

Функциональное программирование

Лекция 8. Аппликативные функторы

Денис Николаевич Москвин

ИТМО, магистратура JB SE

26.10.2020

- 1 Функторы
- 2 Класс типов `Pointed`
- 3 Аппликативные функторы

- 1 Функторы
- 2 Класс типов `Pointed`
- 3 Аппликативные функторы

Класс типов Functor

Представители класса типов `Functor` должны быть конструкторами типа с одним параметром, то есть `f :: * -> *`.

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

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

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap g (Just a) = Just (g a)
```

Задают способ «поднять стрелку на уровень контейнера».

Представитель класса типов Functor для дерева

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)

instance Functor Tree where
  fmap :: (a -> b) -> Tree a -> Tree b
  fmap g (Leaf x)          = Leaf (g x)
  fmap g (Branch l x r) = Branch (fmap g l)
                              (g x)
                              (fmap g r)
```

Расширение `InstanceSigs` позволяет указывать сигнатуры методов в представителе класса типов.

```
GHCi> testTree = Branch (Leaf 2) 3 (Leaf 4)
GHCi> fmap (^2) testTree
Branch (Leaf 4) 9 (Leaf 16)
GHCi> (^3) <$> testTree
Branch (Leaf 8) 27 (Leaf 64)
```

Полное определение класса типов Functor

```
infixl 4 <$, <$>, $>
class Functor f where
  fmap  :: (a -> b) -> f a -> f b
  (<$)  :: a -> f b -> f a
  (<$)  = fmap . const

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

($>)  :: Functor f => f a -> b -> f b
($>)  = flip (<$)

void  :: Functor f => f a -> f ()
void x = () <$ x

infixl 1 <&>      -- Base 4.11.0.0 GHC 8.4.1 (Mar 2018)
(<&>)  :: Functor f => f a -> (a -> b) -> f b
xs <&> f = f <$> xs
```

Представители Functor для двухпараметрических типов

Поскольку `Either`, `(,)`, `(->)` `:: * -> * -> *` требуется связать первый параметр, чтобы можно было объявить их представителями функтора.

```
instance Functor (Either e) where
  fmap :: (a -> b) -> Either e a -> Either e b
  fmap _ (Left x)   = Left x
  fmap g (Right y) = Right (g y)
```

```
instance Functor ((,) s) where
  fmap :: (a -> b) -> (s,a) -> (s,b)
  fmap g (x,y) = (x, g y)
```

А что для частично примененной стрелки?

```
instance Functor ((->) e) where
  fmap :: (a -> b) -> ((->) e a) -> ((->) e b)
  fmap = _
```

- Для любого представитель класса типов `Functor` должно выполняться

Законы класса типов `Functor`

```
fmap id      ≡ id
fmap (f . g) ≡ fmap f . fmap g
```

- Это так для списков, `Maybe`, `IO` и т.д.
- Смысл законов: вызов `fmap g` не должен менять «структуру контейнера», воздействуя только на его элементы.
- **Всегда ли эти законы выполняются?**

- «Плохой» представитель класса `Functor` для списка

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

- Какой закон нарушается для такого объявления представителя и почему?
- «Any Haskell-er worth their salt would reject this code as a gruesome abomination.» Typeclassopedia [Yor09]

Бывают ли не-функторы?

- Все ли типы данных с однопараметрическим конструктором типа являются функторами?

Бывают ли не-функторы?

- Все ли типы данных с однопараметрическим конструктором типа являются функторами? Нет!
- Единственная возможная реализация не удовлетворяет первому закону функторов:

```
newtype Endo a = Endo { appEndo :: a -> a }
```

```
instance Functor Endo where  
  fmap :: (a -> b) -> Endo a -> Endo b  
  fmap _ (Endo _) = Endo id
```

- А стрелка со связанным вторым аргументом вообще не допускает реализации `fmap`:

```
newtype RevArr c a = RevArr { appRevArr :: a -> c }
```

```
instance Functor (RevArr c) where ???
```

- 1 Функторы
- 2 Класс типов `Pointed`
- 3 Аппликативные функторы

Pointed: класс типов, которого нет

Даёт возможность «вложить значение в контекст»

```
class Functor f => Pointed f where  
  pure :: a -> f a -- aka singleton, return, unit, point
```

```
instance Pointed Maybe where  
  pure x = Just x
```

```
instance Pointed [] where  
  pure x = [x]
```

```
instance Pointed (Either e) where  
  pure =
```

Представители Pointed

```
class Functor f => Pointed f where  
  pure :: a -> f a
```

Всегда ли возможно объявления представителя для `Pointed`?

```
instance Pointed ((->) e) where  
  pure =
```

```
instance Pointed Tree where  
  pure =
```

```
instance Pointed ((,) s) where  
  pure =
```

Закон для класса Pointed

Закон для класса типов `Pointed` один:

$$\text{fmap } g \cdot \text{pure} \equiv \text{pure} \cdot g$$

Он выполняется всегда, являясь *свободной теоремой* для типа `Functor f => a -> f a`. Это значит, что диаграмма коммутативна для любой `g :: a -> b`

$$\begin{array}{ccc} a & \xrightarrow{\text{pure}} & f\ a \\ g \downarrow & & \downarrow \text{fmap } g \\ b & \xrightarrow{\text{pure}} & f\ b \end{array}$$

- 1 Функторы
- 2 Класс типов `Pointed`
- 3 **Аппликативные функторы**

Расширяемость класса типов Functor

Можно ли для произвольного функтора универсально сконструировать

```
fmap2 :: (a -> b -> c) -> f a -> f b -> f c  
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d  
...
```

Расширяемость класса типов Functor

Можно ли для произвольного функтора универсально сконструировать

```
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
...
```

Попробуем построить `fmap2 g as bs`. Поскольку `as :: f a`, `bs :: f b` и `g :: a -> (b -> c)`, применение `fmap g` даст

```
fmap g as :: f (b -> c)
```

Расширяемость класса типов Functor

Можно ли для произвольного функтора универсально сконструировать

```
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
...
```

Попробуем построить `fmap2 g as bs`. Поскольку `as :: f a`, `bs :: f b` и `g :: a -> (b -> c)`, применение `fmap g` даст

```
fmap g as :: f (b -> c)
```

Стрелка «забралась» в контейнер, нужен способ «вынуть» её:

```
ap :: f (b -> c) -> f b -> f c
```

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

infixl 4 <*>
```

Функция `pure` обсуждалась выше; если бы `Pointed` существовал, то определение было бы таким

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

Оператор `<*>` похож на `($\$$) :: (a -> b) -> a -> b`, но в «вычислительном контексте», задаваемым функтором.

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

Теперь можем работать в вычислительном контексте с возможно отсутствующим значением:

```
GHCi> Just (+2) <*> Just 5
Just 7
GHCi> Just (+2) <*> Nothing
Nothing
GHCi> Just (+) <*> Just 2 <*> Just 5
Just 7
```

Закон, связывающий `Applicative` и `Functor`

Рассмотрим произвольную $g :: a \rightarrow b$ и $xs :: f\ a$ для некоторого аппликативного функтора f . Тогда

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

Это совпадает по типу с `fmap g`.

Для любого представителя `Applicative` требуют, чтобы для функций `pure` и `<*>` выполнялся

Закон, связывающий `Applicative` и `Functor`

```
fmap g xs ≡ pure g <*> xs
```

Закон, связывающий Applicative и Functor

Или, используя инфиксный синоним fmap

Закон, связывающий Applicative и Functor

$$g \langle \$ \rangle xs \equiv \text{pure } g \langle * \rangle xs$$

Ниже все три вызова идентичны:

```
GHCi> Just (+) <*> Just 2 <*> Just 5
Just 7
GHCi> pure (+) <*> Just 2 <*> Just 5
Just 7
GHCi> (+) <$> Just 2 <*> Just 5
Just 7
```

Identity

$$\text{pure id } \langle * \rangle v \equiv v$$

Homomorphism

$$\text{pure } g \langle * \rangle \text{pure } x \equiv \text{pure } (g \ x)$$

Interchange

$$u \langle * \rangle \text{pure } x \equiv \text{pure } (\$ \ x) \langle * \rangle u$$

Composition

$$\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle x \equiv u \langle * \rangle (v \langle * \rangle x)$$

Подробнее см. в McBride and Paterson [MP08].

Списки как аппликативные функторы

Рассмотрим список функций и список значений

```
fs = [(2*), (3+), (4-)]
```

```
xs = [1, 2]
```

Каким смыслом можно наделить аппликацию `fs <*> xs`?

Списки как аппликативные функторы

Рассмотрим список функций и список значений

```
fs = [(2*), (3+), (4-)]  
xs = [1, 2]
```

Каким смыслом можно наделять аппликацию `fs <*> xs`?

Двумя разными!

- Список — контекст, задающий множественные результаты недетерминированного вычисления:

```
fs <*> xs → [(2*)1, (2*)2, (3+)1, (3+)2, (4-)1, (4-)2]  
           → [2, 4, 4, 5, 3, 2]
```

- Список — это коллекция упорядоченных элементов:

```
fs <*> xs → [(2*)1, (3+)2]  
           → [2, 5]
```

Оператор (`<*>`) в этом случае должен реализовывать модель «каждый с каждым»:

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

```
GHCi> fs = [(2*), (3+), (4-)]
GHCi> xs = [1,2]
GHCi> fs <*> xs
[2,4,4,5,3,2]
```

Два представителя для одного типа недопустимы, поэтому

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

instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (map f xs)

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

```
GHCi> fs = [(2*), (3+), (4-)]
GHCi> xs = [1,2]
GHCi> getZipList $ ZipList fs <*> ZipList xs
[2,5]
```

Можно ли пару сделать представителем `Applicative`?

Можно ли пару сделать представителем `Applicative`?

```
instance Monoid s => Applicative ((,) s) where
  pure x          = (mempty, x)
  (u, f) <*> (v, x) = (u <> v, f x)
```

```
GHCi> ("Answer to ",(*)) <*> ("the Ultimate ",6) <*> (
"Question",7)
("Answer to the Ultimate Question",42)
```

Полное определение класса типов Applicative

```
infixl 4 <*>, *>, <*, <***>
class Functor f => Applicative f where
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
  pure :: a -> f a

  (<*>) :: f (a -> b) -> f a -> f b
  (<*>) = liftA2 id

  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  liftA2 g a b = g <$> a <*> b

  (*>) :: f a -> f b -> f b
  a1 *> a2 = (id <$> a1) <*> a2

  (<*) :: f a -> f b -> f a
  (<*) = liftA2 const
```

```
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = pure f <*> a
```

Реализация `liftA = fmap` не очень хороша, поскольку часто наоборот содержательно реализуют `Applicative`, а для функтора пишут `fmap = liftA`.

```
liftA3 :: Applicative f =>
  (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 g a b c = g <$> a <*> b <*> c
```

```
(<***>) :: Applicative f => f a -> f (a -> b) -> f b
(<***>) = liftA2 (&)
```


Порядок эффектов в <*>

```
(<*>) :: Applicative f => f a -> f (a -> b) -> f b  
<*> = liftA2 (&)
```

```
(<***>) :: Applicative f => f a -> f (a -> b) -> f b  
<***> = flip (<*>)
```

(Оператора <***> не существует в Control.Applicative, мы его реализовали с иллюстративными целями.)

```
GHCi> ("Answer to ",5) <*> ("the Ultimate ",(*8)) <***>  
  ("Question",(+2))  
("Answer to the Ultimate Question",42)  
GHCi> ("Answer to ",5) <***> ("the Ultimate ",(*8)) <***>  
*> ("Question",(+2))  
("Questionthe Ultimate Answer to ",42)
```



Conor McBride and Ross Paterson.

Applicative programming with effects.

J. Funct. Program., 18(1):1–13, January 2008.



Brent Yorgey.

Typeclassopedia.

The Monad.Reader, (13):17–68, March 2009.