Introdução à Programação Funcional com Haskell

IX FLISOL (2015) – Marcelo Haskell Camargo

github.com/haskellcamargo

<marcelocamargo@linuxmail.org>

Conteúdo

1. O que é programação funcional?
2. Linguagens funcionais versus imperativas
3. Características das linguagens funcionais
   - Funções de mais alta ordem
   - Purismo
     - Imutabilidade de dados
     - Transparência referencial
     - Avaliação preguiçosa
     - Purismo e efeitos
       - Efeitos colaterais na linguagem
       - Efeitos colaterais através de monads
   - Recursividade
4. Benefícios da programação funcional
5. Referências e livros

1. O que é programação funcional? (A)

Programação funcional é um estilo de programação que modela computações como avaliações de expressões, um outro paradigma de programação, uma outra maneira de se pensar e de se produzir computações. Diferentemente do padrão assertivo-imperativo, que segue instruções rígidas e lineares, em linguagens puramente funcionais você não informa ao computador **como** alguma coisa deve ser feita, mas **o quê** deve ser feito. Ao contrário de linguagens imperativas, que são compostas de **declarações** (*statements*), linguagens funcionais são compostas por **expressões**.

1. O que é programação funcional? (B)

Funcionalmente, te é assegurado que o estado global de um programa deve ser imutável quando este é executado – estados mutáveis são comumente evitados por linguagens funcionais. Tipicamente, programação funcional requer que funções sejam **cidadãos de primeira classe**, o que significa que elas são tratadas como quaisquer outros valores e podem ser passadas como argumentos para outras funções, ou serem retornadas como o resultado de uma função. Ser de primeira classe também significa que é possível definir e manipular funções através de outras funções. Uma atenção especial deve ser dada a funções que referenciam **variáveis locais** de seu escopo. Se uma função escapa de seu bloco após ser retornada deste, as variáveis locais precisam ser retidas na memória, isso porque serão necessárias posteriormente quando houver chamada de uma função que depende delas. Isso não costuma ser muito simples de se fazer manualmente, mas fiquem tranquilos, a maioria dos compiladores é capaz de gerenciar a memória automaticamente para a gente!

1. O que é programação funcional? (C)

Exemplo de gerenciamento automático de memória para retenção de valores:

```
var add = function (x) {
  return function(y) {
    return x + y;
  }
}
```

Note que **add** é uma função que recebe um parâmetro **x** e retorna uma função que recebe um parâmetro **y** e essa última função retorna os valores de **x + y**. A função retornada referencia uma variável em um escopo superior e esta deve ser armazenada na memória para utilização posterior.

1. O que é programação funcional? (D)

O uso de nossa função previamente definida, que também é uma **curried-function**, é:

```
var add_3 = add(3); // y → x + y
var ten   = add_3(7); // 10
```

Nós não temos o valor de **x** explicitamente denotado ali. Ele será manipulado pelo gerenciador de memória do compilador/interpretador.

2. Linguagens funcionais versus imperativas (A)

Muitas linguagens de programação suportam tanto o estilo funcional, quanto o imperativo, mas a sintaxe e as facilidades da linguagem são tipicamente otimizadas para **somente** um desses estilos, e fatores sociais, como convenções de código e bibliotecas, frequentemente forçam o programador a um desses estilos. Portanto, faz sentido categorizarmos, em nosso domínio, linguagens funcionais e imperativas.

2. Linguagens funcionais versus imperativas (B)

A seguinte tabela mostra o suporte a algumas linguagens à programação funcional:

Linguagem	Closures	Funcional
C	Não	Não
Pascal	Não	Não
C++	Sim	Não
Java	Sim	Não
Modula-3	Sim	Não
Python	Sim	Não
Ruby	Sim	Não
D (2.0)	Sim	Não
OCaml	Sim	Sim
Erlang	Sim	Sim
Haskell	Sim	Sim

2. Linguagens funcionais versus imperativas (C)

Exemplo simples com uma implementação naïve da função fatorial:

- Javascript

```
var fact = var (n) {
  var product = 1;
  for (i = 2; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

2. Linguagens funcionais versus imperativas (D)

Exemplo simples com uma implementação naïve da função fatorial:

- Haskell

```
fact :: Int → Int
fact 0 = 0
fact n = n * fact (n - 1)
```

3. Características das linguagens funcionais
3.1 Funções de mais alta ordem (A)

Funções de mais alta ordem (higher-order functions), ou **HOF**s, são funções que recebem funções como argumentos. Um exemplo básico de uma HOF é a função *map*, que recebe uma função e uma lista como argumentos, aplica a função para todos os elementos da lista, e retorna a lista de resultados. Por exemplo, nós podemos escrever uma função que subtrai 2 de todos os elementos de uma lista sem a utiliação de *loops* ou recursão, em Haskell:

```
subtractTwoFromList :: [Int] → [Int]
subtractTwoFromList xs = map (\x → x – 2) xs
```

3. Características das linguagens funcionais
3.1 Funções de mais alta ordem (B)

HOFs são muito úteis para refatoração de código e reduzir repetições *boilerplate*. Por exemplo, a maioria dos *loops* pode ser expressada através de *maps* ou *folds*. Esquemas customizados de iteração, tais como loops paralelos, podem ser facilmente expressados usando HOFs.

HOFs são frequentemente usadas para implementar linguagens de domínio específico embutidas em Haskell, como bibliotecas de combinadores.

3. Características das linguagens funcionais
3.2 Purismo

Algumas linguagens funcionais permitem expressões que façam *yield* de ações além de retornar esses valores. Essas ações são o que chamamos de *side effects*, ou efeitos colaterais, isso para determinar que o valor de retorno é mais importante do que o que é externo a uma função, o que é o oposto de linguagens imperativas.

É frequentemente benéfico escrever uma porção significante de código de um programa funcional de maneira puramente funcional, e envolver tudo o que lida com estados e I/O especialmente e o mínimo necessário, levando em conta que código impuro tem uma tendência maior a erros

3.2 Purismo
3.2.1 Imutabilidade de dados

Linguagens puramente funcionais operam, tipicamente, em dados **imutáveis**. Ao invés de alterar valores existentes, cópias alterados são criadas enquanto o original é preservado. Levando em conta que as partes não alteradas de uma estrutura não podem ser modificadas, elas podem ser compartilhadas entre a antiga e novas cópias, o que guarda memória. Se você disse ao seu programa que **x** é 1 e depois disse que **x** é 2, ele vai te gritar. Por acaso você é algum tipo de mentiroso?

3.2 Purismo
3.2.2 Transparência referencial

Computações puras retornam o mesmo valor cada vez que elas são invocadas. Essa propriedade é chamada transparência referencial e torna possível conduzir raciocínio equacional ao código. Por exemplo, se **y = f x** e **g = h y y**, então devemos ser capazes de substituir a definição de **g** com **g = h (f x) (f x)** e obtermos o mesmo resultado, contudo, apenas a eficiência pode mudar.

3.2 Purismo
3.2.3 Avaliação preguiçosa

Levando em conta que computações puras são referencialmente transparentes, elas podem ser executadas a qualquer momento e ainda nos retornarem o mesmo resultado. Isso torna possível diferir as computações de valores até quando elas são necessárias, isto, computá-las preguiçosamente. Avaliação preguiçosa evita computações desnecessárias e permite, por exemplo, **estruturas de dados infinitas** a serem definidas e usadas. Por exemplo, que tal se quisermos pegar os 50 primeiros números de uma lista que tende de 5, 7, 9 ... ao infinito?

```
fromInfiniteList :: [Int]
fromInfiniteList = take 50 [1, 3 .. ]
```

3.2 Purismo
3.2.4 Purismo e efeitos

Ainda que programação puramente funcional seja muito benéfica, o programador deve ser capaz de usar características que não estão disponíveis em programas puros, como *arrays* mutáveis eficientes ou a conveniente I/O. Há duas abordagens para esse problema.

3.2.4 Purismo e efeitos
3.2.4.1 Efeitos colaterais na linguagem

Algumas linguagens estendem seu núcleo puramente funcional com efeitos. O programador deve cuidar para não utilizar efeitos colaterais onde somente funções puras são aceitas. Um exemplo clássico seria o simples "obter algo do console" ou "escrever algo na tela". No caso, vamos perguntar o nome e saudar nosso usuário?

```
module Main where

main :: IO ()

main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name
```

3.2.4 Purismo e efeitos
3.2.4.2 Efeitos colaterais através de *monads*

Uma outra maneira, mas um tanto mais complexa, de se introduzir efeitos colaterais é simulá-los utilizando monads. Enquanto a linguagem permanece pura e referencialmente transparente, *monads* podem prover um estado implícito carregando valores dentro deles. O compilador não precisa saber sobre características imperativas porque a linguagem continua pura, no entanto, frequentemente as implementações **sabem** disso por questões de performance, por exemplo, para prover *arrays* mutáveis.

Permitir efeitos colaterais através de *monads* e manter a linguagem tão pura torna possível que usemos avaliação preguiçosa que não conflita com efeitos de código impuro. Ainda que avaliações preguiçosas possam ser avaliadas em **qualquer ordem**, a estrutura monádica garante a execução na ordem correta.

3.3. Recursividade

"Para se entender recursividade, é necessário se entender recursividade".

Recursividade é usada fortemente em programação funcional, levando em conta que não há *loops* como em linguagens imperativas e é, frequentemente, a única maneira de se iterar. Implementação de linguagens funcionais irão frequentemente incluir *tail call optimisation* para assegurar que a recursão não consuma excessivamente a memória.

3.4. Benefícios da programação funcional

Programação funcional é conhecida por prover melhor suporte à programação estruturada até mesmo do que linguagens imperativas. Para se fazer um programa estruturado é necessário desenvolver-se abstrações e dividir em componente que são interface para cada outro com essas abstrações. Linguagens funcionais ajudam nisso por tornarem limpas e simples as abstrações. É fácil, por exemplo, abstrair um trecho de código através de uma HOF, o que tornará o código mais declarativo e até compreensível. Programas funcionais são usualmente mais curtos e fáceis de se entender do que seus concorrentes imperativos. Levando em conta que vários estudos mostraram que a produtividade média do programador, em termos de linhas de código, é mais ou menos a mesma para a maioria das linguagens de programação, isso se traduz em produtividade superior.

3.5. Referências

https://wiki.haskell.org/Functional_programming

http://learnyouahaskell.com/chapters

Dúvidas? Perguntas?