

Про функторы, аппликативы и монады

Косарев Дмитрий а.к.а. Kakadu

матмех СПбГУ

31 октября 2019 г.

ACHTUNG!

Я хотел сделать слайды про монады, а получилось про более простые штуки с уклоном в парсеры.

В этих слайдах

1. Функторы (ковариантные)
 - Законы функторов
 - Примеры функторов
2. Аппликативные функторы
 - Законы аппликативов
 - Аппликативные парсеры
3. Монады
 - Законы монад
 - Монады и парсеры
 - Монада List
 - Монада Writer
 - Монада IO

Функторы (ковариантные)

```
Prelude> :i Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a          -> f b -> f a
  {-# MINIMAL fmap #-}    -- Defined in 'GHC.Base'
```

Функторы – это тип с операцией `fmap`, удовлетворяющий законам функторов. Функторы иногда аллегорично называют “контейнерами”.

Часто встречающийся синоним для `fmap`:

```
Prelude> :i <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
-- Defined in 'Data.Functor'
```

Законы (ковариантных) функторов

```
-- preservation of identity
fmap id == id
-- preservation of composition
fmap (f . g) == fmap f . fmap g
```

Пример реализации `fmap`, которая **не согласуется** с законами:

```
data PPP a = PPP a a

instance Functor PPP where
    fmap f (PPP a b) = PPP (f b) (f a) -- N.B. Swap
```

Но обычно для всех наших типов возможно написать `fmap`.

(Ковариантные) Функторы в стандартной библиотеке

```
-- identity functor
```

```
data Id a = Id a
```

```
instance Functor Id where
```

```
    fmap f (Id a) = Id (f a)
```

```
-- Maybe
```

```
data Maybe a = Just a | Nothing
```

```
instance Functor Maybe where
```

```
    fmap _ Nothing = Nothing
```

```
    fmap f (Just x) = Just (f x)
```

```
-- List
```

```
instance Functor [] where
```

```
    fmap _ [] = []
```

```
    fmap f (x:xs) = (f x) : (fmap f xs)
```

```
-- Arrow
```

```
data Arrow a b = Arr (a -> b)
```

```
instance Functor (Arrow a) where
```

```
    fmap f (Arr g) = Arr (f . g)
```

```
-- Defined in 'GHC.Tuple'
```

```
data (,) a b = (,) a b
```

```
instance Functor ((,)a) where
```

```
    fmap f (a,b) = (a, f b)
```

Пользовательские типы тоже могут быть функторами

```
data L name =  
    Var name  
  | Abs name (L name)  
  | App (L name) (L name)  
  
instance Functor L where  
    fmap f (Var n) = Var (f n)  
    fmap f (Abs n b) =  
        Abs (f n) (fmap f b)  
    fmap f (App l r) =  
        App (fmap f l) (fmap f r)
```

С помощью `fmap` можно сделать какое-нибудь безконтекстное переименование

Не все типы являются ковариантными функторами

Например, для

```
-- Arrow
data ArrowSwapped a b = ArrowSwapped (b -> a)
instance Functor (ArrowSwapped a) where
    fmap f (Arr g) = ...
```

не получится написать код, удовлетворяющий законам. Зато можно сделать контравариантный функтор.

```
> :i Contravariant
class Contravariant f where
    contramap :: (a -> b) -> f b -> f a
    (>$) :: b -> f b -> f a
    {-# MINIMAL contramap #-}
    -- Defined in 'Data.Functor.Contravariant'
```


Шаг 2. Аппликативные функторы

Документация [тут](#). А в [2] можно почитать как они появились и как с помощью них писать интерпретатор выражений.

```
--      ┌─────────── каждый Applicative также и функтор
--      │
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
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  {-# MINIMAL pure, ((<*>) / liftA2) #-}
```

Аппликативы в стандартной библиотеке

```
data Id a = Id a
instance Functor Id where
    pure = Id
    (Id f) <*> (Id x) = Id (f x)
```

```
data Maybe a = Just a | Nothing
instance Applicative Maybe where
    pure = Just
    Just f <*> m      = fmap f m
    Nothing <*> _      = Nothing
```

```
instance Applicative ((->) a) where
    pure  = \x _ -> x           -- K
    (<*>) = \f g x -> f x (g x) -- S
```

```
-- non-empty list
data NonEmpty a = a :| [a]
instance Applicative NonEmpty where
    pure a = a :| []
    (f :| fs) <*> (x :| xs) =
        (f x) :| (fs <*> xs)
```

Как переписать код на аппликативные функторы?

```
foo :: (a -> b -> c) -> a -> b -> c  
foo f a b = f a b
```



```
foo2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c  
foo2 f a b = pure f <*> a <*> b
```

И далее, если `foo2` использовать с аппликативом `Id`, то `foo2` будет вести себя как `foo`.

Идея: `pure` встраивает чистое вычисление в контекст, где совершается какой-то эффект, а дальше вычисления проходят абстрагируясь от этого эффекта

- identity

`pure id <*> v = v`

- composition

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

- homomorphism

`pure f <*> pure x = pure (f x)`

- interchange

`u <*> pure y = pure (\ f -> f y) <*> u`

Пример: аппликативные парсеры (1/4)

```
type Stream s = [s]
newtype Parser s a = P (Stream s -> [(a, Stream s)])

-- running parser on some input
run :: Parser s a -> Stream s -> [(a, Stream s)]
run (P f) = f

-- parser that returns constant value
pReturn :: a -> Parser s a
pReturn x = P $ \s -> [ (x,s) ]
```

Пример: аппликативные парсеры (2/4)

```
newtype Parser s a = P (Stream s -> [(a, Stream s)])
```

```
instance Functor (Parser s) where
```

```
  fmap :: (a -> b) -> Parser s a -> Parser s b
```

```
  -- fmap: apply function to every parser result
```

```
  fmap f (P p) = P $ \inp -> map (\ (a,t1) -> (f a, t1)) $ p inp
```

А ещё можно вспомнить, что наш парсер – композиция (почти) трех функторов: $((->)a)$, $[]$ и $((,)a)$ и сделать так:

```
instance Functor (Parser s) where
```

```
  fmap f (P p) = P $ fmap (fmap (first f)) p
```

```
first :: (a -> a') -> (a, b) -> (a', b)  -- Defined in 'Data.Tuple.Extra'
```

Пример: аппликативные парсеры (3/4)

```
pReturn :: a -> Parser s a
pReturn x = P $ \s -> [ (x,s) ]

-- apply parser that contains function to another parser
apply :: Parser s (a -> b) -> Parser s a -> Parser s b
apply (P pf) (P px) =
    P $ \inp ->
        flip concatMap (pf inp) $ \ (f, tl1) ->
        flip concatMap (px tl1) $ \ (x, tl2) ->
        [(f x, tl2)]

instance Applicative (Parser s) where
    pure = pReturn
    (<*>) = apply
```

Пример: аппликативные парсеры (4/4)

```
pFail :: Parser s a
pFail = P (const [])
```

```
alt :: Parser s a -> Parser s a -> Parser s a
alt (P p1) (P p2) = P (\ inp -> p1 inp ++ p2 inp)
```

```
-- monoid on applicative parsers
instance Alternative (Parser s) where
    empty = pFail
    (<|>) = alt
```


Аппликативные парсеры. Пример. Парсим несколько раз

```
-- запустить парсер, а если не удалось -- вернуть значение по-умолчанию  
opt :: forall s a . Parser s a -> a -> Parser s a  
p `opt` v = p <|> pReturn v
```

```
-- ноль или более вхождений парсера  
pMany :: forall s a . Parser s a -> Parser s [a]  
pMany p = pure (:) <*> p <*> opt (pMany p) []
```

```
-- одно или более вхождений парсера  
pMany1 :: forall s a . Parser s a -> Parser s [a]  
pMany1 p = pure (:) <*> p <*> pMany p
```

Аппликативные парсеры. Пример. Парсим буковки

```
pSym :: Eq s => s -> Parser s s
pSym c = P $ \case
  c2 : tail | c2 == c -> [(c, tail)]
  otherwise           -> []
```

```
pLettera :: Parser Char Char
```

```
pLettera = pSym 'a'
```

```
-- applicative style parsing
```

```
pString_aa :: Parser Char [Char]
```

```
pString_aa =
```

```
  pure (:) <*> pLettera <*> (pure (:[])) <*> pLettera)
```

Итог: аппликативные парсеры

Считается, что

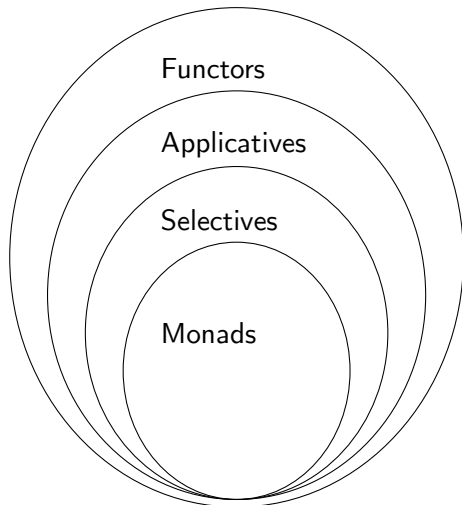
- Они имеют более стрёмный синтаксис, чем монадические
- Они не умеют в контекстно-зависимые языки, но это не совсем так [4]

Монады

```
--      |----- каждая монада также
--      |         u Applicative, u функтор
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
```

Для минимальной реализации нужно иметь (>>=), быть **Applicative** и соблюдать законы монад.

[Документация.](#)



Аллегория про конвейер

Монадическое значение $m\ a$ – конвейер, который создал a и готов его передать дальше.

Операция $return :: a \rightarrow m\ a$ "кладет" значение типа a на конвейер, чтобы с ним можно было дальше работать.

Вычисления типа $a \rightarrow m\ b$ обозначают конвейер, который готов принять значение типа a и на выходе из конвейера предоставить результат типа b .

Операция $(>>=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ состыковывает два конвейера в один большой типа $m\ b$.

Законы монад

- Левый нейтральный: $\text{return } a \gg= f \equiv f a$
- Правый нейтральный: $m \gg= \text{return} \equiv m$
- Ассоциативность (почти): $(m \gg= f) \gg= g \equiv m \gg= (\backslash x \rightarrow f x \gg= g)$

Монады должны соотноситься с аппликативами таким образом:

- `pure` = `return`
- `(<*>)` = `ap`

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap m1 m2 = m1 >>= \f -> m2 >>= \x -> return (f x)
```

Монады в стандартной библиотеке

```
-- identity
data Id a = Id a
instance Monad Id where
    return      = Id
    (Id x) >>= f = f x

-- Maybe
instance Monad Maybe where
    return = Just
    Just x  >>= f      = f x
    Nothing >>= _      = Nothing
```

```
instance Monad ((->) a) where
    f >>= k = \r -> k (f r) r
```

do-нотация

```
thing1 >>= (\x -> func1 x >>= (\y -> thing2  
  >>= (\() -> func2 y >>= (\z -> return z))))
```



```
thing1 >>= \x ->  
func1 x >>= \y ->  
thing2 >>= \() ->  
func2 y >>= \z ->  
return z
```



do

```
x <- thing1  
y <- func1 x  
thing2  
z <- func2 y  
return z
```

См. также

`{-# LANGUAGE ApplicativeDo #-}` [3]

Монады и парсеры

```
instance Monad (Parser s) where
  return :: a -> Parser s a
  return = pReturn

(>>=) :: Parser s a -> (a -> Parser s b) -> Parser s b
P pa >>= a2pb = P $
  flip concatMap (pa input) $ \ (x,t1) ->
    run (a2pb x) t1
  -- \input -> [ input2 | (a, input1) <- pa input
  --              , input2 <- run (a2pb a) input1 ]

  -- concatMap @[] :: (a -> [b]) -> [a] -> [b]
```

Парсим буковки ещё раз

```
pString_aa :: Parser Char [Char]    -- applicative style parsing
```

```
pString_aa =
```

```
  pReturn(·)
```

```
    <*> pSym 'a'
```

```
    <*> (pReturn (\x -> [x]) <*> pSym 'a')
```

```
pString_bb :: Parser Char [Char]    -- monadic style
```

```
pString_bb = pSym 'b' >=> \l -> pSym 'b' >=> \r -> return [l,r]
```

```
pString_cc :: Parser Char [Char]    -- monadic with do-notation
```

```
pString_cc = do
```

```
  l <- pSym 'c'
```

```
  r <- pSym 'c'
```

```
  return [l,r]
```

Монада List

```
instance Monad [] where
  return :: forall a . a -> [a]
  return x = [x]

  (>>=) :: forall a b . [a] -> (a -> [b]) -> [b]
  gs >>= f = concat $ fmap f gs
  -- for every element apply function f
  -- and convert [[b]] to [b]
  -- also know as `concatMap`
```

List comprehensions

```
[ (x,y) | x <- [1..3], y <- [1..2] ]
```



do

```
x <- [1..3]  
y <- [1..2]  
return (x,y)
```



```
[1..3] >>= \x -> [1..2] >>= \y -> return (x,y)
```

Всё это можно обобщить до *monad comprehensions*

Монада Writer (чтобы, например, писать в лог)

```
data MyWriterMonad a = W { unST :: (String, a) }
```

```
out :: String -> MyWriterMonad ()
```

```
out s = W (s, ())
```

```
instance Monad MyWriterMonad where
```

```
  return :: forall a . a -> MyWriterMonad a
```

```
  return x = W ([], x)
```

```
(>>=) :: forall a b .
```

```
  MyWriterMonad a -> (a -> MyWriterMonad b) -> MyWriterMonad b
```

```
W (s1,a) >>= f = W (printf "%s%s" s1 s2, b)
```

```
  where
```

```
    (s2, b) = unST (f a)
```

Монада Writer. Пример

```
-- Example 1
```

```
data Term = Con Int | Div Term Term
```

```
eval :: Term -> MyWriterMonad Int
```

```
eval (Con a) = do
```

```
  out $ printf "(const %d) " a
```

```
  return a
```

```
eval (Div l r) = do
```

```
  a <- eval l
```

```
  b <- eval r
```

```
  out $ printf "(div %d %d) " a b
```

```
  return (a `div` b)
```

```
print $ eval (Con 10)
```

```
-- (10,"(const 10) ")
```

```
print $ eval (Con 10 `Div` Con 2)
```

```
-- (5,"(const 10) (const 2) (div 10 2) ")
```

Монада State (1/2)

Позволяет делать вид, что мы работаем с изменяемыми переменными, хотя на самом деле это не так

```
data MyStateMonad s a = ST { unST :: s -> (a, s) }
```

```
runState :: MyStateMonad s a -> s -> (a, s)
```

```
runState st init = unST st init
```

```
get :: forall s . MyStateMonad s s
```

```
get = ST (\s -> (s,s))
```

```
put :: forall s . s -> MyStateMonad s ()
```

```
put s = ST $ \_ -> ((), s)
```

Монада State (2/2)

```
data MyStateMonad s a = ST { unST :: s -> (a, s) }
```

```
instance Monad (MyStateMonad s) where
```

```
  return :: forall a . a -> MyStateMonad s a
```

```
  return x = ST (\st -> (x, st))
```

```
(>>=) :: forall a b .
```

```
  MyStateMonad s a -> (a -> MyStateMonad s b) -> MyStateMonad s b
```

```
ST g >>= f = ST $
```

```
  \st0 ->
```

```
    let (ans1,st1) = g st0 in
```

```
    runState (f ans1) st1
```


Монада State. Пример. Считаем количество делений

```
data Term = Con Int | Div Term Term
```

```
eval :: Term -> MyStateMonad Int Int
```

```
eval (Con a) = return a
```

```
eval (Div l r) = do
```

```
  a <- eval l
```

```
  b <- eval r
```

```
  old <- get
```

```
  put (1+old)
```

```
  return (a `div` b)
```

```
...
```

```
print $ runState (eval (Con 10)) 111
```

```
print $ runState (eval (Con 10 `Div` Con 2)) 111
```

Монада IO – частный случай монады State

На монаду **IO** можно смотреть как на **State**, где состояние – окружающий мир

```
type IO a = RealWorld -> (a, RealWorld)
```

```
readLn :: RealWorld -> (String, RealWorld)      -- IO String
putStr  :: String -> RealWorld -> ((), RealWorld) -- String -> IO ()
```

Пример:

```
main = do a <- ask "What is your name?"
          b <- ask "How old are you?"
          return ()
```

```
ask s = do putStr s
           readLn
```



Демки на Haskell
Gitlab repo



Applicative programming with effects (Functional Pearl)
Conor McBride & Ross Paterson
PDF



Documentation

```
{-# LANGUAGE ApplicativeDo #-}
```



Parsing context-sensitive languages with Applicative
Brent Yorgey
Blog post



Question: What is a Comonad and how are they useful?

Cool intuition about monads is in
StackOverflow question