Definição de Tipos em Haskell

Prof. Rodrigo Ribeiro

Programação Funcional em Haskell

Definição de Tipos em Haskell — (I)

- Definindo tipos de dados em Haskell
  - Sinônimos de tipos.
  - Renomeamento de tipos.
  - Tipos de dados algébricos. - Sintaxe para registros.

Definição de Tipos em Haskell — (II)

- Sinônimos de tipos
- Usado para dar um novo "nome" para um tipo.
- O tipo original e o "novo" podem ser intercambiados.

```haskell
type Name = String

doubleName :: Name -> Name
doubleName n = n ++ n

f :: String -> Name
f = doubleName
```

Definição de Tipos em Haskell — (III)

- Renomeamento de tipos
  - Permite dar um novo "nome" a um tipo existente.
  - Novo tipo não é intercambiável com novo tipo.

```
newtype Age = Age Int

g :: Age -> Bool
g (Age x) = x > 0
```

Definição de Tipos em Haskell — (IV)

- Note que Age e Int são tipos diferentes. . .

```
h = g 0 -- wrong

h = g (Age 0) -- ok
```

Definição de Tipos em Haskell — (V)

- Tipos de dados algébricos
  - Mecanismo mais expressivo para definição de tipos
  - Tipos podem ser definidos como um conjunto de construtores de dados
  - Cada construtor de dados, especifica um formato de um valor do tipo.

Definição de Tipos em Haskell — (VI)

- Exemplo:
  - Construtor INil: árvore "vazia"
  - Construtor INode: árvore contendo um inteiro e duas subárvores.

```
data IntTree = INil | INode Int IntTree IntTree
```

Definição de Tipos em Haskell — (VII)

- Definindo funções sobre tipos de dados algébricos

```
sumIntTree :: IntTree -> Int
sumIntTree INil = 0
sumIntTree (INode n l r) = n + sumIntTree l + sumIntTree r
```

Definição de Tipos em Haskell — (VIII)

- Mais exemplos...
  - Um tipo de dados algébrico para representar clientes:

```
data Client = Client String   -- nome
                     String   -- endereço
                     Int      -- idade
```

Definição de Tipos em Haskell — (IX)

- Sobre a definição anterior...
  - Pouco informativa. Como diferenciar a primeira da segunda String?

```
type Name = String
type Address = String
type Age = Int

data Client = Client Name
                     Address
                     Age
```

Definição de Tipos em Haskell — (X)

- Funções sobre o tipo Client:

```haskell
type Name = String
type Address = String
type Age = Int

data Client = Client Name Address Age

name :: Client -> Name
name (Client n _ _) = n

address :: Client -> Address
address (Client _ a _) = a

age :: Client -> Age
age (Client _ _ a) = a
```

Definição de Tipos em Haskell — (XI)

- Definições repetitivas...
  - Somente para obter parte da informação de um tipo de dados.
- Solução: Definição usando sintaxe de registro.
  - Vantagens: funções de projeção definidas automaticamente.

```
type Address = String
data Client = Client {
                name :: Name,
                address :: Address,
                age :: Age
              }
```

Definição de Tipos em Haskell — (XII)

- Tipos de dados podem ser polimórficos!
- Exemplos:

```
type IntRight a = (a,Int)

newtype DiffList a = DiffList { out :: [a] -> [a] }

data MyList a = Empty | OneMore a (MyList a)
              deriving (Eq, Ord, Show)
```

Definição de Tipos em Haskell — (XIII)

- Tipos polimórficos:
  - Variáveis de tipos representam "qualquer" tipo.
- Exemplos de tipos polimóricos: listas, tuplas
- Mais exemplos

```
Tree Int, Tree (a,Bool),
[Tree Char], MyList Char
([Char], Tree String),...
```

Definição de Tipos em Haskell — (XIV)

- Continuando com o tipo MyList...

```
sumMyList :: Num a => MyList a -> a
sumMyList Empty = 0
sumMyList (OneMore x xs) = x + sumMyList xs

filterMyList :: (a -> Bool) -> MyList a -> MyList a
filterMyList p Empty = Empty
filterMyList p (OneMore x xs)
              | p x = OneMore x (filterMyList p xs)
              | otherwise = filterMyList p xs
```

Definição de Tipos em Haskell — (XV)

- Note que o tipo MyList a é equivalente ao tipo [a]...
  - Isso pode ser mostrado por funções que convertem um tipo em outro:

```
toList :: MyList a -> [a]
toList Empty = []
toList (OneMore x xs) = x : toList xs

toMyList :: [a] -> MyList a
toMyList [] = Empty
toMyList (x:xs) = OneMore x (toMyList xs)
```

Definição de Tipos em Haskell — (XVI)

- Note que a função toMyList pode ser definida usando foldr:

```
toMyList' :: [a] -> MyList a
toMyList' = foldr OneMore Empty
```

- Dessa forma, cabe a pergunta: toList pode ser definida usando foldr?
  - Sim! Mas para isso precisamos definir a função foldr para o tipo MyList.

Definição de Tipos em Haskell — (XVII)

- Definição de fold para o tipo MyList

```
foldMyList :: (a -> b -> b) -> b -> MyList a -> b
foldMyList f v Empty = v
foldMyList f v (OneMore x xs) = f x (foldMyList f v xs)
```

- Com isso, a definição de toList ficaria...

```
toList' :: MyList a -> [a]
toList' = foldMyList (:) []
```

Definição de Tipos em Haskell — (XVIII)

- Mais um exemplo: árvores binárias

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
              deriving (Eq, Ord, Show)
```

- Exemplo

```
t :: Tree Int
t = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
```

Definição de Tipos em Haskell — (XIX)

- Exemplos: Funções sobre árvores polimórficas

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r)
      | x < y = Node y (insert x l) r
      | otherwise = Node y l (insert x r)
```

Definição de Tipos em Haskell — (XX)

- Mais funções...

```
member :: Ord a => a -> Tree a -> Bool
x `member` Leaf = False
x `member` (Node y l r) = x == y        ||
                          x `member` l  ||
                          x `member` r

size :: Tree a -> Int
size Leaf = 0
size (Node x l r) = 1 + size l + size r
```

Definição de Tipos em Haskell — (XXI)

- Note que as funções member e size, possuem um padrão:
  - Retornar um valor quando a árvore é vazia
  - Aplicar uma função aos valores do nó e aos resultados de processar recursivamente as subárvores.
- Esse padrão é similar ao foldr para listas!
  - Substitua o construtor [] por um valor fixo
  - Substitua o construtor (:) por uma função de dois parâmetros

Definição de Tipos em Haskell — (XXII)

- A idéia da função fold para árvores binárias é similar:
  - Substitua o construtor Leaf por um valor fixo
  - Substitua o construtor Node por uma função de *três* parâmetros.
- Porquê três parâmetros?
  - Um representa o valor associado ao nó
  - Outros dois representam as duas subárvores esquerda e direita.

Definição de Tipos em Haskell — (XXIII)

- Eis a definição de fold para árvores binárias.

```
fold :: (a -> b -> b -> b) -> b -> Tree a -> b
fold f v Leaf = v
fold f v (Node x l r) = f x (fold f v l) (fold f v r)
```

Definição de Tipos em Haskell — (XXIV)

- Com isso, size e member ficam:

```
size' :: Tree a -> Int
size' = fold (\x y z -> 1 + y + z) 0

member' :: Ord a => a -> Tree a -> Bool
member' x = fold (\k y z -> x == k || y || z) False
```

Definição de Tipos em Haskell — (XXV)

- Mas exemplos, usando fold:
  - Caminhamento em pré-ordem

```
preorder :: Tree a -> [a]
preorder = fold (\x l r -> x : (l ++ r)) []
```

Definição de Tipos em Haskell — (XXVI)

- Próxima semana...
  - Um exemplo de tipos de dados algébricos: Um compilador de expressões aritméticas para uma máquina de pilha!
  - Reunião na quarta-feira às 18:00 hrs!

Definição de Tipos em Haskell — (XXVII)

- Tarefas para o recesso:
  - Exercícios para serem resolvidos e apresentados na primeira sexta-feira.
  - Todo o código produzido por vocês deve ser disponibilizado no github.
- Como organizar seu código no github:
  - Crie um repositório chamado cursoHaskell2013
  - Coloque suas soluções para cada lista de exercícios em uma pasta chamada listaX, onde X é o número da lista de exercícios em questão.