Programação Lógica e Funcional

Paradigma Funcional com Haskell (Parte 3)

Introdução

Este material contém notas de aula sobre o *Paradigma Funcional*. O objetivo é apresentar os conceitos deste paradigma utilizando a linguagem de programação Haskell, em continuidade aos materiais Parte 1 e Parte 2. Propõe-se a continuação do conteúdo de funções de alta ordem, explicando funções não cobertas no material da parte 2, bem como abordando os conceitos de currificação e de composição de funções.

A Linguagem Haskell (continuação)

Anteriormente, aspectos do Paradigma Funcional e uma introdução à Linguagem Haskell foram vistos. Neste material, segue-se apresentando os principais recursos da linguagem.

Funções de Alta Ordem (continuação)

O material anterior definiu função de alta ordem, citou algumas delas e explicou as funções `filter` e `map`. No entanto, o material não cobria as funções `foldr`, `foldl`, `flip` e `zipWith`. Essas funções estão explicadas na sequência desta seção.

RELEMBRANDO

A função `filter` recebe um predicado e uma lista e faz uma filtragem na lista com base no predicado.

O que retorna o código das duas linhas a seguir?

```
import Data.Char
filter isDigit "f9aehhoi34j59"
```

A função `map` realiza o mapeamento de uma função recebida como parâmetro a cada elemento de uma lista também recebida como parâmetro.

O que retornam o códigos a seguir?

```
map round (map sqrt [1, 4, 9, 16, 25])
```

E este código, o que retorna?

```
map succ "HASKELL"
```

- **foldr**: aplica uma função e um valor a uma lista, reduzindo-a. A associatividade das operações é à direita.

```
Prelude> foldr (+) 0 [1..5]
15
Prelude> foldr (-) 0 [1..5]
3
```

- **foldl**: aplica uma função e um valor a uma lista, reduzindo-a. A associatividade das operações é à esquerda.

```
Prelude> foldl (+) 0 [1..5]
15
Prelude> foldl (-) 0 [1..5]
-15
```

- **flip**: recebe uma função e dois valores e aplica a função aos valores em ordem invertida.

```
Prelude> flip (-) 4 5
1
Prelude> flip (-) 5 4
-1
```

- **`zipWith`**: recebe três parâmetros, uma função e duas listas, junta as listas e aplica a função à nova lista. A lista resultante terá o tamanho da menor lista. Exemplos:

```
Prelude> zipWith (+) [1,2] [3,4]
[4,6]
Prelude> zipWith (max) [9,1,11] [3,4]
[9,4] -- Por que não pegou o 11?
```

No caso das funções **`foldr`** e **`foldl`**, a diferença na função de subtração ajuda a compreender suas diferenças. Isso acontece porque a subtração não é associativa, diferentemente da adição. Na adição, (1+2)+3 = 1+(2+3). Na subtração, 5-(4-3) = 4, enquanto (5-4)-3 = -2.

`foldr (-) 0 [1..5]` é o mesmo que 1-(2-(3-(4-(5-0)))), que é o mesmo que 3.
`foldl (-) 0 [1..5]` é o mesmo que ((((0-1)-2)-3)-4)-5, que é o mesmo que -15.

Currificação (baseado em LIPOVAČA)

Currificação (*currying*) é um termo que homenageia o matemático Haskell Curry.

Boa parte das funções utilizadas era composta por uma lista com vários argumentos. Quando a função dispõe de muitos argumentos, a técnica faz com que se crie diversas funções intermediárias, cada uma com um argumento (fazendo, portanto, com que se aplique um argumento por vez).

Na prática, as funções em Haskell recebem somente um argumento por vez.

Um exemplo simples em Haskell está disponível no quadro a seguir:

```
soma :: Int -> Int -> Int
soma x y = x + y

soma2 :: Int -> Int
soma2 = soma 2

result :: Int
result = soma2 3 -- resultado é 5
```

Aqui, a função **soma** recebeu dois argumentos, **x** e **y**, e retornou a soma dos dois (**x+y**). A função **soma2** foi definida a partir da aplicação parcial da função **soma** com o argumento **2**. Isso criou uma nova função que recebe um único argumento **y** e retorna a soma de **2** e **y**.

A função **soma2** pode ser chamada com um único argumento **y** e retorna o resultado da soma de **2** e **y**. No exemplo acima, a chamada **soma2 3** retorna o resultado **5**.

Seja f uma função definida como $f(x, y) = z$, a currificação desta função seria como

$f(x, y) = z$

$h = g(x)$

$z = h(y))$

ou

$(g(x))(y) = z$

Exemplos:

```
Prelude> (compare 10) 1
GT
Prelude> ((-) (length [1])) (length [1,2])
-1
Prelude> (min 8) (-9)
-9
```

Em todos os casos, utilizando o parêntese entre os itens mais à esquerda ou não, o resultado será o mesmo.

Mas faz sentido chamar a função `min 8`?

Se a função for chamada isoladamente, ocorrerá um erro. Com a técnica de currificação, pegando o exemplo utilizando a função **`min`**, é gerada uma nova função chamada **`min 8`** (tal como mostrado no parêntese) que recebe o parâmetro **`(-9)`** (mais uma vez a função recebe só um parâmetro).

Portanto, a função **`min`**, assim como a função **`max`**, tem tipo

```
min :: (Ord a) => a -> a -> a
```

Mas isso também pode ser escrito como

```
min :: (Ord a) => a -> (a -> a)
```

Uma forma de ler a definição acima pode ser: **`min`** recebe um **`a`** e retorna (com o símbolo de seta **`->`**) uma função que recebe um **`a`** e retorna um **`a`**. É por isso que o tipo do retorno e os parâmetros das funções são simplesmente separados por setas (LIPOVAČA, s.d.).

A currificação é vantajosa por facilitar o reúso ao servir de parâmetro para funções de ordem superior. Por exemplo:

```
Prelude> map (+1) [1..3]
[2,3,4]
```

Aqui houve currificação em +1, caso contrário, seria necessário fazer algo como:

```
somaUm x = x + 1
map somaUm [1, 2, 3]
```

Ou então:

```
soma x y = x + y
somaUm = soma 1
map somaUm [1, 2, 3]
```

Claramente, ambas resultam em trabalho maior. Outra possibilidade seria utilizar lambda:

```
map (\x -> x + 1) [1, 2, 3]
```

Esta opção é mais simples?

Funções *currificadas* são mais flexíveis do que as funções com tuplas, porque muitas vezes funções úteis podem ser obtidas pela aplicação parcial de uma função currificada (MALAQUIAS, s.d.).

Composição de Funções

Em Haskell, pode-se compor funções de forma explícita utilizando parênteses, na forma $f(g(x))$. Por exemplo:

```
Prelude> sqrt (abs (16-25))
3.0
```

No entanto, existe um operador, geralmente utilizado de forma infixa, denominado ponto ("**.**"), que realiza esta composição. O exemplo que compõe as funções **sqrt** e **abs** ficaria da seguinte forma com o uso do operador de composição:

```
Prelude> (sqrt . abs ) (16-25)
3.0
```

O operador pode ser utilizado também de forma prefixada, mas isso não traria vantagens, já que continua utilizando parênteses:

```
Prelude> ((.) sqrt abs ) (16-25)
3.0
```

Os três casos descritos são equivalentes e possuem o mesmo tipo. O ganho na legibilidade fica evidente quando a quantidade de funções compostas é maior. Exemplo:

$w = f(g(h(i(j(k\ x)))))$ pode ser escrito como **w = (f . g . h . i . j . k).**

A função **w** é de ordem superior?

Conclusão

Esta terceira parte sobre o Paradigma Funcional com Haskell concluiu a exploração de ***funções de alta ordem***, permitindo a manipulação de funções como dados e proporcionando flexibilidade e expressividade ao código. Apresentou-se também o conceito de ***currificação***, uma técnica poderosa que permite criar funções parcialmente aplicadas, simplificando o código e promovendo o reúso de lógica.

Por fim, explorou-se a composição de funções, tanto de forma explícita quanto utilizando o operador ponto ("`.`"). Esta técnica permite combinar funções de maneira legível, contribuindo para a clareza e para a organização do código.

Exercícios

1) Considerando a representação matemática das funções foldr e foldl,
    a) Qual é o resultado de **`foldr (*) 1 [1..5]`**?
    b) Qual é o resultado de **`foldl (*) 1 [1..5]`**?
    c) O que as funções dos exercícios *a* e *b* fazem?
    d) Resolva demonstrando matematicamente como foram realizadas as operações na questão *a*.
    e) Resolva demonstrando matematicamente como foram realizadas as operações na questão *b*.

2) Resolva os exercícios da Lista #7.

Referências

LIPOVAČA, Miran. High Order Functions. Disponível em: http://learnyouahaskell.com/higher-order-functions. Acesso em: 02 mai. 2024.

MALAQUIAS, José Romildo. Programação Funcional em Haskell. Disponível em: http://www.decom.ufop.br/romildo/2018-1/bcc222/slides/progfunc.pdf. Acesso em: 02 mai. 2024.