Лекция 6. Рекурсивные функции в Haskell

Пережогин А.С.

8 октября 2012 г.

Рекурсивные функции. Введение

Многие функции в Haskell определяются в терминах других функций. В примере функция вычисления факториала представляется через функцию вычисления произведения чисел из списка от 1 до n.

```
factorial :: Int -> Int
factorial n = product [1..n]
```

Выражение вычисляется по шагам

```
  factorial 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

Рекурсивные функции

В Haskell функции могут быть определены в терминах самих себя. Такие функции называются *рекурсивные*.

```
factorial 0 = 1
factorial (n+1) = (n+1)*factorial n
```

```
  factorial 3
=
  3 * factorial 2
=
  3 * (2 * factorial 1)
=
  3 * (2 * (1 *(factorial 0)))
=
  3 * (2 * (1 *1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Рекурсивные функции

Замечания:

- `factorial 0 = 1`

  является базой рекурсии, так как 1 – нейтральный элемент для умножения
- рекурсивное определение функции факториала при отрицательных целых числах дает расходимость и переполнение стека

Удобство использования рекурсии

- некоторые функции более компактно объявляются в терминах рекурсивных функций
- свойства рекурсивных функций доказываются с помощью метода математической индукции

Рекурсия списков

Рекурсия используется не только для чисел, но так же применяется для списков

```
product        :: [Int] -> Int
product []      = 1
product (n:ns) = n * product ns
```

Функция product подставляет вместо пустого списка единицу и производит перемножение головы на произведение элементов хвоста.

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * ( 3 * product [4])
=
  2 * ( 3 * ( 4 * product []))
=
  2 * ( 3 * ( 4 * 1))
=
  24
```

Рекурсия списков

Рекурсия используется не только для чисел, но так же применяется для списков

```
product        :: [Int] -> Int
product []      = 1
product (n:ns) = n * product ns
```

Функция product подставляет вместо пустого списка единицу и производит перемножение головы на произведение элементов хвоста.

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * ( 3 * product [4])
=
  2 * ( 3 * ( 4 * product []))
=
  2 * ( 3 * ( 4 * 1))
=
  24
```

Рекурсия списков

Используя подобную схему рекурсии, можно определить функцию длины списка

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

Пример. Функция обращения порядка в списке

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

Рекурсия в случае нескольких аргументов

Функции нескольких аргументов могут использовать рекурсивное объявление.

Пример. Функция zipping

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

Пример. Функция drop

```
drop             :: Int -> [a] -> [a]
drop 0     xs     = xs
drop (n+1) []     = []
drop (n+1) (_:xs) = drop n xs
```

Пример. Функция $(++)$ соединения двух списков

```
drop             :: Int -> [a] -> [a]
drop 0     xs     = xs
drop (n+1) []     = []
drop (n+1) (_:xs) = drop n xs
```

Алгоритм быстрой сортировки

Алгоритм быстрой сортировки списка целых чисел может быть выполнен с помощью двух правил:

- Пустой список – отсортирован
- Список с элементами: сортируем элементы хвоста, которые меньше головы, и сортируем элементы хвоста, которые больше головы. Добавляем результирующие списки с одной и другой стороны от головы.

```
qsort        :: [Int] -> [Int]
qsort []      = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a <- xs, a <= x]
      larger  = [b | b <- xs, b > x]
```

Контрольные задания

Определить функции

- Логическое and

  ```
  and :: [Bool] -> Bool
  ```

- Логическое or

  ```
  or :: [Bool] -> Bool
  ```

- Выбор n-элемента из списка

  ```
  (!!) :: [a] -> Int -> Bool
  ```

- Функция принадлежности элемента списку

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

- Рекурсивная функция

  ```
  msort :: [Int] -> [Int]
  ```

  Список сортирован, если длина $<= 1$.
  Остальные списки сортируются как 2 части и объединяются в результирующий список.