Criandos Tipos e Classes

João Marcelo Uchôa de Alencar

Universidade Federal do Ceará - Quixadá

25 de Maio de 2017

Introdução a Tipos de Dados Algébricos

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

- **data** significa que vamos definir um novo tipo.
- O nome antes do operador = informa o nome do tipo.
- Após o operador =, temos os valores possíveis, separados pelo operador |.

```
data Int = -2147483648 | -2147483647 | ... | -1
           | 0 | 1 | 2 | ... | 2147483647
```

Por que usar **data**?

- Podemos representar um Círculo através de uma tupla:
  - Primeiro elemento seria a coordenada x do centro.
  - Segundo elemento seria a coordenada y do centro.
  - O último elemento seria o raio.
  - $(1.0, 3.5, 5.0)$
- Mas como saber, lá no meio de um programa bem maior, que a tupla representa de fato um Círculo? Ela poderia representar um ponto no espaço 3D, três notas de alunos, etc...

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

O que são os valores?

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

- O que passamos como valores, na verdade são funções.
- Ao contrário das funções que vimos até agora, para ser um valor de um tipo, o nome da função começa com letra maiúscula.

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) *
                                  (abs $ y2 - y1)
```

Poderia fazer *surface :: Circle -> Float*?

Derivando Classes

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

O operador $ altera a precedência na invocação de funções.
O que acontece se você digitar apenas *Circle 10 20 5* ?

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving (Show)
```

E agora, digitando *Circle 10 20 5*, o que acontece?

Criando Várias Intâncias de um Valor

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
           deriving(Show)
create_circles a b [] = []
create_circles a b (c:x) = (Circle a b c) :
                           create_circles a b x
```

Criando Tipos Mais Detalhados

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float
           | Rectangle Point Point
           deriving (Show)

surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) =
      (abs $ x2 - x1) * (abs $ y2 - y1)
```

No caso do retângulo, se agora eu chamasse *surface* como anteriormente, daria certo?

Uma Função para Mover a *Shape*

```haskell
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b =
    Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b =
    Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))

-- Funções constantes para criar formas iniciais
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height =
    Rectangle (Point 0 0) (Point width height)
```

Sintaxe de Registro

Como definir um registro (*struct*) com vários campos?

```
-- Registro Person
---- Primeiro Nome
---- Segundo Nome
---- Idade
---- Altura
---- Telefone
---- Sabor de sorvete favorito
data Person = Person String String Int Float String String
              deriving (Show)
```

Recuperando Informações de Registros

```haskell
firstName :: Person -> String
firstName (Person firstname _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Sintaxe Simplificada

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

As funções de acesso são criadas automaticamente.

Criação de Instâncias na Sintaxe Simplificada

```
-- Forma tradicional
data Car = Car String String Int deriving (Show)
ghci> Car "Volks" "Fusca" 1967
Car "Volks" "Fusca" 1967

-- Forma simplificada
data Car = Car { company :: String,
                 model :: String,
                 year :: Int
               } deriving (Show)
ghci> Car {company="Volks", model="Fusca", year=1967}
Car {company = "Volks", model = "Fusca", year = 1967}
```

Tipos Paramétricos

- Os construtores de valores podem receber parâmetros e retornar uma função.
- Por exemplo: Car, Circle, Rectangle, etc.
- Um caso mais abstrato ocorre quando um tipo é informado do lugar de um valor.
- Temos assim um **construtor de tipo**.

Just Maybe Nothing

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

- *a* é o parâmetro de tipo.
- *Maybe* é um construtor de tipo.
- *Nothing* é sempre *Nothing*, pois não há parâmetros, sejam eles valores ou tipos.
- Já *Maybe* pode assumir *Maybe Int*, *Maybe Char*, *Maybe String*.
- *Just 'a'* é do tipo *Maybe Char*.
- Nenhum valor pode ter o tipo apenas *Maybe*.
- O construtor [] também é um tipo paramétrico. Afinal, temos listas de inteiros, caracteres, etc.

Usando *Maybe* para Funções Parciais

```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
        | x > 0      = Just (log x)
        | otherwise = Nothing
```

Através do *Maybe*, podemos retornar funções que "deram"errado e não retornam valores válidos.

Criando Tipos Paramétricos

```haskell
-- Definindo os tipos
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
   "Esse " ++ c ++ " " ++ m ++ " foi feito em " ++ show y

-- Parametrizando
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) =
   "Esse " ++ c ++ " " ++ m ++ " foi feito em " ++ show y
```

Um Vetor para Qualquer Tipo de Número

```haskell
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

Instâncias Derivadas

- Um tipo pode ser instância de uma classe se ele suportar o comportamento (funções) da classe.
- **Int** é instância de **Eq** porque inteiros podem ser comparados.
- Em outras palavras, podemos usar $==$ e $/=$ em valores do tipo **Int**.
- Haskell vai automatizar o comportamento para tipos derivados das classes **Eq, Ord, Enum, Bounded, Show, Read**.

Definição Automática de Comportamento

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

- Neste caso, existe um construtor de valor: *Person*.
- Para definir o comportamento de ==:
  - Primeiro, Haskell verifica se o construtor de valores batem.
  - Depois, Haskell verifica se cada parâmetro passado para o construtor é o mesmo.
  - Para funcionar, os parâmetros também deve pertencem a classe **Eq**.
- No estágio atual, podemos digitar uma variável do tipo *Person* no *ghci* e ter o valor impresso?

Conversão de/em *Strings*

```
data Person = Person { firstName :: String,
                       lastName :: String,
                       age :: Int
                     } deriving (Eq, Show, Read)
```

- **Show** e **Read** são classes para coisas que podem ser convertidas para/a partir de *Strings*.
- Assim como **Eq**, Haskell faz a conversão automática para valores construtores com tipos compatíveis.
- **Show** transforma um tipo em *String*.
- **Read** transforma uma *String* em um tipo.

Exemplos de Show/Read

```
> let joao = Person{firstName = "Joao", lastName="Marcelo", age=34}
> putStr $ "João é " ++ show joao ++ "\n"
João é Person {firstName = "Joao", lastName = "Marcelo", age = 34}
> read "Person {firstName =\"Joao\", lastName = \"Marcelo\", age = 34}"
                                                              :: Person
Person {firstName = "Joao", lastName = "Marcelo", age = 34}
> read "Person {firstName =\"Joao\", lastName = \"Marcelo\", age = 34}"
                                                              == joao
True
```

Classe Ord

- A classe **Ord** contém tipos que podem ser comparados.
- Se comparar dois valores do mesmo tipo que foram definidos com construtores diferentes, o valor do construtor definido primeiro é considerado o menor.

```
data Bool = False | True deriving (Ord)
```

- Por terem sido definidos primeiro, valores *False* são "menores" que valores *True*.

```
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday
         | Sunday deriving(Ord)
```

Enum e Bounded

- **Enum** é para tipos com valores sem parâmetros que tem predecessores e sucessores.
- **Bounded** é para tipos com valor máximo e mínimo.

```
data Day = Monday | Tuesday | Wednesday |
           Thursday | Friday | Saturday | Sunday
           deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Tipos Sinônimos

```
type String = [Char]
```

- *[Char]* e *String* representam a mesma coisa.
- **Tipos Sinônimos**
- O objetivo é ajudar na legibilidade do código.
- Apesar da palavra reservada **type**, não estamos criando um novo tipo, apenas um sinônimo.

```
toUpperString :: [Char] -> [Char]
toUpperString :: String -> String
```

Exemplo de Legibilidade Aprimorada

```haskell
-- O que a função faz?
estaNaAgenda :: [Char] -> [Char] -> [([Char],[Char])] -> Bool

-- Versão mais clara
type Nome = String
type Telefone = String
type Agenda = [(Nome, Telefone)]

estaNaAgenda :: Nome -> Telefone -> Agenda -> Bool
```

Mais uma Possibilidade para Retornar Valores Diferentes

```haskell
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
type Nome = String
type Nota = Int
type Alunos = [String]
type Resultado = [(Nome, Nota)]

 -- Função que retorna a nota do aluno ou falso
pertence :: Nome -> Resultado -> Either Nota Bool
pertence n [] = Right False
pertence n (a:x) | n == fst a = Left $ snd a
                 | otherwise = pertence n x
```

Estruturas de Dados Recursivas

- Vimos que um construtor de tipos pode ter vários valores, cada um com parâmetros de tipos concretos.
- Podemos avançar um passo e definir um construtor cujo valores recebem parâmetros do tipo do próprio construtor.
- Definição recursiva de tipo.

```
data List a = Empty | Cons a (List a)
                      deriving (Show, Read, Eq, Ord)
-- ou
data List a = Empty |
              Cons { listHead :: a,
                     listTail :: List a
                     } deriving (Show, Read, Eq, Ord)
```

Árvore de Busca Binária

- Um elemento central aponta para dois elementos.
- O elemento da esquerda é menor que o central.
- O elemento da direita é maior que o central.
- Cada elemento pode ter novos sub-elementos.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
                                    deriving (Show, Read, Eq)
```

- Em C, trabalhar com árvores é manipular ponteiros.
- Em Haskell, construímos novas árvores.

Inserindo em uma Árvore

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
            | x == a = Node x left right
            | x < a  = Node a (treeInsert x left) right
            | x > a  = Node a left (treeInsert x right)
```

Buscando em uma Árvore

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
           | x == a = True
           | x < a  = treeElem x left
           | x > a  = treeElem x right
```

Typeclasses 102

- Já vimos várias classes padrão do Haskell: Eq, Show, Read, etc...
- Também já vimos como fazer nossos tipos estenderem essas classes.
- Falta ver como fazer nossas próprias classes de tipo.

Reforçando Conceitos

- Classes de tipo são como interfaces.
- Classes de tipo definem um comportamento (comparação, ordenação, enumeração, etc).
- Tipos que apresentam tal comportamento são instâncias da classe.
- Um tipo fazer parte de uma classe significa dizer que podemos usar as funções da classe no tipo.

Reforçando Conceitos

```haskell
-- class é para definir novas classes de tipo.
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)

-- instance faz tipo instância de uma classe.
data Semaforo = Vermelho | Amarelo | Verde
instance Eq Semaforo where
    Vermelho == Vermelho = True
    Verde == Verde = True
    Amarelo == Amarelo = True
    _ == _ = False
```

Reforçando Conceitos

```
instance Show Semaforo where
    show Vermelho = "Luz Vermelha"
    show Amarelo = "Luz Amarela"
    show Verde = "Luz Verde"
```

Reforçando Conceitos

```haskell
-- "herança"
class (Eq a) => Num a where
...
-- restrições sobre os tipos paramétricos
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False

instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Uma Classe de Tipos Sim-Não

```haskell
class SimNao a where
    simnao :: a -> Bool

instance SimNao Int where
    simnao 0 = False
    simnao _ = True

instance SimNao [a] where
    simnao [] = False
    simnao _ = True

instance SimNao Bool where
    simnao = id -- função identidade
```

Uma Classe de Tipos Sim-Não

```
instance SimNao (Maybe a) where
    simnao (Just _) = True
    simnao Nothing = False

instance SimNao (Tree a) where
    simnao EmptyTree = False
    simnao _ = True

instance Semaforo  where
    simnao Vermelho = False
    simnao _ = True

simnaoSe :: (SimNao y) => y -> a -> a -> a
simnaoSe valorSimNao resultadoSim resultadoNao =
       if simnao valorSimNao then resultadoSim
                             else resultadoNao
```