Курс: Функциональное программирование

Лекция 7. Аппликативные функторы и свёртки

Денис Николаевич Москвин

11.11.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

План лекции

- Свёртки
- Аппликативные функторы
- Моноиды

План лекции

- Свёртки
- Аппликативные функторы
- Моноиды

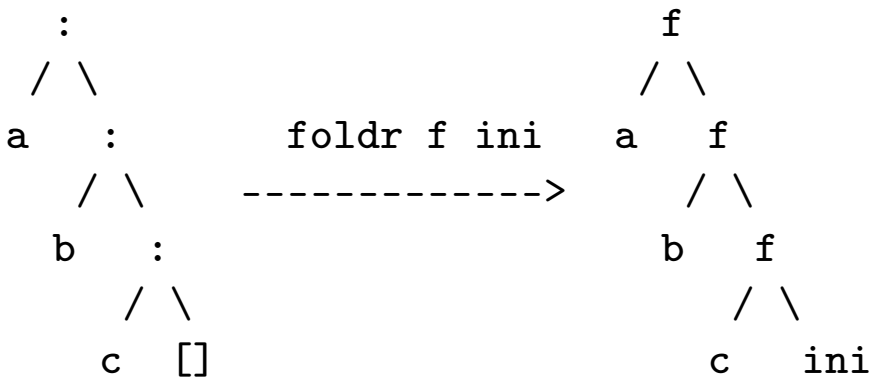
Свёртки

```
sum :: [Integer] -> Integer
sum []     = 0
sum (x:xs) = x + sum xs

product :: [Integer] -> Integer
product []     = 1
product (x:xs) = x * product xs

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

Виден общий паттерн рекурсии.

Правая свёртка

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

a : b : c : [] ------> f a (f b (f c ini))
```



Свёртки через `foldr`

```
sum :: [Integer] -> Integer
sum = foldr (+) 0

product :: [Integer] -> Integer
product = foldr (*) 1

concat :: [[a]] -> [a]
concat = foldr (++) []
```

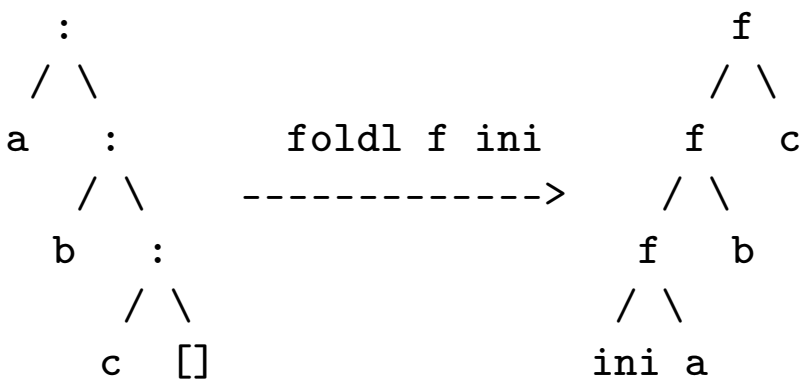
А что получится из

```
foldr (:) []
```

?

Левая свёртка

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

a : b : c : [] ------> f (f (f ini a) b) c
```



Рекурсия хвостовая — оптимизируется. Однако thunk из цепочки `f` нарастает.

Строгая версия левой свёртки

```
foldl'                :: (a -> b -> a) -> a -> [b] -> a
foldl' f ini []       =  ini
foldl' f ini (x:xs)   =  ini' `seq` foldl' f ini' xs
                           where ini' = f ini x
```

Теперь thunk из цепочки аппликаций `f` не нарастает — `ini'` вычисляется на каждом шаге.

Это самая эффективная из свёрток, но все левые свёртки не умеют работать с бесконечными списками!

Ленивость правой свёртки

```
any    :: (a -> Bool) -> [a] -> Bool
any p  =  foldr (\x b -> p x || b) False
```

Правая свёртка на каждом шаге «даёт поработать» используемой функции

```
any (==2) [1..] ~>
foldr (\x b -> (==2) x || b) False (1:[2..]) ~>
(\x b -> (==2) x || b) 1 (foldr (\x b -> (==2) x || b) False [2..]) ~>
False || (foldr (\x b -> (==2) x || b) False [2..]) ~>
foldr (\x b -> (==2) x || b) False 2:[3..] ~>
True || (foldr (\x b -> (==2) x || b) False [3..]) ~>
True
```

Версии свёрток без начального значения

```
foldr1             :: (a -> a -> a) -> [a] -> a
foldr1 _ [x]       =  x
foldr1 f (x:xs)    =  f x (foldr1 f xs)
foldr1 _ []        =  error "foldr1: EmptyList"
```

Имеются аналогичные `foldl1` и `foldl1'`.

План лекции

- Свёртки
- Аппликативные функторы
- Моноиды

Функторы

Функторы позволяют «поднять стрелку в контейнер»

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

Представители `f` должны быть однопараметрическими конструкторами типа, то есть `f :: * -> *`.

Представители функторов

```
instance Functor ((->) r) where
  fmap = (.)

instance Functor ((,) a) where
  fmap g (x,y) = (x, g y)

instance Functor (Either a) where
  fmap _ (Left x)  = Left x
  fmap g (Right y) = Right (g y)

data Tree a = Leaf a | Branch (Tree a) a (Tree a)

instance Functor Tree where
  fmap g (Leaf x)       = Leaf (g x)
  fmap g (Branch l x r) = Branch (fmap g l) (g x) (fmap g r)
```

Законы для функторов

Для любого представитель класса типов `Functor` должно выполняться

```
      fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Это так для списков, `Maybe`, `IO` и т.д.

Смысл законов: вызов `fmap g` не должен менять «структуру контейнера», воздействуя только на его элементы.

Всегда ли эти законы выполняются?

Законы для функторов: контрпример

«Плохой» представитель класса `Functor` для списка

```
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : g x : fmap g xs
```

Какой закон нарушается для такого объявления представителя и почему?

Pointed: класс типов, которого нет

Даёт возможность «вложить значение в контекст»

```
class Functor f => Pointed f where
  pure :: a -> f a -- aka singleton, return, unit, point

instance Pointed Maybe where
  pure x  = Just x

instance Pointed [] where
  pure x  = [x]

instance Pointed (Either l) where
  pure   =
```

Класс `Pointed` (2)

```
class Functor f => Pointed f where
  pure :: a -> f a
```

Всегда ли возможно объявления представителя для `Pointed`?

```
instance Pointed ((->) r) where
  pure  =

instance Pointed ((,) e) where
  pure  =

instance Pointed Tree where
  pure  =
```

Закон для класса `Pointed` один: `fmap g . pure = pure . g`

Аппликативные функторы

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

infixl 4 <*>
```

Функция `pure` обсуждалась выше; если бы `Pointed` существовал, то определение было бы таким

```
class Pointed f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

Оператор `(<*>)` — это, фактически, `($)` но в «вычислительном контексте», задаваемым функтором.

Аппликативные функторы: объявление представителя

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _  = Nothing
  (Just f) <*> x  = fmap f x
```

Теперь можем работать в вычислительном контексте с возможно отсутствующим значением:

```
*Fp072> (Just (*2)) <*> Just 5
Just 10
*Fp072> (Just (*2)) <*> Nothing
Nothing
*Fp072> Nothing <*> Just 5
Nothing
```

Аппликативные функторы: законы

```
pure id <*> v = v                            -- Identity
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
pure g <*> pure x = pure (g x)               -- Homomorphism
g <*> pure x = pure ($ x) <*> g              -- Interchange
```

Закон, связывающий `Applicative` и `Functor`

```
fmap g x = pure g <*> x
```

В `Control.Applicative` введён инфиксный синоним `g <$> x = fmap g x`

```
g <$> x = pure g <*> x
```

Аппликативные функторы: мотивация

Рассмотрим список функций и список значений

```
fs = [\x->2*x, \x->3+x, \x->4-x]
as = [1,2]
```

Каким смыслом можно наделить аппликацию `fs <*> as`?
Двумя разными!

▶ Список — это коллекция упорядоченных элементов.

```
fs <*> as == [(\x->2*x) 1, (\x->3+x) 2] == [2,5]
```

▶ Список — контекст, задающий множественные результаты недетерминистического вычисления.

```
fs <*> as == [(\x->2*x) 1, (\x->2*x) 2, (\x->3+x) 1, (\x->3+x) 2,
              (\x->4-x) 1, (\x->4-x) 2] == [2,4,4,5,3,2]
```

Список как коллекция элементов

Два представителя для одного типа недопустимы — переупаковываем список:

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (map f xs)

instance Applicative ZipList where
  pure x = ???
  ZipList gs <*> ZipList xs = ZipList (zipWith ($) gs xs)
```

```
*Fp072> getZipList $ ZipList fs <*> ZipList as
[2,5]
```

Список как результат многозначного вычисления

Оператор `(<*>)` в этом случае должен реализовывать модель «каждый с каждым»:

```
instance Applicative [] where
  pure x = [x]
  gs <*> xs = [ g x | g <- gs, x <- xs ]

fs = [\x->2*x, \x->3+x, \x->4-x]
as = [1,2]

*Fp072> fs <*> as
[2,4,4,5,3,2]
```

Расширение на большее число «аргументов»

Можно строить цепочки аппликаций любой длины, важно чтобы крайний левый аппликанд имел подходящую арность:

```
u              :: f (a -> b -> c)
v              :: f a
u <*> v        :: f (b -> c)

w              :: f b
u <*> v <*> w  :: f c

*Fp072> pure (*) <*> Just 7 <*> Just 6
Just 42
*Fp072> pure (+) <*> [20,30] <*> [4,5,6]
[24,25,26,34,35,36]
*Fp072> (+) <$> [20,30] <*> [4,5,6]
[24,25,26,34,35,36]
```

Пример: линейная комбинация трёх векторов

```
*Fp072> let f x y z = 3*x+2*y-7*z
*Fp072> let xz = ZipList [2,7,5]
*Fp072> let yz = ZipList [1,-4,3]
*Fp072> let zz = ZipList [5,0,1]
*Fp072> getZipList $ f <$> xz <*> yz <*> zz
[-27,13,14]
```

Удобная замена семейству функций `zipWith`, `zipWith3`, `zipWith4`...

```
*Fp072> zipWith3 f [2,7,5] [1,-4,3] [5,0,1]
[-27,13,14]
```

План лекции

- Свёртки
- Аппликативные функторы
- Моноиды

Определение моноида

Моноид — это множество с ассоциативной бинарной операцией над ним и единицей для этой операции.

```
class Monoid a where
  mempty  :: a             -- единица
  mappend :: a -> a -> a   -- операция

  mconcat :: [a] -> a      -- свёртка
  mconcat = foldr mappend mempty
```

Для любого моноида должны безусловно выполняться законы:

```
mempty `mappend` x == x
x `mappend` mempty == x
(x `mappend` y) `mappend` z == x `mappend` (y `mappend` z)
```

Реализация представителей моноида

Список — моноид относительно конкатенации `(++)`, единица — это пустой список.

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

А числа?
Да, причём дважды: относительно сложения (единица это 0) и относительно умножения (единица это 1).

Реализация представителей моноида: числа

```
newtype Sum a = Sum { getSum :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x `mappend` Sum y = Sum (x + y)

newtype Product a = Product { getProduct :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
  mempty = Product 1
  Product x `mappend` Product y = Product (x * y)

*Fp072> Product 3 `mappend` Product 2
Product {getProduct = 6}
```

Реализация представителей моноида: `Bool`

`Bool` — моноид относительно конъюнкции и дизъюнкции.

```
newtype All = All { getAll :: Bool }  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid All where
        mempty = ????
        All x `mappend` All y = All (x && y)

-- | Boolean monoid under disjunction.
newtype Any = Any { getAny :: Bool }  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
        mempty = ????
        Any x `mappend` Any y = Any (x || y)
```

Какова должна быть реализация для единицы?

Использование моноидов

```
instance Monoid e => Applicative ((,) e) where
  pure x             = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

```
*Fp072> ("Answer to ",(*)) <*> ("the Ultimate ",6) <*> ("Question",7)
("Answer to the Ultimate Question",42)
```