell for Great Good!: A Beginner's Guide" (1st Edition), No Starch Press, 2011.