Множества

Списки в Haskell, как уже говорилось на прошлой лекции, используются порой для представления других различных структур данных.

Уже упоминалось, что определители списков имеют представление, схожее с описаниям множеств в теории множеств. В этих случаях мы можем описать потенциально бесконечные множества, например:

```
[ (2n+1)^3 | n<-[1..] ]
```

Для представления конечных множеств в виде списков, есть ряд полезных функций, которые содержатся (помимо других полезных определений) в модуле Data.List.

```
import Data.List

...
```

или прямо в ghci:

```
Prelude> :m Data.List
Prelude Data.List>
```

Это следующие функции:

- **nub** xs — возвращает список, в котором удалены повторяющиеся элементы;

```
> nub [1,2,3,4,3,2,1,2,4,3,5]
[1,2,3,4,5]
```

- **delete** x xs — удаляет первое вхождение заданного элемента x из списка xs

```
> delete True [True,False,True]
[False,True]
```

- **union** xs ys — возвращает объединение двух списков;

```
Prelude Data.List> [1,2,2,2] `union`  [2,2,3]
[1,2,2,2,3]
```

видим, что всё-таки, списки — это не множества! (Дубликаты и элементы первого списка удаляются из второго списка, но если первый список содержит дубликаты, то и результат будет таким же).

- **intersect** xs ys — возвращает пересечение двух списков

```
> [1,2,2,2] `intersect`  [2,2,3]
[2,2,2]
```

то же самое, это не множества (операция также сохраняет дубликаты первого списка).

- (\\) — возвращает разность (неассоциативную) двух списков, таким образом: (xs ++ ys) \\ xs == ys (если нет общих элементов);

```
> [1,2,2,2] \\  [2,2,3]
[1,2]
```

- **elem** x xs — предикат, возвращающий истину, если элемент x принадлежит списку xs;

- **tails** xs – функция возвращает список хвостов данного списка, напр.:

```
>tails ”abc”
[”abc”, ”bc”, ”c”, ””]
>tails [1,2,2,3]
[[1,2,2,3],[2,2,3],[2,3],[3],[]]
```

эта функция будет полезна для определения множества подмножеств данного множества.

Таким образом, мы видим, что списки, если они содержат только уникальные элементы, могут представлять соответствующие множества. И если нам предстоит делать различные теоретико-множественные операции над списками, представляющими множества, то мы должны каждый раз «очищать» списки от дублей с помощью функции **nub**.

Стоит также отметить, что в Haskell есть специализированные модули для работы с конечными множествами, самый известный из которых Data.Set, основанный на понятии сбалансированных бинарных деревьев (т.е. конечные множества имеют внутреннее представление как раз в виде таких деревьев). Модуль вводит достаточное количество новых (помимо известных) функций: для вычисления декартового произведения, множества подмножеств, обычной и симметрической разности, отношения подмножества и т.п.

Data.Set

Sets

Свёртки

В первой лекции мы обсуждали декларативность, как суть и достоинство Haskell.

Что по этому поводу думают «императивщики»?

> Э.Гарольд «Почему вы ненавидите цикл for?»
>
> Что не так с циклами for?
>
> What’s Wrong with the For Loop
>
> Why Hate the for Loop?

Вот вам распространенный пример на языке Java:

```
double sum = 0;
for (int i = 0; i < array.length; i++) {
    sum += array[i];
}
```

Что он делает? Понимание куска этого кода не займёт много времени — это обыкновенное суммирование элементов массива. Однако для того, чтобы прочесть этот пример, надо последовательно просмотреть около тридцати лексем, распылённых по четырём

строкам. Разумеется, при написании этого кода можно было бы воспользоваться синтаксическим сахаром, который сократил бы его объём. Однако суть в том, что есть множество мест, где вы потенциально можете допустить ошибку при написании обычного суммирования.

Желаете доказательств? Следующий же пример из статьи Гарольда:

```
String s = ””;
for (int i = 0; i < args.length; i++) {
    s += array[i];
}
```

Замечаете баг? Даже если этот код скомпилируется и пройдёт ревизию, могут пройти недели до момента получения первого сообщения об ошибке и ещё несколько до выпуска исправления. И это всего-навсего простые циклы `for`. А теперь представьте, насколько более сложной становится работа в случае с сильно разросшимися, и, быть может, вложенными циклами. (Если вы все ещё не беспокоитесь о багах, подумайте об обыкновенных опечатках. А затем подумайте о том, как часто вы пишите подобные циклы.)

Как в этой ситуации мог бы помочь декларативный стиль? Вот вам первый пример на Haskell:

```
total = sum array
```

Собственно, **`sum`** разворачивается в конструкцию вида:

```
sum = foldl (+) 0
```

и **`concat`**:

```
concat = foldr (++) []
```

Два приведённых выше примера показывают, как можно взять список и свернуть его в один элемент. В мире функционального программирования подобные операции называются *свёртками*. Для своей работы свёртка (опишем *левую свёртку*) принимает функцию, начальное значение (который далее будет выполнять роль аккумулятора) и посещает первый элемент списка. Функция применяется к аккумулятору и первому элементу списка, результат кладётся в аккумулятор. Затем свёртка применяет замыкание к обновленному значению аккумулятора и второму элементу списка. Этот процесс продолжается вплоть до конца списка, когда последнее значение аккумулятора и становится результатом применения свёртки.

Правая свёртка будет действовать немного иначе: она сначала разложит весь список, отщепляя на каждом шаге первый элемент в уменьшающемся хвосте; затем применит функцию к начальному значению и к последнему элементу; затем применит полученное значение к предпоследнему элементу, и так до тех пор, пока не вернется к самому первому элементу.

Каким образом мы могли бы получить такие хорошие функции-свёртки?? Допустим, для суммы списка `[1,2,3,4,5]`:

```
1 + 2 + 3 + 4 + 5 = 0 + 1 + 2 + 3 + 4 + 5
= ((((0 + 1) + 2) + 3) + 4) + 5
```

```
= 1 + (2 + (3 + (4 + (5 + 0))))
```

На месте `(+)` могла быть любая бинарная операция.

```
f (f (... (f (f x0 x1) x2) ...) xn-1) xn,
f x1 (f x2 (... (f xn-1 (f xn x0)) ...)).
```

значение `x0` дописывается в конце или в начале, после чего мы начинаем производить операции или *свёртку*. Определим теперь функции *высокого порядка*, которые обобщают это действие, т.е. задают его шаблон:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f x0 []      =  x0
foldl f x0 (x:xs)  =  foldl f (f x0 x) xs -- отщепляет

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x0 []      =  x0
foldr f x0 (x:xs)  =  f x (foldr f x0 xs) -- вгрызается
```

Обратим внимание на сигнатуру, на то, что начальный параметр (который в первом случае является и аккумулятором) может быть другого типа.

Кроме того, отметим, что foldl по сути является *хвостовой рекурсией*, и может быть в какой-то степени оптимизирована.

Рассмотрим вновь, как они работают:

```
foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
= foldl (+) (((0+1)+2)+3) []
= ((0 + 1) + 2) + 3
= (1 + 2) + 3
= 3 + 3
= 6

foldr (+) 0 [1,2,3]
= 1 + (foldr (+) 0 [2,3])
= 1 + (2 + (foldr (+) 0 [3]))
= 1 + (2 + (3 + (foldr (+) 0 [])))
= 1 + (2 + (3 + 0)))
= 1 + (2 + 3))
= 1 + 5
= 6
```

И в данной ситуации пока разницы нет, так как функция сложения ассоциативна, да и список наш короток.

Вариант без начальных данных

Часто при обработке списка в качестве начального значения будет естественным взять значение первого или последнего элемента (как в примере со сложением в предыдущем разделе). Тогда можно определить более простую версию свёрток, однако они должны применяться только для непустых списков.

```
foldl1 f (x:xs) = foldl f x xs
```

Задавать по аналогии **foldr** было бы довольно глупо:

```
foldr1 f xs = foldr f (last xs) (init xs)
```

где операции **last** и **init** сами были бы достаточно трудоёмки.

По аналогии с исходными определениями **foldl** и **foldr**, можно сделать так:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f [x]         =  x
foldl1 f (x:(y:ys))  =  foldl1 f (f x y : ys)

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
```

они немного проще, без аккумулятора, начальное значение выполняет первый или последний элемент. Но, тем не менее, в Haskell используется для **foldl1** именно первая версия.

Разница в неассоциативном случае

Порядок вычисления левой и правой свёрток может оказаться существенным, если используемая операция не ассоциативна, как, например, (-).

```
> foldr1 (-) [1..3]
2
> foldl1 (-) [1..3]
-4
```

Бесконечные списки

Рассмотрим разницу в вычислениях при обработке бесконечных списков.

Зададим функцию

```
g x y = x
```

Компилятор достаточно сообразителен, чтобы рассматривать ее в данном случае как `g x _`.

Рассмотрим применение **foldr1** к бесконечному списку с указанной выше функцией.

```
Prelude> foldr1 g [1..]
1
```

Теперь, если мы рассмотрим применение **foldl1** к бесконечному списку с указанной выше функцией, то зависнет не только процесс вычисления, но даже и сам ghci :(

```
Prelude> foldl1 g [1..]
```

Давайте попробуем понять, что произошло. В первом случае, сообразительность компилятора и ленивость Haskell позволяет вычислить только первый элемент, чтобы понять чему будет равен результат.

```
foldr1 g [1..]
= g 1 (foldr1 g [2..])
```

Во втором случае, несмотря на то, что в силу ленивости, вычисления, связанные с функцией g откладываются, все равно приходится ждать исчерпания бесконечного списка:

```
foldl1 g [1..]
=  foldl1 g ((g 1 2) : [3..]) =
=  foldl1 g ((g (g 1 2) 3) : [4..]) =
=  foldl1 g ((g (g (g 1 2) 3) 4) : [5..]) =
=  ...
```

Отметим, что дело тут именно в организации вычислений, так как сама функция g является ассоциативной:

```
(x `g` y) `g` z == x `g` z == x
==
x `g` (y `g` z) == x `g` y == x
```

(можно ещё рассмотреть случаи `g x y = y`, что тоже приводит к зависанию)

Очень большие списки

Ещё более тонкая ситуация связана с ленивостью или энергичностью вычислений последовательности функций при свёртках.

Рассмотрим пример не бесконечного, но всё-таки очень большого списка, с функцией, которая в отличие от g, определённой выше, требует реального использования обоих аргументов. Например, с уже ранее рассмотренным сложением (+).

Следующий пример из-за переполнения стека вызвал зависание Windows-7, для выхода из которого не помогли Ctrl-Alt-Del (было актуально для компилятора 2018 года и на ноутбуке с 4 Гб ОЗУ, сейчас, в 2020 году, выдаёт ошибку без зависания и на бОльших списках).

```
foldr1 (+) [1..10^8]
= 1 + (foldr1 (+) [2..10^8])
= 1 + (2 + (foldr1 (+) [3..10^8])) =
= 1 + (2 + (3 + (foldr1 (+) [4..10^8]))) =
=  ...
```

В этом случае **foldr1** должно пройти до правого конца списка, а потом начнет сворачивать список с правой стороны. При вычислении заполняется стек, и выражение вроде **foldr1** (+) [1..10^8] может вообще не вычислиться — будет выдано сообщение «Exception: stack overflow».

Попытка вычислить `foldl1 (+) [1..10^8]` тоже приведет к зависанию (и Windows в том числе), но по другой причине:

```
foldl1 (+) [1..10^8]
= foldl1 (+) (1 + 2) : [3..10^8]
= foldl1 (+) ((1 + 2) + 3) : [4..10^8]
= foldl1 (+) (((1 + 2) + 3) + 4) : [5..10^8] =
=  ...
```

Кстати, мы знаем, что `foldl1` реально определена в терминах `foldl` и использует помимо хвостовой рекурсии и аккумулятор.

Однако, в силу определения левой свёртки и ленивости языка, накапливающийся параметр не будет вычисляться, так как левая свёртка будет «стремиться раскрутить» список. Таким образом, мы вновь получаем переполнение стека.

Из общей теории известно, что хвостовые рекурсии лучше работают в энергичных вычислениях. И в этом случае, левая свёртка в виде хвостовой рекурсии была бы лучше — хотя энергичные вычисления сразу бы стали вычислять и весь список `[1..10^8]`, что опять же бы привело к переполнению стека.

Для этого случая есть особый *строгий* (но для нужного аргумента) вариант левой свёртки `foldl1'`, который может справиться с данным вычислением:

```
Prelude> :m Data.List
Prelude Data.List> foldl1' (+) [1..10^8]
5000000050000000
```

(его нужно подгружать отдельно в составе модуля)

wikibooks: Lists III

wiki.haskell: Foldr Foldl Foldl'

wiki.haskell: Fold

wikipedia: Fold (higher-order function)

Matt Parsons. The Magic of Folds.

Примеры использования свёрток

Рассмотрим несколько примеров использования свёрток, помимо уже описанных.

Попробуем вычислить с их помощью длину списка. Для этого нам нужно понять, какую функцию должна использовать свёртка на элементах. Посмотрим на сигнатуру свёрток — тип начального значения может отличаться от типа элементов списка.

```
f n _ = n + 1
```

Тогда

```
import Data.List hiding (foldr,foldl)

f :: Int -> Int -> Int
f n _ = n + 1
```

```
-- определения ниже заданы по типу частичных, список подразумевается)
len1 :: [Int] -> Int
len1 = foldr f 0
len2 :: [Int] -> Int
len2 = foldl f 0
len3 :: [Int] -> Int
len3 = foldl' f 0
```

Заметим, что `len1` из-за того, что функция `f` игнорирует второй параметр, не разворачивает весь список, и ответ будет равен `(len1 !! 0) + 1` для любого непустого списка:

```
> len1 []
0
> len1 [1]
2
> len1 [1,1]
2
> len1 [1..10^8]
2
```

В самом деле, для вычисления `len1 [2,3]` получим

```
len1 [2,3] == foldr f 0 [2,3] ==
  f 2 (foldr f 0 [3]) -- далее второй аргумент игнорируется
  == f 2 _ == 2 + 1 == 3
```

Функции `len2` и `len3` работают корректно, используют хвостовую рекурсию, но `len3` получается за счет энергичных вычислений по первому аргументу более эффективной и быстрой.

```
*Main> len3 [1..10^7]
10000000
*Main> len2 [1..10^7]
10000000
```

```
g :: Int -> Int -> Int
g _ n = n + 1

len4 :: [Int] -> Int
len4 = foldr g 0
```

Этот вариант с правой свёрткой работает корректно, по скорости сравним с `len2`, но хуже `len3`.

Интересные определения функций **`head`** и **`last`**:

```
head = foldr (\a _ -> a) undefined
last = foldl (\_ x -> x) undefined
```

wiki.haskell: Fold/Examples

И рассмотрим такую конструкцию функции **`filter`**:

```
filter :: (a -> Bool) -> [a] -> [a]
filter f  =
  foldr (\x xs -> if f x then x:xs else xs) []
```

Здесь описана частичная функция (без аргумента-списка), которая организована правой свёрткой с помощью анонимной функции. В роли исходного значения — пустой список `[]`, а функция-обработчик будет собирать (делать свёртку) новый список, добавляя элементы, если они удовлетворяют предикату.

Прогонка (сканирование)

Рассмотрим близкие к свёрткам операции, которые вместо одного свернутого значения возвращает список последовательных свёрток.

Функции **`scanl`** и **`scanr`** похожи на **`foldl`** и **`foldr`**, но они сохраняют все промежуточные значения в список. Также существуют функции **`scanl1`** и **`scanr1`**, которые являются аналогами **`foldl1`** и **`foldr1`**.

В левой прогонке первым элементом будет начальное значение, а последним — результат левой свёртки.

```
scanl f x0 [x1, x2, ...] =
  [x0, x0 `f` x1, (x0 `f` x1) `f` x2, ...]
```

Определение **`scanl`**:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl _ x0 []      =  [x0]
scanl f x0 (x:xs)  =  x0 : scanl f (f x0 x) xs
```

Определение **`scanl1`**:

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanl1 _ []      =  []
scanl1 f (x:xs)  =  scanl f x xs
```

Как можно догадаться, правая прогонка должна содержать начальное значение в качестве последнего элемента и результат правой свёртки в качестве первого.

```
scanr f x0 [x1, x2, ...] =
  [..., (x0 `f` x1) `f` x2, x0 `f` x1, x0]
```

И соответствующие определения:

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ x0 []      =  [x0]
scanr f x0 (x:xs)  =  f x x' : (x':xs')
                      where (x':xs') = scanr f x0 xs

scanr1 :: (a -> a -> a) -> [a] -> [a]
scanr1 _ []      =  []
scanr1 _ [x0]    =  [x0]
scanr1 f (x:xs)  =  f x x' : (x':xs')
                    where (x':xs') = scanr1 f xs
```

Обратите внимание, что, по соглашению, прогоны **scanl1** и **scanr1** пустого списка определены и равны пустому списку, а свёртки **foldl1** и **foldr1** для пустых списков не определены.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
```

unfoldr

Подключенный модуль `Data.List` позволяет выполнять обратную операцию — *развёртку*, **unfoldr** (реализована только такая версия).

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f = \b -> case (f b) of
    Just (a, b') -> a : unfoldr f b'
    Nothing      -> []
```

Развёртка по данной функции и «зерну» (т.е. начальному значению) генерирует список, возможно бесконечный. По принципу: функция принимает начальное значение, и если возвращается пара (обёрнутая тегом **Just**), то первый элемент пары даёт первый элемент будущего списка, а второй элемент пары выступает в качестве зерна для вычисления следующего элемента. Если возвращается значение **Nothing**, то на данном этапе присоединяется пустой список `[]` и дальнейшие вычисления прекращаются.

```
> f = (\b -> if b == 0 then Nothing else Just (b, b-1))
>unfoldr f 10
[10,9,8,7,6,5,4,3,2,1]
```

Изменения модуля Prelude

Если мы глянем на исходники разобранных выше функций в нынешних версиях библиотек компилятора Haskell (ghc), то мы удивимся тому, что они значительно отличаются. Не вдаваясь в особые детали этого вопроса, отметим, что с января 2015 (ghc ver. 7.10), модуль **Prelude** был существенно переделан так, чтобы была возможность работать тем же свёрткам не только на списках, но и на других структурах данных, таким как, например, деревья, множества, последовательности и т.п. Иными словами, была обеспечена возможность перегружать эти функции при создании новых структур данных.

wiki.haskell: Foldable Traversable In_Prelude

Отметим, что несмотря на изменения кода, в большинстве случаев, указанные функции работают именно как было указано выше.

Другие типы данных, подобные спискам

Помимо списков в Haskell есть и другие типы данных, которые так или иначе позволяют организовывать конечные коллекции значений определенного типа.

Для некоторых из них (Array, Tree, Map, Vector, Sequence) реализованы или свои версии свёрток, или воплощения класса Foldable.

Вот примеры некоторых из популярных коллекций:

- Data.Array — неизменяемые (вообще) массивы
- Data.Map — реализация *отображений*, или как их чаще называют *ассоциативных массивов*
- Data.Set — о множествах уже говорили выше, построены на так называемых «бинарных сбалансированных деревьях»
- Data.Sequence — более эффективная реализация конечных списков, построенных на так называемых «пальчиковых деревьях»
- Data.Tree — общее представление деревьев и «леса» деревьев
- Data.Tuple — дополнительные функции для работы с кортежами, существуют дополнительные пакеты, как, например, tuple, существенно расширяющие число и возможности таких функций
- Data.Vector в vector package — эффективная реализация массивов, в том числе и мутабельных (без эмуляции) — проект активно развивается, см. напр., тут: The vector package

Есть ещё ряд структур, определённых специально для списков-строк, для байтовых строк и текстов, для сериализации данных и т.п.