Programação Funcional

Fundamentos

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Visão geral

- A programação funcional é um paradigma de programação que trata a computação como a valoração de funções matemáticas e que evita estados e dados mutáveis
- Tem como fundamento o cálculo lambda
- Os programas consistem em expressões, de modo que é um paradigma declarativo, não imperativo
- O conceito principal é de que o retorno de uma função depende única e exclusivamente de seus parâmetros (ou argumentos)
- Isto significa que uma função chamada duas ou mais vezes, com os mesmos argumentos, terá sempre o mesmo retorno, o que não necessariamente ocorre em paradigmas imperativos

Exemplo em C de mesmos argumentos, retornos distintos

```c
#include <stdio.h>

int y = 0;

int f(int x)
{
    return x + y++;
}

int main()
{
    printf("f(2) = %d\n", f(2));    // f(2) = 2
    printf("f(2) = %d\n", f(2));    // f(2) = 3

    return 0;
}
```

Histórico da programação funcional

- A programação funcional tem raízes no cálculo lambda, que surgiu na década de 1930
- Na década de 1950, Lisp foi uma das primeiras linguagens a incorporar conceitos e características da programação funcional
- Na década de 1960 a APL (*A Programming Language*) foi desenvolvida e influenciou o desenvolvimento da programação funcional na década seguinte
- Também na década de 1970 foi desenvolvida a linguagem ML e em 1987 teve início o desenvolvimento da linguagem Haskell
- Na década de 1990 surgiram as linguagens J, K e Q
- Com o passar do tempo e da maturação destas linguagens, elas deixaram de ser usadas apenas em pesquisas e na academia e hoje são utilizadas em aplicações reais

Principais características da programação funcional

(a) As funções são cidadãs de primeira classe: as funções podem aceitar funções como parâmetros e podem retornar funções

(b) As funções são puras, isto é, não há efeitos colaterais

(c) Uso extensivo de recursão

(d) As variáveis são imutáveis

(e) A valoração das expressões é não-estrita, de modo que o valor de uma variável ou função só é computado quando for utilizado (também chamada *lazy evaluation*)

(f) As expressões dos programas tem um valor de retorno

(g) Uso de *pattern matching* para extração dos membros de objetos ou para condicionais

Variáveis imutáveis

- Variáveis **imutáveis** são aquelas cujo valor não pode ser modificado após sua inicialização
- Em linguagens imperativas, tais variáveis são denominadas **constantes**
- Variáveis imutáveis são úteis em ambientes *multithread*, pois elas não levam a problemas de concorrência
- Em linguagens funcionais, uma variável é atada a um valor ou expressão em sua atribuição inicial, e ao longo do programa ela não pode ser desatada ou atada a uma outra expressão ou valor
- O código Haskell abaixo gera um erro de compilação, pois a variável `x` é imutável:

```
main = print x where
    x = 1
    y = 2 * x
    x = 3 * y + 1
```

Funções de primeira classe

- **Funções de primeira classe** são funções que recebem o mesmo tratamento que os tipos primitivos da linguagem
- Assim, elas podem ser parâmetros ou retornos de outras funções, ou podem ser armazenadas em variáveis
- **Funções de alta ordem** são funções que recebem uma ou mais funções como parâmetros, ou que retornam uma função
- Funções que não são de alta ordem são denominadas funções de **primeira ordem**
- No cálculo lambda, todas as funções de são alta ordem
- Um exemplo típico de função de alta ordem é a função `map()`, que recebe como parâmetros uma função `f()` e uma lista `xs`, e e retorna uma lista cujo i-ésimo elemento é $f(x_i)$
- Exemplo em Haskell:

```
-- Saída: [1.0,1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]
main = print xs where
    xs = map sqrt [1, 2, 3, 4, 5]
```

Funções anônimas

- As funções contém quatro partes: nome, lista de parâmetros, corpo e retorno
- **Funções anônimas** são funções que não possuem um nome
- Há dois tipos de funções anônimas: **funções lambda** e ***closures***
- Em geral, funções anônimas são utilizadas como parâmetros de funções de alta ordem, ou para a construção dos retornos destas
- Funções lambda são funções compostas por uma lista de parâmetros, um corpo e um retorno
- *Closures* são semelhantes às funções lambda, exceto pelo fato de que elas referenciam variáveis foram do escopo do seu corpo e de sua lista de parâmetros
- Em geral, os *closures* são implementados como estruturas de dados que contém um ponteiro para o código da função e para todas as variáveis necessárias para sua execução

Funções puras e efeitos colaterais

- Uma função ou expressão tem um **efeito colateral** se, além de computar o valor de retorno, ela modifica o estado do programa
- Os efeitos colaterais mais comuns são: alteração de variáveis globais, estáticas ou parâmetros, escrita ou leitura em periféricos, e invocação de funções que tenham efeitos colaterais
- A ausência de efeitos colaterais é uma condição necessária, porém não suficiente, para a transparência referencial
- A **transparência referencial** significa que uma expressão pode ser substituída por seu valor
- Funções **puras** não tem efeitos colaterais e geram o mesmo retorno para o mesmo conjunto de parâmetros
- Em outras palavras, o retorno depende única e exclusivamente dos parâmetros

Currying

- *Currying* é uma técnica de transformação que converte uma função com múltiplos parâmetros em uma sequência de funções de um único parâmetro cada, cujos retornos são as próximas funções da sequência
- Por exemplo, considere a função

$$\begin{aligned} f : \mathbb{Z} \times \mathbb{Z} &\to \mathbb{Z} \\ (x, y) &\mapsto f(x, y) = x + y \end{aligned}$$

- A aplicação do *currying* em f leva as funções

$$\begin{aligned} g : \mathbb{Z} &\to (\mathbb{Z} \to \mathbb{Z}) \\ x &\mapsto g(x) = h_x(y) \end{aligned}$$

e

$$\begin{aligned} h_x : \mathbb{Z} &\to \mathbb{Z} \\ y &\mapsto h_x(y) = x + y \end{aligned}$$

Valoração não-estrita

- No contexto de implementação de linguagens de programação, valoração é o processo de computar o valor de uma expressão
- Valoração **estrita** significa que as expressões são valoradas imediatamente e atribuídas à variável que conterá o retorno tão logo esta última seja definida
- Em chamadas de função, a valoração estrita computa o valor de todos os parâmetros antes da execução do bloco da função
- Já a valoração **não-estrita** não computa o valor de um variável, mesmo em sua definição, postergando este cálculo para o seu primeiro uso
- A valoração não-estrita também é conhecida como *lazy evaluation*
- Por exemplo, o código abaixo não gera um laço infinito, mesmo sendo a lista infinita, pois ela só é computada até o valor especificado

```
-- A saída será [1, 2, 3, 4, 5]
main = print xs where
    xs = take 5 [1..]   -- [1..] corresponde a todos os naturais
```

Pattern matching

- Em programação funcional, *pattern matching* diz respeito ao processo de verificação de objeto contra objeto
- Ela é utilizada para extrair membros dos objetos ou verificar se os objetos tem um tipo específico em uma única expressão
- Isto leva a menos linhas de código com atribuições de variáveis e uma melhor leitura e entendimento do código
- Em geral, são escritas vários padrões possíveis para o objeto, e o objeto será confrontado com cada um deles, na sequência apresentada, até que se encontre um padrão que corresponda ao objeto
- É comum o uso de um caractere ou palavra-chave que corresponda a um padrão universal, que corresponderá a qualquer objeto (*wildcard*)

```
-- Cada uma das duas linhas abaixo corresponde um padrão a ser checado
factorial 0 = 1
factorial n = n * factorial (n - 1)
main = print (factorial 10)
```

Haskell

- Haskell é uma linguagem de programação moderna, puramente funcional
- Ela implementa valoração não-estrita, polimorfismo de tipos, funções de alta-ordem e um sistema de tipagem forte e estrito
- Em termos numéricos, ela oferece inteiros de precisão arbitrária, números racionais, números em ponto flutuante e variáveis booleanas
- Ela foi desenvolvida por um comitê, cuja primeira versão data de 1990
- Dentre as principais motivações para a criação do Haskell estavam:
  i. unificar os esforços de dezenas de diferentes linguagens funcionais,
  ii. ter uma linguagem funcional simples e apropriada para ensino, e
  iii. criar uma linguagem funcional livre

GHC

- GHC (*Glasgow Haskell Compiler*) é um compilador (`ghc`), interpretador (`runghc`) e um ambiente interativo (`ghci`), de código aberto, para a linguagem Haskell
- Ele tem bom suporte para paralelismo e gera códigos rápidos, em especial em programação concorrente
- O GHC oferece, por padrão, uma variedade de bibliotecas, e outras podem ser encontradas em Hackage
- Em Linux, ele pode ser instalado com o comando

```
$ sudo apt-get install ghc
```

- Para rodar o GHCi, use o comando

```
$ ghci
```

- Para mudar o *prompt* do GHCi, use o comando `:set`

```
Prelude> :set prompt "ghci> "
```

Operadores Aritméticos

- Em Haskell, expressões utilizando os operadores aritméticos binários podem ser escritas tanto em forma prefixada quanto na forma infixada:

```
ghci> (+) 6 3        -- Forma prefixada da expressão 6 + 3
```

- Além da adição, a subtração, a multiplicação, a divisão e a exponenciação também estão disponíveis

```
ghci> (-) 6 3        -- 6 - 3 = 3
ghci> (*) 6 3        -- 6 * 3 = 18
ghci> (/) 6 3        -- 6 / 3 = 2.0
```

Operadores Aritméticos

- O operador / representa a divisão em ponto flutuante, não inteira
- Os resultados não geram *overflow*, sendo a aritmética estendida implementada nativamente

```
ghci> 2 ^ 70        -- 2 ^ 70 = 1180591620717411303424
ghci> 2 ** 70       -- pow(2, 70) = 1.1805916207174113e21
```

- Números negativos devem vir entre parêntesis, para evitar ambiguidades

```
ghci> 6 - (-3)      -- 6 + 3 = 9
```

Operadores lógicos e relacionais

- Em Haskell os valores booleanos são `True` e `False`
- Os operadores lógicos são: **e** (`&&`), **ou** (`||`) e **não** (`not`)
- Os operadores relacionais são: **igual** (`==`), **diferente** (`/=`), **menor** (`<`), **menor ou igual** (`<=`), **maior** (`>`) e **maior ou igual** (`>=`)
- No ghci, a precedência dos operadores pode ser consultada por meio do comando `:info`
- O valor 1 significa a menor precedência possível; 9 é a maior precedência possível

```
ghci> :info (+)     -- infixl 6 +
ghci> :info (*)     -- infixl 7 *
ghci> :info (^)     -- infixr 8 *
```

Listas

- As listas são tipos primitivos em Haskell
- Elas são delimitadas por colchetes e os seus elementos são separados por vírgulas

```
ghci> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

- A lista vazia é representada por []
- Todos os elementos de uma lista devem ser do mesmo tipo
- Haskell oferece uma notação que permite a enumeração dos elementos da lista

```
ghci> [1..5]                -- [1, 2, 3, 4, 5]
ghci> [2, 5..18]            -- [2, 5, 8, 11, 14, 17]
```

- Listas podem ser concatenadas por meio do operador ++

```
ghci> [5..6] ++ [1..4]      -- [5, 6, 1, 2, 3, 4]
```

- O operador `cons` (:) adiciona um elemento ao início da lista:

```
ghci> 1 : [2..4]            -- [1, 2, 3, 4]
```

Caracteres e strings

- Um caractere é delimitado por aspas simples

```
ghci> 'a'
```

- Uma string (de caracteres) é delimitada por aspas duplas

```
ghci> "Exemplo de string"
```

- Efetivamente, uma string é uma lista de caracteres

```
ghci> "ABC" == ['A', 'B', 'C']                    -- True
```

- Vale a igualdade: `"" == []`
- Observe também que `"A"` e `'A'` tem tipos distintos
- Como as strings são listas, a notação de enumeração pode ser utilizada:

```
ghci> ['a'..'z'] == "abcdefghijklmnopqrstuvwxyz"    -- True
```

Tipos de dados em Haskell

- O **tipo** de um dado é uma abstração sobre a cadeia de *bytes* que armazena o valor da variável ou constante
- Haskell é uma linguagem com tipagem de dados **forte** e **estática**, onde os tipos das expressões pode ser inferidos **automaticamente**
- Em um sistema de tipagem estática, os tipos dos dados e das expressões devem ser conhecidos em tempo de compilação
- Em um sistema forte, as regras identificação, conversão e validação dos tipos são estritas e aplicadas em tempo de compilação
- Em Haskell, se uma expressão violar as regras de tipagem ela será considerada mal formada e levará a um erro de tipo
- Também não há promoção de tipos ou conversões implícitas de tipos dentro de uma expressão

Tipos de dados em Haskell

- Conversões entre tipos envolvem cópias, o que pode impactar na performance dos programas
- A combinação de tipagem forte e estática faz com que os erros de tipos em Haskell jamais aconteçam em tempo de execução
- O fato de ter um sistema forte e estático torna Haskell uma linguagem segura; a inferência de tipos a torna uma linguagem concisa
- A convenção em Haskell é que tipos de dados iniciem em letras maísculas e as variáveis iniciem em letra minúscula
- No ghci, o tipo de uma expressão pode ser determinado por meio do comando `:type`
- A assinatura de um tipo é

```
expression :: Type
```

Scripts

- Programas em Haskell também podem ser escritos em arquivos, chamados ***scripts***
- Estes *scripts* podem ser interpretados pelo programa `runghc`, ou compilados pelo `ghc`
- O *script* abaixo reproduz parcialmente o comportamento do comando `wc` do Linux, que conta o número de palavras da entrada:

```
-- Para rodar use o comando
--      $ runghc wc.hs
-- ou compile com o comando
--      $ ghc wc.hs
main = interact wc
    where wc input = ((show . length . words) input) ++ "\n"
```

- Comentários iniciam com dois traços (`--`)

Referências

1. **BARENDREGT**, Henk; **BARENDSEN**, Erik. *Introduction to Lambda Calculus*, March 2000.
2. Haskell.org. GHC – The Glasgow Haskell Compiler, acesso em 28/02/2020.
3. Haskell Wiki. What is Haskell?, acesso em 28/02/2020.
4. **HUDAK**, Paul; **HUGHES**, John; **JONES**, Simon P.; **WALDER**, Phillip. *A history of Haskell: being lazy with class*, HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, June 2007, Pages 12-1–12-55, https://doi.org/10.1145/1238844.1238856
5. **SHALOM**, Elad. *A Review of Programming Paradigms Througout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
6. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Relly.