CS314. Функциональное программирование

Лекция 11. Функторы и аппликативные функторы

В. Н. Брагилевский

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

21 октября 2016 г.

Пример

```
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]
```

Пример (2)

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

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

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

Пример (3)

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

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

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

Содержание

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).

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.

Идеи функторов

- Обобщение некоторого поведения — преобразование значения с сохранением структуры контейнера или 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
```

Содержание

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 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.

Содержание

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)
```

Содержание

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
```

- Задание: проверить, выполняются ли эти законы для обсуждавшихся экземпляров аппликативных функторов.

Содержание

Обобщённый код

- Цель: написание кода, работающего одинаково в различных условиях (контекстах или контейнерах).
- Средства: моноиды, функторы, аппликативные функторы, ...
- Порядок действий:
  - написание обобщённого кода и его применение в существующих контекстах;
  - разработка нового контекста (контейнера);
  - реализация необходимых экземпляров классов типов;
  - применение обобщённого кода.

Пример 1: создание тройки

```
makeTriplet :: Applicative f =>
               f a -> f a -> f a -> f (a,a,a)
makeTriplet c1 c2 c3 = (,,) <$> c1 <*> c2 <*> c3
```

```
ghci> makeTriplet (Just 4) (Just 1) (Just 8)
Just (4,1,8)
ghci> makeTriplet (Just 4) Nothing (Just 8)
Nothing
ghci> makeTriplet (Right 6) (Right 7) (Left "Error!")
Left "Error!"
ghci> makeTriplet (Right 6) (Right 7) (Right 8)
Right (6,7,8)
```

Пример 1: создание тройки

```
makeTriplet :: Applicative f =>
               f a -> f a -> f a -> f (a,a,a)
makeTriplet c1 c2 c3 = (,,) <$> c1 <*> c2 <*> c3
```

```
ghci> makeTriplet [1] [2] [3]
[(1,2,3)]
ghci> makeTriplet [1,2] [3,4] []
[]
ghci> makeTriplet [1,2] [3,4] [5,6]
[(1,3,5),(1,3,6),(1,4,5),(1,4,6),(2,3,5),(2,3,6)
                                ,(2,4,5),(2,4,6)]
ghci> getZipList $ makeTriplet (ZipList [1,2])
                               (ZipList [3,4])
                               (ZipList [5,6])
[(1,3,5),(2,4,6)]
```

Пример 1: создание тройки

```
makeTriplet :: Applicative f =>
               f a -> f a -> f a -> f (a,a,a)
makeTriplet c1 c2 c3 = (,,) <$> c1 <*> c2 <*> c3
```

```
ghci> makeTriplet getLine getLine getLine
12
34
56
("12 ","34","56")
```

Пример 2: вычисления в моноиде

```
combine :: (Applicative f, Monoid a) => f a -> f a -> f a
combine c1 c2 = mappend <$> c1 <*> c2
```

```
ghci> combine getLine getLine
123
456
"123456"
ghci> combine (Just "123") Nothing
Nothing
ghci> combine [Sum 1] [Sum 2, Sum 3]
[Sum {getSum = 3},Sum {getSum = 4}]
```

Класс Alternative — моноид на аппликативных функторах

```
class Applicative f => Alternative f where
  Control.Applicative.empty :: f a
  (<|>) :: f a -> f a -> f a
```

```
ghci> Just 3 <|> Nothing
Just 3
ghci> Nothing <|> Just 3
Just 3
ghci> Just 5 <|> Just 3
Just 5
ghci> [1,2,3] <|> [4,5,6]
[1,2,3,4,5,6]
```

Пример 3: абстрактное вычисление с Alternative

```
process :: Alternative f =>  (a -> b) -> f a -> f a -> f b
process f c1 c2 = f <$> (c1 <|> c2)
```

```
ghci> process (+1) (Just 1) (Just 3)
Just 2
ghci> process (+1) Nothing (Just 3)
Just 4
ghci> process (+1) [] [2,3]
[3,4]
ghci> process (+1) [1,2] [2,3]
[2,3,3,4]
```