Definindo tipos de dados

Programação Funcional

Prof. Rodrigo Ribeiro

Objetivos

- Definir novos tipos de dados em Haskell.
  - Simples, polimórficos e recursivos.
- Definir funções por casamento de padrão sobre tipos de dados.

Setup

```haskell
{-# LANGUAGE StandaloneDeriving #-}
module Aula05 where

main :: IO ()
main = return ()

deriving instance Show Point
deriving instance Show Shape
deriving instance Show IntList
deriving instance Show IntTree
deriving instance Show Client
```

Anteriormente...

- Usamos somente tipos pré-definidos da linguagem Haskell.
- Tipos básicos: Int, Char, Bool ...
- Tipos compostos: Listas, tuplas.
- Nesta aula, veremos como definir novos tipos de dados.

Enumerações

- O tipo de dados mais simples é o de enumeração.
- Consiste de uma sequência finita de valores.
- Exemplo canônico: `Bool`

Enumerações

- Tipo de dados para direções.

```
data Direction = North | South
               | East  | West
```

- Declaração inicia com a palavra reservada **data** seguida de um nome iniciado com letra maiúscula.
- Seguido de um ou mais **construtores de dados**.
  - Nomes de construtores também iniciam com letras maiúsculas.

Enumerações

- Cada construtor define um valor do tipo de dados.

```
:t North
North :: Direction
```

- Podemos usar o tipo `Direction` para formar listas e tuplas.

```
:t [North, South]
[North, South] :: [Direction]

:t (South, True)
(South, True) :: (Direction, Bool)
```

Casamento de padrão

- A definição de funções sobre novos tipos é feita por casamento de padrão sobre seus construtores.
- Vamos definir uma função para converter valores de tipo `Direction` em `Strings`.

Casamento de padrão

- 1. Definir o tipo

```
directionName :: Direction -> String
```

Casamento de padrão

- 2. Enumerar os casos.
  - Definimos uma equação para cada construtor.

```
directionName :: Direction -> String
directionName North = _
directionName South = _
directionName East  = _
directionName West  = _
```

Casamento de padrão

- 3. Definindo os casos.

```
directionName :: Direction -> String
directionName North = "N"
directionName South = "S"
directionName East  = "E"
directionName West  = "W"
```

Tipos de dados básicos

- Tipos de dados básicos podem ser vistos como enumerações.

```
data Bool = True | False
data Int = ... | -2 | -1 | 0 | 1 | 2 ...
data Char = ... | 'A' | 'B' ...
```

- A definição do tipo `Bool` é exatamente como acima.
- Tipos `Int` e `Char` são tratados pelo compilador de maneira especial.

Pontos

- Tipos de dados podem armazenar informações em construtores.

```
data Point = Point Float Float
```

- O nome do construtor é seguido por uma lista de seus parâmetros.
- O nome de um construtor pode ser igual ao seu tipo.

Criando Pontos

- Para criar um valor do tipo `Point` usamos o construtor `Point` seguido dos valores para cada um de seus argumentos.

```
:t Point 1.0 3.0
```

```
(Point 1.0 3.0) :: Point
```

- Para casamento de padrão, usamos o nome do construtor seguido de padrões para seus argumentos.

```
norm :: Point -> Float
norm (Point x y) = sqrt (x * x + y * y)
```

Construtores são funções

- Cada construtor de um tipo de dados é uma função que constrói valores do tipo em questão.

```
:t North
North :: Direction -- sem argumentos

:t Point
Point :: Float -> Float -> Point -- 2 argumentos
```

Formas geométricas

- Tipos de dados podem possuir zero ou mais construtores.
- Cada construtor pode ter zero ou mais argumentos.

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

- Chamamos esses tipos de **Tipos de Dados Algébricos**.

Perímetro de formas

- Vamos definir uma função para calcular o perímetro de formas geométricas definidas pelo tipo `Shape`.
- Fórmulas para calcular o perímetro.

$$
\begin{aligned}
P_{rect} &= 2w + 2h \\
P_{circle} &= 2\pi r \\
P_{triangle} &= dist(a, b) + dist(b, c) + dist(c, a)
\end{aligned}
$$

Perímetro de formas

- Cada caso define um padrão para cada construtor do tipo Shape.

```
perimeter :: Shape -> Float
perimeter (Rectangle _ w h) = 2 * w + 2 * h
perimeter (Circle _ r) = 2 * pi * r
perimeter (Triangle a b c) = dist (a,b) + dist (b,c) + dist

dist :: (Point, Point) -> Float
dist ((Point x1 y1), (Point x2 y2))
  = sqrt (x11 * x11 + y11 * y11)
   where
     x11 = x1 - x2
     y11 = y1 - y2
```

Tipos recursivos

- Usando o mecanismo de definição de tipos, podemos definir tipos para listas e árvores de números inteiros.

```
data IntList = INil | ICons Int IntList

data IntTree = ILeaf | INode Int IntTree IntTree
```

Busca sequencial

1. Definindo o tipo

```
elemIntList :: Int -> IntList -> Bool
```

Busca sequencial

2. Enumerando os casos

```
elemIntList :: Int -> IntList -> Bool
elemIntList _ INil = _
elemIntList x (ICons y ys) = _
```

Busca sequencial

3. Definindo caso base

```
elemIntList :: Int -> IntList -> Bool
elemIntList _ INil = False
elemIntList x (ICons y ys) = _
```

Busca sequencial

4. Definindo caso recursivo

```
elemIntList :: Int -> IntList -> Bool
elemIntList _ INil = False
elemIntList x (ICons y ys) = x == y || elemIntList x ys
```

Busca em árvores

1. Definindo o tipo.

```
elemIntTree :: Int -> IntTree -> Bool
```

Busca em árvores

2. Enumerando os casos.

```
elemIntTree :: Int -> IntTree -> Bool
elemIntTree _ ILeaf = _
elemIntTree x (INode y l r) = _
```

Busca em árvores

3. Definindo o caso base

```
elemIntTree :: Int -> IntTree -> Bool
elemIntTree _ ILeaf = False
elemIntTree x (INode y l r) = _
```

Busca em árvores

4. Definindo o caso recursivo

```
elemIntTree :: Int -> IntTree -> Bool
elemIntTree _ ILeaf = False
elemIntTree x (INode y l r)
  | x < y = elemIntTree x l
  | x > y = elemIntTree x r
  | otherwise = True
```

Altura em árvores

- Altura de uma folha (`ILeaf`) é igual a zero.
- Altura de um nó é igual a 1 mais o maior valor dentre as alturas das subárvores.

Altura em árvores

1. Definindo o tipo.

```
itreeHeight :: IntTree -> Int
```

Altura em árvores

2. Enumerando os casos.

```
itreeHeight :: IntTree -> Int
itreeHeight ILeaf = _
itreeHeight (INode _ l r) = _
```

Altura em árvores

3. Definindo o caso base.

```
itreeHeight :: IntTree -> Int
itreeHeight ILeaf = 0
itreeHeight (INode _ l r) = _
```

Altura em árvores

4. Definindo o caso recursivo.

```
itreeHeight :: IntTree -> Int
itreeHeight ILeaf = 0
itreeHeight (INode _ l r)
  = 1 + max (itreeHeight l) (itreeHeight r)
```

Convertendo em listas

- Primeiramente, vamos precisar de uma função para concatenar listas (`IntList`).

1. Definindo o tipo.

```
concatIntList :: IntList -> IntList -> IntList
```

Convertendo em listas

2. Enumerando os casos.

```
concatIntList :: IntList -> IntList -> IntList
concatIntList INil ys = _
concatIntList (ICons x xs) ys = _
```

Convertendo em listas

3. Definindo o caso base.

```
concatIntList :: IntList -> IntList -> IntList
concatIntList INil ys = ys
concatIntList (ICons x xs) ys = _
```

Convertendo em listas

4. Definindo o caso recursivo.

```
concatIntList :: IntList -> IntList -> IntList
concatIntList INil ys = ys
concatIntList (ICons x xs) ys
  = ICons x (concatIntList xs ys)
```

Convertendo em listas

- Usando a função de concatenação, podemos definir a conversão de árvores em listas.

1. Definindo o tipo.

```
intTreeToList :: IntTree -> IntList
```

Convertendo em listas

2. Enumerando os casos.

```
intTreeToList :: IntTree -> IntList
intTreeToList ILeaf = _
intTreeToList (INode x l r) = _
```

Convertendo em listas

3. Definindo o caso base.

```
intTreeToList :: IntTree -> IntList
intTreeToList ILeaf = INil
intTreeToList (INode x l r) = _
```

Convertendo em listas

4. Definindo o caso recursivo.

```
intTreeToList :: IntTree -> IntList
intTreeToList ILeaf = INil
intTreeToList (INode x l r)
  = ICons x (concatIntList l' r')
    where
      l' = intTreeToList l
      r' = intTreeToList r
```

Tipos polimórficos

- Vimos que listas e tuplas são tipos polimórficos previamente definidos em Haskell.
- Como definir um novo tipo polimórfico?

Tipos polimórficos

- Tipo `Maybe a` representa um possível valor de tipo `a`.

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

- Na declaração de um tipo polimórfico, o nome do tipo é seguido por uma ou mais variáveis de tipo.
  - Variáveis de tipo são identificadores formados por letras minúsculas.
  - Construtores podem referenciar variáveis em seus argumentos.

Tipo Maybe

```
:t Just True
(Just True) :: Maybe Bool

:t Nothing
Nothing :: Maybe a
```

- Observe que no primeiro exemplo, o valor `True` permitiu o GHC determinar que o parâmetro `a` de `Maybe a` deve ser `a = Bool`.
- No segundo exemplo, o construtor `Nothing` manteve seu tipo polimórfico por não haver informação para determinar uma possível instanciação da variável `a`.

Exemplo

- Usando o tipo `Maybe` podemos implementar uma versão segura da função `head`.

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : _) = Just x
```

- Observe que ao usarmos `safeHead` sobre uma lista vazia, não incorremos em erro de tempo de execução, como em `head`.

```
head []
*** Exception: Prelude.head: empty list
```

Exemplo

- Determinando a posição de um elemento em uma lista.
  - Necessário a restrição `Eq a` para permitir o uso da igualdade em tipos polimórficos.
  - Retornamos `Nothing` quando o elemento não pertence a lista.

1. Definindo o tipo

```
position :: Eq a => a -> [a] -> Maybe Int
```

Exemplo

- Usaremos uma função auxiliar para permitir o uso de um acumulador.

2. Enumerando os casos

```
position :: Eq a => a -> [a] -> Maybe Int
position x xs = pos x xs 0
   where
     pos _ [] _          = _
     pos x (y : ys) ac = _
```

Exemplo

3. Definindo o caso base

```
position :: Eq a => a -> [a] -> Maybe Int
position x xs = pos x xs 0
   where
     pos _ [] _         = Nothing
     pos x (y : ys) ac = _
```

Exemplo

4. Definindo o caso recursivo

```
position :: Eq a => a -> [a] -> Maybe Int
position x xs = pos x xs 0
   where
     pos _ [] _          = Nothing
     pos x (y : ys) ac
       | x == y          = Just ac
       | otherwise       = pos x ys (ac + 1)
```

Problema

- Representar informação sobre um cliente.
- Informação composta por:
  - Nome
  - Sobrenome
  - Aceita receber informações sobre promoções?

Modelagem

- Informações representadas pelos tipos:
  - Nome :: String
  - Sobrenome :: String
  - Ofertas :: Bool
- Como definir o tipo cliente?

Modelagem

- Podemos utilizar uma tupla!

```
ex1 :: (String, String, Bool)
ex1 = ("José", "Silva", False)
```

Problemas

- Uso do tipo String é pouco informativo!
  - Como diferenciar de nome e sobrenome?
  - Usar a posição na tripla é algo propenso a erros.

Sinônimos

- Podemos melhorar a legibilidade do código usando sinônimos de tipos.

```
type Name = String
type Surname = String
type SendOffer = Bool
```

Modelagem

- Representando o exemplo anterior.

```
type Client = (Name, Surname, SendOffer)

ex2 :: Client
ex2 = ("José", "Silva", False)
```

Modelagem

- Agora, o código é mais informativo!
- Porém, ainda dependente da posição de componentes no par.

Modelagem

- O uso de sinônimos não define um novo tipo de dados!
- Logo, os tipos Client e (String, String, Bool) são idênticos!
- Podemos melhorar isso?

Tipos Algébricos

- Podemos definir um novo tipo de dados em Haskell.

```
data Client
 = Customer Name Surname SendOffer

ex3 :: Client
ex3 = Customer "José" "Silva" False
```

Problema

- Como exibir uma mensagem de boas vindas para um cliente assim que ele acessa a aplicação?
- Tarefa: definir uma função, que a partir de um cliente, produza uma mensagem de boas vindas.

Problema

- A implementação de uma função de uma função para gerar a a mensagem pode ser feita por casamento de padrão.

```
greet :: Client -> String
greet (Customer n _ _) = "Welcome, " ++ n ++ "!"
```

Registros

- Haskell permite a definição de registros.

```
data Client
  = Customer {
      name    :: Name
    , surname :: Surname
    , offers  :: SendOffer
    }
```

Registros

- Cada campo do registro define uma função de projeção

```
name :: Client -> Name
surname :: Client -> Surname
offers :: Client -> SendOffer
```

- Exemplo.

```
ex3 :: Client
ex3 = Customer "José" "Silva" False

name ex3 == "José"
```

Exercício

- Desenvolva uma função para calcular a área de formas geométricas descritas pelo tipo `Shape`.

Exercício

- Desenvolva funções para calcular o número de elementos e o número de folhas de uma árvore de tipo `IntTree`.

Exercício

- Desenvolva uma função para converter uma lista (tipo `IntList`) em uma árvore binária (tipo `IntTree`).

Exercício

- Desenvolva uma função que a partir de uma lista de clientes, retorne uma lista contendo todos os clientes que desejam receber mensagens de ofertas.