3ª Aula Prática de PLC:
Tipos Algébricos em Haskell

Roteiro

- Tipos Sinônimos
- Tipos Algébricos

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

1. 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

2. 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)

Exercícios Práticos

1. Uma pilha é uma estrutura de dados em que todas as operações de inserção e remoção são realizadas pela mesma extremidade denominada topo. Geralmente, para manipular uma pilha, utilizam-se as operações push(insere um novo elemento no topo da pilha), pop(remove o elemento do topo da pilha) e top (acessa/retorna o elemento do topo da pilha).

Crie os tipos algébricos polimórficos Pilha1 (utilizando apenas tipos recursivos) e Pilha2 (utilizando apenas listas) que podem ser exibidos. Implemente também as operações push1, pop1 e top1 que manipulam o tipo Pilha1 e push2, pop2 e top2 que manipulam o tipo Pilha2.

Caso a pilha esteja vazia e seja utilizada alguma das operações pop ou top, encerre a execução do programa e exiba a mensagem de erro: “Pilha vazia!”.

Sugestão:

```
data Pilha1 t = Null | Elem t (Pilha1 t) deriving Show
data Pilha2 t = Void | Casa t [Pilha2 t] deriving Show
```

Exercícios Práticos

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 Práticos

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;
- delete, que remove um nó da árvore.

• Os exercícios práticos devem ser realizados individualmente e enviados por e-mail com o assunto "**[IF686EC] EXERCÍCIOS PRÁTICOS 03**" até 12:00h de domingo (01.10.2017).
• As resoluções dos exercícios devem estar em arquivos diferentes, um arquivo por exercício, com os nomes no formato "**Q[número do exercício].hs**".