

## Функторы

Идея функторов пришла из очень современной математики. Функторы действуют подобно гомоморфизмам между алгебраическими системами (а именно, моноидами): сохраняют операции (композиции) и единицу (тождественное отображение). Только они отображают категории в категории... Таким образом, функторы сохраняют операции в новом окружении или «контексте».

Или другими словами, добавляют новый контекст к нашему типу! Сейчас, разберем подробно, что это означает...

---

### Определение 1.

Категория  $\mathcal{C}$  — это:

- класс объектов  $Ob_{\mathcal{C}}$ ;
- для каждой пары объектов  $A, B$  задано множество морфизмов (или стрелок)  $Hom_{\mathcal{C}}(A, B)$ , причём каждому морфизму соответствуют единственные  $A$  и  $B$ ;
- для пары морфизмов  $f \in Hom(A, B)$  и  $g \in Hom(B, C)$  определена композиция  $g \circ f \in Hom(A, C)$ ;
- для каждого объекта  $A$  задан тождественный морфизм  $id_A \in Hom(A, A)$ ;

причём выполняются две аксиомы:

- операция композиции ассоциативна:  $h \circ (g \circ f) = (h \circ g) \circ f$  и
- тождественный морфизм действует тривиально:  $f \circ id_A = id_B \circ f = f$  для  $f \in Hom(A, B)$ .

[wikipedia: Теория категорий](#)

### Определение 2.

Функтор  $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{D}$  ставит в соответствие каждому объекту категории  $\mathcal{C}$  объект категории  $\mathcal{D}$  и каждому морфизму  $f : A \rightarrow B$  морфизм  $F(f) : F(A) \rightarrow F(B)$  так, что

- $F(id_A) = id_{F(A)}$  и
- $F(g) \circ F(f) = F(g \circ f)$ .

---

Функторы в Haskell «закручены» вокруг основной категории `Hask`. Её объектами являются все типы языка Haskell, как встроенные, так и любые, объявленные пользователем. А морфизмами — все одноместные функции между ними. За счет каррирования — этого вполне достаточно. Единичными морфизмами являются функции `id` за счет полиморфизма перегруженные для каждого типа.

Таким образом, чтобы определить функтор, нам нужно задать два отображения: на объектах и на морфизмах. На объектах, т.е. на типах языка Haskell, отображение могут задавать полиморфные типы, которые требуют переменную типа (ещё их называют конструкторами типов — см. лекцию 6).

Если рассматривать так называемые «виды» (kinds), то речь идет о видах `*->*`, которые как раз соответствуют конструкторам типов **Maybe**, `[]`, в отличие от простых типов, вроде **Int**, **Double**, которые имеют вид `*`.

А на морфизмах отображение для каждого соответствующего типа данных задает перегружаемая функция `fmap`.

Таким образом, вводится класс типов **Functor**:

```
class Functor f where

    fmap :: (a -> b) -> f a -> f b
```

### [Data.Functor](#)

Но по большей части, начиная примерно с версии 7 `ghc` этот функционал уже встроен в **Prelude**.

Собственно, функтором в определённом выше математическом смысле здесь будет и тип-параметр (или конструктор типов) `f` (он будет задавать отображение на объектах, т.е. типах в `Hask`), и функция `fmap` (она будет задавать отображение на морфизмах, т.е. одноместных функциях между типами). Класс **Functor** описывает эти две части математического понятия функтора в терминах языка `Haskell`, и тоже будет называться функтором, но уже не в математическом смысле.

Для удобства определён синоним `<$>`, который в некотором смысле является аллюзией к `$`. Точнее говоря, оператор `<$>` является инфиксным синонимом функции `fmap`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
infixl 4 <$>
```

Примеры:

```
> show <$> Nothing
Nothing
> show <$> Just 3
Just "3"
```

то же самое что и

```
> fmap show Nothing
Nothing
> fmap show (Just 3)
Just "3"
```

кстати, сравнить с

```
> show $ Just 3
"Just 3"
```

Продолжим о функторах. Отметим, что программист обязан проверять для воплощений типов следующие законы функторов:

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

## Примеры (воплощений класса функторов)

### Списки

Воплощение класса **Functor** для типа списков наиболее понятное и естественно. Оно полностью повторяет возможности функции **map**.

```
instance Functor [] where
    fmap = map
```

Таким образом, мы можем использовать примеры из лекции 4

```
Prelude> xs = map (+1) [1..10] :: [Int]
```

и записать это как

```
Prelude> xs = fmap (+1) [1..10] :: [Int]
```

и даже

```
Prelude> xs = (+1) <$> [1..10] :: [Int]
```

### Maybe

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

Такой же пример, приспособленный к контейнеру **Maybe**:

```
Prelude> (+1) <$> Maybe 10 :: Maybe 10 Int
```

даст

```
Prelude> (+1) <$> Just 10 :: Maybe Int
Just 11
```

Более содержательный и практичный пример использования функтора **Maybe** разберем во второй половине лекции.

### Деревья

Для общего типа деревьев (розовых кустов) в [Data.Tree](#) есть следующее воплощение:

```
instance Functor Tree where
    fmap = fmapTree
```

```
fmapTree :: (a -> b) -> Tree a -> Tree b
fmapTree f (Node x ts) = Node (f x) (map (fmapTree f) ts)
```

Но для типа бинарных деревьев, разобранных на прошлой лекции (11-я или 8-я):

```
data ATree a = Empty |
    ABranch (ATree a) a (ATree a)
    deriving (Eq, Ord, Show, Read)
```

мы можем ввести своё воплощение:

```
instance Functor ATree where
  fmap _ Empty = Empty
  fmap f (ABranch ltr a rtr) =
    ABranch (fmap f ltr) (f a) (fmap f rtr)
```

и для конкретного дерева провести вычисление:

```
tree :: ATree Int
tree = ABranch
  (
    ABranch
      (ABranch Empty 1 Empty)
      2
      (ABranch Empty 3 Empty)
    )
  4
  (
    ABranch (ABranch Empty 5 Empty)
      6
      (ABranch
        (ABranch Empty 7 Empty)
        8
        (ABranch Empty 9 Empty)
      )
    )
  )
```

```
test = fmap (+10) tree
```

с таким выводом:

```
*Main> test
ABranch (ABranch (ABranch Empty 11 Empty) 12 (ABranch
Empty 13 Empty)) 14 (ABranch (ABranch Empty 15 Empty) 16
(ABranch (ABranch Empty 17 Empty) 18 (ABranch Empty 19 Empty)))
```

## Функции и Map

.... (доделать!!!)

## IO

```
instance Functor IO where
  fmap f x = x >>= (pure . f)
```

(это из документации, см. Source)

которую мы запишем в более привычных нам терминах:

```
instance Functor IO where
  fmap f action = do
    result <- action
    return (f result)
```

(Липовача, с.301)

Результатом отображения действия ввода-вывода с помощью чего-либо будет действие ввода-вывода, так что мы сразу же используем синтаксис **do** для склеивания двух действий и создания одного нового. В реализации для метода **fmap** мы создаём новое действие ввода-вывода, которое сначала выполняет первоначальное действие ввода-вывода, давая результату имя **result**. Затем мы выполняем **return (f result)**. Вспомните, что **return** — это функция, создающая действие ввода-вывода, которое ничего не делает, а только возвращает что-либо в качестве своего результата (обёрнутое типом **IO** в данном случае).

Действие, которое производит блок **do**, будет всегда возвращать результирующее значение своего последнего действия. Вот почему мы используем функцию **return**, чтобы создать действие ввода-вывода, которое в действительности ничего не делает, а просто возвращает применение **f result** в качестве результата нового действия ввода-вывода.

Рассмотрим пример:

```
main = do
  line <- getLine
  let line' = reverse line
  putStrLn $ "Вы сказали " ++ line' ++ " наоборот!"
```

С помощью **fmap** код будет проще:

```
main = do
  line <- fmap reverse getLine
  putStrLn $ "Вы сказали " ++ line ++ "наоборот!"
```

### Нетривиальный пример

...мой?!!! (не работает пока): из БА в группу и в кольцо...

### Якобы рабочий пример:

```
data Foo = Foo deriving Show
data Bar = Bar deriving Show
data Baz = Baz deriving Show
```

```
x :: Foo
x = Foo
```

```
y :: Foo -> Bar
y _ = Bar
```

```
z :: Bar -> Baz
z _ = Baz
```

```
main = print (z . y $ x)
```

Требования меняются, и вы обнаруживаете, что **x** может потерпеть неудачу, поэтому вы должны обновить его и все вызывающие его программы для прохождения через **Maybe Monad** следующим образом:

```
data Foo = Foo deriving Show
data Bar = Bar deriving Show
data Baz = Baz deriving Show
```

```
x :: Maybe Foo
x = Just Foo
```

```
y :: Maybe Foo -> Maybe Bar
y _ = Just Bar
```

```
z :: Maybe Bar -> Maybe Baz
z _ = Just Baz
```

```
main = print (z . y $ x)
```

Но... вам не нужно обновлять свои функции, чтобы «пролистать» значения в типе **Maybe**!

```
-- import Control.Applicative (<$>) - must import explicitly before GHC
7.10
-- import Data.Functor (<$>) - for now
-- or don't import anything, because Prelude has got (<$>) nowadays
```

```
data Foo = Foo deriving Show
data Bar = Bar deriving Show
data Baz = Baz deriving Show
```

```
x :: Maybe Foo
x = Just Foo
```

```
y :: Foo -> Bar
y _ = Bar
```

```
z :: Bar -> Baz
z _ = Baz
```

```
main = print (z . y <$> x)
```

[Real world fmap example](#)

[Функторы в языках программирования](#)

## Аппликативные функторы

Аппликативные функторы вводить из математических представлений уже сложно. Обычно, в литературе по Haskell их вводят исходя из частичного применения функций. Так, `fmap` работает с функцией одного аргумента (и об этом говорилось выше), но вот как раз настал черед рассмотреть возможность работы с функцией, например, двух аргументов.

Если у нас есть `Just 3`, и мы выполняем выражение `fmap (*) (Just 3)`, что мы получим? Из реализации экземпляра типа **Maybe** для класса **Functor** мы знаем, что если это значение **Just**, то функция будет применена к значению внутри **Just**. Следовательно, выполнение выражения

```
fmap (*) (Just 3)
```

```
> :t fmap (*) (Just 3)
```

```
fmap (*) (Just 3) :: Num a => Maybe (a -> a)
```

вернёт **Just** `((*) 3)`, что может быть также записано в виде **Just** `(3*)`, если мы используем сечения. Мы получаем функцию, обернутую в конструктор **Just**.

Но что если у нас есть значение функтора **Just** `(3*)` и значение функтора **Just** `5`, и мы хотим извлечь функцию из **Just** `(3*)` и отобразить с её помощью **Just** `5`? С обычными функторами у нас этого не получится, потому что они поддерживают только отображение имеющихся функторов с помощью обычных функций. Мы могли бы произвести сопоставление конструктора **Just** по образцу для извлечения из него функции, а затем отобразить с её помощью **Just** `5`, но мы ищем более общий и абстрактный подход, работающий с функторами.

(Липовача, С.313)

Вот именно для такой задачи и вводятся аппликативные функторы. Инструментарий для них определен в классе типов **Applicative**, который изначально был задан в модуле [Control.Applicative](#), но начиная с примерно 7-й версии `ghc`, большая часть функционала вынесена сразу **Prelude**

```
class (Functor f) => Applicative f where
```

```
  pure :: a -> f a
```

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

Функция `pure` фактически производит упаковку с помощью функтора `f`, переданного как тип-параметр (это конструктор типа). Или говорят, что функция `pure` помещает значение в некоторый контекст (контекст по умолчанию). Функция `<*>` является отдалённым «родственником» `<$>`, определённого для обычных функторов, но в отличие от `<$>`, который принимает первым аргументом одноместную функцию из `a` в `b`, `<*>` принимает такую функцию, но упакованную функтором `f`.

или вводится `liftA2` (его надо экспортировать с модулем [Control.Applicative](#))

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

```
liftA2 f x y = f <$> x <*> y
```

Она просто применяет функцию между двумя аппликативными значениями, скрывая при этом аппликативный стиль, который мы обсуждали. Однако она ясно демонстрирует, почему аппликативные функторы более мощны по сравнению с обычными. При использовании обычных функторов мы можем просто отображать одно значение функтора с помощью функций. При использовании аппликативных функторов мы можем применять функцию между несколькими значениями функторов. (Липовача, С.330)

Или можно рассматривать эту сигнатуру иначе:

```
liftA2 :: (a -> b -> c) -> (f a -> f b -> f c)
```

т.е. можно сказать, что функция `liftA2` берёт обычную бинарную функцию и преобразует её в функцию, которая работает с двумя аппликативными значениями.

И обратно:

```
(<*>) = liftA2 id
```

## Законы аппликативных функторов

### 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 ($ y) <*> u
```

Как следствие, должно выполняться условие:

```
pure f <*> x == fmap f x
```

Для примеров ниже эти законы выполнены, проверка ложится на программиста...

Утилиты: ... (доделать...): liftA (аналог fmap, но в контексте Applicative), liftA3, ... Alternatives, optional...

## Воплощения-примеры

### Списки

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [fx | f<-fs, x<-xs]
```

Вот такой пример находим в документации [haskell.wikibooks: Applicative functors](http://haskell.wikibooks.org/Applicative_functors)

```
> [(2*), (3*)] <*> [4,5]
[8,10,12,15]
```

Для одной функции можно сделать так:

```
> pure (2*) <*> [4..8]
[8,10,12,14,16]
```

Чаще встречается такое сочетание:

```
> (*) <$> [1..3] <*> [5..8]
[5,6,7,8,10,12,14,16,15,18,21,24]
```

или



```
> :m Control.Applicative
> liftA2 (*) [1..3] [5..8]
[5,6,7,8,10,12,14,16,15,18,21,24]
```

Можно рассмотреть и функции от большого числа аргументов:

```
> (\x y z -> x*y + z) <$> [1..3] <*> [5..8] <*> [20..30]
[25,26,27,28,29,30,31,32,33,34,35,26,27,28,29,30,
31,32,33,34,35,36,27,28,29,30,31,32,33,34,35,36,
37,28,29,30,31,32,33,34,35,36,37,38,30,31,32,33,
34,35,36,37,38,39,40,32,33,34,35,36,37,38,39,40,
41,42,34,35,36,37,38,39,40,41,42,43,44,36,37,38,
39,40,41,42,43,44,45,46,35,36,37,38,39,40,41,42,
43,44,45,38,39,40,41,42,43,44,45,46,47,48,41,42,
43,44,45,46,47,48,49,50,51,44,45,46,47,48,49,50,
51,52,53,54]
```

или так:

```
> Prelude Control.Applicative> (liftA2 (\x y z -> x*y + z)
[1..3] [5..8]) <*> [20..30]
```

## Maybe

Рассмотрим, как делается воплощение класса для этого типа данных:

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

Рассмотрим функцию `pure`, она должна что-то принять и обернуть в аппликативное значение. Мы написали `pure = Just`, потому что конструкторы данных вроде **Just** являются по сути обычными функциями, и здесь мы делаем частичное применение. Также можно было бы написать `pure x = Just x`.

Ещё есть определение оператора `<*>`. Извлечь функцию из значения **Nothing** нельзя, поскольку внутри него нет функции. Поэтому мы говорим, что если мы пробуем извлечь функцию из значения **Nothing**, результатом будет то же самое значение **Nothing**.

Вспомним, выше было определение класса `Applicative`:

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

и там было ограничение контекста **Functor**, поэтому в определении `<*>` мы можем применить отображение функтора `fmap`. Теперь можно считать, что первый параметр-функция обернут конструктором **Just**, а второй — либо **Nothing**, либо некоторое значение, тоже обернутое конструктором **Just**. Далее, уже работает `fmap`, применяя в итоге распакованную функцию к распакованному значению, и окончательно оборачивая всё в итоге конструктором **Just**.

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
```

Заметим, что `pure` действует в этом случае в точности как оборачивание конструктором `Just`. Сами примеры достаточно надуманны, сложно подобрать пример, где сама функция (а не её результат) находится в контейнере `Just`. Скорее в работе мы встретим что-то подобное

```
ghci> (*) <$> Just 3 <*> Just 9
Just 27
```

или

```
ghci> Prelude> :m Control.Applicative
Prelude Control.Applicative> liftA2 (*) (Just 3) (Just 9)
Just 27
```

---

Зачем нужны аппликативные функторы? Вопрос этот не очень простой. Чаще всего, в реальном программировании на Haskell вы их явно использовать будете не слишком часто и главное, не слишком скоро. Тем не менее, в мире Haskell в последние годы появилась совершенно четкая тенденция на выделение аппликативных функторов как промежуточных программных сущностей между обычными функторами и миром монад. Считается, что они проще чем монады, имеют меньше накладных расходов, и рекомендуется использовать их, где вся мощь функционала монад не нужна. Ловушка в том, что для использования монад уже подготовлен мощный синтаксический аппарат — как раз то, с чем мы с вами знакомимся как с «императивным языком» внутри языка Haskell. Но уже есть возможность использовать аппликативные функторы с `do`-нотацией.

### Бонусы...

- Не такие «напряжные» как монады в освоении и в программной реализации, но близкие им по возможностям.
- Хорошо вкладываются, как матрешки, разные типы. В монадах это чрезвычайно сложная, и понятийно, и программно, техника монадных трансформеров.

```
Prelude> :m Control.Applicative
Prelude Control.Applicative> liftA2 (*) <$> Just [1,2,3] <*> Just [4,5,6,7]
Just [4,5,6,7,8,10,12,14,12,15,18,21]
> liftA2 (liftA2 (*)) (Just [1,2,3]) (Just [4,5,6,7])
Just [4,5,6,7,8,10,12,14,12,15,18,21]
> (liftA2 . liftA2) (*) (Just [1..3]) (Just [4..7])
Just [4,5,6,7,8,10,12,14,12,15,18,21]
```

- Могут [Traversable](#)
- Есть `do`-нотация!

## do-нотация аппликативных функторов

В последних версиях компилятора ghc появилось расширение и соответствующая прагма

### [ApplicativeDo](#)

которые вносят семантику аппликативных функторов в привычную всем программистам Haskell **do**-нотацию из мира монад.

Рассмотрим пример, который был разобран выше:

```
test = (*) <$> Just 3 <*> Just 9
```

Теперь мы можем его представить в таком удобном виде:

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

```
test = do
  x <- Just 3
  y <- Just 9
  pure (x*y)
```

или даже таком:

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

```
test = do
  x <- Just 3
  y <- Just 9
  return (x*y)
```

при этом семантически мы работаем с аппликативным функтором, а не монадой! И это совсем недавнее решение.

Теоретические основы для данного расширения были подготовлены и рассмотрены в статье [Desugaring Haskell's do-Notation into Applicative Operations](#). Какие-то моменты можно прочесть на [StackOverflow](#):

### [ApplicativeDo in Haskell](#)

### [ApplicativeDo pragma and Applicative Functor in Haskell](#)

## Разбор реального примера

Ряд примеров есть на [wikibooks haskell](#):

[Haskell/Prologue: IO, an applicative functor](#)

[Haskell/Applicative functors](#)

[Haskell/Solutions/Applicative functors](#)

## Рабочий пример

Разберем такой рабочий пример (абстрагируемся от сброса буфера):

```
main = do
  putStrLn "Please enter your birth year"
  year <- getLine
  putStrLn $ "In 2020, you will be: " ++ show (2020 - read year)
```

Что случится, если мы введем не число? Программа вылетит по исключению:

```
Please enter your birth year
hello
main.hs: Prelude.read: no parse
```

Проблема в том, что ввод пользователя — это строка типа **String** (точнее **IO String**), и функция **read** пытается преобразовать (распарсить) эту строку в целое (напр., **Integer**) и выдает ошибку, если эта строка не может быть распознана как целое число. Но, не все строки могут быть целыми числами!

Более надежно будет подключить модуль **Safe**:

```
cabal install safe
```

(этот модуль заменяет многие привычные нам функции, особенно по обработке списков, на различные безопасные в плане выброса исключений варианты)

и использовать **readMay**, которая будет возвращать значение **Maybe Integer**. Посмотрим, как это работает:

```
Prelude> :m Safe
Prelude Safe> readMay "1980" :: Maybe Integer
Just 1980

Prelude Safe> readMay "hello" :: Maybe Integer
Nothing
```

Видим, вместо исключений возвращается либо **Just...**, либо **Nothing**.

Вот следующая, более надежная версия программы:

```
import Safe (readMay)

main = do
  putStrLn "Please enter your birth year"
  yearString <- getLine
  case readMay yearString of
    Nothing -> putStrLn "You provided an invalid year"
    Just year -> putStrLn $ "In 2020, you will be: " ++ show (2020 - year)
```

Разобьем код «по-хаскеловски» на чистую часть, вычисляющую возраст и формирующую ответ, и на «грязную часть» взаимодействия с пользователем.

```
import Safe (readMay)

displayAge maybeAge =
```

```

    case maybeAge of
      Nothing -> putStrLn "You provided an invalid year"
      Just age -> putStrLn $ "In 2020, you will be: " ++ show age

yearToAge year = 2020 - year

main = do
  putStrLn "Please enter your birth year"
  yearString <- getLine
  let maybeAge =
    case readMay yearString of
      Nothing -> Nothing
      Just year -> Just (yearToAge year)
  displayAge maybeAge

```

Функция `maybeAge` выглядит немного избыточной в плане «упаковок-распаковок **Just** и **Nothing**».

Но на наше счастье, **Maybe** является функтором, и мы можем «протащить» чистую функцию `yearToAge` внутрь контейнера **Maybe** с помощью функторного отображения `fmap` и избежать лишних «упаковок-распаковок **Just** и **Nothing**».

```

import Safe (readMay)

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided an invalid year"
    Just age -> putStrLn $ "In 2020, you will be: " ++ show age

yearToAge year = 2020 - year

main = do
  putStrLn "Please enter your birth year"
  yearString <- getLine
  let maybeAge = fmap yearToAge (readMay yearString)
  displayAge maybeAge

```

или могли записать строку так:

```
let maybeAge = yearToAge <$> (readMay yearString)
```

Усложним наш пример: вместо жёстко заданного 2020 года попросим пользователя ввести произвольный год:

```

import Safe (readMay)

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

main = do
  putStrLn "Please enter your birth year"

```

```

birthYearString <- getLine
putStrLn "Please enter some year in the future"
futureYearString <- getLine
let maybeAge =
    case readMay birthYearString of
        Nothing -> Nothing
        Just birthYear ->
            case readMay futureYearString of
                Nothing -> Nothing
                Just futureYear -> Just (futureYear - birthYear)
displayAge maybeAge

```

Тут, к сожалению, `fmap` нам уже не поможет...

Давайте, всё-таки попробуем!

Мы работаем с двумя значениями: `readMay birthYearString` и `readMay futureYearString`. Оба эти значения имеют тип **Maybe Integer**. И мы хотим применить к ним функцию `yearDiff`:

```

yearDiff :: Integer -> Integer -> Integer
yearDiff futureYear birthYear = futureYear - birthYear

```

Если мы вернемся к попытке использовать `fmap`, то, похоже, столкнемся с небольшой проблемой. Тип `fmap` — специализированный для **Maybe** и **Integer** — это

```
(Integer -> a) -> Maybe Integer -> Maybe a
```

(в предыдущем примере тип `a` был **Integer**)

Другими словами, он принимает функцию, которая принимает один аргумент **Integer** и возвращает значение некоторого типа `a`, принимает второй аргумент **Maybe Integer** и возвращает значение типа **Maybe a**. Но наша функция `yearDiff` самом деле принимает два аргумента, а не один. Так что `fmap` вообще нельзя использовать, верно?

На самом деле это не так. В Haskell есть замечательное свойство «частичное применение функций», в нашем случае, это будет вот так:

```

yearDiff :: Integer -> Integer -> Integer
yearDiff :: Integer -> (Integer -> Integer)

```

Тогда мы можем рассмотреть `fmap` таким образом:

```

fmap :: (Integer -> (Integer -> Integer))
      -> Maybe Integer -> Maybe (Integer -> Integer)

```

(здесь тип `a` становится **Integer -> Integer**)

Теперь частично применим `fmap` к `yearDiff`, получим:

```
fmap yearDiff :: Maybe Integer -> Maybe (Integer -> Integer)
```

Мы можем применить это к `readMay futureYearString` и получим:

```
fmap yearDiff (readMay futureYearString) :: Maybe (Integer -> Integer)
```

Но беда в том, что нам нужно как-то применить это значение типа `Maybe (Integer -> Integer)` к нашему `readMay birthYearString` типа `Maybe Integer`. Но в рамках только функторов непосредственно применить функцию типа `Maybe (Integer -> Integer)` к значению типа `Maybe Integer` мы не можем.

Для этого нам необходимо использовать аппликативные функторы!

И теперь мы подошли к нашей окончательной концепции: аппликативные функторы. Идея проста: мы хотим иметь возможность применить функцию, которая находится внутри функтора, к значению внутри функтора. Магический оператор для этого `<*>`. Давайте посмотрим, как это работает в нашем примере

```
import Safe (readMay)
-- import Control.Applicative ((<*>))

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

main = do
  putStrLn "Please enter your birth year"
  birthYearString <- getLine
  putStrLn "Please enter some year in the future"
  futureYearString <- getLine
  let maybeAge =
      fmap yearDiff (readMay futureYearString)
      <*> readMay birthYearString
  displayAge maybeAge
```

или заменить на более компактный код в нужной нам строке:

```
...
  let maybeAge = yearDiff
      <$> readMay futureYearString
      <*> readMay birthYearString
...
```

Но если мы хотим сделать проверку на отсеивание заведомо глупых отрицательных результатов:

```
backToFuture futureYear birthYear = if (futureYear - birthYear) < 0
                                     then Nothing
                                     else Just (futureYear - birthYear)
```

вместо

```
yearDiff futureYear birthYear = futureYear - birthYear
```

то это уже требует большей функциональности, чем дает аппликативный функтор (пришлось делать проверки и запаковки). Получается такой код:

```

import Safe (readMay)
import Data.Maybe
-- import Control.Applicative ((<$>), (<*>))

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

backToFuture futureYear birthYear = if (futureYear - birthYear) < 0
  then Nothing
  else Just (futureYear - birthYear)

main = do
  putStrLn "Please enter your birth year"
  birthYearString <- getLine
  putStrLn "Please enter some year in the future"
  futureYearString <- getLine
  let maybeAge = backToFuture
    <$> readMay futureYearString
    <*> readMay birthYearString
  displayAge maybeAge

```

и тестируем:

```

Please enter your birth year
1968
Please enter some year in the future
1980
In that year, you will be: Just 12

```

или получаем:

```

Please enter your birth year
1968
Please enter some year in the future
1900
In that year, you will be: Nothing

```

Но зато можно легко расширить по числу переменных:

```

import Safe (readMay)
-- import Control.Applicative ((<*>))

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear koef = (futureYear - birthYear) * koef

main = do

```



```

putStrLn "Please enter your birth year"
birthYearString <- getLine
putStrLn "Please enter some year in the future"
futureYearString <- getLine
putStrLn "Please enter some koef"
koefString <- getLine
let maybeAge = yearDiff
    <$> readMay futureYearString
    <*> readMay birthYearString
    <*> readMay koefString
displayAge maybeAge

```

Используем для нашего предыдущего примера **do**-нотацию:

```

{-# LANGUAGE ApplicativeDo #-}

import Safe (readMay)
-- import Control.Applicative ((<$>), (<*>))

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

maybeAge fS bS = do
  fI <- readMay fS
  bI <- readMay bS
  pure $ yearDiff fI bI

maybeAge fS bS = yearDiff <$> readMay fS <*> readMay bS

main = do
  putStrLn "Please enter your birth year"
  birthYearString <- getLine
  putStrLn "Please enter some year in the future"
  futureYearString <- getLine
  displayAge $ maybeAge futureYearString birthYearString

```

Отметим, что эта **do**-нотация специализированна для работы с аппликативными функторами.

Можно перейти к обычной **do**-нотации для монад:

```

import Safe (readMay)

displayAge maybeAge =
  case maybeAge of
    Nothing -> putStrLn "You provided invalid input"
    Just age -> putStrLn $ "In that year, you will be: " ++ show age

yearDiff futureYear birthYear = futureYear - birthYear

```

```

main = do
    putStrLn "Please enter your birth year"
    birthYearString <- getLine
    putStrLn "Please enter some year in the future"
    futureYearString <- getLine
    let maybeAge = do
        fI <- readMay futureYearString
        bI <- readMay birthYearString
        return $ yearDiff fI bI
    displayAge maybeAge

```

Она даст больше естественных расходов (в нашем случае, особо незаметных) и преимущества в гибкости по использованию императивного кода.

Упражнения по мотивам статьи

[M.Snoyman, Functor, Applicative, and Monad](#)

### Литература

1. C.McBride, R.Paterson. Applicative programming with effects
2. C.McBride, R.Paterson. Idioms: applicative programming with effects
3. Michael Rüegg. Applicative Functors in Haskell
4. S.Marlow, S.P.Jones, E.Kmett, A.Mokhov. Desugaring Haskell's do-Notation into Applicative Operations
5. Д.Н.Москвин. Функциональное программирование. Лекция 8. Аппликативные функторы

- 
1. [habr: Категория Hask](#)
  2. [habr: Эндофункторы категории Hask и их моноидальная структура](#)
  3. [habr: Монады с точки зрения теории категорий](#)

**Цель:** подготовка к монадам.