Алгебрични типове в Haskell

Общи сведения за алгебричните типове

Дефиницията на един алгебричен тип започва с ключовата дума data, след която се записват името на типа, знак за равенство и ***конструкторите*** на типа. Името на типа и имената на конструкторите задължително започват с главни букви.

Пример

```
data Day = Monday | Tuesday | Wednesday | Thursday
               | Friday | Saturday | Sunday
```

Изброени типове

Най-простата разновидност на алгебричен тип се дефинира чрез изброяване на елементите на типа, както беше направено в последния пример.

Следват още примери за изброени типове:

```
data Temp = Cold | Hot
data Season = Spring | Summer | Autumn | Winter
```

Дефинирането на функции върху такива типове се извършва с помощта на стандартните техники, например с използване на подходящи образци:

```
weather :: Season -> Temp
weather Summer = Hot
weather _      = Cold
```

Производни типове

Вместо използването на вектори можем да дефинираме тип с определен брой компоненти като алгебричен тип. Такива типове често се наричат ***производни типове*** (***резултатни типове***; ***product types***).

Пример

```
data People = Person Name Age
```

Тук Name е синоним на String, а Age е синоним на Int:

```
type Name = String
type Age  = Int
```

Горната дефиниция на People може да бъде интерпретирана както следва:

За да се конструира елемент на типа People, е необходимо да се предвидят (дадат като аргументи) две стойности: едната (нека я наречем st) от тип Name, а другата (нека я наречем n) – от тип Age.

Елементът на People, конструиран по този начин, ще има вида Person st n.

Примери за стойности от тип People:

Person “Aunt Jemima” 77

Person “Ronnie” 14

Алтернативи

Геометричните фигури могат да имат различна форма, например кръгла или правоъгълна. Тези алтернативи могат да бъдат включени в дефиниция на тип от вида

```
data Shape = Circle Float |
             Rectangle Float Float
```

Дефиниция от вида на посочената означава, че съществуват два алтернативни начина за конструиране на елемент на Shape.

Примерни данни (обекти) от тип Shape:

```
Circle 3.0
Rectangle 45.9 87.6
```

Дефиниции на функции върху типа Shape:

```
isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound (Rectangle _ _) = False

area :: Shape -> Float
area (Circle r)      = pi*r*r
area (Rectangle h w) = h*w
```

Производни екземпляри на класове

Възможно е да се дефинира нов алгебричен тип като например Season или Shape, който да бъде екземпляр на множество вградени класове.

Примерни дефиниции от посочения вид:

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq,Ord,Enum,Show,Read)

data Shape = Circle Float |
             Rectangle Float Float
             deriving (Eq,Ord,Show,Read)
```

Рекурсивни алгебрични типове

Често характерът на решаваните задачи е такъв, че е естествено някои от алгебричните типове, които потребителят дефинира, да се описват в термините на самите себе си. Такива алгебрични типове се наричат ***рекурсивни***.

Например понятието „израз“ може да се дефинира или като ***литерал*** – цяло число, или като комбинация на два израза, в която се използва аритметичен оператор като + или –.

Примерна дефиниция на Haskell:

```
data Expr = Lit Int |
            Add Expr Expr |
            Sub Expr Expr
```

Аналогично понятието „двоично дърво“ може да се дефинира или като nil, или като комбинация от стойност и две поддървета.

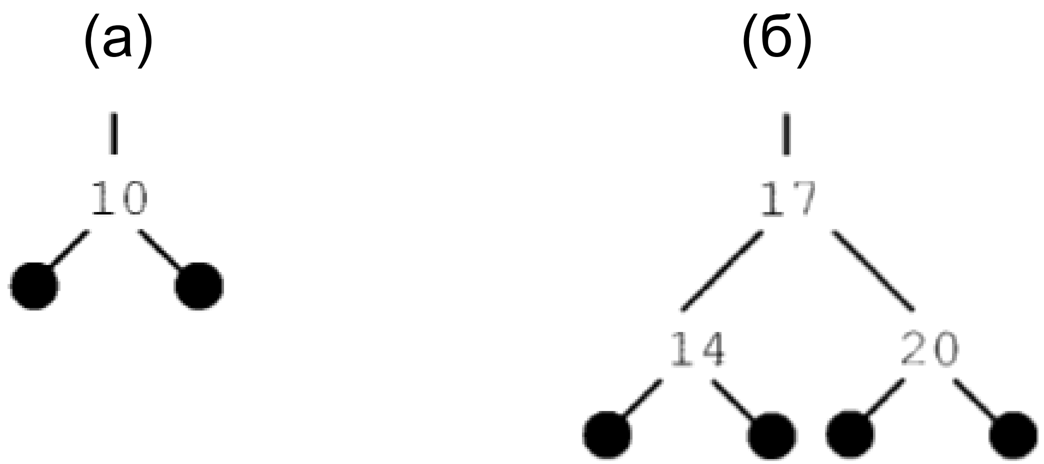
Съответната дефиниция на Haskell изглежда по следния начин:

```
data NTree = NilT |
             Node Int NTree NTree
```

Тази дефиниция е подходяща за моделирането на двоични дървета от цели числа (двоични дървета от тип Int).

Празното дърво се представя чрез **`NilT`**, а дърветата от фиг. (а) и (б) се представят чрез

```
-- (a)
Node 10 NilT NilT
-- (б)
Node 17 (Node 14 NilT NilT) (Node 20 NilT NilT)
```



Дефиниции на някои функции за работа с двоични дървета от цели числа:

```
sumTree,depth :: NTree -> Int

sumTree NilT             = 0
sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2

depth NilT = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

```
occurs :: NTree -> Int -> Int

occurs NilT p = 0
occurs (Node n t1 t2) p
  | n==p      = 1 + occurs t1 p + occurs t2 p
  | otherwise = occurs t1 p + occurs t2 p
```

Взаимно рекурсивни типове

Често е полезно при описанието на един тип да бъдат използвани други типове. Някои от тези типове от своя страна биха могли да цитират първия. В такива случаи се говори за ***взаимно рекурсивни типове***.

Например описанието на даден възрастен човек може да включва биографични детайли, които биха могли да съдържат информация за други хора или поне да цитират други хора.

Примерни дефиниции от посочения вид:

```
data Person = Adult Name Address Biog |
              Child Name
data Biog   = Parent String [Person] |
              NonParent String
```

Тук в случая, когато човекът е родител, биографията му включва подходящ текст и списък от неговите деца, разглеждани като елементи на типа Person.

Дефиниция на функция, която извежда информация за даден човек под формата на символен низ:

```
showPerson (Adult nm ad bio)
   = show nm ++ show ad ++ showBiog bio
...
showBiog (Parent st parList)
   = st ++ concat (map showPerson parList)
...
```

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 предишния параграф, 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 в предишния параграф, 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,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 _ 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 + sumOfElements leftTree +
         sumOfElements 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 tl t2)
   = Node (f x) (mapTree f tl) (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
```

Приложение на алгебричните типове в Haskell: Работа с произволни дървета

Дефиниране на типа:

```
data NTree a = Nil | Node a [(NTree a)]


exTree :: NTree Int
exTree = Node 1 [(Node 2 [Nil, (Node 3 [Nil])]),
(Node 4 [(Node 5 [Nil])])]
```

Намиране на броя на върховете на дадено дърво:

```
numberOfElements :: NTree a -> Int
numberOfElements Nil = 0
numberOfElements (Node _ subTrees)
   = 1 + sum (map numberOfElements subTrees)
```