Лекция 11

Оценяване на изрази.
Работа с безкрайни списъци в Haskell

“Мързеливо” оценяване (lazy evaluation)

“Мързеливото” оценяване (lazy evaluation) е стратегия на оценяване, която по стандарт стои в основата на работата на всички интерпретатори на Haskell. Същността на тази стратегия е, че интерпретаторът оценява даден аргумент на дадена функция само ако (и доколкото) стойността на този аргумент е необходима за пресмятането на целия резултат. Нещо повече, ако даден аргумент е съставен (например е вектор или списък), то се оценяват само тези негови компоненти, чиито стойности са необходими от гледна точка на получаването на резултата. При това дублиращите се подизрази се оценяват по не повече от един път.

Най-важно от гледна точка на оценяването на изрази в Haskell е *прилагането на функции*. Основната идея тук е еквивалентна на т. нар. ***оценяване чрез заместване*** в "традиционните" езици за функционално програмиране.

Оценяването на израз, който е обръщение към функцията f с аргументи a_1, a_2, … , a_k, се състои в заместване на формалните параметри от дефиницията на f съответно с изразите a_1, a_2, … , a_k и оценяване на така получения частен случай на тялото на дефиницията.

Например, ако

```
f x y = x+y
```

, то

```
f (9-3) (f 34 3)
⟶ (9-3) + (f 34 3)
```

При това изразите (9-3) и (f 34 3) не се оценяват преди да бъдат предадени като аргументи на f.

Доколкото операцията събиране изисква аргументите й да бъдат оценени, оценяването на горния израз продължава по следния начин:

```
f (9-3) (f 34 3)
⟶  ...
⟶6 + (34 + 3)
⟶6 + 37
⟶43
```

В конкретния случай бяха оценени и двата аргумента (фактически параметъра), но това не винаги е така.

Ако например дефинираме

```
g x y = x + 12
```
, то

```
g (9-3) (g 34 3)
⟶(9-3) + 12
⟶6 + 12
⟶18
```

Тук x се замества с (9-3), но y не участва в дясната страна на равенството, определящо стойността на g, следователно аргументът (g 34 3) не се оценява.

Така демонстрирахме едно от преимуществата на "мързеливото" оценяване: ***аргументи, които не са необходими, не се оценяват***.

Последният пример не е особено смислен, тъй като вторият аргумент не се използва в никакъв случай и по същество е излишен в дефиницията на g.

По-интересен е следният пример:

```
switch :: Int -> a -> a -> a
switch n x y
   | n>0       = x
   | otherwise = y
```

Ако цялото число n е положително, резултатът съвпада с оценката на x; в противен случай тя съвпада с оценката на y. С други думи, винаги при оценяване на обръщение към функцията switch се оценяват аргументът n (т.е. първият аргумент) и точно един от останалите аргументи.

Нека сега функцията h е дефинирана както следва:

```
h :: Int -> Int -> Int
h x y = x + x
```

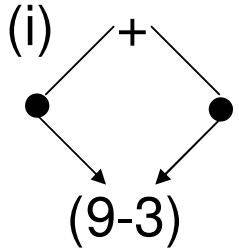
Тогава

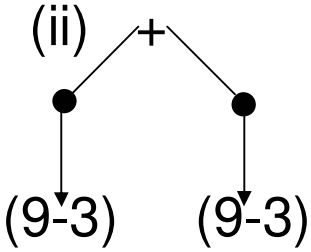
```
h (9-3) (h 34 3)
⟶ (9-3) + (9-3)
⟶ 6 + 6
⟶ 12
```

Изглежда, че в последния пример аргументът (9-3) се оценява двукратно, но принципите на “мързеливото” оценяване гарантират, че ***дублираните аргументи (т.е. многократните включвания на аргументи) се оценяват по не повече от един път***.

В реализацията на Hugs това се постига, като изразите се представят чрез подходящи графи и пресмятанията се извършват върху тези графи. В такъв граф всеки аргумент се представя чрез единствен възел и към този възел може да сочат много дъги.

Например изразът, до който се свежда оценяването на h (9-3) (h 34 3), се представя чрез (i), а не чрез (ii).





Като следващ пример ще проследим оценяването на обръщение към функцията
pm (x,y) = x+1 :

```
pm (3+2,4-17)
⟶ (3+2) + 1
⟶ 6
```

Тук се оценява само част от (първият елемент на) двойката, която е аргумент на обръщението към функцията pm. С други думи, аргументът се използва, но се оценява само реално необходимата негова част.

Правила за оценяване и “мързеливо” оценяване

Дефиницията на една функция се състои от поредица от условни равенства. Всяко условно равенство може да съдържа множество клаузи и може да включва в специална where клауза произволен брой локални дефиниции. Всяко равенство описва в лявата си страна различни случаи на действието на функцията, която е предмет на дефиницията, т.е. описва резултата, който се връща при прилагане на функцията към различни образци.

Общ вид на дефиниция на функция

```
f p1 p2 ... pk
   | g1          = e1
   | g2          = e2
     ...
   | otherwise = em
   where
v1 a1,1     = r1
  ...
f q1 q2 ... qk
   = ...
.....
```

Оценяването на f a_1 a_2 … a_k има три основни аспекта.

1. Съпоставане по образец

Аргументите се оценяват с цел да се установи кое от условните равенства е приложимо. При това оценяването на аргументите не се извършва изцяло, а само до степен, която е достатъчна, за да се прецени дали те са съпоставими със съответните образци.

Ако аргументите са съпоставими с образците p_1, p_2, … , p_k, то оценяването продължава с използване на първото равенство; в противен случай се прави проверка за съпоставимост на аргументите с образците от второто равенство и тази проверка от своя страна може да предизвика по-нататъшно оценяване на аргументите.

Този процес продължава, докато се намери множество от съпоставими с аргументите образци или докато се изчерпат условните равенства от дефиницията (тогава се получава **`Program error`**).

Нека например е дадена дефиницията:

```
f :: [Int] -> [Int] -> Int
f []     ys     = 0                    (f.1)
f (x:xs) []     = 0                    (f.2)
f (x:xs) (y:ys) = x+y                  (f.3)
```

Тогава оценяването на израза f [1..3] [1..3] се извършва по следния начин:

```
f [1..3] [1..3]                                    (1)
⟶ f (1:[2..3]) [1..3]                              (2)
⟶ f (1:[2..3]) (1:[2..3])                          (3)
⟶ 1 + 1                                            (4)
⟶ 2
```

На стъпка (1) няма достатъчно информация, за да може да се прецени дали има съпоставимост с (f.1). Следващата стъпка на оценяване води до (2) и показва, че няма съпоставимост с (f.1).

Първият аргумент от (2) е съпоставим с първия образец от (f.2), следователно трябва да се направи проверка за втория аргумент от (2) и втория образец от (f.2). Следващата стъпка на оценяването, извършена в (3), показва, че вторият аргумент не е съпоставим с втория образец от (f.2). На тази стъпка се вижда също, че е налице съпоставимост на аргументите с (f.3), следователно е налице (4).

2. Условия (guards)

Нека за определеност предположим, че първото условно равенство от дефиницията е съпоставимо с обръщението към f, което трябва да се оцени. Тогава образците p_1, p_2, … , p_k в условното равенство се заместват с изразите a_1, a_2, … , a_k. След това трябва да се определи коя от клаузите в дясната страна е приложима. За целта условията се оценяват последователно, докато се намери първото условие със стойност True; като резултат се връща стойността на съответната на това условие клауза.

Нека например е дадена дефиницията:

```
f :: Int -> Int -> Int -> Int
f m n p
    | m>=n && m>=p = m
    | n>=m && n>=p = n
    | otherwise    = p
```

Тогава

```
f (2+3) (4-1) (3+9)
 ?? (2+3)>=(4-1) && (2+3)>=(3+9)
 ?? ⟶ 5>=3 && 5>=(3+9)
 ?? ⟶ True && 5>=(3+9)
 ?? ⟶ 5>=(3+9)
 ?? ⟶ 5>=12
 ?? ⟶ False
```

```
?? 3>=5 && 3>=12
?? ⟶ False && 3>=12
?? ⟶ False
?? otherwise   True
⟶ 12
```

3. Локални дефиниции

Стойностите в клаузите where се пресмятат при необходимост (“при поискване”): пресмятането на дадена стойност започва едва когато се окаже, че тази стойност е необходима.

Ако са дадени дефинициите:

```
f :: Int -> Int -> Int
f m n
   | notNil xs = front xs
   | otherwise = n
   where
   xs = [m .. n]
```

```
front (x:y:zs) = x+y
front [x]      = x

notNil []     = False
notNil (_:_) = True
```

Тогава процесът на оценяване на f 3 5 изглежда по следния начин:

```
f 3 5
 ?? notNil xs
 ?? | where
 ?? | xs = [3 .. 5]
 ?? |   ⟶ 3:[4 .. 5]                                  (1)
 ?? ⟶ notNil (3:[4 .. 5])
 ?? ⟶ True
```

```
→ front xs
    where
    xs = 3:[4 .. 5]
     → 3:4:[5]                                    (2)
→ 3+4                                             (3)
→ 7
```

За да може да се оцени условието notNil xs, започва оценяване на xs и след една стъпка (1) показва, че това условие има стойност True.

Оценяването на front xs изисква повече информация за xs, затова оценяването на xs се извършва на (продължава с) още една стъпка и така се получава (2). Успешното съпоставяне по образец в дефиницията на front в този случай дава (3) и така се получава окончателният резултат.

Ред на оценяването

Това, което характеризира оценяването в Haskell, освен обстоятелството, че аргументите се оценяват по не повече от един път, е ***редът, в който се прилагат функции***, когато има възможност за избор.

- Оценяването се извършва в посока от външните към вътрешните изрази.

В ситуации от типа на

$$f_1\ e_1\ \underline{(f_2\ e_2\ 10)}\ ,$$

където едно прилагане на функция включва друго, външното обръщение $f_1\ e_1\ (f_2\ e_2\ 10)$ се избира за оценяване.

- В останалите случаи оценяването се извършва в посока от ляво на дясно.

В израза

$$\underline{f_1\ e_1} + \underline{f_2\ e_2}$$

трябва да бъдат оценени и двата подчертани израза. При това най-напред се оценява левият израз $f_1\ e_1$.

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 ==.