Функциональное программирование
Лекция 7. Свертки

Денис Николаевич Москвин

ИТМО, магистратура JB SE

19.10.2020

План лекции

1. Свертки
2. Полугруппы и моноиды
3. Класс типов `Foldable`

План лекции

Наблюдение

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

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

```
allOdd :: [Integer] -> Bool
allOdd []     = ???
allOdd (x:xs) = odd x && allOdd xs
```

Виден общий шаблон рекурсии.

Правая свертка

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

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

Конкретные свертки через foldr

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

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

```haskell
allOdd :: [Integer] -> Bool
allOdd = foldr (\n b -> odd n && b) True
```

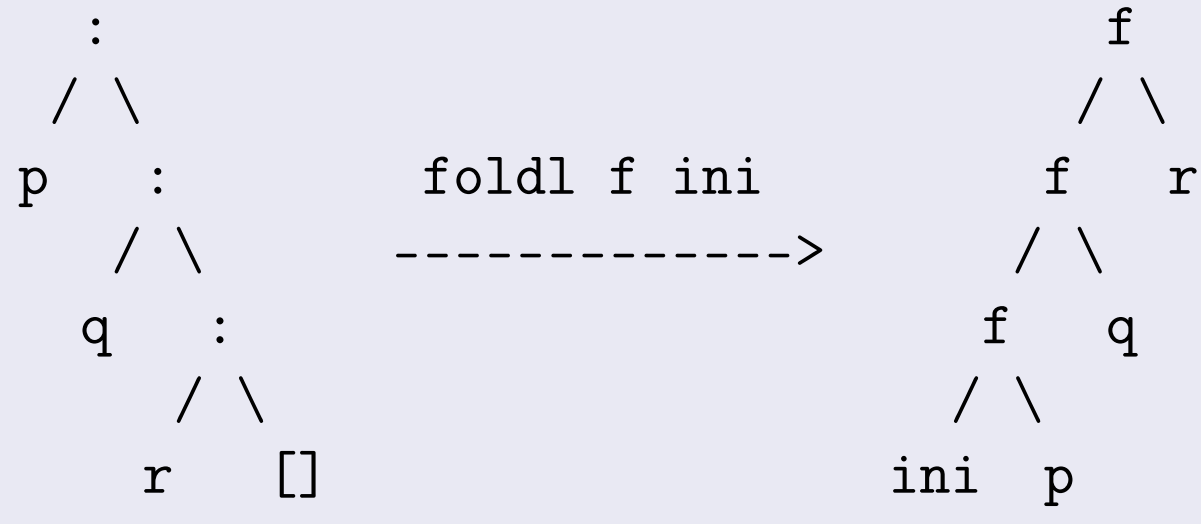
А что получится в результате такой свертки?

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

Левая свертка

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

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



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

Строгая версия левой свертки

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

```
foldl'              :: (b -> a -> b) -> b -> [a] -> b
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 сверток, но все левые свертки не умеют работать с бесконечными списками.

«Продуктивность» правой свертки

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

Правая свертка на каждом шаге «дает 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 сверток без начального значения

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'`.

Сканы

Представляют собой списки последовательных шагов свертки.

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

```
scanl          :: (b -> a -> b) -> b -> [a] -> [b]
scanl  _  z []     =  [z]
scanl (#) z (x:xs) =  z : scanl (#) (z # x) xs
```

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

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

Правый скан

Правый скан накапливает результаты справа налево.

```
scanr               :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ z []        =  [z]
scanr f z (x:xs)    =  f x q : qs
                       where qs@(q:_) = scanr f z xs
```

```
GHCi> scanr (+) 0 [1,2,3]
[6,5,3,0]
GHCi> scanr (++) "!" ["aa","bb","cc"]
["aabbcc!","bbcc!","cc!","!"]
```

Для сканов выполняются следующие тождества

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

Развертка

Операция двойственная к свертке.

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr g ini
  | Nothing    <- next = []
  | Just (a,b) <- next = a : unfoldr g b
  where next = g ini
```

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

Пример использования: возможное определение `iterate`

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

План лекции

1. Свертки
2. Полугруппы и моноиды
3. Класс типов `Foldable`

Определение полугруппы

Полугруппа — это множество с ассоциативной бинарной операцией над ним.

```
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a
```

Для любой полугруппы должен выполняться закон:

```
(x <> y) <> z  ≡  x <> (y <> z)
```

Список — полугруппа относительно конкатенации `(++)`.

```
instance Semigroup [a] where
  (<>) = (++)
```

Полное определение полугруппы

```haskell
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a
  sconcat :: NonEmpty a -> a
  stimes :: Integral b => b -> a -> a

infixr 5 :|
data NonEmpty a = a :| [a]
```

Используем не обычный список, потому что непонятно, что возвращать на пустом списке.

```
GHCi> import Data.List.NonEmpty
GHCi> sconcat $ "AB" :| ["CDE","FG"]
"ABCDEFG"
GHCi> stimes 5 "Ab"
"AbAbAbAbAb"
```

Определение моноида

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

```
class Semigroup a => Monoid a where
  mempty  :: a

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

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

Для любого моноида должны безусловно выполняться законы:

```
mempty <> x  ≡  x
x <> 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 Semigroup [a] where
  (<>) = (++)
instance Monoid [a] where
  mempty  = []
  mconcat = concat
```

Реализация представителей моноида

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

```
instance Semigroup [a] where
  (<>) = (++)
instance Monoid [a] where
  mempty  = []
  mconcat = concat
```

А числа — моноид?

Реализация представителей моноида

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

```
instance Semigroup [a] where
  (<>) = (++)
instance Monoid [a] where
  mempty  = []
  mconcat = concat
```

А числа — моноид?

Да, причем четырежды: относительно сложения (нейтральный элемент это 0), умножения (нейтральный элемент это ?), `min` (?) и `max` (?).

Числа как моноид относительно сложения

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

instance Num a => Semigroup (Sum a) where
  Sum x <> Sum y = Sum (x + y)

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
```

```
GHCi> Sum 3 <> Sum 2
Sum {getSum = 5}
```

Что такое `mconcat` для `Sum a`? `stimes` для `Sum a`?

Числа как моноид относительно умножения

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

instance Num a => Semigroup (Product a) where
  (<>) = coerce ((*) :: a -> a -> a)  -- Data.Coerce

instance Num a => Monoid (Product a) where
  mempty = Product 1
```

Для использования coerce из Data.Coerce нужно подключить расширение ScopedTypeVariables.

```
GHCi> Product 3 <> Product 2
Product {getProduct = 6}
```

Что такое mconcat для Product a? stimes для Product a?

Моноид относительно min

Моноид относительно min формируют не только числа:

```haskell
newtype Min a = Min { getMin :: a }
  deriving (Eq, Ord, Read, Show, Bounded)

instance Ord a => Semigroup (Min a) where
  (<>) = coerce (min :: a -> a -> a)
  stimes = stimesIdempotent

instance (Ord a, Bounded a) => Monoid (Min a) where
  mempty = maxBound
```

```
GHCi> Min "Hello" <> Min "Hi"
Min {getMin = "Hello"}
GHCi> mempty :: Min Int
Min {getMin = 9223372036854775807}
```

Что такое mconcat для Min a? stimes для Min a?

Моноид и полугруппа Min

```
GHCi> (getMin . mconcat . fmap Min)  [7,3,2,12] :: Int
2
GHCi> (getMin . mconcat . fmap Min)  [] :: Int
9223372036854775807
```

Некоторые типы данных не формируют моноида `Min`, оставаясь полугруппой:

```
GHCi> mempty :: Min Integer
<interactive>: error: No instance for (Bounded Integer)...
GHCi> (getMin . mconcat . fmap Min) ["Hello","Hi"]
<interactive>: error: No instance for (Bounded [Char])...
```

Это можно исправить перйдя от моноидальной `mconcat` к полугрупповой `sconcat` (и от списка к `NoEmpty`)

```
GHCi> (getMin . sconcat . fromList . fmap Min) ["Hello","Hi"]
"Hello"
```

Реализация представителей моноида: Bool

Булев тип — моноид относительно конъюнкции и дизъюнкции.

```
newtype All = All { getAll :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
instance Semigroup All where
  (<>) = coerce (&&)
```

```
newtype Any = Any { getAny :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
instance Semigroup Any where
  (<>) = coerce (||)
```

Каковы должны быть реализации моноида (то есть нейтральные элементы)?

План лекции

1. Свертки
2. Полугруппы и моноиды
3. Класс типов `Foldable`

Класс Foldable

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, foldr' :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl, 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
```

Класс Foldable (продолжение)

```haskell
class Foldable t where
  ...
  toList :: t a -> [a]

  null :: t a -> Bool
  null = foldr (\_ _ -> False) True

  length :: t a -> Int
  length = foldl' (\c _ -> c + 1) 0

  elem :: Eq a => a -> t a -> Bool

  maximum, minimum :: Ord a => t a -> a

  sum, product :: Num a => t a -> a
  sum = getSum . foldMap Sum
```

Представители класса Foldable

```
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.

Двухпараметрические представители Foldable

Чтобы объявить представителя для двухпараметрического типа данных, первый параметр нужно связать.

```
GHCi> foldr (+) 5 (Right 37)
42
GHCi> foldr (+) 5 (Left 37)
5
GHCi> foldr (+) 5 ("Answer",37)
42
```

```
GHCi> maximum (Right 37)
37
GHCi> maximum (Left 37)
*** Exception: maximum: empty structure
GHCi> maximum (100,42)
???
```

Законы Foldable

```
foldr f z t ≡  appEndo (foldMap (Endo . f) t ) z
foldl f z t ≡  appEndo
  (getDual (foldMap (Dual . Endo . flip f) t)) z
fold        ≡  foldMap id
length      ≡  getSum . foldMap (Sum . const 1)
sum         ≡  getSum . foldMap Sum
product     ≡  getProduct . foldMap Product
minimum     ≡  getMin . foldMap Min
maximum     ≡  getMax . foldMap Max
foldr f z   ≡  foldr f z . toList
foldl f z   ≡  foldl f z . toList
```

Если контейнер `t` не только `Foldable`, но и `Functor`, то

```
foldMap f ≡ fold . fmap f
foldMap f . fmap g ≡ foldMap (f . g)
```

Второе следует из первого благодаря закону `Functor`.

Обобщенные специальные свертки

Многие функции, исторически реализованные для списков, были обобщены до `Foldable`:

```
concat :: Foldable t => t [a] -> [a]
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]

and,or :: Foldable t => t Bool -> Bool
any,all :: Foldable t => (a -> Bool) -> t a -> Bool
maximumBy,minimumBy
    :: Foldable t => (a -> a -> Ordering) -> t a -> a

notElem :: (Foldable t, Eq a) => a -> t a -> Bool

find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```