

Функциональное программирование

Лекция 7. Свёртки

Денис Николаевич Москвин

СПбАУ РАН, CSC

26.03.2015

- 1 Свёртки
- 2 Моноиды
- 3 Класс типов Foldable
- 4 Свойство слияния для `foldr`

- 1 Свёртки
- 2 Моноиды
- 3 Класс типов `Foldable`
- 4 Свойство слияния для `foldr`

```
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)
```

```
p : q : r : []      ----->      f p (f q (f r ini))
```

```
      :
     / \
    p  :
       / \
      q  :
         / \
        r []

      foldr f ini
      ----->
      f
     / \
    p  f
       / \
      q  f
         / \
        r ini
```

Конкретные свёртки через foldr

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

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

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

А что получится в результате такой свёртки?

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

Свойство универсальности

Если функция g удовлетворяет системе уравнений

$$g [] = v$$

$$g (x:xs) = f x (g xs)$$

то

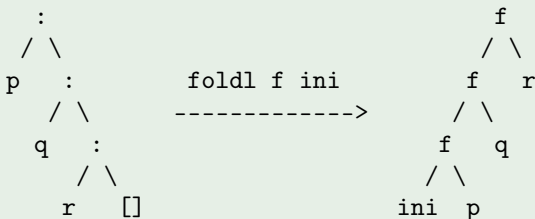
$$g = \text{foldr } f \ v$$

- Доказывается простой индукцией.
- Практический смысл в том, что `foldr` является *единственным* решением системы.
- Обратное утверждение тривиально, поскольку прямо следует из определения `foldr`.

Левая свёртка

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

$p : q : r : [] \quad \text{-----} \rightarrow \quad f (f (f \text{ ini } p) q) r$



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

Строгая версия левой свёртки

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

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

- Теперь thunk из цепочки вызовов f **не** нарастает — вычисление arg форсируется на каждом шаге.
- Это самая эффективная из свёрток, но все левые свёртки не умеют работать с бесконечными списками.

«Продуктивность» 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
```

Для непустых списков можно обойтись без инициализатора:

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

```
foldl1      :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
foldl1 _ []     = error "foldl1: EmptyList"
```

Аналогично реализована строгая версия `foldl1'`.

Представляют собой списки последовательных шагов свёртки.

```
scanl f z [a, b, ...] ≡ [z, z 'f' a, (z 'f' a) 'f' b, ...]
```

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

GHCi

```
Prelude> scanl (++) "!" ["a","b","c"]
["!","!a","!ab","!abc"]
Prelude> scanl (*) 1 [1..] !! 5
120
```

Можно и с бесконечными списками (в отличие от `foldl`).

Правый скан накапливает результаты справа налево.

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

GHCi

```
Prelude> scanr (++) "!" ["aa","bb","cc"]  
["aabbcc!", "bbcc!", "cc!", "!"]  
Prelude> scanr (:) [] [1,2,3]  
[[1,2,3], [2,3], [3], []]
```

Для сканов выполняются следующие тождества

```
head (scanr f z xs)  ≡ foldr f z xs  
last (scanl f z xs) ≡ foldl f z xs
```

Операция двойственная к свёртке.

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

GHCi

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

Пример использования (возможное определение `iterate`)

```
iterate f = unfoldr (\x -> Just (x, f x))
```

- 1 Свёртки
- 2 Моноиды
- 3 Класс типов Foldable
- 4 Свойство слияния для foldr

Определение моноида

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

```
class Monoid a where
  mempty  :: a           -- нейтральный элемент
  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 (`++`),
нейтральный элемент — это пустой список.

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

Что такое `mconcat` для списков?

Реализация представителей моноида

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

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

Что такое `mconcat` для списков?

Свёртка конкатенацией!

```
mconcat :: [[a]] -> [a]
mconcat = concat
```

Реализация представителей моноида

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

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

Что такое `mconcat` для списков?

Свёртка конкатенацией!

```
mconcat :: [[a]] -> [a]
mconcat = concat
```

А числа — моноид?

Реализация представителей моноида

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

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

Что такое `mconcat` для списков?

Свёртка конкатенацией!

```
mconcat :: [[a]] -> [a]
mconcat = concat
```

А числа — моноид?

Да, причём дважды: относительно сложения (нейтральный элемент это 0) и относительно умножения (нейтральный элемент это 1).

Числа как моноид относительно сложения

```
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)
```

GHCi

```
*Fp07> Sum 3 `mappend` Sum 2
Sum {getSum = 5}
```

Что такое `mconcat` для `Sum a`?

Числа как моноид относительно умножения

```
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)
```

GHCi

```
*Fp07> Product 3 `mappend` Product 2
Product {getProduct = 6}
```

Что такое `mconcat` для `Product a`?

Реализация представителей моноида: Bool

Булев тип — 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)
```

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

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

Каковы должны быть реализации для нейтральных элементов?

- 1 Свёртки
- 2 Моноиды
- 3 Класс типов Foldable**
- 4 Свойство слияния для `foldr`

Минимальное полное определение: `foldMap` или `foldr`.

```
class Foldable t where
  fold :: Monoid m => t m -> m
  fold = foldMap id

  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap f = foldr (mappend . f) mempty

  foldr :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl :: (a -> b -> a) -> a -> t b -> a
  foldl f z t =
    appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z

  foldr1, foldl1 :: (a -> a -> a) -> t a -> a
```

```
instance Foldable [] where
  foldr = Prelude.foldr
  foldl = Prelude.foldl
  foldr1 = Prelude.foldr1
  foldl1 = Prelude.foldl1
```

```
instance Foldable Maybe where
  foldr _ z Nothing = z
  foldr f z (Just x) = f x z

  foldl _ z Nothing = z
  foldl f z (Just x) = f z x
```

А также Set из Data.Set, Map k из Data.Map, Seq из Data.Sequence, Tree из Data.Tree и т.п.

- 1 Свёртки
- 2 Моноиды
- 3 Класс типов `Foldable`
- 4 Свойство слияния для `foldr`

Как «протащить» функцию через `foldr`?

Рассмотрим равенство

```
h . foldr g w = foldr f v
```

Какими свойствами должны обладать входящие в него функции, чтобы это равенство выполнялось?

Как «протащить» функцию через `foldr`?

Рассмотрим равенство

```
h . foldr g w = foldr f v
```

Какими свойствами должны обладать входящие в него функции, чтобы это равенство выполнялось?

Воспользуемся свойством универсальности

```
(h . foldr g w) [] = v  
(h . foldr g w) (x:xs) = f x ((h . foldr g w) xs)
```

Свойство слияния для foldr (foldr fusion)

Первое равенство даст

$$(h \ . \ foldr \ g \ w) \ [] = v \Leftrightarrow h \ (foldr \ g \ w \ []) = v \Leftrightarrow h \ w = v$$

Второе равенство даст

$$\begin{aligned}(h \ . \ foldr \ g \ w) \ (x:xs) &= f \ x \ ((h \ . \ foldr \ g \ w) \ xs) \\ \Leftrightarrow h \ (foldr \ g \ w \ (x:xs)) &= f \ x \ (h \ (foldr \ g \ w \ xs)) \\ \Leftrightarrow h \ (g \ x \ (foldr \ g \ w \ xs)) &= f \ x \ (h \ (foldr \ g \ w \ xs)) \\ \Leftarrow h \ (g \ x \ y) = f \ x \ (h \ y)\end{aligned}$$

Итак, имеем *свойство слияния* для foldr

foldr fusion property

$$h \ (g \ x \ y) = f \ x \ (h \ y) \quad \Rightarrow \quad h \ . \ foldr \ g \ w = foldr \ f \ (h \ w)$$

Свойство слияния для ассоциативного оператора

Положим в свойстве слияния $f=g=(\otimes)$ и $h=(\otimes z)$. Тогда условие $h (g x y) = f x (h y)$ превратится в

$$\begin{aligned}(\otimes z) ((\otimes) x y) &= (\otimes) x ((\otimes z) y) \\ \Leftrightarrow (\otimes z) (x \otimes y) &= x \otimes ((\otimes z) y) \\ \Leftrightarrow (x \otimes y) \otimes z &= x \otimes (y \otimes z)\end{aligned}$$

Для любого ассоциативного оператора \otimes свойство слияния выполняется безусловно и имеет вид

Свойство слияния для ассоциативного оператора

$$(\otimes z) . \text{foldr } (\otimes) w = \text{foldr } (\otimes) (w \otimes z)$$

$$(+42) . \text{sum} = \text{foldr } (+) 42$$

Если стартовать с равенства

```
foldr g w . map h = foldr f v
```

то окажется, что безо всяких дополнительных условий на пустых списках получится $v = w$, а непустые дадут $f = g . h$.

foldr-map fusion property

```
foldr g w . map h = foldr (g . h) w
```