Ленивые вычисления

Для начала рассмотрим эффекты, связанные с ленивостью вычислений в Haskell. Вспомним функцию вычисления факториала, которую мы записывали на одном из первых занятий:

```
fact' :: Integer -> Integer -> Integer
fact' 0 f = f
fact' n f = fact' (n-1) (n*f)
```

Хотя она и реализована с использованием хвостовой рекурсии, ее использование все равно требует линейного количества памяти в связи с ленивостью вычислений. Рассмотрим пример вызова:

```
fact 5
fact' 5 1
```

Здесь просто происходит вызов функции с подстановкой аргумента. Однако, стоит обратить внимание на то, что выражения внутри функции не вычисляются пока не станут необходимы. Таким образом мы получаем запись:

```
fact' (5-1) (5*1)
```

Теперь происходит вызов очередной функции, при этом вычисляется только выражение `(5-1)`, поскольку его результат должен быть сопоставлен с `0`, а `(5*1)` передается в функцию в неизменном виде:

```
fact' (4-1) (4*(5*1))
```

Далее процесс идет аналогично:

```
fact' (3-1) (3*(4*(5*1)))
fact' (2-1) (2*(3*(4*(5*1))))
fact' (1-1) (1*(2*(3*(4*(5*1)))))
1*(2*(3*(4*(5*1))))
```

Как видно из-за ленивости вычислений, функция fact действительно требует линейного количества памяти.

Операторы $ и $!

Как известно, операция применения функции в аргументу имеет в Haskell наивысший приоритет. Из-за этого длинные функциональные вызовы необходимо записывать с большим числом скобок:

```
head (filter (>0) (map (*2) [1..10]))
```

Работа с такими вызовами может быть несколько упрощена за счет использования оператора `$`. Этот оператора определен в стандартной библиотеке следующим образом:

```
a $ b = a b
```

То есть он является просто другим способом записи функционального вызова. Основное отличие этого способа состоит в том, что оператор `$` имеет наинизший приоритет, поэтому можно писать конструкции вида:

```
head $ filter (>0) $ map (*2) [1..10]
```

Здесь уже нет необходимости в скобках и запись выглядит чище.

Но значительно полезней другой похожий оператор - $!. Он так же осуществляет функциональный вызов, но при этом аргумент (перед которым стоит этот оператор?) обязательно вычисляется до передачи в функцию. За счет этого можно обойти проблемы, связанные с ленивым вычислением в факториале и записать следующую версию:

```
fact1b :: Integer -> Integer -> Integer
fact1b 0 f = f
fact1b n f = fact1b (n-1) $! (n*f)
```

Можно проверить, что результат его вычисления действительно требует константной памяти.

Бесконечные списки

Теперь перейдем к работе с бесконечными списками. Предположим, что нам необходимо описать бесконечный список нулей. Как это может быть сделано? Первым элементом этого списка будет 0, а хвост будет так же представлять из себя бесконечный список нулей, потому можно записать определение следующим образом:

```
zeros = 0:zeros
```

Чтобы проверить, что мы действительно получили то, что нужно рассмотрим первые 5 элементов списка. Для этого применим к списку функцию `take 5`. При этом будем использовать следующее ее определение:

```
take 0 _ = []
take n (h:t) = h:take (n-1) t
```

Применение `take 5` к списку:

```
take 5 zeros
take 5 (0:zeros)
0:take 4 zeros
0:take 4 (0:zeros)
0:0:take 3 zeros
…
0:0:0:0:0take 0 zeros
[0,0,0,0,0]
```

Как видно из процесса построения точно такой же результат будет получен для любого числа элементов, а значит список действительно бесконечный. Теперь рассмотрим другой бесконечный список — степеней двойки.

Его можно получить как результат вычисления функции powersOf2 с аргументом 1, определенной следующим образом:

powersOf2 x = x:powersOf2 (x*2)

То, что действительно получается нужный список, можно проверить аналогично предыдущему

случаю - взяв несколько первых элементов списка.

Функции для конструирования бесконечных списков

Теперь рассмотрим основные функции для конструирования бесконечных списков.

- Функция `repeat` создает список, состоящий из бесконечного числа повторяющихся элементов, например:

```
zeros1 = repeat 0 – [0,0,0,0,0...]
```

- Функция `cycle` создает список, состоящий из бесконечного числа повторений заданного списка:

```
l1 = cycle [1,2,3] – [1,2,3,1,2,3,1,2,3...]
```

- Функция `iterate` создает список, каждый элемент которого получается применением некоторой функции к предыдущему элементу. Примерами таких списков являются арифметическая и геометрическая прогрессии:

```
arithSeq = iterate (+5) 0
geomSeq = iterate (*3) 1
```

Теперь рассмотрим еще несколько примеров бесконечных списков.

Бесконечный список чисел Фиббоначи получается как результат применения функции fibSeq:

```
fibSeq :: Integer -> Integer -> [Integer]
fibSeq m n = m : (fibSeq n (m+n))


fib = fibSeq 1 1
```

Бесконечный список простых чисел может быть получен путем фильтрации списка всех целых чисел, начинающегося с двойки.

Для осуществления такой фильтрации мы будем использовать функцию `isPrime`, определенную следующим образом:

```
isPrime x = noDivisors [2..x-1] x
```

В ней мы обращаемся к функции `noDivisors` куда передаем список всех чисел от `2` до `(x-1)` — тех, которые могут быть делителями числа `x`.

Функция `noDivisors` определяется следующим образом:

```
noDivisors l x = all (\n -> x `mod` n /= 0) l
```

И сам список простых чисел:

```
primes = filter isPrime [2..]
```

Теперь рассмотрим несколько возможных оптимизаций этого метода.

Во-первых, нет смысла перебирать четные числа (кроме 2). А значит можно организовать перебор в два раза быстрее заменив определение списка на:

```
primes = 2:filter isPrime [3,5..]
```

Далее, известно, что все простые числа кроме 2 и 3 могут быть записаны как `6k±1`. Тогда, мы можем ускорить процесс перебора еще в полтора раза за счет перебора только таких чисел:

```
primes = 2:3:filter isPrime [6*k+z | k <- [1..], z ← [-1,1] ]
```

Еще ускорение в два раза может получится, если учесть, что делителями числа могут быть только числа меньше либо равные его половины:

```
isPrime x = noDivisors [2..x `div` 2] x
```

И даже более того, если есть хотя бы один делитель больший корня квадратного из числа, то есть соответствующий меньше корня. А значит можно перебирать числа до квадратного корня из числа:

```
isPrime x = noDivisors (takeWhile (\n -> n*n <= x) [2..]) x
```

Это даст самое значительное ускорение, особенно для больших чисел.

Корекурсия

Мы рассмотрели три основных способа построения бесконечных списков: определение через себя (`zeros`), как результат вычисления некоторой функции (которая реализует бесконечную рекурсию) и как результат фильтрации других бесконечных списков.

Первый из этих способов носит достаточно фундаментальный характер и называется корекурсией. Корекурсия представляет собой операцию, двойственную к рекурсии (в рекурсии мы сводим вычисление большей задачи к меньшим, а в корекурсии мы строим бесконечную структуру данных путем разворачивания от меньшей к большей).

Хорошим примером корекурсии является следующее определение бесконечного списка чисел Фиббоначи.

Нам надо определить список чисел Фиббоначи через себя. Как это сделать? Начнем с того, что первые два элемента списка фиксированы и равны 1. Вопрос в том, как получается хвост. Рассмотрим его по элементам: третий элемент получается как сумма первого и второго, четвертый — как сумма второго и третьего и так далее. То есть получается, что хвост списка чисел Фиббоначи получается как поэлементная сумма двух списков — списка чисел Фиббоначи и его же, но без первого элемента. Так и запишем это с использованием функции `zipWith`:

```
fib = 1:1:zipWith (+) fib (tail fib)
```

Теперь, чтобы проверить, что мы получили действительно нужный список попробуем применить к нему функцию `take 5`. При этом воспользуемся следующими определениями:

```
take 0 _ = []
take n (h:t) = h:take (n-1) t
zipWith f (h1:t1) (h2:t2) = f h1 h2:zipWith f t1 t2
```

Здесь мы не рассматриваем другие строчки определения функции `zipWith`, поскольку работа идет с бесконечными списками и они нам не пригодятся. В процессе у нас неоднократно возникнет вычисление хвоста `fib`. Будем использовать обозначение `fib_1` для `fib` без первого элемента, `fib_3` для `fib` без первых трех и т. д. Перейдем к применению функции:

```
take 5 fib
take 5 (1:1:zipWith (+) fib (tail fib))
```

```
1:take 4 (1:zipWith (+) fib (tail fib))
1:1:take 3 (zipWith (+) fib (tail fib))
```

Здесь используем то, что к этому моменту первые два элемента списка чисел Фиббоначи уже вычислены и потому мы можем разбить `fib` и `tail fib` на голову и хвост.

```
1:1:take 3 ((1+1):zipWith (+) fib_1 fib_2)
1:1:2:take 2 (zipWith (+) fib_1 fib_2)
```

Аналогично, третий элемент уже вычислен (далее будем сокращать `tail` до `t`):

```
1:1:2:take 2 ((1+2):zipWith (+) fib_2 fib_3)
1:1:2:3:take 1 (zipWith (+) fib_2 fib_3)
1:1:2:3:take 1 ((2+3):zipWith (+) fib_3 fib_4)
1:1:2:3:5:take 0 (zipWith (+) fib_3 fib_4)
[1,1,2,3,5]
```

Теперь рассмотрим вычисление списка простых чисел с использованием корекурсии. Определим функцию `noDivisors` так, чтобы она могла работать с бесконечным упорядоченным списком — добавим в нее отсечение тех элементов, которые больше корня:

```
noDivisors l x =  all (\n -> x `mod` n /= 0) $
                      takeWhile (\n -> n*n < x) l
```

Теперь определим простые числа следующим образом: первые два простых числа 2 и 3, а дальше простые числа — это те среди чисел 6k±1, которые не имеют делителей среди простых чисел:

```
primes = 2:3:filter (noDivisors primes)
                  [6*k+z | k <- [1..], z <- [-1,1]]
```

И напоследок еще один пример вычисления списка простых чисел — полностью эквивалентный решету Эратосфена:

```
eratosthenes (x:xs) = x:eratosthenes (filter ((/= 0).(`mod` x)) xs)
primes = eratosthenes [2..]
```