Работа с безкрайни списъци в Haskell

Едно важно следствие от “мързеливото” оценяване в Haskell е обстоятелството, че езикът позволява да се работи с ***безкрайни структури***. Пълното оценяване на такава структура по принцип изисква безкрайно време, т.е. не може да завърши, но механизмът на “мързеливото” оценяване позволява да бъдат оценявани само тези части (“порции”) на безкрайните структури, които са реално необходими.

Най-прост пример за безкраен списък: безкраен списък от еднакви елементи, например безкраен списък от единици.

```
ones :: [Int]
ones = 1 : ones
```

Оценяването на ones ще продължи безкрайно дълго и следователно ще трябва да бъде прекъснато от потребителя.

Възможно е обаче съвсем коректно да бъдат оценени обръщения към функции с аргумент ones.

Пример

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x:y:zs) = x+y
```

Тогава

```
addFirstTwo ones
 ⟶ addFirstTwo (1:ones)
 ⟶ addFirstTwo (1:1:ones)
 ⟶ 1+1
 ⟶ 2
```

Вградени за Haskell са списъците от вида [n ..] и [n,m ..] .

Например

[3 ..] = [3,4,5,6, …]

[3,5 ..] = [3,5,7,9, …]

Примерни дефиниции на функции – генератори на горните списъци:

```
from :: Int -> [Int]
from n = n : from (n+1)

fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```

Тогава

```
fromStep 3 2
⟶ 3 : fromStep 5 2
⟶ 3 : 5 : fromStep 7 2
⟶   ...
```

Безкрайни списъци могат да се дефинират и чрез определяне на обхвата им (чрез list comprehension). Например списък от всички Питагорови тройки може да бъде генериран чрез избор на стойност на z от [2 ..], следван от избор на подходящи стойности на x и y, по-малки от тази на z.

```
pythagTriples :: [(Int,Int,Int)]
pythagTriples =
   [ (x,y,z) | z <- [2 .. ], y <- [2 .. z-1],
               x <- [2 .. y-1], x*x + y*y == z*z ]
```

Така pythagTriples = [(3,4,5),(6,8,10),(5,12,13),(9,12,15),
(8,15,17),(12,16,20), …]

Забележка. Предложената дефиниция на pythagTriples е коректна. Не е коректна обаче следната дефиниция:

```
pythagTriples2 :: [(Int,Int,Int)]
pythagTriples2 =
   [ (x,y,z) | x <- [2 .. ],
               y <- [x+1 .. ],
               z <- [y+1 .. ],
               x*x + y*y == z*z ]
```

Последната дефиниция не произвежда резултат, защото редът на избор на стойности на елементите на тройките е неподходящ. Първата избрана стойност за x е 2, за y е 3 и при тези фиксирани стойности на x и y следват безброй много неуспешни опити за избор на стойност на z.

Като по-сложен пример ще разгледаме реализация на решетото на Ератостен, което представлява генератор на безкраен списък от простите числа.

```
primes :: [Int]
primes = sieve [2 .. ]

sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs,
                               y `mod` x > 0]
```

Тогава

```
primes
⟶ sieve [2 .. ]
⟶ 2 : sieve [ y | y <- [3 .. ], y `mod` 2 > 0]
⟶ 2 : sieve (3 : [ y | y <- [4 .. ],
                       y `mod` 2 > 0])
⟶ 2 : 3 : sieve [ z | z <- [ y | y <- [4 .. ],
                                 y `mod` 2 > 0],
                       z `mod` 3 > 0]
⟶  ...
⟶ 2 : 3 : sieve [ z | z <- [5,7,9, ... ],
                       z `mod` 3 > 0]
⟶  ...
⟶ 2 : 3 : sieve [5,7,11, ... ]
⟶  ...
```

Можем ли да използваме primes за проверка дали дадено число е просто?

Нека member е функцията за проверка на принадлежност към списък, дефинирана както следва:

```
member :: Eq a => [a] -> a -> Bool
member []     x = False
member (y:ys) x
    | x==y       = True
    | otherwise = member ys x
```

Ако оценим member primes 7, се получава резултат True, но member primes 6 не дава резултат. Причината отново е в това, че трябва да се проверят безброй много елементи на primes преди да се направи заключение, че 6 не е елемент на този списък.

Проблемът може да се реши с отчитане на факта, че списъкът primes е нареден. За целта може да се дефинира нова функция, която проверява дали вторият й аргумент се съдържа в наредения списък, който е неин първи аргумент:

```
memberOrd :: Ord a => [a] -> a -> Bool
memberOrd (x:xs) n
    | x<n        = memberOrd xs n
    | x==n       = True
    | otherwise = False
```

Забележки

1. Записът “Eq a => “ в декларацията на типа на member означава изискването типът a да бъде екземпляр на класа Eq, в който е дефинирана операцията “еквивалентност” (проверката за равенство ==).
2. Записът “Ord a => “ в декларацията на типа на memberOrd означава изискването типът a да бъде нареден (a да бъде екземпляр на класа на наредените типове Ord), т.е. за него да са дефинирани операциите за сравнение >, >=, <, <= и операцията за проверка на равенство ==.

Типове и класове в Haskell.
Дефиниране на нови типове

Генерични и полиморфни функции.
Класове от/на типове

Досега разгледахме два типа функции, които могат да работят с данни от повече от един тип.

Една ***полиморфна функция***, например length (намиране на дължина на списък, чийто елементи могат да бъдат от произволен тип), има единствена дефиниция, която работи върху всички нейни типове.

Генеричните функции, например == (проверка за равенство), + (събиране на числа от един и същ тип) и show (конвертиране на число, булева стойност и др. в низ), могат да бъдат прилагани към данни от много типове, но за различните типове в действителност се използват различни дефиниции (различни ***методи*** на генеричната функция).

Генерични функции и overloading (пренатоварване; додефиниране)

Аритметичният оператор + предизвиква пресмятане на сумата на произволни две числа от един и същ тип. Например, този оператор може да бъде използван за пресмятане на сумата на две цели числа и тогава резултатът ще бъде също цяло число. При събирането на две реални числа (числа с плаваща точка) се получава друго реално число и т.н.

Примери

```
> 1 + 2
3
> 1.1 + 2.2
3.3
```

С други думи, операторът + може да бъде прилаган към аргументи от всеки числов тип и резултатът от прилагането му ще бъде от същия тип.

Тази идея може да бъде прецизирана с използване на т. нар. *ограничение върху класа* (*class constraint*) при дефинирането на типа на оператора +:

```
(+) :: Num a => a -> a -> a
```

Горното ограничение означава, че за всеки тип a, който е ***екземпляр*** на класа Num на/от числови(те) типове, функцията (+) е от тип a -> a -> a.

Всеки тип, който включва едно или повече ограничения върху класа, се нарича ***overloaded*** (*пренатоварен*; *додефиниран*). Следователно, Num a => a -> a -> a е ***overloaded*** (додефиниран) тип и (+) е ***генерична*** (додефинирана) функция.

В действителност повечето от аритметичните функции, дефинирани в стандартния прелюд на Haskell, са генерични.

Например:

```
(-)         :: Num a => a -> a -> a
(*)         :: Num a => a -> a -> a
negate  :: Num a => a -> a
abs         :: Num a => a -> a
signum :: Num a => a -> a
```

Нещо повече, числата в Haskell също са додефинирани. Например, **3 :: Num a => a** означава, че числото 3 е от всеки числов тип *a* (за всеки числов тип *a* числото 3 има тип *a*).

Общи сведения за класовете в Haskell

Най-общо, понятието ***клас*** в езика Haskell се определя като колекция от типове, за които се поддържа множество додефинирани операции, наречени ***методи***.

Например, за функцията elem, която е “вградена” в Haskell, може да се предположи, че е от тип

```
elem :: a -> [a] -> Bool
```

Това обаче ще бъде вярно само за такива типове a, за които е дефинирана операцията (функцията) за проверка за равенство ==.

Следователно, би било полезно да се разполага със средства, които позволяват да се зададат експлицитно определени ограничения върху даден тип, описан с помощта на типова променлива (type variable).

Множеството (колекцията) от типове, за които са дефинирани съответно множество от функции, се нарича ***клас от/на типове*** (***type class***) или накратко ***клас***.

Например, множеството от типове, за които е дефинирана функцията за проверка на равенство (==), се означава като клас Eq.

Дефиниране на класа Eq

За да може да се дефинира един клас, е необходимо да се избере (зададе) неговото име и да се опишат ограниченията, които трябва да удовлетворява даден тип a, за да принадлежи на този клас.

Типовете, които принадлежат на даден клас, се наричат ***екземпляри*** на този клас.

Най-важно (определящо) за класа Eq е наличието на функцията == от тип a **->** a **->** Bool, която проверява дали два елемента на даден клас a, който е екземпляр на Eq, са равни:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Примерни дефиниции на функции върху екземпляри на Eq

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)

elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem x (y:ys) = (x == y) || (elem x ys)
```

Signatures („подписи”; функции, характеризиращи даден клас)

Както показахме по-горе, дефиницията на даден клас включва декларация от вида

```
class Visible a where
  toString :: a -> String
  size :: a -> Int
```

Декларацията включва името на класа (Visible) и т. нар. ***signature*** („подпис”) на класа, т.е. списък от имената и типовете на функциите, които еднозначно определят (характеризират) класа – това са функциите, които следва задължително да бъдат дефинирани за всички типове, които са екземпляри на този клас.

Следователно, дефиницята на клас има следния общ вид:

```
class Name ty where
... signature involving the type variable ty ...
```

Дефиниране на екземпляри на клас

Един тип се определя като екземпляр на даден клас, като се дефинират функциите – „подписи” на класа за елементите на този тип.

Например дефиницията

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

определя вградения тип Bool като екземпляр на класа Eq.

Примери за дефиниции на екземпляри на класа Visible:

```
instance Visible Char where
  toString ch  = [ch]
  size _       = 1

instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _         = 1
```

```
-- Дефиниция на екземпляр на клас с контекст.

instance Visible a => Visible [a] where
  toString = concat . map toString
  size     = foldr (+) 1 . map size
```

Горната дефиниция използва ***контекста*** **`Visible a`**, за да означи факта, че видими са всички списъци от обекти, които сами по себе си са видими. В десните страни на равенствата, дефиниращи функциите toString и size за типа [a], са използвани едноименните функции, които действат върху типа a.

Дефиниции по подразбиране

Нека отново се върнем към дефиницията на класа Eq. В Haskell този клас е дефиниран по следния начин:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y     = not (x==y)
  x == y     = not (x/=y)
```

Към операцията за сравнение с цел проверка на равенство се добавя и проверката за неравенство (различие). Освен това са включени и ***дефиниции по подразбиране*** на /= чрез == и на == чрез /=.

Тези дефиниции са валидни по подразбиране за всички екземпляри на класа Eq, но ако за даден тип, който е екземпляр на Eq, някоя от функциите == или /= има конкретна дефиниция, тази дефиниция има приоритет над (припокрива, overrides) дефиницията по подразбиране.

Нещо повече, за всеки екземпляр на класа Eq е необходимо поне едната от двете функции == и /= да има конкретна дефиниция (иначе двойката дефиниции по подразбиране е неизползваема, тъй като ще генерира бездънна рекурсия). Ако е дефинирана конкретно само едната от двете функции (или ==, или /=), то тази дефиниция би била достатъчна и за другата функция, тъй като дефиницията по подразбиране определя връзката между двете функции.

Производни класове

Езикът Haskell позволява да бъдат дефинирани класове, които са подкласове (производни класове, derived classes) на други класове.

Най-прост пример в това отношение е класът на наредените типове, Ord. За да бъде нареден, един тип трябва да поддържа операциите за сравнение ==, /=, >, >=, <, <=. Наличието на първите две операции означава, че наредените типове образуват подмножество на класа Eq, т.е. те образуват ***подклас*** (***производен клас***, ***derived class***) на класа Eq.

Това обстоятелство се записва формално по следния начин:

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

От друга страна може да се каже, че класът Ord ***наследява*** операциите на Eq. ***Наследяването*** (***inheritance***) е една от централните идеи на **обектно ориентираното програмиране**.

Множествени ограничения и множествено наследяване

В една от предишните лекции в курса дефинирахме функция с име iSort, която сортираше даден списък от цели числа, като за целта използваше метода за сортиране чрез вмъкване. В действителност тази функция има по-общ тип:

```
iSort :: Ord a => [a] -> [a]
```

Да предположим, че задачата ни е да сортираме даден списък и след това да визуализираме резултата като символен низ. Тогава функцията, която следва да дефинираме, ще бъде от тип

```
vSort :: (Ord a, Visible a) => [a] -> String
```

Подобни ***множествени ограничения*** могат да се появят и в дефиницията на екземпляр, например

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x,y) == (z,w)  =  x==z && y==w
```

Възможно е също множествени ограничения да бъдат включени в дефиницията на клас, например

```
class (Ord a,Visible a) => OrdVis a
```

В случаите, когато даден клас се дефинира на базата на два или повече класа, се казва, че е налице ***множествено наследяване*** (***multiple inheritance***).

Кратки сведения за вградените класове в Haskell

Haskell поддържа голям брой базови (вградени) класове, част от които ще представим накратко в настоящата лекция.

Eq – клас на типовете, за които са дефинирани операциите за проверка на равенство и неравенство

Този клас включва типове, чиито стойности могат да бъдат сравнявани за равенство и неравенство (различие), като за целта се използват следните методи:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Разгледаните досега типове Bool, Char, String, Int, Integer и Float са ***екземпляри*** на класа Eq. Такива са също и типовете, обхващащи списъци и вектори, чиито елементи са от тип, който е екземпляр на Eq.

Ще припомним, че класът Eq е дефиниран както следва:

```
class Eq a where
(==), (/=) :: a -> a -> Bool
x /= y = not (x==y)
x == y = not (x/=y)
```

Следва да отбележим изрично, че функционалните типове в общия случай не са екземпляри на класа Eq, тъй като няма подходящ механизъм за сравняване на две функции с цел установяване на равенстнво или неравенство (различие) между тях.

Ord – клас на наредените типове

Този клас включва типове, които са екземпляри на класа Eq, между елементите на които съществува (е дефинирана) линейна наредба, следователно техните елементи могат да бъдат сравнявани посредством следните методи:

```
(<)   :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>)   :: a -> a -> Bool
(>=) :: a -> a -> Bool
min  :: a -> a -> Bool
max :: a -> a -> Bool
compare :: a -> a -> Ordering
```

Типът Ordering включва стойностите LT, EQ и GT, които представят трите възможни резултата от сравняването на два елемента на даден нареден тип.

Тогава функцията compare може да бъде дефинирана например по следния начин:

```
compare x y
  | x==y       = EQ
  | x<=y       = LT
  | otherwise = GT
```

Ползата от такава функция е, че тя помага с помощта на една проверка (с едно обръщение към нея) да се определи точната релация между произволни два елемента на наредения тип, докато използването на оператори, които връщат булев резултат, би изисквало две сравнения.

Дефинициите по подразбиране на операторите за сравнение също използват стойностите от compare:

```
x <= y = compare x y /= GT
x < y  = compare x y == LT
x >= y = compare x y /= LT
x > y  = compare x y == GT
```

Дефинициите по подразбиране на функциите max и min изглеждат по следния начин:

```
max x y
   | x >= y    = x
   | otherwise = y

min x y
   | x <= y    = x
   | otherwise = y
```

Enum – клас на изброимите типове

Дефиницията на този клас изглежда по следния начин:

```
class (Ord a) => Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]                    -- [n .. ]
  enumFromThen :: a -> a -> [a]           -- [n,m .. ]
  enumFromTo :: a -> a -> [a]             -- [n .. m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n'.. m]
```

Тук са включени също подходящи дефиниции по подразбиране на функциите **`enumFromTo`** и **`enumFromThenTo`**.

“Подписът” на класа включва също функциите **`fromEnum`** и **`toEnum`**, които преобразуват стойности между съответния тип и Int.

В случая на типа Char тези функции са известни също като ord и chr:

```
ord :: Char -> Int
ord = fromEnum

chr :: Int -> Char
chr = toEnum
```

Bounded – клас на ограничените типове

Този клас се специфицира посредством декларацията

```
class Bounded a where
  minBound, maxBound :: a
```

Тук **`minBound`** и **`maxBound`** определят най-малката и най-голямата допустима стойност на елементите на съответния тип.

Show – клас на “видимите” типове
(типовете, чиито елементи могат да се преобразуват
в символни низове)

В стандартния прелюд на Haskell е дефиниран класът Show, който съдържа всички типове, чиито елементи могат да се преобразуват в символни низове (и в този смисъл могат да бъдат визуализирани, т.е. по принцип са “видими”).

Дефиницията на класа Show изглежда по следния начин:

```
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList  :: [a] -> ShowS
```

Функцията **showsPrec** е предназначена за гъвкаво и ефективно преобразуване на “дълги” стойности; като начало е достатъчна функцията
show :: a -> String ,
която реализира конвертирането (преобразуването) в символен низ.

Read – клас на типовете, чиито стойности
могат да бъдат четени от низове

Класът Read съдържа типове, чиито стойности могат да бъдат четени от символни низове. Като начало за използването на този клас е достатъчно да се познава функцията

```
read :: (Read a) => String -> a
```

Резултатът от изпълнението на тази функция може да не бъде добре дефиниран: необходимо е “входният” низ да включва точно един обект от съответния тип.

Освен това, в много случаи е от съществено значение типът на резултата от изпълнението на read да бъде точно специфициран, тъй като този резултат потенциално би могъл да бъде от различни типове.

Например може да се запише

```
(read "   1   " ) :: Int
```

с цел да се посочи явно, че резултатът трябва да бъде от тип Int.

Алгебрични типове

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:

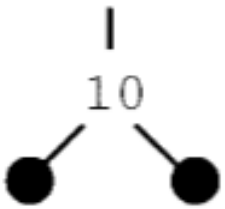
```
data NTree = NilT |
             Node Int NTree NTree
```

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

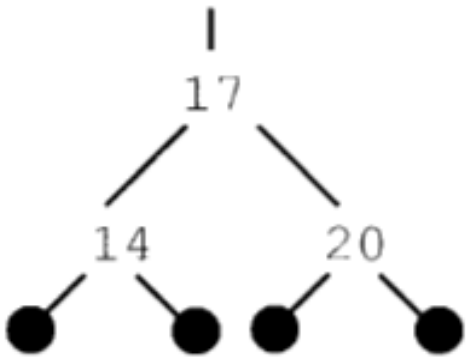
Празното дърво се представя чрез **`NilT`**, а дърветата от фиг. (а) и (б) се представят чрез

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

(а)



(б)



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
```