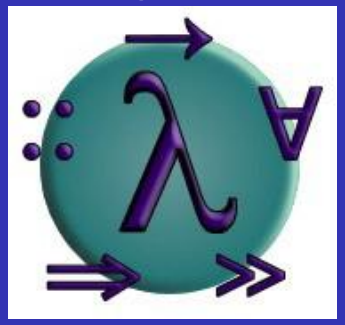
Programação Funcional



Capítulo 11

Tipos Algébricos

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2012.1

1 Tipos Algébricos

Tópicos

Novos tipos de dados

- **Tipos básicos**:
  - `Bool`
  - `Char`
  - `Int`
  - `Integer`
  - `Float`
  - `Double`
- **Tipos Compostos**:
  - tuplas: $(t_1, t_2, \ldots, t_n)$
  - listas: $[t]$
  - funções: t_1 `->` t_2
- **Novos tipos**: como definir?
  - dias da semana
  - estações do ano
  - figuras geométricas
  - árvores
  - tipos cujos elementos são inteiros ou strings
  - ...

Tipos algébricos

Uma **declaração de tipo algébrico** é da forma:

$$\texttt{data}\ cx\ \texttt{=>}\ T\ u_1\ \cdots\ u_k\ \texttt{=}\ C_1\ t_{11}\ \cdots\ t_{1n_1}$$
$$\vdots$$
$$\texttt{|}\ C_m\ t_{m2}\ \cdots\ t_{mn_m}$$

onde:

- cx é um contexto
- $u_1\ \cdots\ u_k$ são variáveis de tipo
- T é o **construtor de tipo**
- $T\ u_1\ \cdots\ u_k$ é um novo tipo introduzido pela declaração `data`
- $C_1, \ldots, C_m$ são **construtores de dados**
- t_{ij} são tipos
- Construtores de tipo e construtores de dados são identificadores alfanuméricos começando com letra maiúscula, ou identificadores simbólicos.

Tipos algébricos (cont.)

- Um **construtor de dados** é utilizado para
  - construir valores do tipo definido, funcionando como uma função (eventualmente, constante) que recebe argumentos (do tipo indicado para o construtor), e constrói um valor do novo tipo de dados;
  - decompor um valor do tipo em seus componentes, através de casamento de padrão
- Construtores de dados são funções especiais, pois não tem nenhuma definição (algoritmo) associada.

Exemplo: formas geométricas

- Definição de um novo tipo para representar formas geométricas:

```
data Figura = Circulo Double
            | Retangulo Double Double
```

- O **construtor de tipo** é `Figura`.
- Os **construtores de dados** deste tipo são:

```
Circulo   :: Double -> Figura
Retangulo :: Double -> Double -> Figura
```

e com eles é possível construir todo e qualquer valor do tipo `Figura`:

```
a :: Figura
a = Circulo 2.3 {- um círculo de raio 2.3 -}
```

```
b :: Figura
b = Retangulo 2.8 3.1 {- um retângulo de
                         base 2.8 e altura 3.1 -}
```

```
lfig :: [Figura]
lfig = [Retangulo 5 3, Circulo 5.7, Retangulo 2 2]
```

Exemplo: formas geométricas (cont.)

- Expressões como `Circulo 2.3` ou `Retangulo 2.8 3.1` não podem ser reduzidas, pois já estão em sua forma mais simples.
- Os construtores são utilizados em casamento de padrões para acessar os componentes de um valor do tipo algébrico.
- Podemos definir funções envolvendo os tipos algébricos.

Exemplo: formas geométricas (cont.)

```
eRedondo :: Figura -> Bool
eRedondo (Circulo _)      = True
eRedondo (Retangulo _ _) = False
```

```
eRedondo (Circulo 3.2)      ⇝ True
eRedondo (Retangulo 2 5.1) ⇝ False
```

Exemplo: formas geométricas (cont.)

```
area :: Figura -> Double
area (Circulo r)      = pi * r^2
area (Retangulo b a) = b * a
```

```
area (Circulo 2.5)      ⇝ 19.634954084936208
area (Retangulo 2 5.1) ⇝ 10.2
```

Exemplo: formas geométricas (cont.)

```
quadrado :: Double -> Figura
quadrado lado = Retangulo lado lado
```

```
area (quadrado 2.5) ⇝ 6.25
```

Exemplo: direção de movimento

- Definição de um novo tipo para representar direções de movimento:

```
data Dir = Esquerda | Direita | Acima | Abaixo
```

- O **construtor de tipo** é `Dir`.
- Os **construtores de dados** deste tipo, todos constantes, são:

```
Esquerda :: Dir
Direita  :: Dir
Acima    :: Dir
Abaixo   :: Dir
```

- Quando os construtores de dados são **constantes**, (ou seja, não tem argumentos), dizemos que o tipo é uma **enumeração**.
- Neste exemplo os únicos valores do tipo `Dir` são `Direita`, `Esquerda`, `Acima` e `Abaixo`.
- Podemos definir funções envolvendo o tipo algébrico:

Exemplo: direção de movimento (cont.)

```haskell
type Pos = (Double,Double)
```

```haskell
move :: Dir -> Pos -> Pos
move Esquerda  (x,y) = (x-1,y  )
move Direita   (x,y) = (x+1,y  )
move Acima     (x,y) = (x  ,y+1)
move Abaixo    (x,y) = (x  ,y-1)
```

```haskell
moves :: [Dir] -> Pos -> Pos
moves []     p = p
moves (m:ms) p = moves ms (move m p)
```

```haskell
moves [Direita,Acima,Acima,Abaixo,Acima,Direita,Acima] (0,0)
⇝ (2.0,3.0)
```

Exemplo: direção de movimento (cont.)

```
flipDir :: Dir -> Dir
flipDir Direita = Esquerda
flipDir Esquerda = Direita
flipDir Acima = Abaixo
flipDir Abaixo = Acima
```

```
flipDir Direita ⇝ erro:
No instance for (Show Dir) arising from a use of ‘print’
```

Oops!

Exemplo: direção de movimento (cont.)

- A princípio Haskell não sabe como exibir valores dos novos tipos.
- O compilador pode definir automaticamente funções necessárias para exibição:

```
data Dir = Esquerda | Direita | Acima | Abaixo
           deriving (Show)
```

- A cláusula **deriving** permite declarar as classes das quais o novo tipo será instância, automaticamente.
- Logo, segundo a declaração dada, o tipo `Dir` é uma instância da classe `Show`, e a função `show` é sobrecarregada para o tipo `Dir`.

```
show Direita    ⇝ "Direita"
flipDir Direita ⇝ Esquerda
```

Exemplo: cor

- Definição de um novo tipo para representar cores:

```
data Cor = Azul | Amarelo | Verde | Vermelho
```

- O **construtor de tipo** é `Cor`.
- Os **construtores de dados** deste tipo são:

```
Azul     :: Cor
Amarelo  :: Cor
Verde    :: Cor
Vermelho :: Cor
```

Exemplo: cor (cont.)

- Podemos agora definir funções envolvendo cores:

```
fria :: Cor -> Bool
fria Azul  = True
fria Verde = True
fria _     = False
```

```
fria Amarelo ⇝ False
```

```
quente :: Cor -> Bool
quente Amarelo  = True
quente Vermelho = True
quente _        = False
```

```
quente Amarelo ⇝ True
```

Exemplo: coordenadas cartesianas

- Definição de um novo tipo para representar coordenadas cartesianas:

```
data Coord = Coord Double Double
```

- O **construtor de tipo** é `Coord`.
- O **construtor de dados** deste tipo é:

```
Coord :: Double -> Double -> Coord
```

- Podemos agora definir funções envolvendo coordenadas:

```
somaVet :: Coord -> Coord -> Coord
somaVet (Coord x1 y1) (Coord x2 y2) = Coord (x1+x2) (y1+y2)
```

Exemplo: horário

- Definição de um novo tipo para representar horários:

```
data Horario = AM Int Int Int | PM Int Int Int
```

- Os **construtores do tipo** `Horario` são:

```
AM :: Int -> Int -> Int -> Horario
PM :: Int -> Int -> Int -> Horario
```

  e podem ser vistos como uma etiqueta (*tag*) que indica de que forma os argumentos a que são aplicados devem ser entendidos.

- Os valores `AM 5 10 30`, `PM 5 10 30` e `(5,10,30)` não contém a mesma informação. Os construtores `AM` e `PM` tem um papel essencial na interpretação que fazemos destes termos.

- Podemos agora definir funções envolvendo horários:

```
totalSegundos :: Horario -> Int
totalSegundos (AM h m s) = (h*60 + m)*60 + s
totalSegundos (PM h m s) = ((h+12)*60 + m)*60 + s
```

Exemplo: booleanos

- O tipo `Bool` da biblioteca padrão é um tipo algébrico:

```
data Bool = True | False
```

- O **construtor de tipo** é `Bool`.
- Os **construtores de dados** deste tipo são:

```
True  :: Bool
False :: Bool
```

Exemplo: booleanos (cont.)

- Exemplos de uso do tipo:

```
infixr 3 &&
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
infixr 3 ||
(||) :: Bool -> Bool -> Bool
False || False = False
_     || _     = True
```

```
not :: Bool -> Bool
not True  = False
not False = True
```

Exemplo: números naturais

- As definições de tipos algébricos podem ser **recursivas**.
- O tipo `Nat` representa números naturais:

```
data Nat = Zero | Suc Nat
         deriving (Show)
```

- Os **construtores de dados** do tipo `Nat` são:

```
Zero :: Nat
Suc  :: Nat -> Nat
```

  isto é,

  - `Zero` é um valor to tipo `Nat`, e
  - se *n* é um valor do tipo `Nat`, então `Suc` *n* também é um valor do tipo `Nat`

- Exemplos de valores do tipo `Nat`:

```
Zero
Suc Zero
Suc (Suc Zero)
Suc (Suc (Suc Zero))
Suc (Suc (Suc (Suc Zero)))
```

- Exemplos de operações:

Exemplo: números naturais (cont.)

```
nat2int :: Nat -> Int
nat2int Zero    = 0
nat2int (Suc n) = 1 + nat2int n
```

Exemplo: números naturais (cont.)

```
int2nat 0 = Zero
int2nat n | n > 0 = Suc (int2nat (n - 1))
```

Exemplo: números naturais (cont.)

```
somaNat :: Nat -> Nat -> Nat
somaNat m Zero     = m
somaNat m (Suc n) = Suc (somaNat m n)
```

Exemplo: números naturais (cont.)

```
subNat :: Nat -> Nat -> Nat
subNat m Zero           = m
subNat (Suc m) (Suc n) = subNat m n
```

Exemplo: listas

- Um tipo algébrico pode ser **polimórfico**.
- O tipo `Lista a` é um tipo algébrico polimórfico:

```
data Lista a = Nil | Cons a (Lista a)
```

- Os **construtores de dados** são:
  - `Nil :: Lista a`
    um construtor constante representando a lista vazia
  - `Cons :: a -> Lista a -> Lista a`
    um construtor para listas não vazias, formadas por uma cabeça e uma cauda.
- **Exemplo**: a lista do tipo `Lista Int` formada pelos elementos 3, 7 e 1 é representada por
  `Cons 3 (Cons 7 (Cons 1 Nil))`.
- O **construtor de tipo** `Lista` está parametrizado com uma variável de tipo `a`, que poderá ser substituída por um tipo qualquer. É neste sentido que se diz que `Lista` é um construtor de tipo.
- Operações com lista:

```
comprimento :: Lista a -> Int
comprimento Nil = 0
comprimento (Cons _ xs) = 1 + comprimento xs
```

Exemplo: listas (cont.)

```
elemento :: Eq a => a -> Lista a -> Bool
elemento _ Nil          = False
elemento x (Cons y xs) = x == y || elemento x xs
```

Exemplo: listas (cont.)

- O tipo `Lista a` deste exemplo é similar ao tipo `[a]` da bilioteca padrão do Haskell:

```haskell
data [a] = [] | a : [a]
```

- Observe apenas que Haskell usa:
  - uma notação especial para o construtor de tipo: `[a]`
  - uma notação especial para o construtor de lista vazia: `[]`
  - um identificador simbólico com status de operador infixo para o construtor de lista não vazia: `(:)`

Exemplo: *maybe*

- O tipo `Maybe a` da biblioteca padrão é um tipo algébrico polimórfico:

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

- O **construtor de tipo** é `Maybe`, que espera um argumento de tipo representando o tipo do dado encapsulado por `Just`.
- Os **construtores de dados** deste tipo são:

```
Nothing :: Maybe a
Just    :: a -> Maybe a
```

- O tipo `Maybe` é muito usado para indicar sucesso ou falha de alguma operação.
- Exemplos:

Exemplo: *maybe* (cont.)

```haskell
safediv :: Double -> Double -> Maybe Double
safediv _ 0 = Nothing
safediv x y = Just (x / y)

test :: IO ()
test =
  do putStrLn "digite dois números"
     a <- readLn
     b <- readLn
     case safediv x y of
       Nothing -> do putStrLn "divisão por zero"
                     putStrLn "tente novamente"
                     test
       Just z  -> putStrLn ("resposta: " ++ show z)
```

Exercícios

Exercício 1

1. Defina um tipo algébrico para representar uma expressão booleana. Uma expressão boolena pode ser
   - uma constante booleana (verdadeiro ou falso)
   - uma variável
   - a negação de uma expressão booleana
   - a conjunção de duas expressões booleanas
   - a disjunção de duas expressões booleanas
2. Defina um tipo para representar uma *memória*, isto é, um mapeamento de identificadores a valores booleanos.
   Dica: Use listas de associações. Uma lista de associação é uma lista de pares.
3. Defina uma função que recebe uma memória e uma expressão boolena e calcula o valor da expressão booleana usando a memória. Considere que o valor de uma variável indefinida é falso.
   Dica: Utilize a função `lookup` do prelúdio para encontrar o valor associado a uma chave em uma lista de associações.

Fim