Tipos de dados abstratos - *Abstract Data Types* (ADT)

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Definindo novos tipos

Novos tipos de dados

UFABC

- A definição de novos tipos de dados, além dos tipos primitivos, permite manter a legibilidade do código e facilita a organização de seu programa.

Declaração de tipo

UFABC

- A forma mais simples de definir um novo tipo é criando *apelidos* para tipos existentes:

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

Declaração de tipo

UFABC

- Todo nome de tipo deve começar com uma letra maiúscula. As definições de tipo podem ser encadeadas!
- Suponha a definição de um tipo que armazena uma coordenada e queremos definir um tipo de função que transforma uma coordenada em outra:

```haskell
type Coord = (Int, Int)
type Trans = Coord -> Coord
```

Declaração de tipo

UFABC

- Porém, não podemos definir tipos recursivos:

```haskell
type Tree = (Int, [Tree])
```

- Mas temos outras formas de definir tais tipos...

Declaração de tipo

UFABC

- A declaração de tipos pode conter variáveis de tipo:

```haskell
type Pair a = (a, a)

type Assoc k v = [(k,v)]
```

Declaração de tipo

UFABC

- Com isso podemos definir funções utilizando esses tipos:

```haskell
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']

> find 2 [(1,3), (5,4), (2,3), (1,1)]
3
```

Exercício 1

UFABC

- Crie uma função `paraCima` do tipo `Trans` definido anteriormente que ande para cima dado uma coordenada (some +1 em y).

Declaração de tipo

UFABC

- Como esses tipos são apenas apelidos, eu posso fazer:

```
1  array = [(1,3), (5,4), (2,3), (1,1)] :: [(Int, Int)]
2  > find 2 array
3  3
4
5  array' = [(1,3), (5,4), (2,3), (1,1)] :: Assoc Int Int
6  > find 2 array
7  3
```

- O compilador não distingue um do outro.

Tipos de Dados Algébricos

Tipos de Dados Algébricos

UFABC

- Tipos completamente novos.
- Pode conter tipos primitivos.
- Permite expressividade.
- Permite checagem em tempo de compilação

Tipos Soma

UFABC

- Tipo soma:

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

- `data`: declara que é um novo tipo
- `Bool`: nome do tipo
- `True | False`: poder assumir ou True ou False

Exemplo

UFABC

- Vamos criar um tipo que define a direção que quero andar:

```haskell
data Dir = Norte | Sul | Leste | Oeste
```

Exemplo

UFABC

- Com isso podemos criar a função `para`:

```haskell
data Dir = Norte | Sul | Leste | Oeste

para :: Dir -> Trans
para Norte (x,y) = (x,y+1)
para Sul   (x,y) = (x,y-1)
para Leste (x,y) = (x+1,y)
para Oeste (x,y) = (x-1,y)
```

Exemplo

UFABC

E a função `caminhar`:

```haskell
caminhar :: [Dir] -> Trans
caminhar []     coord = coord
caminhar (d:ds) coord = caminhar ds (para d coord)
```

Tipos Produto

UFABC

- Tipo produto:

```haskell
data Ponto = Ponto Double Double
```

- `data`: declara que é um novo tipo
- `Ponto`: nome do tipo
- `Ponto`: construtor (ou envelope)
- `Double Double`: tipos que ele encapsula

UFABC

- Para ser possível imprimir esse tipo:

```haskell
data Ponto = Ponto Double Double
                        deriving (Show)
```

- `deriving`: derivado de outra classe
- `Show`: tipo imprimível
- Isso faz com que o Haskell crie automaticamente uma instância da função `show` para esse tipo de dado.

UFABC

- Para usá-lo em uma função devemos sempre envelopar a variável com o construtor.

```
1  dist :: Ponto -> Ponto -> Double
2  dist (Ponto x y) (Ponto x' y') = sqrt
3                                   $ (x-x')^2 + (y-y')^2
4
5  > dist (Ponto 1 2) (Ponto 1 1)
6  1.0
```

UFABC

- Podemos misturar os tipos soma e produto:

```haskell
data Forma = Circulo Ponto Double
           | Retangulo Ponto Double Double

-- um quadrado é um retângulo com os dois lados iguais
quadrado :: Ponto -> Double
quadrado p n = Retangulo p n n
```

UFABC

- `Circulo` e `Retangulo` são funções construtoras:

```
> :t Circulo
Circulo :: Ponto -> Double -> Forma

> :t Retangulo
Retangulo :: Ponto -> Double -> Double -> Forma
```

Tipos parametrizados

UFABC

- As declarações de tipos também podem ser parametrizados, considere o tipo `Maybe`:

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

- A declaração indica que um tipo `Maybe a` pode não ser nada ou pode ser apenas o valor de um tipo `a`.

Maybe

- Esse tipo pode ser utilizado para ter um melhor controle sobre erros e exceções:

```haskell
-- talvez a divisão retorne um Int
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

UFABC

- Esses erros podem ser capturados com a expressão `case`:

```haskell
divComErro :: Int -> Int -> Int
divComErro m n = case (safeDiv m n) of
                   Nothing -> error "divisão por 0"
                   Just x  -> x
```

- A expressão case nos permite fazer pattern matching dentro do código da função com quaisquer expressões e não apenas nos seus parâmetros

Either

UFABC

- Um outro tipo interessante é o `Either` definido como:

```haskell
data Either a b = Left a | Right b
```

- Esse tipo permite que uma função retorne dois tipos diferentes, dependendo da situação.

UFABC

```haskell
-- ou retorna uma String ou um Int
safeDiv' :: Int -> Int -> Either String Int
safeDiv' _ 0 = Left "divisão por 0"
safeDiv' m n = Right (m `div` n)

> safeDiv' 2 2
1
> safeDiv' 2 0
"divisão por 0"
```

Exercício 2

UFABC

- Crie um tipo `Fuzzy` que pode ter os valores `Verdadeiro`, `Falso`, `Pertinencia Double`, que define um intermediário entre `Verdadeiro` e `Falso`.
- Crie uma função `fuzzifica` que recebe um `Double` e retorna `Falso` caso o valor seja menor ou igual a 0, `Verdadeiro` se for maior ou igual a 1 e `Pertinencia v` caso contrário.

Newtype

- Uma terceira forma de criar um novo tipo é com a função `newtype`, que permite apenas um construtor:

```haskell
newtype Nat = N Int
```

- A diferença entre `newtype` e `type` é que o primeiro define um novo tipo enquanto o segundo é um sinônimo.
- A diferença entre `newtype` e `data` é que o primeiro define um novo tipo até ser compilado, depois ele é substituído como um sinônimo. Isso ajuda a garantir a checagem de tipo em tempo de compilação.

Tipos Recursivos

UFABC

- Lembrando a aula de funções-λ, a definição de números naturais era definida por um Zero e uma sequência de aplicações de uma função `f`. Podemos replicar essa definição como:

```haskell
data Nat = Zero | Succ Nat
```

- Ou seja, ou o número é Zero ou ele é a aplicação do construtor `Succ` em outro valor de `Nat`.

UFABC

- Então os primeiros números são definidos como:

```haskell
zero = Zero
um   = Succ Zero
dois = Succ (Succ Zero)
tres = Succ (Succ (Succ Zero))
```

UFABC

- Podemos então definir uma função `nat2int` e outra `int2nat` como:

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

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Exercício 3

UFABC

Defina a função **add** sem utilizar a conversão:

```haskell
add :: Nat -> Nat -> Nat
```

Resposta - Exercício 3

UFABC

```haskell
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Árvore Binária

UFABC

- Um outro exemplo de tipo recursivo é a árvore binária, que pode ser definida como:

```haskell
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

- Ou seja, ou é um nó folha contendo um valor do tipo `a`, ou é um nó contendo uma árvore à esquerda, um valor do tipo `a` no meior e uma árvore à direita.

UFABC

- Desenhe a seguinte árvore:

```
1  t :: Tree Int
2  t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
3          (Node (Leaf 6) 7 (Leaf 9))
```

UFABC

- Podemos definir uma função `contem` que indica se um elemento `x` está contidado em uma árvore `t`:

```haskell
contem :: Eq a => Tree a -> a -> Bool
contem (Leaf y) x       = x == y
contem (Node l y r) x = x == y || l `contem` x
                               || r `contem` x

> t `contem` 5
True
> t `contem` 0
False
```

Exercício 4

UFABC

- Altere a função `contem` levando em conta que essa é uma árvore de busca, ou seja, os nós da esquerda são menores ao nó atual, e os nós da direita são maiores.

Resposta - Exercício 4

UFABC

```haskell
contem :: Eq a => Tree a -> a -> Bool
contem (Leaf y) x                  = x == y
contem (Node l y r) x | x == y     = True
                      | x < y      = l `contem` x
                      | otherwise  = r `contem` x
```

Classes de Tipo

UFABC

- Aprendemos em uma aula anterior sobre as classes de tipo, classes que definem grupos de tipos que devem conter algumas funções especificadas.
- Para criar um novo tipo utilizamos a função `class`:

```haskell
class Eq a where
   (==), (/=) :: a -> a -> Bool

   x /= y = not (x == y)
```

UFABC

- Essa declaração diz: *para um tipo* `a` *pertencer a classe* `Eq` *deve ter uma implementação das funções* `(==)` *e* `(/=)`.

```haskell
class Eq a where
   (==), (/=) :: a -> a -> Bool

   x /= y = not (x == y)
```

UFABC

- Além disso, ela já traz uma definição padrão da função `(/=)`, então basta definir `(==)`.

```haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool

    x /= y = not (x == y)
```

UFABC

- Para definirmos uma nova instância de uma classe basta declarar:

```haskell
instance Eq Bool where
   False == False = True
   True  == True  = True
   _     == _     = False
```

Instâncias da Classe

UFABC

- Apenas tipos definidos por `data` e `newtype` podem ser instâncias de alguma classe.

UFABC

- Uma classe pode estender outra para formar uma nova classe. Considere a classe `Ord`:

```haskell
class Eq a => Ord a where
   (<), (<=), (>), (>=) :: a -> a -> Bool
   min, max             :: a -> a -> a

   min x y | x <= y     = x
           | otherwise  = y

   max x y | x <= y     = y
           | otherwise  = x
```

- Ou seja, antes de ser uma instância de `Ord`, o tipo deve ser também instância de `Eq`.

Instância de Ord

UFABC

- Seguindo nosso exemplo de Booleano, temos:

```
1  instance Ord Bool where
2      False < True = True
3      _     < _    = False
4
5      b <= c = (b < c) || (b == c)
6      b > c  = c < b
7      b >= c = c <= b
```

Derivação de instâncias

UFABC

- Em muitos casos o Haskell consegue inferir as instâncias das classes mais comuns, nesses casos basta utilizar a palavra-chave `deriving` ao definir um novo tipo:

```haskell
data Bool = False | True
          deriving (Eq, Ord, Show, Read)
```

UFABC

- Implementa as funções:

```
succ, pred, toEnum, fromEnum
```

Classe Enum

UFABC

```haskell
data Dias = Seg | Ter | Qua | Qui | Sex | Sab | Dom
                    deriving (Show, Enum)
```

- Enum é enumerativo:

```haskell
succ Seg == Ter
pred Ter  == Seg
fromEnum Seg == 0
toEnum 1 :: Dias == Ter
```

Exercício 5

UFABC

- Defina um tipo para jogar o jogo Pedra, Papel e Tesoura e defina as funções `ganhaDe`, `perdeDe`.
- Defina também uma função denominada `ganhadores` que recebe uma lista de jogadas e retorna uma lista dos índices das jogadas vencedoras.