Programação com Haskell – Listas Infinitas

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

Listas infinitas

- Já vimos que podemos usar listas para representar sequências finitas

  [1,2,3,4] = 1:2:3:4:[ ]

- Nesta aula vamos ver que também podemos usar listas para representar sequências infinitas

  [1..] = 1:2:3:4:5:...

- Ao contrário de uma lista finita, não podemos descrever uma lista infinita em extensão, usamos representações em compreensão ou definições recursivas

Exemplos

```
-- todos os números naturais
nats :: [Int]
nats = [0..]

-- todos os pares não-negativos
pares :: [Int]
pares = [0,2..]

-- a lista infinita 1, 1, 1, ...
uns :: [Int]
uns = 1 : uns

-- todos os inteiros a partir de n
ints :: Int -> [Int]
ints n = n : ints (n+1)
```

Processamento de listas infinitas (1/2)

- Por causa da **avaliação preguiçosa** (lazy evaluation) podemos trabalhar com listas infinitas, as listas são calculadas à medida da necessidade e apenas até onde for necessário

  head (uns)

  =

  head (1:uns)

  =

  1

Processamento de listas infinitas (2/2)

- Uma computação que necessite percorrer toda a lista infinita não termina

length uns
=
length (1:uns)
=
1 + length uns
=
1 + length (1:uns)
=
1 + (1 + length uns)
=
... não termina

Produzir listas infinitas (1/4)

- Muitas funções do prelude padrão produzem listas infinitas quando os argumentos são listas infinitas

```
> map (2*) [1..]
[2, 4, 6, 8, 10, ...

> filter odd [1..]
[1, 3, 5, 7, 9, ...
```

- Também podemos usar notação em compreensão

```
> [2*x | x<-[1..]]
[2, 4, 6, 8, 10 ...

> [x | x<-[1..], odd x]
[1, 3, 5, 7, 9 ...
```

Produzir listas infinitas (2/4)

- Algumas funções do prelude padrão produzem especificamente listas infinita

```
repeat :: a -> [a]
-- repeat x = [x,x,x, ...]

cycle :: [a] -> [a]
-- cycle xs = xs++xs++xs++...

iterate :: (a -> a) -> a -> [a]
-- iterate f x = [x, f x, f(f x), f(f(f x)), ...]
```

- (Note que iterate é de ordem superior - o 1o argumento é uma função)

Produzir listas infinitas (3/4)

- Podemos testar no interpretador pedido de prefixos finitos

```
> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]

> take 10 (repeat 'a')
"aaaaaaaaaa"

> take 10 (cycle [1,-1])
[1,-1,1,-1,1,1,-1,1,-1,1]

> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512]
```

Produzir listas infinitas (4/4)

- As funções repeat, cycle e iterate estão definidas no prelude padrão usando recursão

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs

cycle :: [a] -> [a]
cycle [] = error "empty list"
cycle xs = xs' where xs' = xs++xs'

iterate :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Por que usar lista infinitas?

- Permite simplificar o processamento de listas combinando listas finitas com infinitas (e.g. evitar especificar comprimentos)

- Permite separar a geração e o consumo de sequências (e.g. aproximação numérica)

- Mais geralmente, permite maior modularidade na escrita de programas

Preenchimento de texto (1/3)

- Um exemplo simples - escrever uma função

```
preencher :: Int -> String -> String
```

que preenche uma cadeia com espaços de forma a somar n caracteres

- Se a cadeia já tiver comprimento n ou maior, deve ser truncada para n caracteres

Preenchimento de texto (2/3)

- Exemplos

```
> preencher 10 "Haskell"
"Haskell   "

> preencher 10 "Haskell B. Curry"
"Haskell B."
```

Preenchimento de texto (3/3)

- Solução usando take e uma lista infinita

```
preencher n xs = take n (xs++repeat ' ')
```

Aproximação da raiz quadrada (1/3)

- Segundo exemplo - calcular uma aproximação de $\sqrt{q}$ pelo método babilônico (um caso particular do método de Newton-Rapson para resolução numérica de equações)
  1. Começamos com $x_0 = q$
  2. Em cada passo, melhoramos a aproximação tomando

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{q}{x_n}\right)$$

  3. Terminamos quando se verificar um critério de parada

erro absoluto $|x_{n+1} - x_n| < \epsilon$

erro relativo $|(x_{n+1} - x_n)/x_n| < \epsilon$

Aproximação da raiz quadrada (2/3)

- sucessão infinita de aproximações da raiz quadrada

```
babilon :: Double -> [Double]
babilon q = iterate f q
     where f x = 0.5*(x+q/x)
```

- critérios de parada

```
absolute :: [Double] -> Double -> Double
absolute xs eps = head [x | (x,x')<-zip xs (tail xs),
                            abs(x-x')<eps]

relative :: [Double] -> Double -> Double
relative xs eps = head [x | (x,x')<-zip xs (tail xs),
                            abs((x-x')/x')<eps]
```

Aproximação da raiz quadrada (3/3)

- Exemplos de uso

```
> take 5 (babilon 2)
[2.0,1.5,1.4166667,1.4142157,1.4142135]

> absolute (babilon 2) 0.01
1.4166667

> relative (babilon 2) 0.001
1.4142157
```

A sequência de Fibbonacci (1/3)

- Terceiro exemplo - a sequência de Fibbonacci
  - começa com 0, 1
  - cada valor seguinte é a soma dos dois anteriores

    0, 1, 1, 2, 3, 5, 8, 13, … , n, m, n+m, …

A sequência de Fibbonacci (2/3)

- Solução em Haskell: uma lista infinita definida recursivamente

```
fibs :: [Int]
fibs = 0 : 1 : [n+m | (n,m)<-zip fibs (tail fibs)]
```

A sequência de Fibbonacci (3/3)

- Os primeiros dez números de Fibbonacci

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

- O nono número de Fibbonacci (índices começam em 0)

```
> fibs!!8
21
```

- O primeiro número de Fibbonacci superior a 100

```
> head (dropWhile (<100) fibs)
144
```

O crivo de Eratóstenes (1/3)

- Outro exemplo - usar o crivo de Eratóstenes para gerar todos os números primos

1. Começar com a lista dos inteiros iniciando em 2
2. Marcar como primo o primeiro número p na lista
3. Remover p e todos os múltiplos de p da lista
4. Repetir o passo 2

- Observar que o passo 3 envolve processar uma lista infinita

O crivo de Eratóstenes (2/3)

- Em Haskell

```
primos :: [Int]
primos = crivo [2..]

crivo :: [Int] -> [Int]
crivo (p:xs) = p : crivo [x | x<-xs, x`mod`p/=0]
```

O crivo de Eratóstenes (3/3)

- Os primeiros 10 primos

```
> take 10 primos
[2,3,5,7,11,13,17,19,23,29]
```

- Quantos primos são inferiores a 100?

```
> length (takeWhile (<100) primos)
25
```