Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Tipos Algébricos

(COMP0393)

Aula 14

dcomp
departamento de computação

Tipos Algébricos (datatypes)

- Mecanismo de definição de tipos novos
- Possibilitam definir tipos enumerados, produtos cartesianos e estruturas dinâmicas

```
data Temp = Frio | Quente
data Semana = Dom | Seg | Ter | Qua | Qui | Sex | Sab
data Estacao = Primavera | Verao | Outono | Inverno

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

`Frio` e `Quente` são "os" valores do tipo `Temp` denotados com identificadores que começam com maiúscula

Produtos Cartesianos

- Uma definição `data`, além de definir um tipo novo, define um conjunto de **construtores**:
  - `Ponto` é um construtor de valores do tipo `Ponto2D`
  - Construtores podem ser usados em padrões
  - Tipos produto têm um único construtor

```
data Ponto2D = Ponto Int Int


Ponto 0 0

Ponto (-3) 5
```

Valores de `Ponto2D`

Produtos Cartesianos

- Construtores podem ser usados como funções

```
data Ponto2D = Ponto Int Int

deslocaX:: Ponto2D -> Int -> Ponto2D
deslocaX (Ponto x y)  delta = Ponto (x + delta) y
```

Produtos Cartesianos

- Mais exemplos

```
type Nome = String
type Idade = Int
data Individuo = Pessoa Nome Idade

Pessoa "Maria" 25
Pessoa "Pedro" 44

idade :: Individuo -> Idade
idade (Pessoa _ y) = y

nome :: Individuo -> Nome
nome (Pessoa n _) = n
```

Valores de `Individuo`

Funções usando o tipo `Individuo`

Tipos Soma (Altenativas)

- O tipo algébrico pode ser mais geral, pois pode prover alternativas diferentes usando diversos construtores

```
data Formato = Circulo Float
             |Retangulo Float Float

Circulo 25 0.5
Retangulo 2.5 5.0

redondo :: Formato -> Bool
redondo (Circulo _ ) = True
redondo _ = False

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

Valores de `Formato`

Funções usando o tipo `Formato`

Formato Geral

```
data NomeTipo = Cons1 t11   t12   ...   t1k1
              |Cons2 t21   t22   ...   t2k2
                 .
                 .
                 .
              |Consm tm1   tm2  ...   tmkn
```

Derivação de Instâncias de Classes de Tipos

- É possível derivar automaticamente instâncias das classes de tipos pré-definidas em Haskell para o novo tipo criado

```
data Estacao = Primavera | Verao | Outono | Inverno
               deriving (Eq, Ord, Enum, Show, Read)

Primavera == Verao
show Primavera
[Primavera .. Outono]
Primavera < Verao
```

Expressões válidas usando `Estação`

Derivação de Instâncias de Classes de Tipos

- (==)
  - Cada construtor constrói valores diferentes
- (<=)
  - Construtores listados primeiro constroem valores menores
  - Comparação lexicográfica da esquerda para a direita
- `Enum`
  - Só se todos os construtores são “nulários”

Derivação de Instâncias de Classes de Tipos

- `Enum` **não pode ser derivada em:**

```
data Formato = Circulo Float
             |Retangulo Float Float
             deriving (Eq, Show, Read, Ord)
```

```
Circulo x <= Retangulo y z, para todo x, y, z
Circulo x <= Circulo y sse x <=y
Retangulo x y <= Retangulo v w sse
                          x < v || x == v && y <=w
```

Condições de comparação de valores no tipo `Formato`

Exercícios Recomendados

- Retome o exemplo da biblioteca (Aula 4, slide 36) para usar um tipo algébrico ao invés do par `(Pessoa, Livro)` . Refaça todas as questões solicitadas para este exemplo.
- Suponha agora que além de livros você pode emprestar CDs e revistas. Livros e CDs possuem autor e título. Revistas possuem o nome da revista apenas. Cada empréstimo varia de acordo com a categoria: um mês para livros, uma semana para CDs e três dias para revistas.
  - Quais são os tipos do seu novo sistema?
  - Elabore funções para:
    - Encontrar todos os itens emprestados a uma pessoa;
    - Encontrar os livros, CDs ou as revistas emprestadas a uma pessoa;
    - Encontrar todos os itens que devem ser retornados até uma data específica;
    - Encontrar todas as pessoas que possuem empréstimos a serem devolvidos até uma determinada data;
    - Atualizar o banco de dados com novos empréstimos. Você pode supor a existência de uma constante hoje que guarda a data do dia de hoje no formato que você definir.

Tipos Algébricos Recursivos

- Tipos algébricos podem também ser descritos em termos deles mesmos, recursivamente. Por exemplo, uma expressão inteira pode ser um literal ou a combinação de literais usando os operadores de adição e subtração.

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

```
Lit 2                                          2
Add (Lit 2) ( Lit 3)                           2+3
Sub (Lit 3) (Lit 1)                            3-1
Add (Sub (Lit 3) (Lit 1)) (Lit 3)             (3-1)+3
```

Exemplos de valores do tipo `Expr`

Tipos Algébricos Recursivos

- Operações possíveis sobre `Expr`:
  - Avaliar uma expressão (`avalia`);
  - Transformar numa string para ser lida (`show`)
  - Contar quantos operadores existem na expressão (`contOp`)
  - ...

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

```
avalia:: Exp -> Int
avalia (Lit n) = n
avalia (Add e1 e2) = (avalia e1) + (avalia e2)
avalia (Sub e1 e2) = (avalia e1) - (avalia e2)
```

Caso base de `avalia`

Casos recursivos de `avalia`

Tipos Algébricos Recursivos

- Operações possíveis sobre `Expr`:
  - Avaliar uma expressão (`avalia`);
  - Transformar numa string para ser lida (`show`)
  - Contar quantos operadores existem na expressão (`contOp`).
  - ...

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

```
show:: Exp -> String
show (Lit n) = show n
show (Add e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
show (Sub e1 e2) = "(" ++ show e1 ++ "-" ++ show e2 ++ ")"
```

Caso base de `show`

Casos recursivos de `show`

Tipos Algébricos Recursivos

- O que aconteceria se tivéssemos derivado da classe Show, sem redefinição?

```
data Expr = Lit Integer
          |Add Expr Expr
          |Sub Expr Expr
          deriving Show
```

```
show (Sub (Add (Lit 4) (Lit 5)) (Lit 6)) ↝
          "Sub (Add (Lit 4) (Lit 5)) (Lit 6)"
```

Tipos Algébricos Recursivos

- Uma outra possibilidade para não redefinir a função `show` para `Expr` seria criar mais uma instância de `Show` para o tipo `Expr`.

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

```
instance Show Exp where
    show (Lit n) = show n
    show (Add e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
    show (Sub e1 e2) = "(" ++ show e1 ++ "-" ++ show e2 ++ ")"
```

```
show (Sub (Add (Lit 4) (Lit 5)) (Lit 6)) ↝ "((4+5)-6)"
```

Tipos Algébricos Recursivos

- É possível definir construtores de tipos com notação infixa.

```
data Expr = Lit Integer
          |Expr :+: Expr
          |Expr :-: Expr
```

`:+:` e `:+:` são construtores que usam o mesmo símbolo dos operadores de adição e subtração, mas construtores necessitam que estes símbolos estejam entre : :

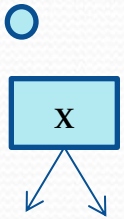
```
Lit 2
(Lit 3 :+: Lit 4) :-: Lit 8
```

Possíveis valores

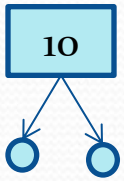
Tipos Algébricos Recursivos

- Tipos recursivos são usados para modelar **árvores**. Árvores são estruturas para organizar dados muito utilizadas em computação.
- Ex: Árvores Binárias de Inteiros

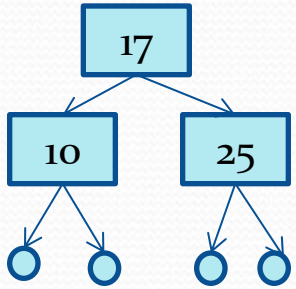
```
data Arv = NoNulo
         |No Integer Arv Arv
```




```
No 10 NoNulo NoNulo
```




```
No 17 (No 10 NoNulo NoNulo) (No 25 NoNulo NoNulo)
```



Tipos Algébricos Recursivos

- Exemplos de funções sobre árvores binárias de inteiros:
  - Somar os elementos da árvore
  - Contar quantas vezes um elemento ocorre na árvore

```
data Arv = NoNulo
         |No Integer Arv Arv
```

```
somaArv :: Arv -> Integer
somarArv NoNulo = 0                                  --caso base
somaArv (No x t1 t2) = x + somaArv t1 + somaArv t2 --caso geral
```

```
ocorreArv :: Arv -> Integer -> Integer
ocorreArv NoNulo _ = 0                               --caso base
ocorreArv (No x t1 t2) y                             --caso geral
   | x == y    = 1 +  ocorreArv t1 + ocorreArv t2
   | otherwise = ocorreArv t1 + ocorreArv t2
```

Tipos Algébricos Recursivos

- Exemplos de funções sobre árvores binárias de inteiros:
  - Somar os elementos da árvore
  - Contar quantas vezes um elemento ocorre na árvore

```
data Arv = NoNulo
         |No Integer Arv Arv
```

```
somaArv :: Arv -> Integer
somarArv NoNulo = 0                                  --caso base
somaArv (No x t1 t2) = x + somaArv t1 + somaArv t2 --caso geral
```

```
ocorreArv :: Arv -> Integer -> Integer
ocorreArv NoNulo _ = 0                               --caso base
ocorreArv (No x t1 t2) y                             --caso geral
   | x == y    = 1 +  ocorreArv t1 + ocorreArv t2
   | otherwise = ocorreArv t1 + ocorreArv t2
```

Tipos Algébricos Mutuamente Recursivos

- Tipos algébricos também podem ser mutuamente recursivos

```
data Pessoa = Adulto Nome Endereco Biog
            | Crianca Nome

data Biog = Pai String [Pessoa]
          | NaoPai String
```

Adultos pode ter biografias do tipo `Biog`.

Biografia de pais podem ter a informação dos seus filhos que são do tipo `Pessoa`!

Exercícios Recomendados

- Elabore uma função `contOp` para contar quantos operadores existem numa expressão.
- Estenda o tipo `Expr` para incluir a operação de multiplicação e refaça as três funções feitas para `Expr: avalia, show e contOp`.
- Suponha agora que você defina

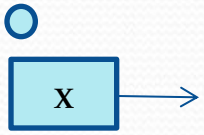
```
data Ops = Add | Sub | Mult
data Expr = Lit Integer | Op Ops Expr Expr
```

  Redefina `avalia, show e contOp` para estes novos tipos.
- Defina uma função que retorna a subárvore esquerda de uma árvore binária de inteiros.
- Defina uma função que checa se um inteiro ocorre alguma vez numa árvore binária de inteiros.
- Defina uma função que retorne o maior e o menor valor de uma árvore binária de inteiros.
- Defina uma função que troca as subárvores esquerdas pelas subárvores direitas em todos os nós não nulos de uma árvore binária de inteiros.

Listas com Tipos Recursivos

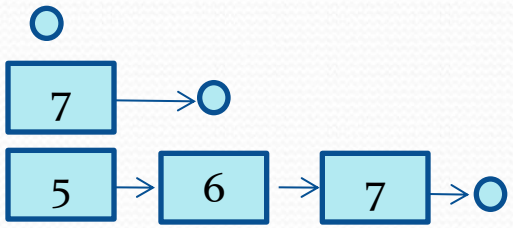
- É possível definir listas encadeadas com tipos recursivos. Listas encadeadas são estruturas dinâmicas muito usadas em computação. Ex: listas encadeadas de inteiros.

```
data ListInt = Vazia
             | Cons Int ListInt
```



Exemplos de Valores.

```
Vazia
Cons 7 Vazia
Cons 5 (Cons 6 (Cons 7 Vazia))
```



Exemplos de Funções.

```
tamLista :: ListInt -> Int
tamLista Vazia = 0
tamLista (Cons x xs) = 1 + tamLista xs

concatenaLista :: ListInt -> ListInt -> ListInt
concatenaLista Vazia xs = xs
concatenaLista (Cons y ys) xs = Cons y (concatenaLista ys xs)
```

Tipos Algébricos Polimórficos

- Na definição de tipos algébricos podemos ter variáveis de tipos e portanto, tipos algébricos polimórficos.

```
data Dupla a = Par a a
```

```
Par 0 1 :: Dupla Int
Par 'a' 'b' :: Dupla Char
Par [] [1,5,4] :: Dupla [Int]

Par 0 'a' :: ??
```

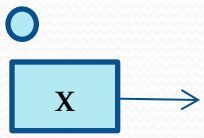
Exemplos de Valores.

ERRO!! Argumentos devem ser de mesmo tipo!!

Estruturas Dinâmicas Polimórficas

- É possível definir listas encadeadas polimórficas.

```
data List a = Vazia
            | Cons a (List a)
```




```
data [a] = []
         | a : [a]
         deriving (Eq, Ord, Show, Read)
```

Listas pré-definidas de Haskell poderiam ser definidas assim!

- Da mesma forma, podemos definir árvores binárias polimórficas.

```
data Arv a = NoNulo
           |No a (Arv a) (Arv a)
           deriving (Eq, Ord, Show, Read)
```

Observe que agora as listas e as árvores podem ser de qualquer tipo!

Tipos Algébricos Polimórficos

- É possível definir mais de um parâmetro em tipos algébricos polimórficos. Ex: Tipo união `Either`

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Show, Read)
```

```
Left 0 :: Either Int b
Right 'a' :: Either a Char
```

Valores possíveis

```
isLeft :: Either a b -> Bool
isLeft (Left _) = True
isLeft _        = False

duasFunc :: (a->c) -> (b->c) -> Either a b ->  c
duasFunc f g (Left x)  = f x
duasFunc f g (Right y) = g y
```

Testa se o elemento está definido como na primeira parte do tipo

Função de alta ordem que aplica a função `f : a ->c` se o tipo de entrada for `a` ou a função `g : b->c` se o tipo de entrada for `b`

Exercícios Recomendados

- Elabore uma função `colapsaArv` para transformar os elementos de uma árvore binária polimórfica em uma lista, de forma que para cada nó, os elementos da subárvore esquerda precedam o elemento do nó e os da subárvore direita sucedam o elemento do nó, na lista.
- Defina uma função `mapTree` que tem a mesma funcionalidade de `map` para listas, só que agora aplicada a uma árvore polimórfica.
- Defina uma função para trocar a ordem do tipo `Either`

```
troca :: Either a b -> Either b a
```

Qual o efeito de `troca.troca` ?