Общее о типах в Haskell

Типы данных — это сильная сторона Haskell. И мы последовательно ее изучим, начиная от встроенных примитивов до создания собственных типов данных. Сегодня рассмотрим кортежи и списки, как примеры сложных типов, неких контейнеров, содержащих в себе другие типы.

Кортежи

Как и в математике, выражения вида

`(a,b), (a,b,c), …, (x1…,,x15)`

называют кортежами. В Haskell подобные объекты описывает специальный алгебраический тип данных (что это такое, будем изучать позже), язык позволяет кортежи до длины 15, функции поддержки до размера 7.

Особенностью кортежей в Haskell является их неизменность и возможность совмещать в себе компоненты разных типов.

```
a = (1, 'a', True)
b = ((-2.0), 3.14)
c = fst b
d = snd b
```

Функции **`fst`** и **`snd`** — наиболее часто употребительные функции для разбора компонет пары.

Для нас кортежи полезны в случае возврата функцией нескольких значений разного типа.

Дополнительная поддержка осуществляется в модуле Data.Tuple.

Для кортежей большей размерности поддержка есть, например в пакете tuple.

Д.Шевченко. О Haskell по-человечески. Кортеж.

wikibooks: Haskell/Tuples.

Списки

Списки в Computer Sci

Абстрактный тип данных, который представляет собой упорядоченную (как правило, конечную) последовательность значений, в которой некоторое значение может встречаться более одного раза.

Происходит из математического понятия конечной последовательности.

В языках программирования обычно реализуется в виде массива или линейного связного списка.

Последнее в виде односвязного списка наиболее популярно в функциональных языках программирования и наиболее приспособлено к ним.

> Односвязный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей.

Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на `NULL`. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

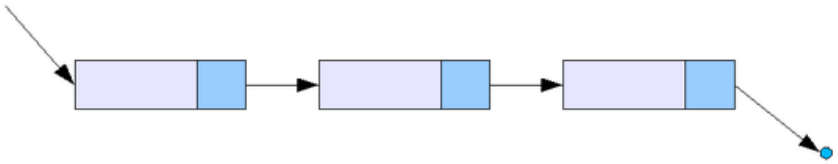

Рис. 1: Односвязный список

ru.wikipedia: Список (информатика)

ru.wikipedia.org: Связный список (односвязный список, однонаправленный связный список)

Списки в Haskell

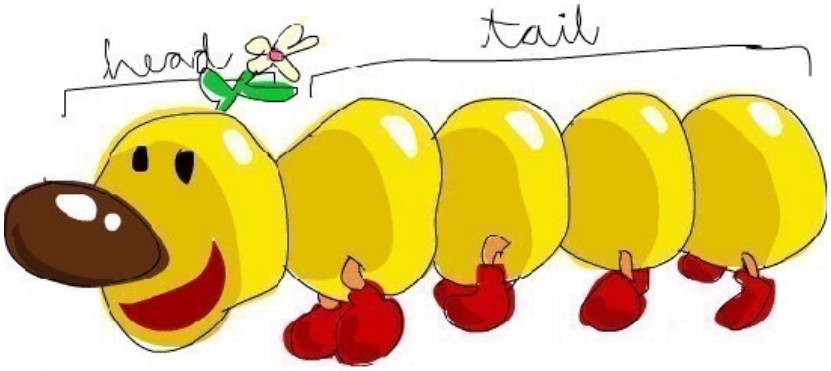

Рис. 2: Списки по Липовачу

Списки в Haskell — это односвязные линейные списки.

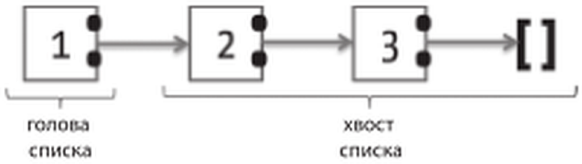

Рис. 3: Представление списков в Haskell (`:` — это конструктор)

Состоит из элементов одного типа. Число элементов не фиксируется, может быть даже потенциально бесконечным! Тем не менее, при каждом изменении списка в Haskell, мы его фактически целиком или частично пересоздаем.

Представление в памяти

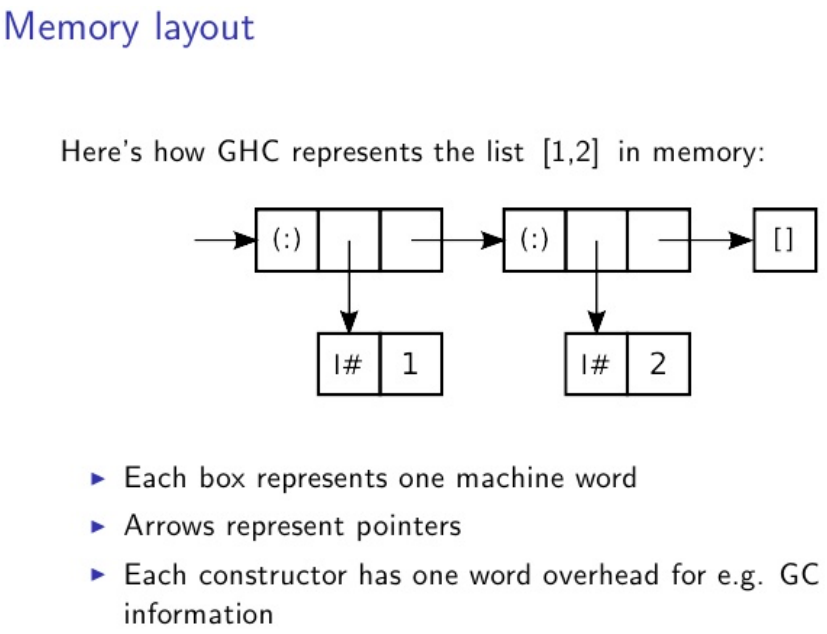

Рис. 4: Haskell списки на «кухне»

Простейший синтаксис

В самой простой ситуации:

```
list = [True,True,False]
```

И так как `list` это в Haskell функция, то проверим её тип:

```
Prelude> :t list
list :: [Bool]
```

Ситуация с числами немного сложнее, так как числа — это перегруженные функции. Поэтому, желательно сразу сообщать тип, если мы не планируем обобщенных конструкций:

```
list2 :: [Int]
list2 = [1,5,21,(-2)]

list3 = [2.4, (-3.2), 0, 4] :: [Double]
```

Возможно создание очень причудливых списков:

```
foolist =
  [(+), (*), (-), (\t s -> (t*s+s))] :: [Int->Int->Int]
```

и даже возможен список «ординалов»:

```
setlist = [ [], [[]], [[],[[]]] ]
```

как в теории множеств:

```
{ ∅, {∅}, {∅,{∅}} }
```

Список символов в Haskell — это стандартное представление строк:

```
hi = ['h','e','l','l','o']
```

И в ghci:

```
*Main> hi
”hello”
*Main> :t hi
hi :: [Char]
```

Таким образом, мы могли бы сразу указать:

```
hi = ”hello”
```

Отметим, что тип **`String`** — это синоним типа **`[Char]`**.

Причины использования

Почему списки так много и часто используются в Haskell?

- традиция функционального программирования
- операции (некоторые) с ними достаточно эффективны
- есть удобная система обозначений для списков, особенно определители списков (list comprehensions)
- большое число встроенных и библиотечных функций
- паттерн-матчинг
- списками представлены строки в Haskell
- другие структуры данных порой малоизвестны
- массивы в Haskell работают часто хуже чем списки

Таким образом, хотя часто неэффективно, но списки в Haskell используются для представления массивов, очередей, множеств, отображений, ассоциативных списков, таблиц…

Рекурсивное представление списка

На самом деле, конструкция вида

```
list2 = [1,5,21,(-2)]
```

не является первичной (это т.н. «синтаксический сахар», en.wikipedia: Syntactic sugar)

Базовыми являются *пустой список* `[]` и *конструктор списка* `:`. И списки формируются рекурсивно:

```
[1,5,21,(-2)] == 1:[5,21,(-2)] == 1:(5:[21,(-2)])
   == 1:(5:(21:[-2])) == 1:(5:(21:((-2):[])))
   == 1:5:21:(-2):[]
```

В следующих лекциях мы уделим бОльшее внимание конструкторам данных и значений. В данном случае, отметим, что наличие такого конструктора позволяет осуществлять паттерн-матчинг (выбор по образцу) и рекурсии со списками. В паттерн-матчинг могут входить конструкторы, а функции — нет.

```haskell
-- очень древние, но простые версии функций

head :: [a] -> a
head (x:_) = x
head []    = error "…"

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

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

init :: [a] -> [a]
init [x]    = []
init (x:xs) = x:(init xs)
init []     = error "…"

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

push :: a -> [a] -> [a]
push x xs = x:xs
```

Отметим тут, что обозначение `xs` (`ys`, `zs` и т.п.) традиционное для Haskell-программистов, обозначает список, т.е. много значений.

Вычисление длины списка организованно по рекурсии, как раз с помощью конструктора списка в левой части и обращении к самой себе с меньшим параметром — в правой.

И ещё отметим тут, что все указанные тут списочные функции *полиморфны*, т.е. они способны работать со списками значений любых типов. И это отражено в сигнатуре функций, где вместо традиционных типов **`Bool`**, **`Int`**, **`Double`** присутствуют переменные типов с маленькой буквы `a`, `[a]`.

Полезные базовые функции для работы со списками

Рассмотрим ряд полезных **базовых** функций.

(везде ниже `xs`, `ys` и т.п. обозначают список)

- `xs !! n` — получим `n`-й произвольный элемент списка `xs`, начиная с нулевого;
- **`head`** `xs` — вернёт *первый* элемент списка `xs`;
- **`last`** `xs` — вернёт *последний* элемент списка `xs`;
- **`tail`** `xs` — вернёт список `xs` без *первого* элемента;
- **`init`** `xs` — вернёт список `xs` без *последнего* элемента;
- **`reverse`** `xs` — вернёт обратный список;
- **`length`** `xs` — вернёт длину списка `xs`.

Добавление к списку

Ряд функций для добавления как отдельных элементов, так и целых списков.

- `x:xs` — добавит `x` в качестве нового головного (нулевого) элемента к списку `xs`;
- `xs ++ [x]` — добавит `x` в качестве последнеого элемента к списку `xs`;
- `list1 ++ list2` — *конкатенация* двух списков `list1` и `list2`.

Если же нам необходимо вставить `x` на произвольную позицию `n`, надо разбить список на два меньших куска, поместить новый элемент между ними и соединить все вместе обратно:

```
let (ys, zs) = splitAt n xs
in ys ++ [x] ++ zs
```

получим новый список, где `x` займет требуемую позицию `n`.

Удаление из списка

- **`drop`** `n xs` — вернёт список, где удалены первых `n` элементов из списка `xs`;
- **`take`** `n xs` — вернёт список из первых `n` элементов из списка `xs`;
- **`splitAt`** `n xs` — вернёт пару списков, полученных из списка `xs` разбиением с `n`-й позиции (причем `n`-й элемент войдет во второй список).

Если мы хотим удалить только `n`-й элемент из списка, нам придется вновь разбить список на два меньших куска, удалить нулевой элемент из второго списка и соединить все вместе обратно:

```
let (ys, zs) = splitAt n xs
in ys ++ (tail zs)
```

Условия на список

Различные функции, которые извлекают подсписок в соответствии с условиями, выполняют проверки и т.п.:

- **`null`** `xs` — проверяет, пуст ли список `xs` (возвращает **`True`** или **`False`**);
- `x` **`` `elem` ``** `xs` — проверит, лежит ли элемент `x` в списке `xs` (возвращает **`True`** или **`False`**);
- **`any`** `test xs` — вернёт **`True`**, если какой-либо элемент списка удовлетворяет условию `test` (если тип списка `[a]`, то тип функции `test` должен быть a -> Bool);
- **`all`** `test xs` — вернёт **`True`**, если все элементы списка удовлетворяет условию `test` (с аналогичными выше требованиями);
- **`filter`** `test xs` — вернёт только те элементы списка, которые удовлетворяют условию `test` (с аналогичными выше требованиями).

Пример использования в ghci:

```
Prelude> let xs = [1,2,322,100]
Prelude> filter (\x -> if x>1 then True else False) xs
Prelude> [2,322,100]
```

Изменение списка или его элементов

- `map f xs` — применит функцию `f` ко всем элементам списка `xs` и вернёт новый список;
- `map (\x -> if p x then f x else x) xs` — применит функцию `f` только к тем элементам списка, для которых функция `p` вернёт `True`;
- `concat [[x,y,z],[a,b,c,d,e],[o,p,r,s,t]]` — из списка списков вернёт список элементов всех списков `[x,y,z,a,b,c,d,e,o,p,r,s,t]`;
- `zip xs ys` — вернёт список пар вида `(x,y)`, в каждой из которых первый элемент `x` является очередным элементом первого списка `xs`, а второй элемент `y` — второго списка `ys`.

сюда же можно добавить

- `sum xs` — суммирует элементы в списке `xs`, при условии, что элементы могут быть просуммированы;
- `maximum xs` — возвращает максимум в списке `xs`, при условии, что элементы могут быть сравнимы между собой (`minimum` — аналогично);

Замечания о скорости работы списочных функций

Следующие функции всегда быстры:

- `:` — добавление первого элемента к началу списка;
- `head` — получение первого элемента списка;
- `tail` — удаление первого элемента из списка, т.е. хвост списка.

Функции, которые имеют дело с `n`-м элементом (или первыми `n` элементами) обычно имеют скорость порядка `n`:

- `xs !! n`;
- `take n xs`;
- `drop n xs`;
- `splitAt n xs`.

Все функции, которые для выполнения нуждаются в целом списке, становятся тем медленнее, чем длиннее становится список:

- `length xs`;
- `list1 ++ list2` — скорость зависит *только* от длины `list1`;
- `last xs`;
- `map my_fn xs`;
- `filter my_test xs`;
- `zip list1 list2` — скорость зависит от *наименьшего* из этих двух списков, поскольку именно он задет размер результирующего списка;
- `sum xs`;
- `minimum xs`;
- `maximum xs`.

Определители списков и диапазоны

list comprehensions (eng.)

иногда говорят о *Списковом включении*

List comprehension

По аналогии с математической теории множеств, в которой множество можно задать из имеющего с помощью *аксиомы выделения*:

```
{ 2x | x ∈ N }
```

где мы задаем множество четных натуральных чисел, в Haskell мы можем задать список четных натуральных чисел с помощью уникального механизма, называемого *определитель списков*:

```
[ 2*x | x <- [1..] ]
```

Отметим, заданный список будет потенциально бесконечный, тем не менее, до тех пор пока он в данной форме, он является всего лишь чистой декларацией. Вычислять его будем только при обращениях вроде:

```
take 8 [ 2*x | x <- [1..] ]
```

После выражений вида `x <- xs`, называемых *генератором*, могут быть дополнительные условия (*охрана*), которые также должны быть выполнены:

```
[ 2*x | x <- [1..], 3 < x, x < 7 ]
```

Отметим, что хотя с математической точки зрения данный список будет формально заданным, его вычисление, быстро выдав три первых ответа: 8,10,12 — зависнет, так как реально программа будет перебирать все натуральные значения `x`, начиная с 1. Поэтому запуск можно делать как

```
take 3 [ 2*x | x <- [1..], 3 < x, x < 7 ]
```

либо как указано выше, вводить ограничение сверху, т.е. использовать `x <- [1..57]`.

В примере также показан способ построения списков, называемых *диапазон*:

- `[1..]` — список всех натуральных чисел;
- `[1,3..]` — список всех нечетных натуральных чисел;
- `[1,3..99]` — список всех нечетных натуральных чисел от 1 до 99.

Похожие записи тоже используются в математике:

```
{2,4,6,..},  {1,..,9}
```

В целом, определители списков позволяют красиво в декларативном стиле описывать математически сложные множества объектов, одновременно задавая реализацию по их генерации на Haskell. Не всегда это будет оптимально по производительности, но всегда ясно и понятно с точки зрения описания.

В лекции 2 мы уже говорили о предикате `isprime`, его наличие позволяет красиво описать множество всех простых чисел:

```
primeset = [x | x <-[2..], isprime x]
```

(реализация `isprime` будет в заданиях).

Замечания об эффективности списков и операций на них

Выше мы уже обсуждали скорость работы некоторых списочных функций и было отмечено, что скорость некоторых из них существенно зависит от длины списка.

Память тоже расходуется очень интенсивно, так как фактически мы передаем копии (ну на самом деле обещания их вычислить) данных.

Рассмотрим вычисление длины некоторого списка с помощью **length**.

```
length [True, True, False] =>
1 + length [True, False] =>
1 + (1 + length [False]) =>
1 + (1 + (1+ length [])) =>
1 + (1 + (1+ 0)) =>
1 + (1 + 1) =>
1 + 2 =>
3
```

Видно, что интенсивно используется память: на хранение многочисленных промежуточных данных и на хранение ссылок для возврата.

За счет ленивости вычислений, исполняющей системе в ряде случаев удается экономить память даже для такой функции.

Воспользуемся вспомогательной командой `:sprint` внутри ghci, которая помогает исследовать, вычислено то или иное выражение до конца, или нет:

```
Prelude> let xs = map (+1) [1..10] :: [Int]
Prelude> :sprint xs
xs = _
Prelude> head xs
2
Prelude> :sprint xs
xs = 2 : _
Prelude> length xs
10
Prelude> :sprint xs
xs = [2,_,_,_,_,_,_,_,_,_]
Prelude> sum xs
65
Prelude> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Мы видим, что исполняющая система Haskell старается по возможности уклоняться от вычислений. Так, для вычисления длины списка нам не нужны были сами значения элементов — они и не считаются (первый элемент, равный 2, был уже вычислен при вызове `head xs`).

Но конечно, это не решает всех проблем, связанных с накладными расходами при работе с рекурсиями. Одним из решений является использование *хвостовой рекурсии*.

Хвостовая рекурсия и аккумулятор

Рассмотрим теперь такой вариант вычисления длины списка

```
len :: [a] -> Int -> Int
len [] n     = n
len (x:xs) n = len xs (n+1)

length' :: [a] -> Int
length' ys = len ys 0
```

Более сложный вариант с двумя функциями имеет то преимущество, что функция `len` сделана как *хвостовая рекурсия*, т.е. она вызывает в последнем выражении сама себя. Функция **`length`** при определении в последнем выражении вызывает `(+)`. Хвостовые рекурсии распознаются компилятором и оптимизируются как циклы, уменьшая накладные расходы.

Кроме того, функция `len` использует *накапливающий параметр* `n`, который тоже может рассматриваться как метод оптимизации рекурсий. Как правило они используются вместе.