Programação Funcional com Haskell – Definições recursivas

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Definições usando outras funções

- Podemos definir uma função usando outras previamente definidas (e.g. no Standard Prelude)

  - fatorial :: Int -> Int
    fatorial n = product [1..n]

Definições recursivas

- Também podemos definir uma função usando a própria função que estamos definindo; tais definições são denominadas recorrências ou recursões

  - ```
    fatorial :: Int -> Int
    fatorial 0 = 1
    fatorial (n+1) = (n+1)*fatorial n
    ```
    ```

Exemplo

fatorial 3

=

3 * fatorial 2

=

3 * (2 * fatorial 1)

=

3 * (2 * (1 * fatorial 0))

=

3 * (2 * (1 * 1))

=

6

Observações

- A primeira equação define o fatorial de zero

- A segunda equação define o fatorial de inteiros positivos (por causa do padrão n + 1)

- Portanto, fatorial fica definido apenas para inteiros não-negativos
  - > factorial (-1)

    Program error: pattern match failure: factorial (-1)

Alternativas

- Duas equações sem padrões n + k
  - fatorial 0 = 1
    fatorial n = n * factorial (n-1)

- Uma equação com guardas
  - fatorial n | n==0 = 1
    | otherwise = n*fatorial (n-1)

- Uma equação com uma condição
  - fatorial n = if n==0 then 1 else n*fatorial (n-1)

Porquê usar recursão?

- É um modelo universal de computação - qualquer algoritmo pode ser escrito usando funções recursivas

- Muitas vezes é mais simples resolver um problema usando a solução de sub-problemas semelhantes mas de menor tamanho

- Podemos demonstrar propriedades de funções definidas recursivamente usando indução matemática

Recursão sobre Listas

- Também podemos definir funções recursivas sobre listas
  - Exemplo: a função que calcula o produto de uma lista de números (Standard Prelude)

```
product :: Num a => [a] -> a
product      [ ]  = 1
product (x:xs) = x*product xs
```

  - Num é um tipo numérico (Ex: Int, Float)
  - Num a (restrição de classe)

Exemplo de recursão

product [2,3,4]

=

2 * product [3,4]

=

2 * (3 * product [4])

=

2 * (3 * (4 * product [ ]))

=

2 * (3 * (4 * 1))

=

24

A função length

- O comprimento de uma lista também pode ser definido por recursão

  - length :: [a] -> Int
    length [ ] = 0
    length (_:xs) = 1 + length xs

Exemplo de recursão

length [1,2,3]

=

1 + length [2,3]

=

1 + (1 + length [3])

=

1 + (1 + (1 + length []))

=

1 + (1 + (1 + 0))

=

3

A função reverse

- A função reverse (que inverte a ordem dos elementos de uma lista) também pode ser definida recursivamente

  - ```
    reverse :: [a] -> [a]
    reverse [ ] = [ ]
    reverse (x:xs) = reverse xs ++ [x]
    ```
    ```

Exemplo de recursão

```
reverse [1,2,3]
=
reverse [2,3] ++ [1]
=
(reverse [3] ++ [2]) ++ [1]
=
((reverse [ ] ++ [3]) ++ [2]) ++ [1]
=
(([ ] ++ [3]) ++ [2]) ++ [1]
=
[3,2,1]
```

Funções com múltiplos argumentos (1/2)

- Podemos definir recursivamente funções com múltiplos argumentos

- A função zip que constroi a lista dos pares de elementos de duas listas
  - ```
    zip :: [a] -> [b] -> [(a,b)]
    zip [ ] _ = [ ]
    zip _ [ ] = [ ]
    zip (x:xs) (y:ys) = (x,y) : zip xs ys
    ```
    ```

Funções com múltiplos argumentos (1/2)

- Podemos definir recursivamente funções com múltiplos argumentos

- A função zip que constroi a lista dos pares de elementos de duas listas
  - zip :: [a] -> [b] -> [(a,b)]
    zip [ ] _ = [ ]
    zip _ [ ] = [ ]
    zip (x:xs) (y:ys) = (x,y) : zip xs ys

Funções com múltiplos argumentos (2/2)

- A função drop que remove um prefixo de uma lista
  - drop :: Int -> [a] -> [a]

    drop 0 xs = xs

    drop (n+1) [ ] = [ ]

    drop (n+1) (x:xs) = drop n xs

Recursão mútua

- Podemos também definir duas ou mais funções que dependem mutamente umas das outras

- Testar se um número natural é par ou impar
  - ```
    par :: Int -> Bool
    par 0 = True
    par (n+1) = impar n
    ```
  - ```
    impar :: Int -> Bool
    impar 0 = False
    impar (n+1) = par n
    ```

Quicksort

- O algoritmo Quicksort para ordenação de uma lista pode ser especificado de forma recursiva
  - Se a lista é vazia então já está ordenada
  - Se a lista não é vazia seja x o primeiro valor e xs os restantes
    1. Recursivamente ordenamos os valores de xs que são menores ou iguais a x
    2. Recursivamente ordenamos os valores de xs que são maiores do que x
    3. Concatenamos os resultados com x no meio

Quicksort em Haskell

- qsort :: [Int] -> [Int]

```
qsort [ ]       = [ ]
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
        where menores = [x' | x'<-xs, x'<=x]
                  maiores = [x' | x'<-xs, x'>x]
```

- Definição de uma lista a partir de outras
  - > [x^2 | x<-[1,2,3,4,5]]

    [1, 4, 9, 16, 25]
  - > [(x,y) | x<-[1,2,3], y<-[4,5]]

    [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

Exemplo de execução

qsort [3,2,4,1,5]

=

qsort [2,1] ++ [3] ++ qsort [4,5]

=

(qsort [1]++[2]++qsort []) ++ [3] ++ (qsort []++[4]++qsort [5])

=

([1]++[2]++[]) ++ [3] ++ ([]++[4]++[5])

=

[1,2,3,4,5]

Como escrever definições recursivas

- Definir o tipo da função
- Enumerar os casos a considerar usando equações com padrões
- Definir o valor nos casos simples
- Definir o valor nos outros casos assumindo que a função está definida para valores de tamanho inferior
- Generalizar e simplificar

Exemplo (1/6)

- Escrever uma definição recursiva da função init que remove o último elemento de uma lista

  - > init [1,2,3,4,5]
    [1,2,3,4]

  - > init [1]
    [ ]

  - > init [ ]
    Program error: pattern match failure: init [ ]

Exemplo (2/6)

- Passo 1: O tipo da função
  - init :: [a] -> [a]

Exemplo (3/6)

- Passo 2: Enumerar os casos
  - init :: [a] -> [a]
    init (x:xs) =
- Note que init **não** está definido para a lista vazia

Exemplo (4/6)

- Passo 3: Definir o caso simples

  - ```
    init :: [a] -> [a]
    init (x:xs) | null xs = [ ]
                | otherwise =
    ```

- Note que se xs é a lista vazia então a lista (x:xs) tem um só elemento
    ```

Exemplo (5/6)

- Passo 4: Definir o caso recursivo
  - init :: [a] -> [a]

```
init (x:xs) | null xs = [ ]
            | otherwise = x : init xs
```

- Note que se xs é a lista vazia então a lista (x:xs) tem um só elemento

Exemplo (6/6)

- Passo 5: Simplificação
  - init :: [a] -> [a]
    init [_] = [ ]
    init (x:xs) = x : init xs
- Podemos separar o caso da lista com um só elemento em uma equação e assim eliminar as guardas