3ª Aula Prática de PLC: Avaliação Preguiçosa, Tipos Algébricos e Classes de Tipos em Haskell

Roteiro

- Avaliação Preguiçosa
- Tipos Sinônimos
- Tipos Algébricos
- Classes de Tipos
- Instâncias de Classes
- Classes Derivadas

Avaliação Preguiçosa

Lazy evaluation – avaliação de uma expressão se dá apenas quando seu valor é necessário

F1 :: Int → Int → Int

F1 a b = a + 12

G :: Int → Int

G c = c + g c

F1 3 (g 0) = ?

F1 (g 0) 3 = ?

Exercícios Resolvidos

1. Considere, as seguintes as funções f, g e h abaixo:

```
f :: Int -> Int
f x = fst (x, x + (f x))

g :: Int -> Int -> Bool
g n r = n>=r && (g (n-1) r)

h :: Int -> Int -> Bool
h n r = (h (n-1) r) && n>=r
```

Qual o resultado da avaliação de f 10, g 3 2 e h 3 2? Alguma dessas expressões entra em loop infinito?

Tipos Sinônimos

Em Haskell, tipos sinônimos não criam tipos de dados novos, apenas são uma maneira de dar nomes diferentes para tipos já existentes.

```
type String = [Char]
type Nome = String
type Telefone = String
type AgendaTelefonica = [(Nome, Telefone)]
```

As funções que manipulam um determinado tipo de dados, também manipularão os tipos sinônimos do mesmo.

Tipos Sinônimos

As funções agenda1 e agenda2 abaixo são essencialmente a mesma função:

```
agenda1 :: [(String, String)]
agenda1 = [(“Miguel”, “96364000”), (“Filipe”, “73313574”), (“Joao”, “96375551”), (“Luiz”, “95561630”), (“Maria”, “85420900”)]

agenda2 :: AgendaTelefonica
agenda2 = [(“Miguel”, “96364000”), (“Filipe”, “73313574”), (“Joao”, “96375551”), (“Luiz”, “95561630”), (“Maria”, “85420900”)]
```

Tipos Sinônimos

Tipos sinônimos também podem ser parametrizados.

```
type Lista t = [t]
type Tupla x y = (x, y)
```

Exemplos de funções para manipular esses tipo seriam:

```
cabeca :: Lista t -> t
cabeca [ ] = error “Lista vazia!”
cabeca (x:xs) = x

inverterTupla :: Tupla a b -> Tupla b a
inverterTupla (x, y) = (y, x)
```

Exercícios Resolvidos

2. Defina o tipo Ponto como sinônimo de uma tupla com três Float. Defina a função distancia :: Ponto -> Ponto -> Float, que calcula a distância entre três pontos em um espaço tridimensional.

Exemplo:
Main> distancia (0, 0, 0) (1, 1, 1)
1.7320508

Tipos Algébricos

Os tipos algébricos, representam um aumento na expressividade e no poder de abstração de Haskell, visto que com eles é possível construir novos tipos de dados (incluindo tipos recursivos, polimórficos, enumerados).

```
data Bool = True | False
data Bit = Zero | Um
data Elemento = Ar | Agua | Fogo | Terra
data Lado = Lado Float
data Triangulo = Escaleno Lado Lado Lado| Isoceles Lado Lado
               | Equilatero Lado
data Vetor t = Nulo | Elemento t (Vetor t) deriving (Show)
data Arvore t = Vazia | Folha t (Arvore t) (Arvore t) deriving (Show)
```

Tipos Algébricos

Um tipo de dado algébrico admite mais de um construtor, desde que tenham nomes diferentes. Além disso, cada construtor tem seus próprios parâmetros (ou não têm nenhum também).

De forma geral, um tipo algébrico é definido da seguinte forma:

```
data NovoTipo = Construtor1 t11 t12 ... t1n1
              | Construtor2 t21 t22 ... t2n2
              ...
              | Construtorm tm1 tm2 ... tmnm
              deriving (TC1, TC2, ..., TCk)
```

Tipos Algébricos

Através do casamento de padrões projetam-se operações que geram e processam os tipos algébricos.

```
numeroBit :: Bit -> Int
numeroBit Zero = 0
numeroBit Um = 1

vetorParaLista :: Vetor t -> [t]
vetorParaLista Nulo = [ ]
vetorParaLista (Elemento x v) = x:(vetorParaLista v)

emOrdem :: Arvore Int -> [Int]
emOrdem Vazia = [ ]
emOrdem (Folha x e d) = emOrdem e ++ [x] ++ emOrdem d
```

Exercícios Resolvidos

3. Diretórios podem ser pastas ou arquivos.
Arquivos contêm informação sobre nome e tamanho.
Pastas possuem nome e um conjunto de diretórios.

Implemente os tipos sinônimos para Nome e Tamanho e o tipo algébrico Diretorio e seus construtores de tipo Arquivo e Pasta. Crie, também, a função size que dado um Diretorio retorne uma tupla com o tamanho ocupado pelo mesmo (soma de todos os tamanhos dos arquivos) e a quantidade de arquivos (não de pastas) nesse diretório.

size :: Diretorio -> (Tamanho, Int)

Classes de Tipos

Em Haskell, uma classe de tipos (typeclass) é uma coleção de tipos para os quais um conjunto de funções está definido. Por exemplo, os tipos que admitem comparação (se são iguais ou diferentes) entre os seus membros são tipos que pertencem a classe Eq, os tipos que admitem uma ordenação de seus membros pertencem a classe Ord, etc.

As funções de uma determinada classe são denominadas funções sobrecarregadas.

```
Prelude> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Classes de Tipos

A forma de declarar uma classe é a seguinte:

```
class NomeClasse tipoVariavel where
    assinaturaDafuncao1
    assinaturaDafuncao2
    ...
    assinaturaDafuncaoN
```

Um exemplo seria a classe Eq, definida como:

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

Instâncias de Classes

Para incorporar um novo tipo a uma classe, é necessário criar uma nova instância da classe e definir explicitamente cada função para os elementos do novo tipo. A instância pode ser realizada através da palavra reservada instance.

```
instance NomeClasse tipoInstanciado where
    implementacaoDafuncao1
    implementacaoDafuncao2
    ...
    implementacaoDafuncaoN
```

Instâncias de Classes

Para que o tipo Bool, por exemplo, seja uma instância da classe Eq, define-se:

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

O mesmo vale para o tipo Bit:

```
instance Eq Bit where
    Zero == Zero = True
    Um == Um = True
    _ == _ = False
```

Instâncias de Classes

É possível, também, criar novos tipos enumerados simplesmente instanciando a classe Enum:

```
instance Enum Elemento where
    fromEnum Ar = 1
    fromEnum Agua = 2
    fromEnum Fogo = 3
    fromEnum Terra = 4
    toEnum 1 = Ar
    toEnum 2 = Agua
    toEnum 3 = Fogo
    toEnum 4 = Terra
```

Instâncias de Classes

Dado que algumas instâncias são óbvias, Haskell providencia uma forma automática de criar instâncias de um novo tipo de dados em suas classes básicas como Enum, Eq, Ord, Show, Read, etc. A linguagem faz isso através da palavra reservada deriving, utilizada na declaração de um tipo algébrico.

```
data Bit = Zero | Um deriving (Show)
data Elemento = Ar | Agua | Fogo | Terra deriving (Enum, Show)
data Arvore t = Vazia | Folha t (Arvore t) (Arvore t) deriving (Show)
```

Classes Derivadas

Uma classe em Haskell pode herdar as propriedades de outras classes. Como exemplo, a classe Ord que possui as operações >, >=, <, <=, max e min, além de herdar a operação == da classe Eq.

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

O algoritmo quicksort pode ser redefinido agora de uma forma mais geral:

```
quicksort :: (Ord t) => [t] -> [t]
quicksort [ ] = [ ]
quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++ [x]
          ++ quicksort [y | y <- xs, y > x]
```

Classes Derivadas

O caso em que um tipo herda de mais de uma classe é conhecido na literatura como herança múltipla.

```
selSort :: (Ord a, Show a) => [a] -> String
selSort [ ] = [ ]
selSort (x:[ ]) = show(x) ++ “.”
selSort x  = show(m) ++ “, ” ++ selSort (remove m x)
        where m = minimum x

remove :: (Ord a) => a -> [a] -> [a]
remove e [ ] = [ ]
remove e (x:xs)       | x == e    = xs
           | otherwise = x:(remove e xs)
```

Exercícios Resolvidos

4. Crie o tipo algébrico Complexo que é representado por uma parte real e uma parte imaginaria. Faça, também, ele ser uma instância de Show de modo que ele seja impresso da seguinte maneira “Real + Imaginaria.j” ou “Real – Imaginaria.j”.

Exercícios

1) Crie o tipo algébrico Complexo que é representado por uma parte real e uma parte imaginaria. Faça, também, ele ser uma instância da classe Num e Eq.

Exercícios

2)Uma estrutura de dados conjuntos-disjuntos é uma coleção D = {D1, ..., Dn} de conjuntos dinâmicos disjuntos, onde cada conjunto Dk é identificado por um representante, que é um membro do conjunto (Considere o primeiro elemento do conjunto). Dados os dois tipos algébricos abaixo que representam conjuntos disjuntos de números inteiros, crie uma classe **OprDisjointSet** que contenham as funções makeSet, union e find.

data LDisjointSet = LDS [[**Int**]] **deriving** (**Eq**, **Show**)

data TDisjointSet = Void | **TDS** [**Int**] **TDisjointSet deriving** (**Eq**, **Show**) Crie também instâncias da classe **OprDisjointSet** para os dois tipos algébricos. Vale, ainda, ressaltar que a ordem dos elementos e/ou dos conjuntos é irrelevante.

Observações:

•A função makeSet **:: Int ->** t **->** t, cria um novo conjunto cujo único elemento é x, desde que x não pertença a outro conjunto da coleção.

•A função union **:: Int -> Int ->** t **->** t, executa a união dos conjuntos que contêm os elementos x e y, transformando os dois conjuntos em um único conjunto.

•A função find **:: Int ->** t **-> Maybe Int**, retorna o representante do conjunto que contêm o elemento procurado x. Caso x não pertença a nenhum dos conjuntos, **Nothing** deve ser retornado.

•A função makeSet nas duas instâncias deve ser implementada utilizando apenas a função find.

Exercícios

Exemplos: **Main>** makeSet **7** (**LDS** [[**1**, **2**, **3**], [**4**, **5**, **6**]])
LDS [[**1**, **2**, **3**], [**4**, **5**, **6**], [**7**]]
Main> makeSet **7** (**LDS** [[**1**, **2**, **3**], [**4**, **5**, **6**], [**7**]])
LDS [[**1**, **2**, **3**], [**4**, **5**, **6**], [**7**]]
Main> union **2 6** (**LDS** [[**1**, **2**, **3**], [**4**, **5**, **6**], [**7**]])
LDS [[**1**, **2**, **3**, **4**, **5**, **6**], [**7**]]
Main> find **8** (**TDS** [**1**, **2**, **3**] (**TDS** [**4**, **5**, **6**] (**TDS** [**7**] **Void**)))
Nothing
Main> makeSet **8** (**TDS** [**1**, **2**, **3**] (**TDS** [**4**, **5**, **6**] (**TDS** [**7**] **Void**)))
TDS [**8**] (**TDS** [**1**, **2**, **3**] (**TDS** [**4**, **5**, **6**] (**TDS** [**7**] **Void**)))

Bônus

Desenvolva o que se pede.
a) Defina um tipo para representar árvore binária. Considere que vamos trabalhar com uma árvore binária em que cada nó possui valor maior que o da subárvore à esquerda e menor que o da subárvore à direita (sem repetições).
b) Implemente as funções:
- insert, que insere um nó em uma árvore;
- arvLista, que transforma a árvore em uma lista;
- somaArv, que devolve a soma dos valores dos nós da árvore;
- listArv, que dada uma lista devolve uma árvore.
- maiorArv, que dada uma árvore retorna a altura dela.

• Os exercícios práticos devem ser realizados individualmente e enviados por e-mail com o assunto "**[IF686EC] EXERCÍCIOS PRÁTICOS 03**" até as **23:59** de hoje (25.04.2017).
• As resoluções dos exercícios devem estar em arquivos diferentes, um arquivo por exercício, com os nomes no formato "**Q[número do exercício].hs**".