

Курс: Функциональное программирование Практика 7. Полугруппы, моноиды, свертки

Полугруппы

- Определите значения выражений без GHCi и проверьте себя, запустив интерпретатор

```
stimes 5 "z"  
  
stimes (-1) "z"  
  
stimes 0 "z"
```

- Можно ли написать эффективный универсальный код, который подходит для реализации `stimes` и для `All` и для `Any`?

Моноиды

Эндоморфизм (стрелка из типа в него же) можно упаковать так

```
newtype Endom a = Endom { appEndom :: a -> a }
```

Эндоморфизм образует моноид относительно композиции. (В стандартной библиотеке такой тип-обертка называется `Endo`.)

- Напишите

```
instance Semigroup (Endom a) where  
  Endom f <> Endom g = undefined  
  
instance Monoid (Endom a) where  
  mempty = undefined
```

- Определите тип функции

```
fn = mconcat $ map Endom [(+5),(*3),(^2)]
```

и вычислите значение выражения

```
appEndom fn 2
```

- Найдите с помощью Google определение типа-обертки `Dual`, объясните семантику представителя моноида для него. Определите тип функции

```
fn' = mconcat $ map (Dual . Endom) [(+5),(*3),(^2)]
```

и вычислите значение выражения

```
appEndom (getDual fn') 2
```

- Можно ли написать представителя моноида для типа `Maybe a`? Сколько разных вариантов можно реализовать? Обязательно ли должно быть `mempty = Nothing`?

Свёртки

- Устно вычислите значения выражений и проверьте результат в GHCi:

```
foldl (/) 480 [3,2,5,2]
```

```
foldr (/) 2 [8,12,24,4]
```

- Используя `foldr1`, напишите функцию, конструирующую из списка строк строку, разделённую запятыми.

```
GHCi> f ["ab","cde","fgh"]  
"ab,cde,fgh"
```

► Напишите реализацию следующих функций стандартной библиотеки через свёртки `foldr` или `foldl` (можно использовать `foldr1` или `foldl1`):

```
or' :: [Bool] -> Bool
or' = fold? undefined undefined

length' :: [a] -> Int
length' = fold? undefined undefined

maximum' :: Ord a => [a] -> a
maximum' = fold? undefined undefined

head' :: [a] -> a
head' = fold? undefined undefined

last' :: [a] -> a
last' = fold? undefined undefined

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = fold? undefined undefined

map' :: (a -> b) -> [a] -> [b]
map' f = fold? undefined undefined
```

Иногда реализация через свертку требует некоторого трюка с дополнительным параметром:

```
take' :: Int -> [a] -> [a]
take' n xs = foldr step ini xs n
  where
    step :: a -> (Int -> [a]) -> Int -> [a]
    step x g 0 = []
    step x g n = x : g (n - 1)
    ini :: Int -> [a]
    ini = const []
```

Работает это так

```

take' 2 "abc" ~> -- take' def
foldr step ini "abc" 2 ~>> -- foldr def (3 times)
step 'a' (step 'b' (step 'c' ini)) 2 ~> -- step def (2)
'a' : ( (step 'b' (step 'c' ini)) (2 - 1) ) ~> -- step def (2)
'a' : 'b' : ( (step 'c' ini) 0 ) ~> -- step def (1)
'a' : 'b' : []

```

Foldr-map fusion law

Найдем какие требования нужно наложить на g , w , h , f и v , чтобы они удовлетворяли уравнению

```
foldr g w . map h ≡ foldr f v -- (Eqv)
```

(1) На пустом списке это дает

```

foldr g w (map h []) ≡ foldr f v [] -- map def
foldr g w [] ≡ foldr f v [] -- foldr def
w ≡ v -- (result 1)

```

(2) На непустом списке

```

foldr g w (map h (x:xs)) ≡ foldr f v (x:xs)
foldr g w (h x : map h xs) ≡ foldr f v (x:xs) -- map def
g (h x) (foldr g w (map h xs)) ≡ f x (foldr f v xs) -- foldr def
g (h x) (foldr f v xs) ≡ f x (foldr f v xs) -- (Eqv, as IH)
g (h x) ≡ f x
g . h ≡ f -- (result 2)

```

Получаем знаменитый foldr-map fusion law

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

Домашнее задание

► (1 балл) Используя `unfoldr`, реализуйте функцию, которая возвращает в обратном алфавитном порядке список символов, попадающих в

заданный парой диапазон. Попадание символа x в диапазон пары (a, b) означает, что $x \geq a$ и $x \leq b$.

```
revRange :: (Char,Char) -> [Char]
revRange = unfoldr fun

fun = undefined
```

► (1 балл) Напишите реализации функций из стандартной библиотеки `tails`, `inits` :: `[a] -> [[a]]` через свёртку `foldr`:

```
tails' :: [a] -> [[a]]
tails' = foldr fun ini
fun = undefined
ini = undefined

inits' :: [a] -> [[a]]
inits' = foldr fun' ini'
fun' = undefined
ini' = undefined
```

► (1 балл) Напишите две реализации функции обращения списка `reverse` :: `[a] -> [a]` — через свёртки `foldr` и `foldl`:

```
reverse' :: [a] -> [a]
reverse' = foldr fun' ini'
fun' = undefined
ini' = undefined

reverse'' :: [a] -> [a]
reverse'' = foldl fun'' ini''
fun'' = undefined
ini'' = undefined
```

► (2 балла) Напишите реализацию оператора «безопасного» поиска элемента списка по индексу (!!!) через `foldr`:

```
GHCi> [1..10] !!! 5
Just 6
GHCi> [1..10] !!! (-1)
Nothing
GHCi> [1..10] !!! 100
Nothing
```

Используйте технику дополнительного параметра:

```
(!!!) :: [a] -> Int -> Maybe a
xs !!! n = foldr fun ini xs n
fun = undefined
ini = undefined
```

► (2 балла) Напишите реализацию `foldl` через `foldr`:

```
foldl'' :: (b -> a -> b) -> b -> [a] -> b
foldl'' f v xs = foldr (fun f) ini xs v
fun = undefined
ini = undefined
```

(Используйте технику дополнительного параметра.)

► (3 балла) Для реализации свертки двоичных деревьев нужно выбрать алгоритм обхода узлов дерева (см., например, http://en.wikipedia.org/wiki/Tree_traversal).

Сделайте двоичное дерево

```
data Tree a = Nil | Branch (Tree a) a (Tree a)
  deriving (Eq, Show)
```

представителем класса типов `Foldable`, реализовав симметричную стратегию (in-order traversal). Реализуйте также три другие стандартные стратегии (pre-order traversal, post-order traversal и level-order traversal), упаковав дерево в типы-обертки

```
newtype Preorder a = Pre0 (Tree a) deriving (Eq, Show)
newtype Postorder a = Post0 (Tree a) deriving (Eq, Show)
newtype Levelorder a = Level0 (Tree a) deriving (Eq, Show)
```

и сделав эти обертки представителями класса `Foldable`.