CS314. Функциональное программирование

Лекция 10. Обобщённые вычисления

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

15 октября 2016 г.

Содержание

Содержание

Сорта типов (type kinds)

```
ghci> :k Int
Int :: *
ghci> :k Maybe
Maybe :: * -> *
ghci> :k Maybe Int
Maybe Int :: *
ghci> :k Either
Either :: * -> * -> *
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
ghci> :k [ ]
[ ] :: * -> *
ghci> :k (->)
(->) :: * -> * -> *
```

Содержание

Вспомним общую алгебру!

Определение

Полугруппа — это множество с заданной на нём ассоциативной бинарной операцией $(S, *)$.

Определение

Моноид — это полугруппа с нейтральным элементом $(M, *, e)$:

$$\forall x \in M \; x * e = e * x = x.$$

Определение

Группа — это моноид $(G, *, e)$, в котором каждый элемент имеет обратный:

$$\forall x \in G \; \exists y \in G \; x * y = y * x = e.$$

Класс типов Monoid (модуль Data.Monoid)

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
```

- Каков сорт m?
- Пример (и источник терминологии): списки с операцией конкатенации.

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

Числовые моноиды

- Сложение и 0 как нейтральный элемент.
- Умножение и 1 как нейтральный элемент.

Механизм newtype

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

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

- Ключевое слово `newtype` создаёт тип-обёртку для значений некоторого типа.
- У нового типа может быть только один конструктор с единственным параметром — оборачиваемым типом.

Экземпляры для Product и Sum

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

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

Примеры

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4
                             `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

Логические моноиды

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

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

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

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

Примеры

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny . mconcat . map Any $ [False, False, True]
True
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

Maybe *a* как моноид

- Использование типа *a* как моноида.
- Предпочтение первого значения (обёртка First).
- Предпочтение последнего значения (обёртка Last).

Экземпляры для Maybe *a*

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)

instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

- Задание: самостоятельно реализовать экземпляр для Last.

Примеры

```
ghci> Nothing `mappend` Just "QQ"
Just "QQ"
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getLast . mconcat . map Last $ [Nothing,Just 9,Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

Законы моноидов

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

Код, написанный в расчёте на наличие экземпляра Monoid, может работать с данными самых разных типов. Это обобщённый код.

Бестолковый пример

```
process :: Monoid m => m -> m -> [m] -> m
```

Содержание

```
class Foldable t where    -- copm t: *->*
  Data.Foldable.fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  Data.Foldable.foldr' :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  Data.Foldable.foldl' :: (b -> a -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
  Data.Foldable.toList :: t a -> [a]
  null :: t a -> Bool
  length :: t a -> Int
  elem :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  minimum :: Ord a => t a -> a
  sum :: Num a => t a -> a
  product :: Num a => t a -> a
```

Использование экземпляров класса типов Foldable

```
diameter :: (Num a, Ord a, Foldable t) => t a -> a
diameter c = maximum c - minimum c
```

```
ghci> diameter [2, 10, 5, 44]
42
ghci> diameter (Data.Sequence.fromList [2, 10, 5, 44])
42
ghci> diameter (Data.Set.fromList [2, 10, 5, 44])
42
ghci> diameter (Just 10)
0
```

- Снова обобщённый код!

Свёртка контейнера с моноидом

```
ghci> :m + Data.Foldable Data.Monoid
ghci> let xs = [1,2,3,4,5]
ghci> let xs_sum = map Sum xs
ghci> let xs_prod = map Product xs
ghci> getSum $ fold xs_sum
15
ghci> getProduct $ fold xs_prod
120
```

Foldable для кортежей

Johan Tibell
@johtib

Читаю

Prelude> length (1,2)
1
#haskell-wat

Показать перевод

РЕТВИТОВ 14 ИЗБРАННОЕ 22

0:31 - 14 окт. 2015 г.

Содержание

Определение класса Functor

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

- Сорт f: * -> *, так как f a и f b должны иметь конкретный тип.
- Функтор — преобразование значения с сохранением структуры (контекста).

Функтор для списка — функция map

```
ghci> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
ghci> :t map
map :: (a -> b) -> [a] -> [b]
```

```
instance Functor [ ] where
   fmap = map
```

```
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Функтор для Maybe

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

```
ghci> fmap (++"!!!") (Just "Hello")
"Hello!!!"
ghci> fmap (++"!!!") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Функтор для дерева

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x left right) = Node (f x) (fmap f left)
                                            (fmap f right)
```

- Структура дерева полностью сохраняется, меняются только значения в узлах.
- Если дерево было деревом поиска, то соответствующее свойство может в результате отображения нарушиться.

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*2) (foldr treeInsert EmptyTree [5,7,3])
Node 10 (Node 6 EmptyTree EmptyTree) (Node 14 EmptyTree EmptyTree)
```

Функтор для Either a

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left y) = Left y
```

```
ghci> fmap (+2) (Right 5)
Right 7
ghci> fmap (+2) (Left "Сообщение об ошибке")
Left "Сообщение об ошибке"
```

Функтор для IO

```
instance Functor IO where
    fmap f action = do
            result <- action
            return (f result)
```

```
main = do
   par1 <- fmap head getArgs
   putStrLn par1
```

```
main = do
   line <- fmap (++"!") getLine
   putStrLn line
```

Функтор для IO

```
main = do
    line <- fmap (reverse . map toUpper) getLine
    putStrLn line
```

```
main = do
    [weight, height] <- (map read) `fmap` getArgs
    putStrLn $ analyze weight height
```

- Как read узнает, к какому типу нужно преобразовывать параметры командной строки?

Функтор для функции

```
ghci> :k (->)
(->) :: * -> * -> *
ghci> :k (->) Int
(->) Int :: * -> *
```

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

```
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Функтор для функции

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

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (+100) 1
"303"
```

Контейнеры и вычислительные контексты

- Maybe a — контейнер и вычисление с возможной неудачей.
- Either a b — контейнер и вычисление с возможной неудачей и сообщением об ошибке.
- [a] — контейнер и результат недетерминированного вычисления.
- IO a — контейнер и вычисление с побочными эффектами.
- (->) r a — контейнер (значения типа a, проиндексированные элементами типа r) и вычисление, в котором можно использовать значение параметра типа r.