

CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 5b. Классы типов для описания абстракций
над вычислительными контекстами

В. Н. Брагилевский

14 марта 2019 г.

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

Понятие вычислительного контекста

- $v :: a$ — значение типа a
- $w :: C a$ — значение типа a в контексте C
- $f :: a_1 \rightarrow a_2 \rightarrow \dots \rightarrow C a$ — функция, результатом которой является значение в контексте

Таким образом, контекст — это тип сорта $* \rightarrow *$.

Примеры вычислительных контекстов

- **IO** – вычисление с потенциально возможными побочными эффектами (пример: `getLine :: IO String`).
- **Maybe** – вычисление с возможной неудачей (пример: `Just 5` или `Nothing`).
- **Either** `a` – вычисление с возможной неудачей и сообщением об ошибке типа `a` (пример: `Right 'x'` или `Left "Incorrect File Format"`).
- **[]** – вычисление с недетерминированным результатом (пример: `[1, 3, 5]`).

Зачем нужно учитывать вычислительный контекст?

- Контроль
- Выразительность

Зачем нужно абстрагировать понятие вычислительного контекста?

- Вычислительные контексты поддерживают операции, общие для всех контекстов.
- Можно описывать процессы, работающие независимо от контекста, и применять их к любым контекстам.

Основной инструмент абстракции в Haskell – классы типов.

Класс Functor

Определение класса Functor

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

- f – это вычислительный контекст.
- Функтор – преобразование значения с сохранением контекста.

Функтор для списка — функция map

```
fmap :: Functor f => (a -> b) -> f a -> f b  
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")
```

```
Just "Hello!!!"
```

```
ghci> fmap (++"!!!") Nothing
```

```
Nothing
```

```
ghci> fmap (*2) (Just 200)
```

```
Just 400
```

```
ghci> fmap (*2) Nothing
```

```
Nothing
```

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"
```

Загадка

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

```
(1,3)
```

```
instance Functor IO where
```

```
  fmap f action = do  
    result <- action  
    pure (f result)
```

```
main = do
```

```
  par1 <- fmap head getArgs  
  putStrLn par1
```

```
main = do
```

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

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

Функтор позволяет применять функцию к результату вычисления без изменения контекста.

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]
```

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

```
readInt :: IO Int
```

```
readInt = do
  s <- getLine
  pure $ read s
```

```
ghci> inc readInt
```

```
10
```

```
11
```

- Функтор — это обобщение некоторого поведения — преобразование значения с сохранением особенностей вычислительного контекста.
- Одинаковый код для преобразования значения в контексте вне зависимости от контекста.
- 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
```

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)
```

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

Класс Applicative

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 в контексте с помощью функции, находящейся в контексте.

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`

- 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 "lololo"
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
```

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
```

```
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 \rightarrow список значений \rightarrow 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)
```

```
instance Applicative IO where
```

```
  a <*> b = do
```

```
    f <- a
```

```
    x <- b
```

```
  pure (f x)
```

Примеры: IO как аппликативный функтор

```
myAction :: IO String
```

```
myAction = do
```

```
  a <- getLine
```

```
  b <- getLine
```

```
  pure $ a ++ b
```

```
myAction :: IO String
```

```
myAction = (++) <$> getLine <*> getLine
```

```
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"
```

```
(*>) :: 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")
```

```
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.