

Функциональное программирование

Лекция 8. Аппликативные функторы

Денис Николаевич Москвин

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

06.04.2012

- 1 Функторы
- 2 Класс типов Pointed
- 3 Аппликативные функторы
- 4 Класс типов Traversable

- 1 Функторы
- 2 Класс типов Pointed
- 3 Аппликативные функторы
- 4 Класс типов Traversable

Представители класса типов Functor должны быть конструкторами типа с одним параметром, то есть $f :: * \rightarrow *$.

```
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».

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

instance Functor Tree where
  fmap g (Leaf x)          = Leaf (g x)
  fmap g (Branch l x r) = Branch (fmap g l) (g x) (fmap g r)
```

GHCi

```
*Fp08> let testTree = Branch (Leaf 2) 3 (Leaf 4)
*Fp08> fmap (^2) testTree
Branch (Leaf 4) 9 (Leaf 16)
```

Функция `fmap` не меняет «структуру» контейнера.

Представители Functor для двухпараметрических типов

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

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

```
instance Functor ((,) a) where
  fmap g (x,y) = (x, g y)
```

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

- Для любого представитель класса типов `Functor` должно выполняться

Законы для функторов

```
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?

- 1 Функторы
- 2 Класс типов Pointed
- 3 Аппликативные функторы
- 4 Класс типов Traversable

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 l) where  
  pure =
```

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

Всегда ли возможно объявления представителя для Pointed?

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

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

```
instance Pointed ((,) e) 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 $\text{Functor } f \Rightarrow a \rightarrow f a$.

Диаграмма коммутативна для любой $g :: a \rightarrow b$

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

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

Можно ли для функтора f 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
...
```

Можно ли для функтора f 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 $g :: a \rightarrow (b \rightarrow c)$, применение $fmap\ g$ даст

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

Можно ли для функтора f 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 $g :: a \rightarrow (b \rightarrow c)$, 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 `<*>` — это, фактически, `($\$$) :: (a -> b) -> a -> b`, но в «вычислительном контексте», задаваемым функтором.

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

Теперь можем работать в вычислительном контексте с возможно отсутствующим значением:

GHCi

```
*Fp08> Just (+2) <*> Just 5
Just 7
*Fp08> Just (+2) <*> Nothing
Nothing
*Fp08> Just (+) <*> Just 2 <*> Just 5
Just 7
```

Закон, связывающий Applicative И Functor

Рассмотрим произвольную $g :: a \rightarrow b$ и $as :: f\ a$ 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 as ≡ pure g <*> as
```

В Control.Applicative введён инфиксный синоним fmap

```
g <$> x = pure g <*> x
```

Ниже все три вызова идентичны:

GHCi

```
*Fp08> Just (+) <*> Just 2 <*> Just 5  
Just 7  
*Fp08> pure (+) <*> Just 2 <*> Just 5  
Just 7  
*Fp08> (+) <$> Just 2 <*> Just 5  
Just 7
```

Identity

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

Composition

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

Homomorphism

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

Interchange

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

Подробнее см. в McBride and Paterson [MP08].

Списки как аппликативные функторы

Рассмотрим список функций и список значений

```
fs = [\x->2*x, \x->3+x, \x->4-x]  
as = [1,2]
```

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

Списки как аппликативные функторы

Рассмотрим список функций и список значений

```
fs = [\x->2*x, \x->3+x, \x->4-x]  
as = [1,2]
```

Каким смыслом можно наделить аппликацию $fs \langle * \rangle as$?

Двумя разными!

- Список — контекст, задающий множественные результаты недетерминированного вычисления.

```
fs \langle * \rangle as \equiv [(\backslash x \rightarrow 2 * x) 1, (\backslash x \rightarrow 2 * x) 2, (\backslash x \rightarrow 3 + x) 1,  
(\backslash x \rightarrow 3 + x) 2, (\backslash x \rightarrow 4 - x) 1, (\backslash x \rightarrow 4 - x) 2] \equiv [2, 4, 4, 5, 3, 2]
```

- Список — это коллекция упорядоченных элементов.

```
fs \langle * \rangle as \equiv [(\backslash x \rightarrow 2 * x) 1, (\backslash x \rightarrow 3 + x) 2] \equiv [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

```
*Fp08> let fs = [\x->2*x, \x->3+x, \x->4-x]
*Fp08> let as = [1,2]
*Fp08> fs <*> as
[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

```
*Fp08> let fs = [\x->2*x, \x->3+x, \x->4-x]
*Fp08> let as = [1,2]
*Fp08> getZipList $ ZipList fs <*> ZipList as
[2,5]
```

Пара как представитель аппликативного функтора

Можно ли пару сделать представителем `Applicative`?

Можно ли пару сделать представителем `Applicative`?

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

GHCi

```
*Fp08> ("Answer to ",(*)) <*> ("the Ultimate ",6) <*> ("Que
stion",7)
("Answer to the Ultimate Question",42)
```

- 1 Функторы
- 2 Класс типов Pointed
- 3 Аппликативные функторы
- 4 Класс типов Traversable

```
dist :: Applicative f => [f a] -> f [a]
dist []           = pure []
dist (x : xs) = (:) <$> x <*> (dist xs)
```

GHCi

```
*Fp08> dist [Just 3,Just 5]
Just [3,5]
*Fp08> dist [Just 3,Nothing]
Nothing
*Fp08> getZipList $ dist $ map ZipList [[1,2,3],[4,5,6]]
[[1,4],[2,5],[3,6]]
*Fp08>
```

Минимальное полное определение: `traverse` или `sequenceA`.

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g = sequenceA . fmap g

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
```

```
instance Traversable Maybe where
  traverse _ Nothing  = pure Nothing
  traverse g (Just x) = Just <$> g x
```

```
instance Traversable [] where
  traverse g = Prelude.foldr cons_f (pure [])
  where cons_f x ys = (:) <$> g x <*> ys
```



Conor McBride and Ross Paterson.

Applicative programming with effects.

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