Programação Funcional
Tipos Algébricos

Vander Alves

Tipos algébricos

- representar meses: Janeiro, ..., Dezembro
- representar um tipo cujos elementos podem ser um inteiro ou uma string
- representar o tipo árvore

Tipos Enumerados

- 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
```

Produtos

```
type Name = String
type Age = Int
data People = Person Name Age

Person ”José” 22
Person ”Maria” 23

showPerson :: People -> String
showPerson (Person n a) = n ++ ” -- ” ++ show a

Person :: Name -> Aqe -> People
```

por que não usar tuplas?

```
type People = (Name, Age)
```

- Com tipos algébricos
  - cada objeto do tipo tem um label explícito
  - não se pode confundir um tipo com outro, devido ao construtor (definições bem tipadas)
  - o tipo aparecerá nas mensagens de erro
- Com tipos sinônimos
  - elementos mais compactos, definições mais curtas
  - possibilidade de reusar funções polimórficas

quando usar?

```
data Age = Years Int
type Age = Int

f1 :: Int -> Int
f2 :: Age -> Int
```

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
```

Alternativas

```
area :: Shape -> Int
area (Circle r) = pi*r*r
area (Rectangle h w) = h * w
```

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 polimórficos

- Tipos de dados polimórficos:

```
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)
```

Derivando instâncias de classes

```
data List t = Nil | Cons t (List t)
              deriving (Eq,Ord,Show)

data Tree t = NilT |
              Node t (Tree t) (Tree t)
              deriving (Eq,Ord,Show)
```

Exercícios

- Defina as seguintes funções

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

Maybe

```
Maybe t = Just t | Nothing
          deriving (Eq,Ord,Show)
```

- Indicar erros

```
Nothing

Nothing /= Just 10

Nothing /= Just (“Maria”, 200)
```

- Possibilita tratamento de erros
  - *Robustez*

Maybe

Indicar erros

```
saldo :: String -> [(String,Float)] -> Maybe Float
saldo _ [] = Nothing
saldo person ((p,s):pss)
             | person == p = Just s
             | otherwise = saldo person pss

saldo “Maria” [(“Jose”, 10), (“Maria”,20)] → Just 20.0
saldo “Pedro” [(“Jose”, 10), (“Maria”,20)] → Nothing
```

Maybe

Definição de Funções Parciais

```
f :: Float -> Float -> Float
f x y = x / y

f 0 1 →  0.0
f 1 0  →  ?

f :: Float -> Float -> Maybe Float
f x y
    | y == 0 = Nothing
    | otherwise = Just (x / y)

f 0 1  →   Just 0.0
f 1 0  →   Nothing
```

Maybe

Definição de Funções Parciais

```
import Data.Maybe    -- fromJust
fat :: Int -> Maybe Int
fat 0 = 1
fat n
    | n < 0 = Nothing
    | n > 0 = Just (n * (fromJust (fat (n-1))))
```