Функциональное программирование
Лекция 8. Аппликативные функторы

Денис Николаевич Москвин

СПбАУ РАН

24.10.2017

План лекции

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

План лекции

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 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)
*Fp08> (^3) <$> testTree
Branch (Leaf 8) 27 (Leaf 64)
```

Функция `fmap` не меняет «структуру» контейнера.

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
```

Представители `Functor` для двухпараметрических типов

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

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

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

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

Законы для функторов

- 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?

Законы для функторов: контрпример

- «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 Haskeller worth their salt would reject this code as a gruesome abomination.» Typeclassopedia [Yor09]

План лекции

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  =
```

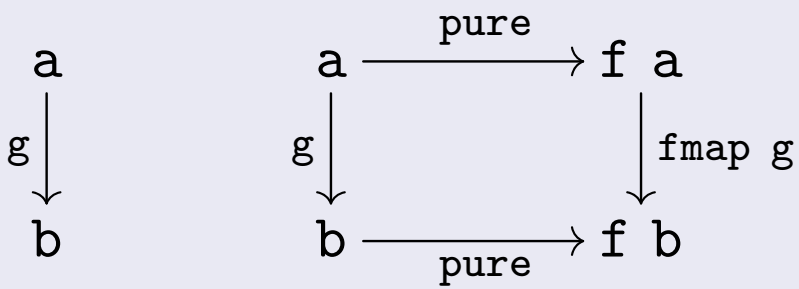
Закон для класса `Pointed`

Закон для класса типов `Pointed` один:

```
fmap g . pure  ≡  pure . g
```

Он выполняется всегда, являясь *свободной теоремой* для типа `Functor f => a -> f a`.

Диаграмма коммутативна для любой `g :: a -> b`



План лекции

Расширяемость класса типов `Functor`

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
...
```

Расширяемость класса типов `Functor`

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 -> (b -> c)`, применение `fmap g` даст

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

Расширяемость класса типов `Functor`

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 -> (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
```

Аппликативные функторы: класс типов

```
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.

Аппликативные функторы: объявление представителей

```
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

```
*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 -> 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 <$> xs  ≡  pure g <*> xs
```

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
```

Законы для `Applicative`

Identity

```
pure id <*> v  ≡  v
```

Homomorphism

```
pure g <*> pure x  ≡  pure (g x)
```

Interchange

```
u <*> pure x  ≡  pure ($ x) <*> u
```

Composition

```
pure (.) <*> u <*> v <*> w  ≡  u <*> (v <*> w)
```

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 <*> as`?

Двумя разными!

- Список — контекст, задающий множественные результаты недетерминированного вычисления.

```
fs <*> as ≡ [(\x->2*x) 1, (\x->2*x) 2, (\x->3+x) 1,
  (\x->3+x) 2, (\x->4-x) 1, (\x->4-x) 2] ≡ [2,4,4,5,3,2]
```

- Список — это коллекция упорядоченных элементов.

```
fs <*> as ≡ [(\x->2*x) 1, (\x->3+x) 2] ≡ [2,5]
```

Список как результат недетерминированного вычисления

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]
```

Список как коллекция элементов

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`?

Пара как представитель аппликативного функтора

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)
```

Полное определение класса типов Applicative

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

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

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

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 (flip ($))
```

Литература

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.