Дефиниране на функции
за работа със списъци

Съпоставяне с образец (pattern matching)

Както вече показахме, дефинициите на функции могат да имат вида на условни равенства като например

```
mystery :: Int -> Int -> Int
mystery x y
   | x == 0    = y
   | otherwise = x
```

Същата дефиниция може да се напише и с използване на поредица от две равенства:

```
mystery 0 y = y
mystery x y = x
```

В тези две равенства последователно са описани отделните възможни случаи за стойността на mystery, като за целта са използвани ***образци*** – в случая литералът 0 и променливите x и y. Равенствата се прилагат **последователно**, като до всяко следващо равенство се достига и то евентуално се прилага само ако всички предишни не са дали резултат (т.е. ако съпоставянето им е завършило с неуспех).

Във второто равенство от дефиницията на mystery променливата y не се използва, затова тя може да бъде заменена от образеца, известен като ***wildcard*** (***специален символ за безусловно съпоставяне***):

```
mystery 0 y = y
mystery x _ = x
```

При описанието на различните възможни случаи за стойността на дадена функция може да се използват и образци, с чиято помощ се именуват елементите на вектори.

Например:

```
joinStrings :: (String,String) -> String
joinStrings (str1,str2) = str1 ++ str2
```

Образци (обобщение)

Образецът (pattern) може да бъде:

- ***Литерал*** като например 24, ‘f’ или True; даден аргумент се съпоставя успешно с такъв образец, ако е равен на неговата стойност;
- ***Променлива*** като например x или longVariableName; образец от този вид се съпоставя успешно с аргумент с произволна стойност;
- ***Специален символ за безусловно съпоставяне*** (***wildcard***) ‘_’, който е съпоставим с произволен аргумент;
- ***Вектор – образец*** $(p_1,p_2, \dots ,p_n)$; за да бъде съпоставим с него, аргументът трябва да има вида $(v_1,v_2, \dots ,v_n)$, като всяко v_i трябва да бъде съпоставимо със съответното p_i;

- ***Конструктор***, приложим към даденото множество от аргументи.

Забележка. Някои видове образци от тип конструктор ще бъдат разгледани по-нататък.

Списъци и образци на списъци

Всеки списък или е празен (т.е. е равен на/съпоставим с **[ ]**), или е непразен. Във втория случай списъкът има глава и опашка, т.е. може да бъде записан във вида ***x:xs***, където ***x*** е първият елемент на този списък, а ***xs*** е списъкът без първия му елемент (опашката на този списък).

Освен това, всеки списък може да бъде конструиран от празния чрез многократно прилагане на ‘:’ за добавяне на елементи и тази идея съответства в максимална степен на вътрешното представяне на списъците в Haskell.

Примери

[ ]   3:[ ] = [3]   2:[3] = [2,3]   4:[2,3] = [4,2,3]

При това последният списък може да бъде записан също като 4:2:3:[ ]

Забележка. Операторът ‘:’ е ***дясно асоциативен***, т.е. за всички стойности на x, y и zs е вярно

x:y:zs = x:(y:zs)

Конструкцията 4:2:3:[ ] определя *единствения начин*, по който списъкът [4,2,3] може да се конструира с използване на [ ] и ‘:’.

Следователно, операторът **`‘:’`** от тип **`a -> [a] -> [a]`** играе изключително важна роля при работата със списъци – той е ***конструктор на списъци***, тъй като всеки списък може да бъде конструиран по единствен начин с използване на [ ] и ‘:’.

По исторически причини (свързани с езика Lisp) този конструктор понякога се нарича **cons**.

Дефиниции на някои функции за работа със списъци с използване на образци

```
head        :: [a] -> a
head (x:_)   = x

tail        :: [a] -> [a]
tail (_:xs) = xs

null        :: [a] -> Bool
null []      = True
null (_:_)   = False
```

Сега вече можем да обясним как изглеждат и каква е ролята на образците от тип конструктор (по-точно, образците – конструктори на списъци).

Образецът – конструктор на списъци може да бъде [ ] или да има вида (p:ps), където p и ps също са образци.

Правилата за съпоставяне с такъв образец могат да бъдат формулирани по следния начин:

- даден списък е съпоставим с образеца [ ] точно когато е празен;
- даден списък е съпоставим с образец от вида (p:ps), когато не е празен, главата му се съпоставя успешно с образеца p и опашката му се съпоставя успешно с образеца ps.

В частност всеки непразен списък е съпоставим с образец от вида (x:xs).

Забележка. Образец, който включва конструктор от вида ‘:’, като правило се загражда в кръгли скоби, тъй като прилагането на функция има по-висок приоритет от останалите операции.

Конструкцията case

Досега показахме как може да се описват различни случаи на съпоставяне на аргументите на дадена функция с определени образци в (чрез) поредица от равенства.

За целта може да се използва и конструкцията case.

Пример. Ще дефинираме функция, която връща като резултат първата цифра от даден низ.

```
firstDigit :: String -> Char
firstDigit str
   = case (digits str) of
      []     -> '\0'
      (x:_) -> x
```

Забележки

1. ***digits*** е функцията (дефинирана в предишна лекция), която намира всички цифри в даден низ:

```
digits :: String -> String
digits st = [ch | ch <- st, isDigit ch]
```

2. Специалният символ от вида ‘\n’ означава знака с ASCII код n.

В горния пример изразът case е използван с цел да се разграничат двата основни случая – тези на празен и непразен списък, както и за да се извлекат (получат) части от търсената стойност чрез асоцииране (свързване) на променливи, участващи в образците, със съпоставените им стойности.

В общия случай изразът от тип ***case*** има следния вид:

```
case e of
   p₁ -> e₁
   p₂ -> e₂
     ...
   pₖ -> eₖ
```

Тук ***e*** е израз, който се съпоставя последователно с образците p_1, p_2, …, p_k. Ако p_i е първият образец от поредицата, който е съпоставим с ***e***, резултатът е e_i. При това променливите, които участват в p_i, се свързват със съответните им части от ***e***.

Примитивна рекурсия върху списъци

Нека предположим, че имаме за задача да намерим сумата на елементите на даден списък от цели числа. Една възможна дефиниция на такава функция изглежда по следния начин:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Тази дефиниция е пример за реализация на т. нар. **примитивна рекурсия върху списъци**. В дефиниция, която реализира този тип рекурсия, се описват следните типове случаи:

- начален (прост, базов) случай: в горния пример тук се описва стойността на sum за [ ];
- общ случай, при който се посочва връзката между стойността на функцията за дадена стойност на аргумента (в случая sum (x:xs)) и стойността на функцията за по-проста в определен смисъл стойност на аргумента (sum xs).

Оценяване на обръщението към sum [3,2,7,5]:

```
sum [3,2,7,5]
⟶ 3 + sum [2,7,5]
⟶ 3 + (2 + sum [7,5])
⟶ 3 + (2 + (7 + sum [5]))
⟶ 3 + (2 + (7 + (5 + sum [])))
⟶ 3 + (2 + (7 + (5 + 0)))
⟶ 17
```

Други дефиниции на функции
чрез примитивна рекурсия върху списъци

Пример 1. Примерна дефиниция (с учебна цел) на функцията concat, която е дефинирана в Prelude.hs:
concat $[e_1, e_2, \ldots, e_n] = e_1 ++ e_2 ++ \ldots ++ e_n$

Дефиниция

```
concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

Пример 2. Примерна учебна дефиниция на функцията ++ (двуаргументен аналог на append в Racket).

```
(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs++ys)
```

Пример 3. Функция, която проверява дали дадено цяло число (число от тип Int) съвпада с някой от елементите на даден списък от цели числа.

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

Важна забележка. Изглежда, че горната дефиниция би могла да се запише и по следния начин:

```
elem x []       = False
elem x (x:ys) = True
elem x (y:ys) = elem x ys
```

Тук проверката за равенство се извършва чрез повторно (многократно) използване на променливата **x** – както това обикновено се прави при програмирането в логически стил на езика Prolog – но за съжаление **многократната употреба на една и съща променлива в образци в лявата страна на едно равенство от дефиницията на функция не е разрешена при програмирането на Haskell**.

Пример 4. Ще дефинираме функция, която удвоява елементите на даден списък от цели числа.

Първо решение

```
doubleAll :: [Int] -> [Int]
doubleAll xs = [2*x | x <- xs]
```

Второ решение

```
doubleAll :: [Int] -> [Int]
doubleAll []     = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

Пример 5. Ще дефинираме функция, която извлича четните числа измежду елементите на даден списък от цели числа (филтрира по четност елементите на даден списък от цели числа).

Първо решение

```
selectEven :: [Int] -> [Int]
selectEven xs = [x | x <- xs, isEven x]
```

Второ решение

```
selectEven :: [Int] -> [Int]
selectEven [] = []
selectEven (x:xs)
   | isEven x  = x : selectEven xs
   | otherwise =     selectEven xs
```

Пример 6. Сортиране във възходящ ред на даден списък от числа чрез вмъкване.

Дефиниция

```
iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)

ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:ys)
   | x <= y    = x:(y:ys)
   | otherwise = y : ins x ys
```

Оценяване на примерно обръщение към функцията iSort

```
iSort [3,9,2]
⟶ ins 3 (iSort [9,2])
⟶ ins 3 (ins 9 (iSort [2]))
⟶ ins 3 (ins 9 (ins 2 (iSort [])))
⟶ ins 3 (ins 9 (ins 2 []))
⟶ ins 3 (ins 9 [2])
⟶ ins 3 (2 : ins 9 [])
⟶ ins 3 [2,9]
⟶ 2 : ins 3 [9]
⟶ 2 : [3,9]
⟶ [2,3,9]
```

Обща рекурсия върху списъци

Тук ще покажем някои примери на дефиниции на функции за работа със списъци, които не се подчиняват на ограниченията на примитивната рекурсия. При тях схемата на дефиниране е специфична и се подчинява на задачата да се даде отговор на въпроса:

Когато се дефинира **`f (x:xs)`**, кои стойности на **`f ys`** биха помогнали за коректното описание на резултата?

Пример 1. Ще дефинираме функцията zip, която трансформира два списъка в списък от двойки от съответните елементи на двата списъка, например

zip [1,3] [2,4]   = [(1,2),(3,4)]
zip [1,2] [‘a’,’b’,’c’] = [(1,’a’),(2,’b’)]

Ще опишем дефиницията на тази функция с рекурсия относно двата аргумента.

Първо решение

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

Второ решение

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip (x:xs) []     = []
zip []     zs     = []
```

Оценяване на примерно обръщение към функцията zip

```
zip [1,5] [‘c’,‘d’,‘e’]
⟶ (1,‘c’) : zip [5] [‘d’,‘e’]
⟶ (1,‘c’) : (5,‘d’) : zip [] [‘e’]
⟶ (1,‘c’) : (5,‘d’) : []
⟶ (1,‘c’) : [(5,‘d’)]
⟶ [(1,‘c’),(5,‘d’)]
```

Пример 2. Функцията take ще връща като резултат списък от първите няколко елемента на даден списък, например

```
take  5 “Hot Rats” = “Hot R”
take 15 “Hot Rats” = “Hot Rats”
```

И тук в дефиницията ще организираме рекурсия по отношение на двата аргумента.

Дефиниция

```
take :: Int -> [a] -> [a]
take 0 _  = []
take _ [] = []
take n (x:xs)
   | n>0 = x : take (n-1) xs
take _ _ = error “take: negative argument”
```

Пример 3. „Бърза“ сортировка. Алгоритъмът за бързо сортиране генерира две рекурсивни обръщения към себе си с аргументи – части от списъка, който е зададен като оригинален аргумент.

Дефиниция

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs)
   = qSort [ y | y <- xs, y<=x] ++ [x]
     ++ qSort [ y | y <- xs, y>x]
```

Изследване на свойствата на програми на Haskell

Едно от най-големите предимства на програмирането във функционален стил е възможността строго да се доказват свойства на функционалните програми.

Тук ще покажем един прост подход за доказване на това, че определено множество от функции е дефинирано така, че дадено свойство, във формулировката на което участват тези функции, е вярно за всички (крайни) списъци.

Принцип на структурната индукция при работата със списъци

В случай, че трябва да се докаже, че дадено свойство P(xs) е вярно за всички крайни списъци xs, доказателството може да се извърши на две стъпки:

- ***Базов случай.*** Доказване, че е вярно P([ ]).
- ***Индуктивна стъпка.*** Доказване, че е вярно P(x:xs) при предположение, че е вярно P(xs).

Твърдението P(xs) във втората стъпка се нарича ***индуктивна хипотеза,*** тъй като за него се предполага, че е вярно в процеса на доказателството на P(x:xs).

Пример 1. Нека sum и doubleAll са функции, дефинирани както следва:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

doubleAll :: [Int] -> [Int]
doubleAll []     = []
doubleAll (z:zs) = 2*z : doubleAll zs
```

Ще покажем, че за всеки списък xs е в сила
sum (doubleAll xs) = 2*sum xs .

Доказателството ще извършим на две стъпки:

1. Базов случай: трябва да покажем, че
   sum (doubleAll [ ]) = 2*sum [ ] .
2. Индуктивна стъпка: ще покажем, че е в сила
   sum (doubleAll (x:xs)) = 2*sum (x:xs),
   използвайки за целта индуктивната хипотеза
   sum (doubleAll xs) = 2*sum xs.

Базов случай

sum (doubleAll [ ]) = sum [ ] = 0;
2 * sum [ ] = 2*0 = 0.

Следователно, sum (doubleAll [ ]) = 2 * sum [ ] .

Индуктивна стъпка

sum (doubleAll (x:xs)) = sum (2*x : doubleAll xs)
= 2*x + sum (doubleAll xs);
2 * sum (x:xs) = 2 * (x + sum xs) = 2*x + 2*sum xs.

Според индуктивното предположение е в сила
sum (doubleAll xs) = 2*sum xs, следователно

sum (doubleAll (x:xs)) = 2*sum (x:xs) .

Пример 2. Връзка между length и ++.

Нека length и ++ са функциите за намиране на дължината на списък и конкатенация на списъци, дефинирани както следва:

```
length :: [a] -> Int
length []     = 0
length (z:zs) = 1 + length zs

(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Ще покажем, че за всеки два списъка xs и ys е в сила
length (xs ++ ys) = length xs + length ys .

Доказателство

Базов случай

Ще покажем, че length ([ ] ++ ys) = length [ ] + length ys .

length ([ ] ++ ys) = length ys;

length [ ] + length ys = 0 + length ys = length ys.

Следователно length ([ ] ++ ys) = length [ ] + length ys .

Индуктивна стъпка

Ще покажем, че е вярно

length ((x:xs) ++ ys) = length (x:xs) + length ys

при условие, че съгласно индуктивното предположение е вярно

length (xs ++ ys) = length xs + length ys.

length ((x:xs) ++ ys) = length (x:(xs ++ ys))
= 1 + length (xs ++ ys) = 1 + length xs + length ys;

length (x:xs) + length ys = 1 + length xs + length ys.

Следователно
length ((x:xs) ++ ys) = length (x:xs) + length ys .

Пример 3. Връзка между reverse и ++.

Нека reverse е функцията, която обръща реда на елементите на даден списък, дефинирана по следния начин:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (z:zs) = reverse zs ++ [z]
```

Ще покажем, че
reverse (xs ++ ys) = reverse ys ++ reverse xs .

Доказателство

Базов случай

reverse ([ ] ++ ys) = reverse ys;

reverse ys ++ reverse [ ] = reverse ys ++ [ ] = reverse ys.

Следователно

reverse ([ ] ++ ys) = reverse ys ++ reverse [ ].

Забележка. Горното доказателство ще е коректно, ако докажем, че добавянето чрез конкатенация (++) на празен списък към даден друг списък е операция – идентитет, т.е.

xs ++ [ ] = xs за всеки списък xs.

Доказателство на това твърдение ще покажем по-нататък в изложението на тази тема.

Индуктивна стъпка

reverse ((x:xs) ++ ys) = reverse (x:(xs ++ ys))
= reverse (xs ++ ys) ++ [x].

Според индуктивното предположение е вярно
reverse (xs ++ ys) = reverse ys ++ reverse xs.

Следователно
reverse ((x:xs) ++ ys) = (reverse ys ++ reverse xs) ++ [x].

От друга страна
reverse ys ++ reverse (x:xs) = reverse ys ++ (reverse xs ++ [x]).

Тъй като операцията конкатенация на списъци (++) е асоциативна, т.е. за всеки три списъка xs, ys и zs е изпълнено
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs, то следователно
reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs).

Забележки

1. Асоциативността на операцията ‘++’ подлежи на строго доказателство (такова е направено в края на изложението на тази тема).
2. Принципът на структурната индукция е подходящ за доказване на свойства на функции, дефинирани с помощта на примитивна рекурсия.

Доказателство на твърдението, че добавянето чрез конкатенация (++) на празен списък към даден друг списък е операция – идентитет, т.е. xs ++ [ ] = xs за всеки списък xs

Базов случай

По дефиниция [ ] ++ [ ] = [ ].

Индуктивна стъпка

По дефиниция (x:xs) ++ [ ] = x:(xs ++ [ ]). Според индуктивното предположение xs ++ [ ] = xs, откъдето следва, че
(x:xs) ++ [ ] = x:(xs ++ [ ]) = x:xs.

Доказателство на твърдението, че операцията конкатенация на списъци (++) е асоциативна, т.е. за всеки три списъка xs, ys и zs е изпълнено xs ++ (ys ++ zs) = (xs ++ ys) ++ zs

Базов случай

[ ] ++ (ys ++ zs) = ys ++ zs;
([ ] ++ ys) ++ zs = ys ++ zs.

Следователно
[ ] ++ (ys ++ zs) = ([ ] ++ ys) ++ zs.

Индуктивна стъпка

По дефиниция (x:xs) ++ (ys ++ zs) = x:(xs ++ (ys ++ zs)).

Според индуктивното предположение е вярно xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.

Следователно (x:xs) ++ (ys ++ zs) = x:((xs ++ ys) ++ zs) = (x:(xs ++ ys)) ++ zs = ((x:xs) ++ ys) ++ zs.