

Haskell: АТД, паттерн матчинг, классы типов, ленивые вычисления

Пугачёв Константин (K.V.Pugachev@inp.nsk.su)

2018-11-16 15h

В предыдущей серии

- ФП: чистота, декларативность, набор идей
- λ -исчисление – ФП, возведённое в абсолют
- Нумералы Чёрча, комбинаторы н.т.: наркоз уже не тот
- Haskell: знакомство

Алгебраические типы данных

data – определение нового типа данных

```
data Number = One | Two | Many Integer
```

Number – конструктор типа

One, Two, Many – конструкторы данных

```
x :: Number
```

```
x = One
```

```
y :: Number
```

```
y = Many 256
```

Некоторые из них – нульарные, но это ничего не меняет. Все значения можно понимать как нульарные функции.

Аналогия с C++:

```
struct One {};
```

```
struct Two {};
```

```
struct Many { int x; };
```

```
typedef std::variant<One, Two, Many> Number;
```

Алгебраические типы данных

N -арные конструкторы типов:

```
data Number a = One | Two | Many a -- a – тип-параметер
```

Number – не тип значения, Number <что-то> – тип значения:

```
x :: Number Int  
x = One
```

```
y :: Number Double  
y = Many 256
```

Аналогия с C++:

```
struct One {};  
struct Two {};  
typedef <typename a> struct Many { a x; };  
  
template <typename a>  
using Number = std::variant<One, Two, Many<a>>;
```

Сумма типов и произведение типов

```
data Sum = First | Second | Third
```

```
data Prod = Ctor T1 T2 T3
```

```
-- можно заменить типом (T1, T2, T3)
```

Можно конструировать сколь угодно сложные типы и значения:

```
-- много аргументов, вариантов, элементов произведения:
```

```
data TypeCtor a b = BinaryDataCtor a b | UnaryDataCtor1 b  
                  | UnaryDataCtor2 Int | NullaryDataCtor
```

```
-- большая вложенность типов:
```

```
value :: TypeCtor Char (TypeCtor [()] (Int, Char))  
value = BinaryDataCtor 'a' (UnaryDataCtor1 (1, '!'))
```

```
-- конструктор типа как аналог ФВП:
```

```
data TypeApply f x = Wrap (f x)
```

```
xs :: TypeApply [] Int
```

```
xs = Wrap [1, 2, 3]
```

Паттерн матчинг (сравнение с образцом)

Функция – набор уравнений:

```
fac 0 = 1           -- 0 – как конструктор данных
fac n = n * fac (n-1) -- n – как переменная
```

Образцы проверяются сверху вниз:

```
badFac n = n * fac (n-1) -- подойдёт всегда
badFac 0 = 1             -- никогда не подойдёт
```

Если образцов недостаточно, ошибка исполнения:

```
dullFac 0 = 1
dullFac 1 = 1
dullFac 2 = 2
dullFac 3 = 6

dullFac 4 -- ошибка
```

Паттерн матчинг

Можно указывать сконструированные значения

```
data Number = One | Many Integer
```

```
negate :: Number -> Number
```

```
negate One = Many (-1)
```

```
negate (Many (-1)) = One
```

```
negate (Many 1) = error "Bad number: Many 1"
```

```
negate (Many n) = Many (-n)
```

```
small :: Number -> Bool
```

```
small One = True
```

```
small _ = True -- nobody cares
```

```
hasZero :: (Int, Int) -> Bool
```

```
hasZero (0, _) = True
```

```
hasZero (_, 0) = True
```

```
hasZero (_, _) = False -- hasZero _ = False
```

Паттерн матчинг – особенности

С маленькой буквы – переменная, с большой – значение:

```
data Number = One | Many Integer
```

```
a = 10
```

```
number One = 1
```

```
number (Many a) = 2 -- не сработает как "10"
```

```
number (Many _) = 3 -- сюда не дойдёт
```

Функция не пройдёт, только конструктор данных:

```
eq5 = (== 5) -- сечение оператора "=="
```

```
isFive (eq5 x) = True -- ошибка компиляции
```

```
isFive _ = False
```

```
isFive x | eq5 x = True -- так можно  
         | otherwise = False
```


Род (kind) – вверх по абстракции

```
data Number a = One | Many a
data Point = Point Int Int    -- одно имя – можно
```

Значение имеет тип (type):

```
Point 2 3 :: Point
```

```
One :: Number a
```

```
Many 'x' :: Number Char
```

Конструктор типов имеет род (kind):

```
Point :: *
```

```
Number :: * -> *
```

```
Number Char :: *
```

Тип любого сконструированного значения имеет kind *. В GHCi :t smth – узнать тип, :k smth – узнать род.

Классы типов – мотивация

Модель из жизни:

- **Числа** можно складывать, вычитать, умножать, ...
- **Сравниваемые объекты** можно проверять на равенство
- **Упорядочиваемые объекты** можно сравнивать

Хотелось бы объявить операторы для чисел, но:

- $1, 2, 3 \in Z$ – числа
 - $1.0, 2.0, 3.0 \in R$ – тоже числа
 - $\lambda f.\lambda x.x, \lambda f.\lambda x.f\ x, \lambda f.\lambda x.f\ (f\ x), \dots$ – странные, но числа
- и все проявляются чуть по-разному.

Классы типов – мотивация

В ЯП с ООП написали бы интерфейсы, от которых нужные типы бы унаследовались:

```
interface Num {  
    Num sum(Num rhs); // lhs = this  
    Num sub(Num rhs);  
    Num prod(Num rhs);  
}
```

```
interface Eq {  
    bool equals(Eq rhs);  
}
```

```
enum Ordering { LT, EQ, GT }
```

```
interface Ord {  
    Ordering compare(Ord rhs);  
    bool less(Ord rhs);  
}
```

Классы типов

В Haskell сделали классы типов:

```
class Num a where           -- К.Т. ЧИСЛО  
  (+)  :: a -> a -> a  
  (-)  :: a -> a -> a  
  (*)  :: a -> a -> a  
  negate :: a -> a  
  abs   :: a -> a  
  signum :: a -> a  
  fromInteger :: Integer -> a
```

Можно читать как *некоторый тип **a** будет относиться к классу типов **Num**, если над ним определить функции (+), (-), (*)...*

Кстати, $(+) \ a \ b \equiv a + b$, и ещё $f \ a \ b \equiv a \ `f` \ b$.

Класс типов можно указать как контекст в функциях:

```
f :: Num a => a -> a -- для любого типа a, который Num  
f x = x + x
```

Классы типов – ещё примеры из std. библиотеки

-- сравниваемое

class Eq a **where**

(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool

data Ordering = LT | EQ | GT

-- упорядочиваемое

class Eq a => Ord a **where** *-- требуется сравниваемость*

compare :: a -> a -> Ordering

(<) :: a -> a -> Bool

(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

(>=) :: a -> a -> Bool

max :: a -> a -> a

min :: a -> a -> a

-- преобразуемое в строку

class Show a **where**

showsPrec :: Int -> a -> ShowS

show :: a -> String

showList :: [a] -> ShowS

Экземпляры/воплощения классов типов

```
data Number = One | Many Integer

instance Eq Number where -- Number вместо a
    One == One = True
    One == (Number 1) = True
    (Number 1) == One = True
    (Number a) == (Number b) = a == b
    _ == _ = False
-- /= будет выведен сам!
```

- Есть минимальное полное определение (для Eq – (==) или (/=)), не нужно расписывать то, что выражается само
- Экземпляры классов типов в коде отделены от типов (как extension methods в C#)
- Можно указать нужный контекст:

```
instance Num a => Eq a where -- для чисел
instance Eq a => Eq (List a) where -- для спис. срвн-ых
```

Классы типов – “общий случай”

```
-- Tuple3 :: * -> * -> * -> *
-- deriving - когда реализация тривиальна
data Tuple3 a b c = T3 a b c deriving (Eq, Show)

-- mul требует t :: * -> *
class Multipliable t where
    mul :: Num n => t n -> n -> t n

-- делаем * -> * из Tuple3 частичным применением
-- Eq a, Eq b - демонстрация сложного контекста
instance (Eq a, Eq b) => Multipliable (Tuple3 a b) where
    (T3 x y z) `mul` n = T3 x y (z * n)
```

Списки

- Список – это голова плюс хвост.
- Конец списка – пустота.
- Конструктор списка соединяет голову и хвост.

```
template <typename Data>
struct List {
    List(Data* head, List* tail):
        head(head), tail(tail) {}
    Data* head;
    List* tail;
};

int i=3, j=2, k=1;
List<int> xs1(&i, nullptr);
List<int> xs2(&j, &xs1);
List<int> xs(&k, &xs2); // xs = {1, 2, 3}
```


Списки

- Список – это голова плюс хвост.
- Конец списка (`[]`) – пустотой список.
- Конструктор списка (`:`) соединяет голову и хвост.

```
xs :: [Int]
xs = 1 : 2 : 3 : []
```

```
xs' :: [Int]
xs' = [1, 2, 3] -- синтаксический сахар
```

Циклы? Рекурсия.

```
function reverse(xs) {  
  var ys = Array(xs.length);  
  
  for(var i=0; i<xs.length; ++i)  
    ys[i] = xs[xs.length - i - 1];  
  
  return ys;  
}
```

Циклов в ФП нет, поэтому используется рекурсия:

```
const reverse = (xs) => xs.length ?  
  reverse(xs.slice(1)).concat([xs[0]]) :  
  [];
```

Haskell:

```
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```

Работа со списками

Паттерн матчинг (конструкторы `[]` и `:` в образцах):

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

Ручной обход:

```
listMul2 :: Num a => [a] -> [a]
listMul2 []      = []
listMul2 (x:xs) = x * 2 : listMul2 xs
```

Функции из Prelude (std.библиотека) или Data.List: head, tail, last, init, null, map, (++) (конкатенация), filter, length, (!!) (кв.скобки), reverse, take, drop, splitAt, zip, intersperse, ...

Свёртки и отображения – замена циклам

`foldl/foldr` – левая/правая свёртка

`foldl1, foldr1` – крайний элемент – начальное значение

`map` – отображение списка

```
foldr  :: (a -> b -> b) -> b -> [a] -> b
foldl  :: (b -> a -> b) -> b -> [a] -> b
foldr1 :: (a -> a -> a) -> [a] -> a
foldl1 :: (a -> a -> a) -> [a] -> a
map     :: (a -> b) -> [a] -> [b]
```

```
sum :: [a] -> a
sum xs = foldl1 (+) xs
-- или sum xs = foldr1 (+) xs
```

`Array.prototype.reduce` в ECMAScript – аналог `foldl, foldl1`; `Array.prototype.map` – аналог `map`.

На самом деле, `fold*` уже в классе типов `Foldable`, и применять их можно не только к списку.

Бесточечный стиль

$\lambda x.f\ x \equiv f$

В выражении $\lambda x.f\ x$ есть точка, в выражении f – нет.

Отсюда пошёл **бесточечный** стиль в Haskell.

```
sum :: [a] -> a
-- sum xs = foldr (+) xs
sum = foldr (+)                                -- бесточечный стиль

listMul2 :: Num a => [a] -> [a]
-- listMul2 xs = map (*2) xs
listMul2 = map (*2)                            -- бесточечный стиль
```

Бесточечному стилю потворствуют каррирование, сечения (напр. $(+2)$), множество вспомогательных функций и операторов для разного рода композиций функций.

Свёртки – это сила

Свёртка может посчитать какое-то интегральное значение:

```
product :: [a] -> a  
product = foldr1 (*)
```

Свёртка может и оставить список после себя:

```
copy :: [a] -> [a]  
copy = foldr (:) []
```

```
flip f x y = f y x -- обмен аргументов местами
```

```
reverse :: [a] -> [a]  
reverse = foldl (flip (:)) []
```

Свёртки – это сила

fold мощнее map:

```
map f = foldr ((:) . f) []
```

Пояснения:

```
f . g = \x -> f (g x) -- КОМПОЗИЦИЯ
```

```
((:) . f) x = ((:) . f) x = (:) (f x) = (f x :)  
((:) . f) x xs = (f x :) xs = f x : xs
```

xs – обработанная правая часть, x – чуть более левый элемент.

Соответственно,

```
listMul2 :: Num a => [a] -> [a]  
listMul2 = foldr ((:) . (*2)) []
```

Ленивые вычисления

Энергичные/строгие вычисления – сначала аргументы:

```
int second(int x, int y) {  
    return y;  
}  
  
int main () {  
    // тормозит, вычисляя fac(1e10)  
    printf("%d", second(fac(1e10), fac(5)));  
}
```

Ленивые вычисления – когда понадобится – тогда и считаем:

`second x y = y`

-- считает fac 5 потому, что его просили вывести
`main = print (second (fac (10^10)) (fac 5))`

Ленивые вычисления

- Всё вычисляется, когда *нужно*
 - *Нужно* – значит когда значение выходит из математического мира в физический (напр. печать)
 - Обычное значение и нуль-арная функция становятся эквивалентными
 - Для вычисления в памяти могут копиться незавершённые вычисления
 - Кстати, есть ещё и ленивое сопоставление с образцом кроме обычного
- $$f \sim (x:xs) = x:xs \equiv f \ ys = \text{head } ys : \text{tail } ys$$

Бесконечные структуры данных

Ленивого конструктора списков достаточно для создания бесконечного списка.

Бесконечный список единиц:

```
ones = 1 : ones
```

Зацикливание списка:

```
cycle x = x : cycle x
```

```
ones = cycle 1
```

В Haskell можно задавать списки через диапазоны: $[1..3] \equiv [1,2,3]$, $[1,3..5] \equiv [1,3,5]$.

Встроенный бесконечный список:

```
nats = [1..]
```

Бесконечные структуры данных

Обычное бинарное дерево:

```
data Tree a = Node (Tree a) a (Tree a) | Leaf a
```

Может стать бесконечным:

```
--          1
--        1   1
--      1 1 1 1
ones = Node ones 1 ones
```

```
--          0
--       0.1   0.2
-- 0.11 0.12 0.21 0.22
nums = nums' 0 1 where
  nums' x k = Node l x r where
    l = nums' (x + k') k'
    r = nums' (x + 2*k') k'
    k' = k / 3
```

-- числа на $[0; 1]$ в троичной
-- почти континуум

Популярное определение списка чисел Фибоначчи:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Просто рекурсивная нуль-арная функция функция, подслащенная ленивыми вычислениями.

В физический мир отправляют, разумеется, подписание:

```
fibSquares = map (^2) fibs
fibSquaresPlusOne = zipWith (+) ones fibSquares

main = print . take 10 $ fibSquaresPlusOne
-- [1,2,2,5,10,26,65,170,442,1157]
```

Бесконечные списки и свёртки

$$\text{foldl}(f, a, x) = f(f(f(a, x_1), x_2), x_3) = f(a', x_{last})$$

`foldl` – это `foldl`: уравнение вида `foldl _ _ _ = foldl _ _ _`

$$\text{foldr}(f, a, x) = f(x_1, f(x_2, f(x_3, a))) = f(x_1, a')$$

`foldr f` – это `f`: уравнение вида `foldr f _ _ = f _ _`

Если в `f` не использовать аргумент, который требует бесконечной рекурсии, можно обработать бесконечный список с помощью `foldr`:

```
takeWhile f xs :: (a -> Bool) -> [a] -> [a]
takeWhile f = foldr func [] where
  func x xs = if f x then x:xs else []
```

```
takeWhile (<10) [1..] -- [1,2,3,4,5,6,7,8,9]
```

Ленивость, БСД и мир

Дополнительная абстракция:

```
const ones = {  
  head: () => 1,  
  tail: () => ones  
};
```

Генераторы:

```
function* ones() {  
  while(1) yield 1;  
}  
  
var i = 0;  
for(var x of ones()) {  
  if(++i > 10) break;  
  console.log(x);  
}
```

Ленивость, БСД и мир

Ссылки, самоссылки:

```
var ones = {head: 1};  
ones.tail = ones;
```

Геттеры:

```
const ones = {  
  get head(){ return 1; },  
  get tail(){ return ones; }  
};
```

Проперти:

```
var ones = {};  
Object.defineProperty(ones, 'head', {get: () => 1});  
Object.defineProperty(ones, 'tail', {get: () => ones});
```

Задачи

Знаний достаточно для реализации следующих задач:

- Задача 1 о реализации
 - `filter`, `map`, `foldl`
 - и `fac` через свёртку бесконечного списка
- Задача 2 о бесконечном списке на JS

Внимание, лекции, задания и сниппеты обновляются.