CS314. Функциональное программирование

Лекция 6. Обобщённые вычисления

В. Н. Брагилевский

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

29 сентября 2017 г.

Содержание

Содержание

Сорта типов (type kinds)

```
ghci> :kind Int
Int :: *
ghci> :k Char
Char :: *
```

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

```
ghci> :k (->)
(->) :: TYPE q -> TYPE r
                  -> *
```

```
type * = TYPE LiftedRep
```

Содержание

Вспомним общую алгебру!

Определение

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

Определение

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

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

Определение

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

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

Классы типов Semigroup и Monoid (в будущем)

```haskell
module Data.Semigroup where

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

```haskell
module Data.Monoid where

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

    mconcat :: [a] -> a
    mconcat = foldr (<>) mempty
    mtimes :: Integral b => b -> a -> a
```

Класс типов Semigroup и Monoid (в GHC 8.2)

```
class Semigroup a where
  (<>) :: a -> a -> a
  sconcat :: Data.List.NonEmpty.NonEmpty a -> a
  stimes :: Integral b => b -> a -> a
```

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

Пример и источник терминологии

```
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 значения с единственным параметром оборачиваемового типа.

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 :: (Eq m, Monoid m) => m -> m -> [m] -> m
process x y zs = if x<>y == mempty then mempty
                                   else mconcat zs
```

```
ghci> process "" "" ["1", "2", "3", "4"]
""
ghci> process "12" "13" ["1", "2", "3", "4"]
"1234"
ghci> process (Sum 5) (Sum (-5)) $ map (Sum) [1..4]
Sum {getSum = 0}
ghci> process (Sum 5) (Sum 1) $ map (Sum) [1..4]
Sum {getSum = 10}
```

Содержание

```
class Foldable ( t :: * -> * )  where
  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
```

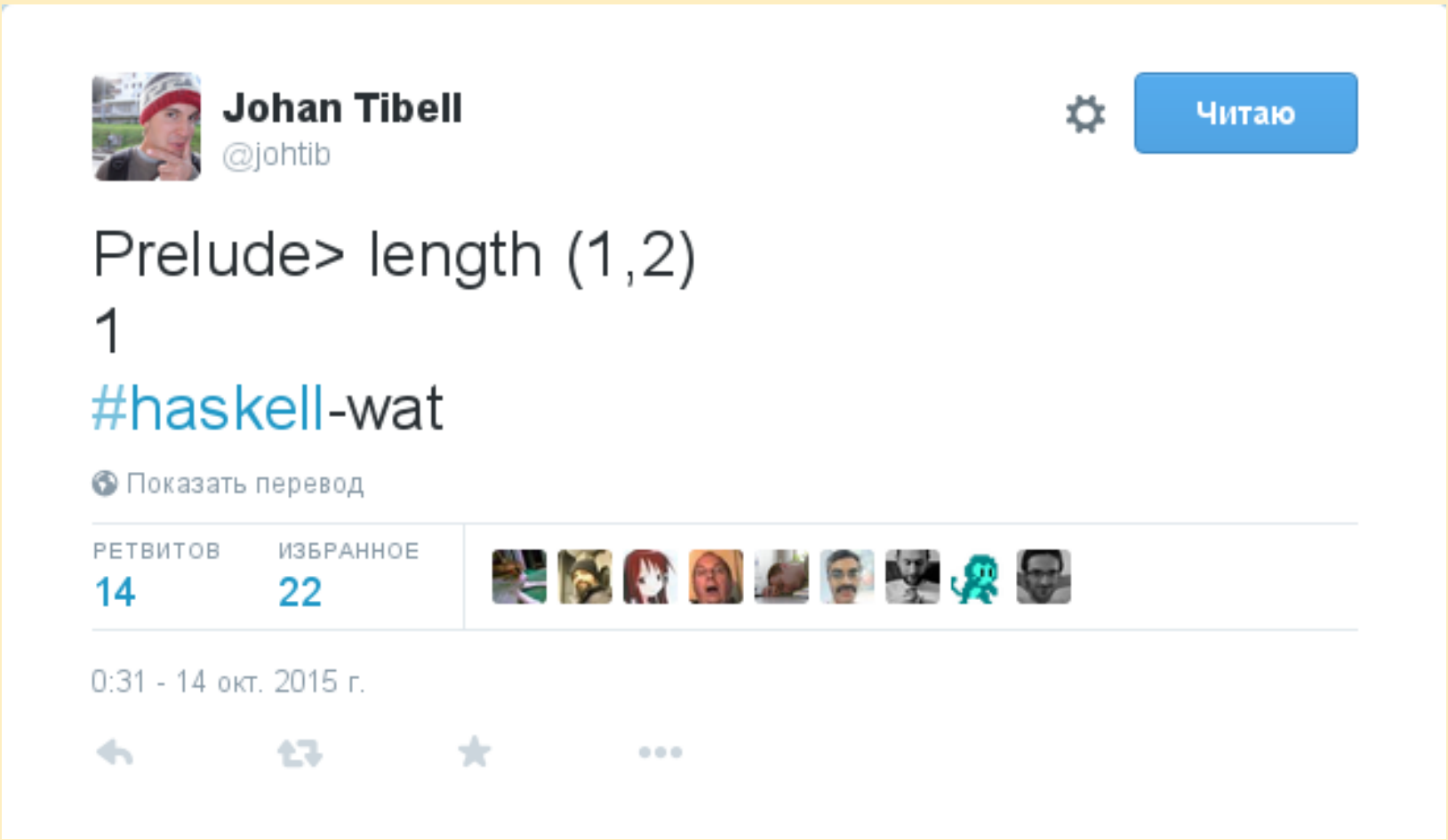
```
ghci> diameter [2, 10, 5, 44]
42
ghci> diameter (Data.Sequence.fromList [2, 10, 5, 44])
42
ghci> diameter (Data.Set.fromList [2, 10, 5, 44])
42
ghci> diameter (Just 10)
0
```

- Снова обобщённый код!

Свёртка контейнера с моноидом

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

Foldable для кортежей



Foldable для кортежей: объяснение

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

```
class Foldable ( t :: * -> * )  where
  ...
  length :: t a -> Int
  ...
```

Содержание

Определение класса Functor

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

- f — это вычислительный контекст (структура).
- Функтор — преобразование значения с сохранением вычислительного 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"
```

```
ghci> fmap (+1) (1,2)

(1,3)
```

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?

Вычислительные контексты и функторы

- `Maybe` — вычисление с возможной неудачей.
- `Either a` — вычисление с возможной неудачей и сообщением об ошибке типа `a`.
- `[]` — вычисление с недетерминированным результатом.
- `IO` — вычисление с побочными эффектами.

- Функтор позволяет применять функцию к результату вычисления без изменения контекста.

Ещё пример: функтор для функции

```
ghci> :k (->)
(->) :: TYPE q -> TYPE r -> *
ghci> :k (->) Int
(->) Int :: * -> *
ghci> :k (->) Char
(->) Char :: * -> *
```

```
instance Functor ((->) r) where
  -- fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
  fmap f g = ...
```

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

Функтор для функции: примеры

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

```
ghci> let f = fmap isDigit pred
ghci> :t f
f :: Char -> Bool
ghci> f '1'
True
ghci> f '0'
False
```

- `(->) r` — это вычисление с использованием параметра типа `r`.
- С применением `fmap` может меняться тип результата вычисления.

Примеры обобщённых вычислений с функторами

```
inc :: (Functor t, Num a) => t a -> t a
inc v = fmap (+1) v
```

```
ghci> inc (Just 10)
Just 11
ghci> inc (Right 10)
Right 11
ghci> inc [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

```haskell
inc :: (Functor t, Num a) => t a -> t a
inc v = fmap (+1) v
```

```haskell
readInt :: IO Int
readInt = do
  s <- getLine
  return $ read s
```

```
ghci> inc readInt
10
11
```

```haskell
inc :: (Functor t, Num a) => t a -> t a
inc v = fmap (+1) v
```

```haskell
doubler :: Double -> Double
doubler d = d * 2
```

```
ghci> inc doubler 10
21.0
```

Идеи функторов

- Функтор — это обобщение некоторого поведения — преобразование значения с сохранением особенностей вычислительного контекста.
- Одинаковый код для преобразования значения в контексте вне зависимости от контекста (fmap f значение_в_контексте).
- С учётом каррирования fmap можно рассматривать как «подъём функции» (lift):

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

Законы функторов

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

- Эти законы гарантируют, что структура контейнера или вычислительный контекст не будут затронуты функтором.

Пример нарушения законов

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

Законы функторов

- Можно доказать, что невозможно построить экземпляр функтора для списка, отличный от стандартного и удовлетворяющий законам функторов.
- Можно доказать, что если функтор удовлетворяет первому закону, то он удовлетворяет и второму, обратное неверно.

Проблема многопараметрических функций

```
ghci> :t fmap (*) (Just 3)
fmap (*) (Just 3) :: Num a => Maybe (a -> a)
ghci> :t Just (3*)
Just (3*) :: Num a => Maybe (a -> a)

ghci> :t fmap compare "ABC"
fmap compare "ABC" :: [Char -> Ordering]
ghci> :t [compare 'A', compare 'B', compare 'C']
[compare 'A', compare 'B', compare 'C'] :: [Char -> Ordering]
```

- В результате применения fmap функция оказывается внутри контекста.
- Применить её дальше средствами функторов не удаётся.

Содержание

Определение аппликативного функтора

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

- Две функции:
  - помещение значения в контекст;
  - преобразование значения с помощью функции, находящейся в контексте.
- Можно ли с помощью `pure` и `<*>` получить fmap?

```
fmap = (<*>) . pure
```

На что похоже <*>?

```
(<*>) :: f (a -> b) -> f a -> f b
($) :: (a -> b) -> a -> b
fmap :: (a -> b) -> f a -> f b
```

Вспомогательные операции из Applicative

```
(*>) :: f a -> f b -> f b
(<*) :: f a -> f b -> f a
```

Законы аппликативных функторов

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Экземпляр для Maybe

- Помещение в контекст (pure).
- Применение функции из контекста (<*>).

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

Примеры

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"!!!") <*> Nothing
Nothing
ghci> Nothing <*> Just "ololo"
Nothing
```

Многопараметрические функции (аппликативный стиль)

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

Аппликативный стиль

Функция <$>

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

```
ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing
ghci> (+) <$> Nothing <*> Just 5
Nothing
```

Экземпляр для списков

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

```
instance Applicative [] where
   pure :: a -> [a]
   (<*>) :: [a->b] -> [a] -> [b]
   ...
```

- Как можно реализовать эти функции?
- Возможны различные реализации, например: склейка списков, недетерминированные вычисления.

Экземпляр для списка: недетерминированные вычисления

- Простейший контекст: одноэлементный список.
- Список функций $\to$ список значений $\to$ применение каждой функции к каждому значению.

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

Пример: недетерминированные вычисления

```
ghci> (*) <$> [1,2,3] <*> [4,5]
[4,5,8,10,12,15]
ghci> :t (*) <$> [1,2,3]
(*) <$> [1,2,3] :: Num a => [a -> a]
ghci> length $  (*) <$> [1,2,3]
3
ghci> (++) <$> ["abc", "def"] <*> pure "!!!"
["abc!!!","def!!!"]
```

Обёртка для типа списка (Control.Applicative)

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

```
ghci> :t ZipList [1,2,3]
ZipList [1,2,3] :: Num a => ZipList a
ghci> getZipList $ ZipList [1,2,3]
[1,2,3]
```

Экземпляр для списка: склейка списков

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

- Что делать с pure?
- Нужен бесконечный список!

```
  pure x = ZipList (repeat x)
```

Примеры: склейка списков

```
ghci> getZipList $ (+) <$> ZipList [1,2,3]
                       <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> pure 100
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3]
                       <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "foo" <*> ZipList "bar"
                                           <*> ZipList "tre"
[('f','b','t'),('o','a','r'),('o','r','e')]
ghci> :t (,,)
(,,) :: a -> b -> c -> (a, b, c)
```

Экземпляр для IO

```
instance Applicative IO where
   pure = return
   a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Примеры: IO как аппликативный функтор

```haskell
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

```haskell
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

```haskell
main = do
    a <- myAction
    putStrLn $ "Две строки, соединённые вместе: " ++ a
```

Примеры: IO как аппликативный функтор

Последовательное выполнение действий

```
ghci> putStr "Your name: " *> getLine
Your name: John
"John"
ghci> :t (*>)
(*>) :: Applicative f => f a -> f b -> f b
```

```
ghci> (,) <$> (putStr "Your name: " *> getLine) <*>
                        (putStr "Your age: " *> getLine)
Your name: John
Your age: 20
("John","20")
```

Проблема

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

- Как ввести строку, а затем вывести её на консоль?
- Наблюдение: второе действие зависит от результата первого.
- Вывод: возможностей аппликативных функторов недостаточно.