Linguagens de Programação

Programação Funcional e Haskell
Funções Recursivas

Thiago Alves

Introdução

Várias funções podem ser definidas naturalmente em termos de outras funções

```
fac  :: Int → Int
fac n = product [1..n]
```

fac mapeia qualquer inteiro n para o produto dos inteiros entre 1 e n.

Expressões são avaliadas por um processo passo-a-passo de aplicações de funções ao seus argumentos.

Por exemplo:

```
    fac 4
=
    product [1..4]
=
    product [1,2,3,4]
=
    1*2*3*4
=
    24
```

Funções Recursiva

Em Haskell, funções podem ser definidas em termos delas mesmas. Tais funções são chamadas recursivas.

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac mapeia 0 para 1, e qualquer outro inteiro para o produto dele mesmo e do fac do seu predecessor.

Por exemplo:

```
    fac 3
=
    3 * fac 2
=
    3 * (2 * fac 1)
=
    3 * (2 * (1 * fac 0))
=
    3 * (2 * (1 * 1))
=
    3 * (2 * 1)
=
    3 * 2
=
    6
```

- fac 0 = 1 é apropriado porque 1 é a identidade da multiplicação: 1*x = x = x*1.

- A definição recursiva diverge em inteiros negativos pois o caso base nunca é alcançado:

```
> fac (-1)

Exception: stack overflow
```

Utilidade da Recursão?

- Algumas funções, como o fatorial, são mais simples de definir em termos de outras funções.
- Entretanto, várias funções podem ser definidas naturalmente em termos delas mesmas.
- Propriedades de funções definidas usando recursão podem ser provadas usando indução.

Podemos definir a multiplicação para inteiros não-negativos de forma recursiva:

```
(*) :: Int → Int → Int
m * 0 = 0
m * (n+1) = m + (m * n)
```

Exemplo:

$$
\begin{aligned}
& 4 * 3 \\
= \; & 4 + (4 * 2) \\
= \; & 4 + (4 + (4 * 1)) \\
= \; & 4 + (4 + (4 + (4 * 0))) \\
= \; & 4 + (4 + (4 + (0))) \\
= \; & 12
\end{aligned}
$$

Recursão em Listas

Recursão também pode ser usada para definir funções em listas.

```
product :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product mapeia a lista vazia para 1, e qualquer outra lista para sua cabeça multiplicada pelo product da sua cauda.

Por exemplo:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Usando o mesmo padrão de recursão como no product, podemos definir a função length.

length :: [a] → Int

Usando o mesmo padrão de recursão como no product, podemos definir a função length.

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length mapeia a lista vazia para 0, e qualquer outra lista ao sucessor do length da cauda.

Por exemplo:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Usando um padrão similar de recursão, podemos definir a função reverse em listas.

```
reverse :: [a] → [a]
```

Usando um padrão similar de recursão, podemos definir a função reverse em listas.

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

reverse mapeia a lista vazia para a lista vazia, e qualquer outra lista para o reverso da sua cauda concatenado com sua cabeça.

Por exemplo:

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

Inserir um elemento em uma lista ordenada na posição correta:

```
insert :: Ord a => a → [a] → [a]
```

Inserir um elemento em uma lista ordenada na posição correta:

```
insert :: Ord a => a → [a] → [a]
insert x [] = [x]
insert x (y:ys) | x <= y = x:y:ys
                | otherwise = y:insert x ys
```

Exemplo:

insert 3 [1,2,4,5]

=

1:insert 3 [2,4,5]

=

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

=

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

=

[1,2,3,4,5]

Podemos definir uma função que implementa o insertion sort usando a função insert:

```
isort :: Ord a => [a] → [a]
```

Podemos definir uma função que implementa o insertion sort usando a função insert:

```
isort :: Ord a => [a] → [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Exemplo:

```
  isort [3,2,1,4]
=
  insert 3 (isort [2,1,4])
=
  insert 3 (insert 2 (insert 1 (insert 4 [])))
=
  insert 3 (insert 2 (insert 1 [4]))
=
  insert 3 (insert 2 [1,4])
=
  insert 3 [1,2,4]
=
  [1,2,3,4]
```

Múltiplos Argumentos

Funções com mais que um argumento também podem ser definidas usando recursão. Por exemplo:

- Zipping os elementos de duas listas:

```
zip :: [a] → [b] → [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Exemplo:

```
  zip ['a','b','c'] [1,2,3,4]
=
  ('a',1):zip ['b','c'] [2,3,4]
=
  ('a',1):('b',2):zip ['c'] [3,4]
=
  ('a',1):('b',2):('c',3):zip [] [4]
=
  ('a',1):('b',2):('c',3):[]
=
  [('a',1),('b',2),('c',3)]
```

- Remover os primeiros n elementos de uma lista:

```
drop :: Int → [a] → [a]
```

- Remover os primeiros n elementos de uma lista:

```
drop :: Int → [a] → [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

- Concatenar duas listas:

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

- Concatenar duas listas:

```
(++) :: [a] → [a] → [a]
[]  ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Exemplo:

```
  [1,2,3] ++ [4,5]
=
  1:([2,3] ++ [4,5])
=
  1:(2:([3] ++ [4,5]))
=
  1:(2:(3:([] ++ [4,5])))
=
  1:(2:(3:[4,5]))
=
  [1,2,3,4,5]
```

Recursão Múltipla

A função é aplicada mais de uma vez na sua definição

- Fibonacci:

```
fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

O algoritmo quicksort para ordenar uma lista de valores pode ser especificado pelas duas regras a seguir:

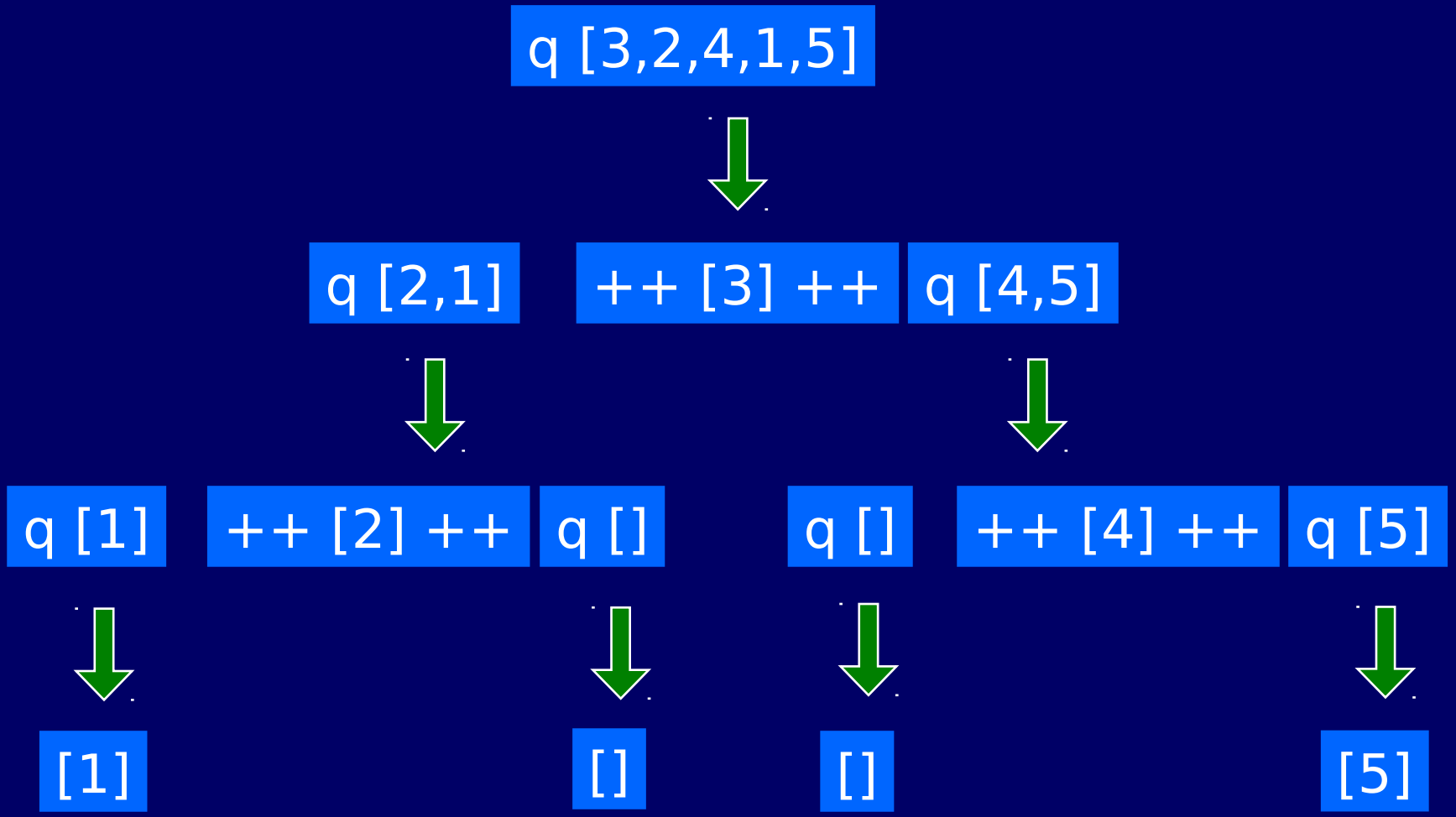
- A lista vazia já está ordenada;
- Listas não-vazias podem ser ordenadas através da ordenação dos valores da cauda $\leq$ que a cabeça, da ordenação dos valores da cauda $>$ que a cabeça, e concatenando as listas resultantes em cada lado do valor da cabeça.

Usando recursão, essa especificação pode ser traduzida diretamente em uma implementação:

```
qsort :: Ord a ⇒ [a] → [a]
qsort [] = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

- Essa é provavelmente a implementação mais simples do quicksort em qualquer linguagem de programação!

Por exemplo (abreviando qsort como q):



Recursão Mútua

Duas ou mais funções são definidas em termos umas das outras

- Exemplo:

```
even :: Int → Bool
even 0 = True
even (n+1) = odd n
odd :: Int → Bool
odd 0 = False
odd (n+1) = even n
```

Funções que selecionam os elementos das posições ímpares e das pares:

```
evens :: [a] → [a]

odds :: [a] → [a]
```

Funções que selecionam os elementos das posições ímpares e das pares:

```
evens :: [a] → [a]
evens [] = []
evens (x:xs) = x:odds xs
odds :: [a] → [a]
odds [] = []
odds (x:xs) = evens xs
```

Exemplo:

evens “abc”

=

'a':odds “bc”

=

'a':evens “c”

=

'a':'c':odds []

=

'a':'c':[]

=

“ac”