Paradigmas de Programação

Fabrício Olivetti de França

19 de Julho de 2018

Definindo novos tipos

Novos tipos de dados

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

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

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

Declaração de tipo

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:

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

Declaração de tipo

Porém, não podemos definir tipos recursivos:

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

mas temos outras formas de definir tais tipos...

Declaração de tipo

A declaração de tipos pode conter variáveis de tipo:

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

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

Declaração de tipo

Com isso podemos definir funções utilizando esses tipos:

```
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

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

Declaração de tipo

Como esses tipos são apenas apelidos, eu posso fazer:

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

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

O compilador não distingue um do outro.

Tipos de Dados Algébricos

Tipos de Dados Algébricos

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

Tipos Soma

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

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

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

Exemplo

Com isso podemos criar a função `para`:

```
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

E a função `caminhar`:

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

Tipos Produto

Tipo produto:

```
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

Tipos Produto

Para ser possível imprimir esse tipo:

```
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.

Tipos Produto

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

```
dist :: Ponto -> Ponto -> Double
dist (Ponto x y) (Ponto x' y') = sqrt
                                 $ (x-x')^2 + (y-y')^2
```

```
> dist (Ponto 1 2) (Ponto 1 1)
1.0
```

Tipos Produto

Podemos misturar os tipos soma e produto:

```
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
```

Tipos Produto

`Circulo` e `Retangulo` são funções construtoras:

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

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

Tipos parametrizados

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)
```

Maybe

Eses erros podem ser capturados com a expressão `case`:

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

Either

Outras 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.

Either

```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

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:

```
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

Números Naturais

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:

```
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`.

Números Naturais

Então os primeiros números são definidos como:

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

Números Naturais

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

```
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 (0.5 pto)

Defina a função add sem utilizar a conversão:

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

Árvore Binária

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.

Árvore Binária

Desenhe a seguinte árvore:

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

Árvore Binária

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

```
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

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.

Classes de Tipo

Clases de Tipo

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`:

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

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

Clases de Tipo

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

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

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

Clases de Tipo

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

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

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

Instâncias da Classe

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

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

Instâncias da Classe

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

Classes de Tipo

Uma classe pode extender outra para forma uma nova classe.

Considere a classe `Ord`:

```
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

Seguindo nosso exemplo de Booleano, temos:

```
instance Ord Bool where
    False < True = True
    _     < _    = False

    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Derivação de instâncias

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)
```

Classe Enum

Implementa as funções:

`succ, pred, toEnum, fromEnum`

Classe Enum

```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 (0.5 pto)

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.