Tipos algébricos e árvores

Tipos algébricos

A construção de tipos álgébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implementação de tipos enumerados, co-produtos (união disjunta de tipos), e tipos indutivos (recursivos).

O tipo das listas é um exemplo de um tipo indutivo (recursivo):
Um lista - ou é vazia,

- ou tem um elemento e a uma sub-estrutura que é também uma lista.

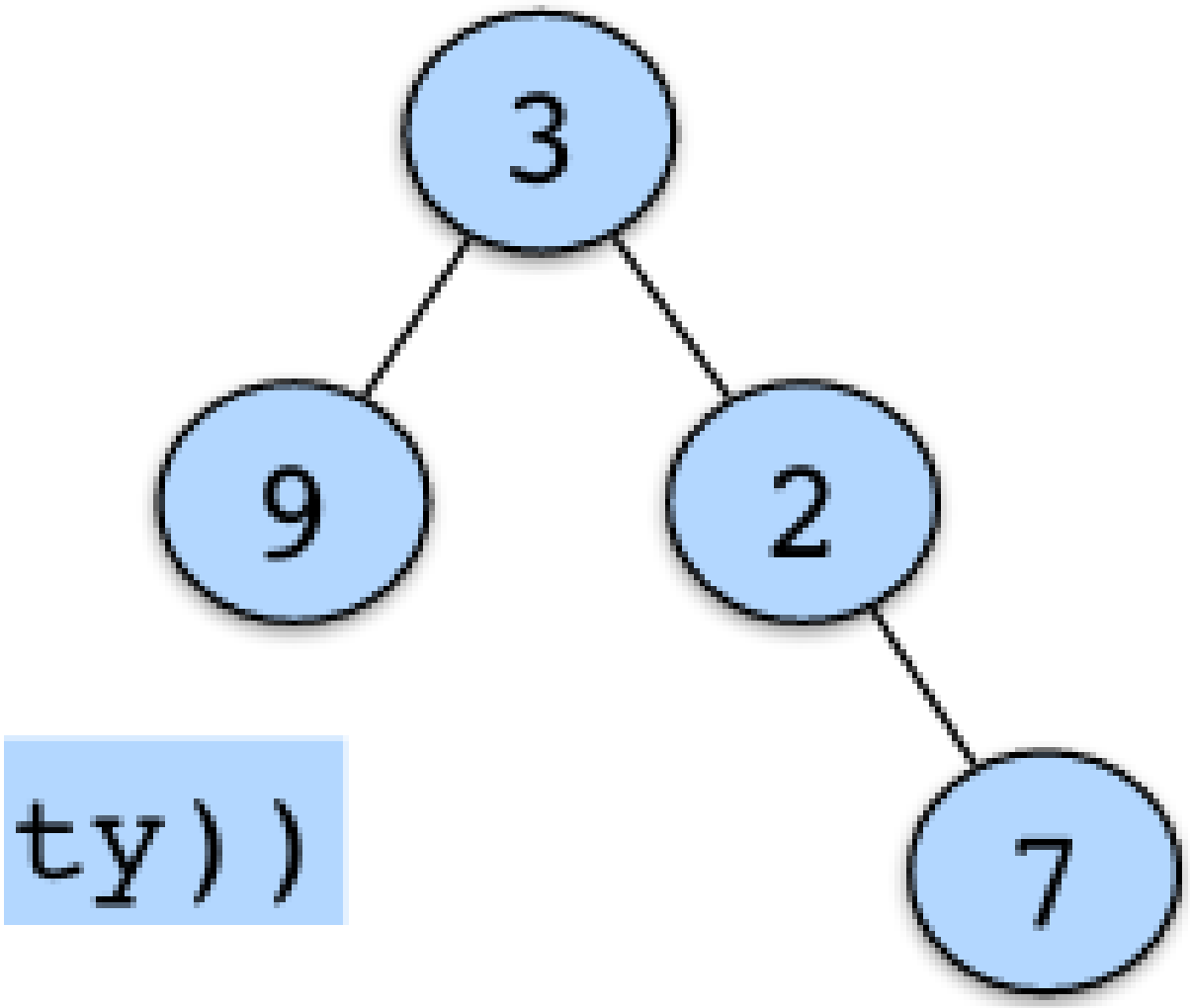
```
[1,2,3] = 1 : [2,3] = 1 : 2 : [3] = 1 : 2 : 3 : []
data [a] = []
         | (:) a [a]
```

A noção de árvore binária expande este conceito.
Uma árvore binária - ou é vazia,

- ou tem um elemento e a duas sub-estruturas que também são árvores.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

```
            Node 3 (Node 9 Empty Empty) (Node 2 Empty (Node 7 Empty Empty))
```



Árvores Binárias

As árvores binárias são extruturas de dados muito úteis para organizar a informação.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
   deriving (Show)
```

Os construtores da árvore são:

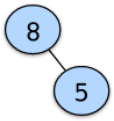
```
Empty :: BTree a                                                    Emmpty
representa a árvore vazia
Node  :: a -> (BTree a) -> (BTree a) -> (BTree a)
```

Node recebe um elemento e duas árvores, e constrói a árvore com esse elemento na raiz, uma arvore no lado esquerdo e outra no lado direito.

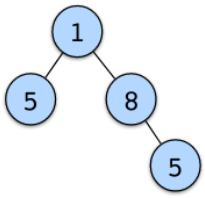
Exemplos de árvores:

```
arv1 = Node 5 Empty Empty
```

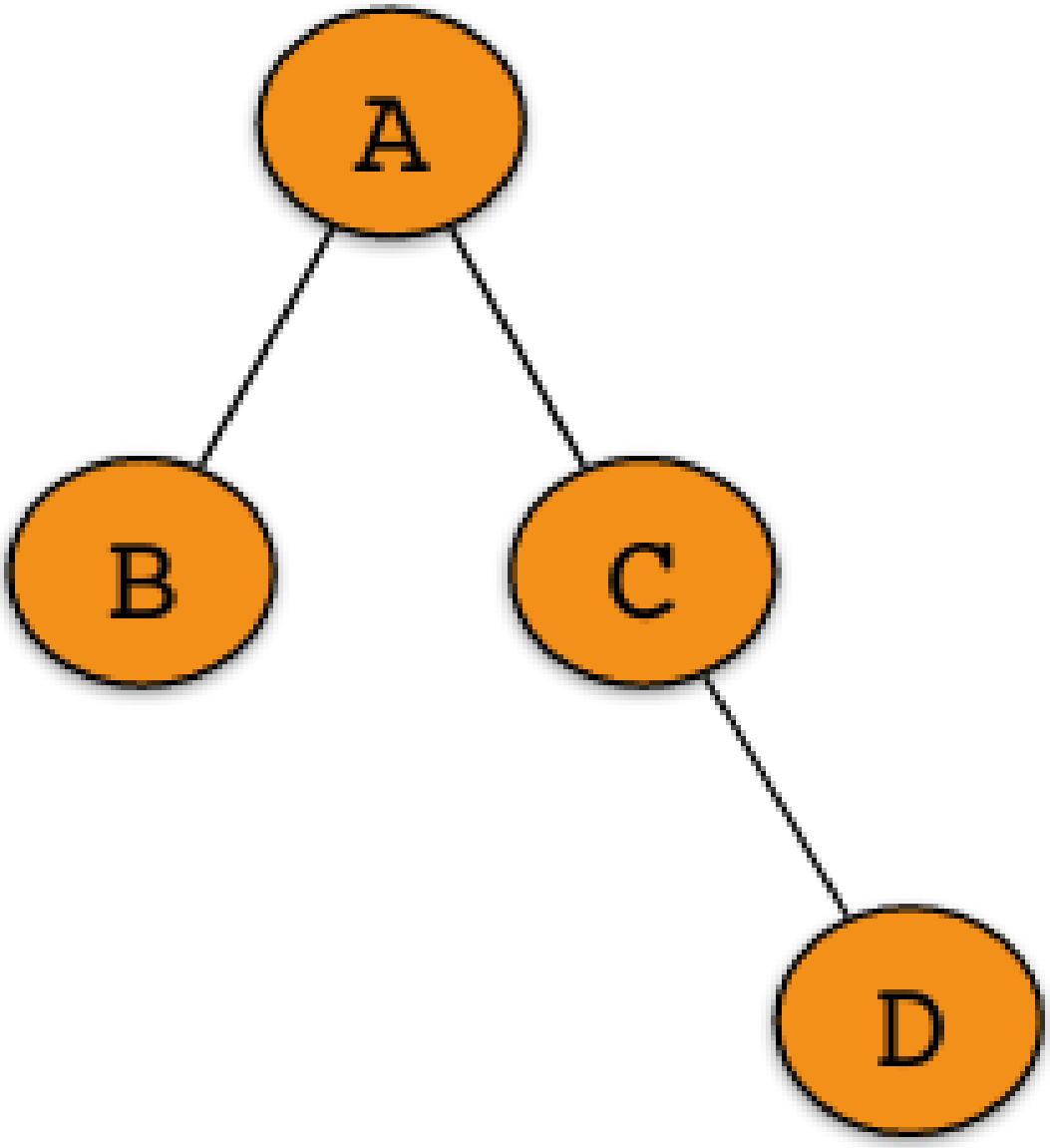

```
arv2 = Node 8 Empty arv1
```




```
arv3 = Node 1 arv1 arv2
```



Terminologia:



O nodo A é a raiz da árvore;
Os nodos B e C são filhos (ou descendentes) de A;
O nodo C é o pai de D;
B e C são folhas da árvore;
O caminho (path) de um nodo é a sequência de nodos da raiz até esse nodo. Por exemplo,

A,C,D é o caminho para o nodo D;
A altura da árvore é o comprimento do caminho mais longo. Esta árvore tem altura 3.

As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com padrões de recursividade semelhantes aos dos tipos de dados.

Exemplos:

Calcular o número de nodos que tem uma árvore:

```
conta :: BTree a -> Int
conta Empty = 0
conta (Node x e d) = 1 + conta e + conta d
```

Somar todos os nodos de uma árvore de números:

```
sumBT :: Num a => BTree a -> a
sumBT Empty = 0
sumBT (Node x e d) = x + sumBT e + sumBT d
```

Calcular a altura de uma árvore:

```
altura :: BTree a -> Int
altura Empty = 0
altura (Node _ e d) = 1 + max (altura e) (altura d)
```

As funções map e zip para árvores binárias:

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT Empty = Empty
mapBT f (Node x e d) = Node (f x) (mapBT f e) (mapBT f d)

zipBT :: BTree a -> BTree b -> BTree (a,b)
zipBT _ _ = Empty
zipBT (Node x e d) (Node x1 e1 d1) = Node (x,x1) (zipBT e e1) (zipBT d d1)
```

Travessias de árvores binárias

Uma árvore pode ser percorrida de várias formas. As principais estratégias são:

- Travessia preorder:
  |--> visitar a raiz, depois a árvore esquerda e a seguir a árvore direita.

```
preorder :: BTree a -> [a]
preorder Empty = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

- Travessia inorder:
  |--> visitar a árvore esquerda, depois a raiz e depois a árvore direita.

```
inorder :: BTree a -> [a]
inorder Empty = []
inorder (Node x e d) = (inorder e)  ++ [x] ++ (inorder d)
```

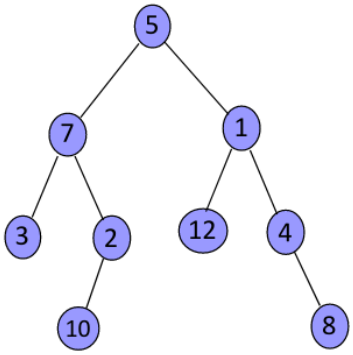
- Travessia postorder:
  |--> visitar árvore esquerda, depois árvore direita, e a seguir a raiz.

```
postorder :: BTree a -> [a]
postorder Empty = []
postorder (Node x e d) = (postorder e) ++ (postorder d) ++ [x]
```

Exemplo:

Travessias de árvores binárias

```
arv = (Node 5 (Node 7 (Node 3 Empty Empty)
                      (Node 2 (Node 10 Empty Empty) Empty)
              )
              (Node 1 (Node 12 Empty Empty)
                      (Node 4 Empty (Node 8 Empty Empty))
              )
      )
```




```
preorder arv  = [5,7,3,2,10,1,12,4,8]

inorder arv   = [3,7,10,2,5,12,1,4,8]

postorder arv = [3,10,2,7,12,8,4,1,5]
```

Árvore Binárias de Procura

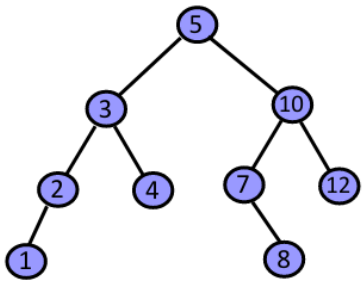
Uma árvore binária em que o valor de cada nodo é maior do que os nodos à sua esquerda, e menor do que os nodos à sua direita diz-se uma árvore binária de procura (ou de pesquisa).

Exemplo:

Uma **árvore binária de procura** é uma árvore binária que verifica as seguinte condição:

- a raiz da árvore é maior do que todos os elementos que estão na sub-árvore esquerda;
- a raiz da árvore é menor do que todos os elementos que estão na sub-árvore direita;
- ambas as sub-árvores são árvores binárias de procura.

Exemplo: Esta é uma árvore binária de procura de procura



Exemplos de utilização:
--> Testar se um elemento pertence a uma árvore:

```
elemBT :: Ord a => a -> BTree a -> Bool
elemBT x Empty = False
elemBT x (Node y e d)
          | x < y  = elemBT x e
          | x > y  = elemBT x d
          | x == y = True
```

--> Inserir um elemento numa árvore binária de procura.

```
insertBT :: Ord a => a -> BTree a -> BTree a
insertBT x Empty = Node x Empty Empty
insertBT x (Node y e d)
        | x < y  = Node y (insertBT x e) d
        | x > y  = Node y e (insertBT x d)
        | x == y = Node y e d
```

Árvores binárias de procura possibilitam pesquisas potencialmente mais eficientes do que as listas. Exemplos:

--> Pesquisa numa lista não ordenada: (o número de comparações de chaves é no máximo igual ao comprimento da lista)

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup x [] = Nothing
lookup x ((y,z):t) | x == y = Just z
                                | x /= y = lookup x t
```

--> Pesquisa numa lista ordenada:

```
lookupSL :: Ord a => a -> [(a,b)] -> Maybe b
lookupSL x [] = Nothing
lookupSL x ((y,z):t) | x < y  = Nothing
                     | x == y = Just z
                     | x > y  = lookupSL x t
```

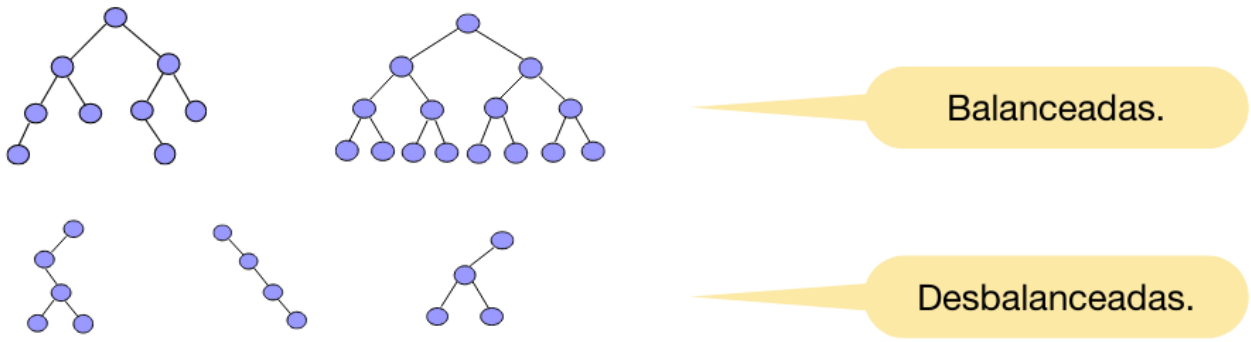
--> Pesquisa numa árvore binária: (o número de comparações de chaves é no máximo igual à altura da árvore)

```
lookupBT :: Ord a => a -> BTree (a,b) -> Maybe b
lookupBT x Empty = Nothing
lookupBT x (Node (y,z) e d) | x < y  = lookupBT x e
                            | x > y  = lookupBT x d
                            | x == y = Just z
```

Árvore balanceadas

Uma árvore binaria diz-se balanceada (ou equilibrada) se é vazia, ou se verifica as seguintes condições:

- as alturas da sub-árvore esquerda e direita diferem no máximo uma unidade;
- ambas as subárvores são balanceadas



A pesquisa em árvores binárias de procura são especialmente mais eficientes se as árvores forem balanceadas. Uma forma de balancear uma árvore de procura consiste em: primeiro gerar uma lista ordenada com os seus elementos e depois, a partir dessa lista, gerar a árvore.

```
balance :: BTree a -> BTree a
balance t = constroi (inorder t)

constroi :: [a] -> BTree a
constroi [] = Empty
constroi l = let n = length l
                 (l1, x:l2) = splitAt (n `div` 2) l
             in Node x (constroi l1) (constroi l2)
```

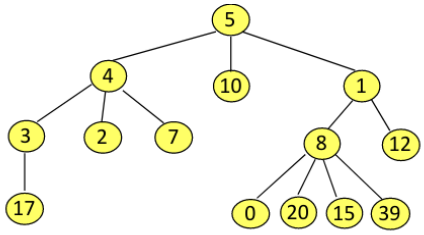
Árvores Irregulares

Nas árvores irregulares cada nodo pode ter um número variável de descendentes. O seguinte tipo de dados é uma implementação de
árvores irregulares, não vazias.

```
data RTree a = R a [RTree a]
  deriving (Show)
```

O único construtor das árvores é: `R :: a -> [RTree a] -> RTree a`
--> R recebe o elemento que fica na raiz da árvore e a lista das sub-árvores com que vai construir a árvore. Exemplo:

```
R 5 [ R 4 [ R 3 [R 17 []], R 2 [], R 7 []],
      R 10 [],
      R 1 [ R 8 [ R 0 [], R 20 [], R 15 [],  R 39 [] ],
            R 12 [] ]
    ]
```



Rose Trees

Como é de esperar, as funções definidas sobre rose trees seguem um padrão de recursividade compatível com a sua definição indutível.

Exemplo:
Contar os nodos de uma árvore:

```
contaRT :: RTree a -> Int
contaRT (R x l) = 1 + sum (map contaRT l)
```

Calcular a altura de uma árvore:

```
alturaRT :: RTree a -> Int
alturaRT (R x []) = 1
alturaRT (R x l ) = 1 + maximum (map alturaRT l)
```

Testar se um elemento pertence a uma árvore.

```
pertenceRT :: Eq a => a -> RTree a -> Bool
pertenceRT x (R y l) = x==y || or (map (pertenceRT x) l)
```

Fazer uma travessia preorder de uma árvore.

```
preorderRT :: RTree a -> [a]
preorderRT (R x l) = x : concat (map preorderRT l)
```

Outras árvores:

- Leaf trees:
  Árvores binárias em que a informação está apenas nas folhas da árvore.
  Os nós intermédios não têm informação.

```
data LTree a = Tip a
                        | Fork (LTree a) (LTree a)
```

- Full trees
  Árvores binárias que têm informação nos nós intermédios e nas folhas.
  A informação guardada nos nós e nas folhas pode ser de tipo diferente.

```
data FTree a b = Leaf b
                          | No a (FTree a b) (FTree a b)
```

#haskell
#SoftwareEngineering