Tipos Algébricos

Márcio Lopes Cornélio

Centro de Informática - UFPE

Tipos algébricos

Útil para

- representar tipos enumerados. Por exemplo, meses: Janeiro, ..., Dezembro
- representar um tipo cujos elementos podem ser um inteiro ou uma string (uniões disjuntas)
- representar o tipo árvore (tipo recursivo)
- Tipos-sinônimo não podem ser recursivos

Tipos enumerados

Permitem criar novos tipos de dados e novos construtores de tipos

```
data Bool = True | False

data Estacao = Inverno | Verao | Outono | Primavera

data Temp = Frio | Quente
```

Tipos enumerados

Funções usam casamento de padrões

```
clima :: Estacao -> Temp
clima Inverno = Frio
clima _       = Quente
```

Relembrando: casamento de padrões utiliza construtores de tipos (listas, tuplas, etc.)

Produtos

```haskell
type Nome = String
type Idade = Int
data Pessoas = Pessoa Nome Idade

Pessoa "Jose" 22
Pessoa "Maria" 23

showPerson :: Pessoas -> String
showPerson (Pessoa n a) = n ++ "--" ++ show a

Pessoa :: Nome -> Idade -> Pessoas
```

Por que não usar tuplas?

```
type Pessoas = (Nome, Idade)
```

Usando tipos algébricos

- cada objeto do tipo tem um rótulo explícito
- não se pode confundir um tipo com outro, devido ao construtor (definições fortemente tipadas)
- tipos recursivos e enumerados

Usando tipos sinônimos

- elementos mais compactos, definições mais curtas
- possibilidade de reusar funções polimórficas

Alternativas

Construtores com argumentos

```
data Shape = Circle Float
           | Rectangle Float Float

Circle 4.9 :: Shape
Rectangle 4.2 2.0 :: Shape


isRound :: Shape -> Bool
isRound (Circle _) = True
isRound (Rectangle _ _) = False
```

Como definir a função abaixo?

```
area :: Shape -> Int
```

Forma geral

```
data Nome_do_Tipo
  = Construtor1 t11 ... t1k1
  | Construtor2 t21 ... t2k2
    ....
  | Construtorn tn1 ... tnkn
```

O tipo pode ser recursivo

A definição pode ser polimórfica, adicionando argumentos ao `Nome_do_Tipo`

Tipos recursivos

Tipos de dados recursivos

```
data Expr = Lit Int         |
          Add Expr Expr     |
          Sub Expr Expr
```

Funções definidas recursivamente

```
eval :: Expr -> Int
eval (Lit n)      = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Tipos de dados polimórficos

Polimorfismo

```
data Pairs t = Pair t t

Pair 6 8 :: Pairs Int
Pair True True :: Pairs Bool
Pair [] [1,3] :: Pair [Int]
```

Listas

```
data List t = Nil | Cons t (List t)
```

Árvores

```
data Tree t = NilT | Node t (Tree t) (Tree t)
```

Tipos polimórficos

Tipo para união

```
data Either a b = Left a | Right b
```

Tipo para lidar com erros

```
data Maybe a = Nothing | Just a
```

Exercícios

Defina as seguintes funções

```
showExpr :: Expr -> String
toList :: List t -> [t]
fromList :: [t] -> List t
depth :: Tree t -> Int
collapse :: Tree t -> [t]
mapTree :: (t -> u) -> Tree t -> Tree u
```

Bibliografia

Slides elaborados a partir de originais por André Santos e Fernando Castor

[1] Simon Thompson.
Haskell: the craft of functional programming.
Addison-Wesley, terceira edition, Julho 2011.