Лекция 13

Полиморфни алгебрични типове

Дефинициите на алгебрични типове могат да съдържат променливи на типове (типови променливи, type variables) ***a***, ***b*** и т.н. По този начин се дефинират *полиморфни типове*.

Тези дефиниции изглеждат така, както беше показано в предишната лекция, като променливите на типове се включват след името на типа в лявата страна на дефиницията.

Пример

```
data Pairs a = Pr a a
```

Примерни елементи на този тип:

```
Pr 2 3 :: Pairs Int
Pr [ ] [3] :: Pairs [Int]
Pr [ ] [ ] :: Pairs a
```

Дефиниция на функция, която проверява дали са равни двете части на дадена двойка:

```
equalPair :: Eq a => Pairs a -> Bool
equalPair (Pr x y) = (x==y)
```

Списъци

Вграденият списъчен тип може да бъде дефиниран като алгебричен например по следния начин:

```
data List a = NilList | Cons a (List a)
              deriving (Eq,Show,Read)
```

Тук синтаксисът [a], [ ] и ‘:’ е аналогичен на List a, NilList и Cons. Така типът “списък” е добър пример за рекурсивен полиморфен тип.

Двоични дървета

Дърветата, които дефинирахме на предишната лекция, бяха дървета от цели числа (дървета от тип Int). Ако искаме да дефинираме двоично дърво от произволен тип ***a***, това може да стане с помощта на конструкция от вида

```
data Tree a = Nil | Node a (Tree a) (Tree a)
              deriving (Eq,Ord,Show,Read)
```

При това някои от вече дискутираните дефиниции на функции за работа с двоични дървета от цели числа могат да бъдат използвани и в общия случай, например:

```
depth :: Tree a -> Int
depth Nil             = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

Дефиниции на някои функции за работа с двоични дървета от произволен тип

Намиране на броя на върховете на двоично дърво:

```
numberOfElements :: Tree a -> Int
numberOfElements Nil = 0
numberOfElements (Node n leftTree rightTree)
   = 1 + numberOfElements leftTree +
         numberOfElements rightTree
```

Намиране на сумата от върховете на двоично дърво от цели числа:

```
sumOfElements :: Tree Int -> Int
sumOfElements Nil = 0
sumOfElements (Node n leftTree rightTree)
   = n + numberOfElements leftTree +
         numberOfElements rightTree
```

Намиране на броя на листата на двоично дърво:

```
countLeaves :: Tree a -> Int
countLeaves Nil = 0
countLeaves (Node _ Nil Nil) = 1
countLeaves (Node _ leftTree rightTree)
   = countLeaves leftTree + countLeaves rightTree
```

Трансформиране на двоично дърво (прилагане на дадена функция към всеки от върховете на дървото):

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Nil = Nil
mapTree f (Node x t1 t2)
   = Node (f x) (mapTree f t1) (mapTree f t2)
```

Намиране на върховете от k-то ниво на дадено двоично дърво:

```
onKLevel :: Tree a -> Int -> [a]
onKLevel Nil _ = []
onKLevel (Node n lt rt) 1 = [n]
onKLevel (Node _ lt rt) k
   = (onKLevel lt (k-1)) ++ (onKLevel rt (k-1))
```

Намиране на броя на листата от k-то ниво на дадено двоично дърво:

```
kLevelLeaves :: Tree a -> Int -> Int
kLevelLeaves Nil _ = 0
kLevelLeaves (Node _ Nil Nil) 1 = 1
kLevelLeaves (Node _ lt rt) k
   = (kLevelLeaves lt (k-1)) +
     (kLevelLeaves rt (k-1))
```

Трансформиране на списък в двоично дърво:

```
createTree :: [a] -> Tree a
createTree [] = Nil
createTree list
   = Node root (createTree leftList)
               (createTree rightList)
     where
       mid = div (length list) 2
       secondPart = drop mid list
       leftList = take mid list
       root = head secondPart
       rightList = tail secondPart
```