

Функциональное программирование

Лекция 9. Использование аппликативных функторов

Денис Николаевич Москвин

СПбАУ РАН

07.11.2017

- 1 Аппликативные парсеры
- 2 Класс типов `Alternative`
- 3 Класс типов `Traversable`

- 1 Аппликативные парсеры
- 2 Класс типов `Alternative`
- 3 Класс типов `Traversable`

- *Парсер* — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противном случае.
- Простейший, но неудобный парсер:
`type Parser a = String -> a`

- *Парсер* — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противном случае.
- Простейший, но неудобный парсер:
`type Parser a = String -> a`
- Версия 2 (храним неразобранный остаток):
`type Parser a = String -> (String,a)`

- *Парсер* — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противном случае.
- Простейший, но неудобный парсер:
`type Parser a = String -> a`
- Версия 2 (храним неразобранный остаток):
`type Parser a = String -> (String,a)`
- Версии 3,4 (умеем обрабатывать ошибки):
`type Parser a = String -> Maybe (String,a)`
`type Parser a = String -> Either String (String,a)`

- *Парсер* — программа, принимающая на вход строку и возвращающая некоторую структуру данных, если строка удовлетворяет заданной грамматике, или сообщение об ошибке в противном случае.
- Простейший, но неудобный парсер:
`type Parser a = String -> a`
- Версия 2 (храним неразобранный остаток):
`type Parser a = String -> (String,a)`
- Версии 3,4 (умеем обрабатывать ошибки):
`type Parser a = String -> Maybe (String,a)`
`type Parser a = String -> Either String (String,a)`
- Версия 5 (неоднозначные грамматики):
`type Parser a = String -> [(String,a)]`

- Выберем в качестве базовой Версию 3:
`type Parser a = String -> Maybe (String,a)`
- Возможное обобщение — абстрагироваться по типу входного потока:
`type Parser tok a = [tok] -> Maybe ([tok],a)`

- Выберем в качестве базовой Версию 3:
`type Parser a = String -> Maybe (String,a)`
- Возможное обобщение — абстрагироваться по типу входного потока:
`type Parser tok a = [tok] -> Maybe ([tok],a)`
- Итоговая версия, годная для реализации представителей:

```
newtype Parser tok a =  
  Parser {runParser :: [tok] -> Maybe ([tok],a)}
```

```
newtype Parser tok a =  
  Parser {runParser :: [tok] -> Maybe ([tok],a)}
```

```
charA :: Parser Char Char  
charA = Parser f where  
  f (c:cs) | c == 'A' = Just (cs,c)  
  f _                  = Nothing
```

Сессия GHCi

```
GHCi> runParser charA "ABC"  
Just ("BC",'A')  
GHCi> runParser charA "BCD"  
Nothing
```

Функции для конструирования парсеров

```
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser f where
  f (c:cs) | pr c = Just (cs,c)
  f _           = Nothing
```

```
GHCi> runParser (satisfy isUpper) "ABC"
Just ("BC",'A')
GHCi> runParser (satisfy isLower) "ABC"
Nothing
```

```
lower :: Parser Char Char
lower = satisfy isLower

char :: Char -> Parser Char Char
char c = satisfy (== c)
```

Парсер как функтор

```
digit :: Parser Char Int
digit = satisfy isDigit    -- returns Char :(
```

Парсер как функтор

```
digit :: Parser Char Int
digit = satisfy isDigit    -- returns Char :(
```

```
digit :: Parser Char Int
digit = digitToInt <$> satisfy isDigit
```

Парсер как функтор

```
digit :: Parser Char Int
digit = satisfy isDigit    -- returns Char :(
```

```
digit :: Parser Char Int
digit = digitToInt <$> satisfy isDigit
```

Для этого нужно сделать парсер функтором:

```
instance Functor (Parser tok) where
--fmap :: (a -> b) -> Parser tok a -> Parser tok b
  fmap g (Parser p) = Parser f where
    f xs = case p xs of
      Just (cs, c) -> Just (cs, g c)
      Nothing      -> Nothing
```

Парсер как функтор (2)

Предыдущее объявление функтора может быть записано компактнее.

```
newtype Parser tok a =  
  Parser {runParser :: [tok] -> Maybe ([tok],a)}
```

Действительно, тип `Parser` — не что иное, как последовательная композиция трех функторов: `(->)` `[tok]`, `Maybe` и `(,)` `[tok]`. Поэтому

```
instance Functor (Parser tok) where  
  --fmap :: (a -> b) -> Parser tok a -> Parser tok b  
  fmap g (Parser p) = Parser $ (fmap . fmap . fmap) g p
```

```
instance Applicative (Parser tok) where
--pure :: a -> Parser tok a
  pure x = Parser $ \s -> Just (s, x)
  ...
```

СЕМАНТИКА: pure — парсер, всегда возвращающий заданное значение; входная строка не «потребляется».

```
GHCi> runParser (pure 42) "ABCD"
Just ("ABCD",42)
```


Аппликативный парсер: (<*>)

```
--(<*>) :: Parser tok (a -> b) -> Parser tok a -> Parser tok b
Parser u <*> Parser v = Parser f where
  f xs = case u xs of
    Nothing      -> Nothing
    Just (xs', g) -> case v xs' of
      Nothing      -> Nothing
      Just (xs'', x) -> Just (xs'', g x)
```

СЕМАНТИКА: получить результат первого парсера, затем второго на остатке строки, и применить первый ко второму. Неудача хотя бы одного парсера приводит к тотальной неудаче.

```
GHCi> runParser (pure (,) <*> digit <*> digit) "12AB"
Just ("AB", (1,2))
GHCi> runParser ((,) <$> digit <*> digit) "1AB2"
Nothing
```

Теперь можем строить «сложные» парсеры:

```
multiplication :: Parser Char Int
multiplication = (*) <$> digit <*> char '*' <*> digit
```

Все операторы левоассоциативны и имеют один (4) приоритет.

GHCi

```
GHCi> runParser multiplication "6*7"
Just ("",42)
```

А как сделать универсальный парсер для "63*796"?

- 1 Аппликативные парсеры
- 2 Класс типов `Alternative`
- 3 Класс типов `Traversable`

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Некоторые аппликативные функторы являются, помимо всего прочего, моноидами (списки, Maybe). Например,

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing 'mappend' m           = m
  m       'mappend' Nothing    = m
  Just m1 'mappend' Just m2    = Just (m1 'mappend' m2)
```

Полезны и другие способы сделать Maybe моноидом, например

```
instance Monoid (Maybe a) where
  mempty = Nothing
  Nothing 'mappend' m = m
  m       'mappend' _ = m
```

Поскольку нельзя объявить двух представителей для одного типа, в стандартной библиотеке используется упаковка

```
newtype First a = First { getFirst :: Maybe a }
```

В отличие от предыдущей реализации, параметризующий Maybe тип `a` совершенно не важен.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

infixl 3 <|>
```

Наделяем аппликативный функтор дополнительной моноидальной операцией с семантикой «сложения».

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

Представитель Alternative для списков полностью повторяет определение моноида для списка.

```
instance Alternative Maybe where
  empty = Nothing

  Nothing <|> m = m
  m       <|> _ = m
```

Представитель Alternative для Maybe ведёт себя, как упаковка First, возвращая первый не-Nothing в цепочке альтернатив:

Сессия GHCi

```
*Fp11> Nothing <|> (Just 3) <|> (Just 5) <|> Nothing
Just 3
```

Законы Alternative (1) и (2)

Помимо законов моноидальной структуры требуют выполнения

(1) Right distributivity of $\langle * \rangle$

$$(f \langle | \rangle g) \langle * \rangle a \equiv (f \langle * \rangle a) \langle | \rangle (g \langle * \rangle a)$$

(2) Right absorption for $\langle * \rangle$

$$\text{empty} \langle * \rangle a \equiv \text{empty}$$

Эта пара законов описывает связь Alternative и Applicative.

Выполняются ли они для Maybe?

Списков?

(3) Left distributivity of fmap

$$f \langle \$ \rangle (a \langle | \rangle b) \equiv (f \langle \$ \rangle a) \langle | \rangle (f \langle \$ \rangle b)$$

(4) Left absorption for fmap

$$f \langle \$ \rangle \text{empty} \equiv \text{empty}$$

Эта пара законов описывает связь Alternative и Functor.

Выполняются ли они для Maybe?

Списков?

```
instance Alternative (Parser tok) where
--empty :: Parser tok a
  empty = Parser $ \_ -> Nothing
--(<|>) :: Parser tok a -> Parser tok a -> Parser tok a
  Parser u <|> Parser v = Parser f where
    f xs = case u xs of
      Nothing -> v xs
      z         -> z
```

СЕМАНТИКА:

- empty — парсер, всегда возвращающий неудачу;
- (<|>) — пробуем первый, при неудаче пробуем второй на исходной строке.

Пример использования альтернативного интерфейса

```
GHCi> runParser (char 'A' <|> char 'B') "ABC"  
Just ("BC", 'A')  
GHCi> runParser (char 'A' <|> char 'B') "BCD"  
Just ("CD", 'B')
```

Теперь можем смонтировать рекурсивный парсер

```
lowers :: Parser Char String  
lowers = (:) <$> lower <*> lowers <|> pure ""
```

```
GHCi> runParser lowers "abCd"  
Just ("Cd", "ab")  
GHCi> runParser lowers "abcd"  
Just ("", "abcd")  
GHCi> runParser lowers "Abcd"  
Just ("Abcd", "")
```

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  -- One or more.
  some :: f a -> f [a]
  some v = some_v where
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v

  -- Zero or more.
  many :: f a -> f [a]
  many v = many_v where
    many_v = some_v <|> pure []
    some_v = (:) <$> v <*> many_v

infixl 3 <|>
```

- 1 Аппликативные парсеры
- 2 Класс типов `Alternative`
- 3 Класс типов `Traversable`

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

GHCi

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

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

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

`sequenceA`: обеспечиваем правило коммутации нашего функтора `t` с произвольным аппликативным функтором `f`. Структура внешнего контейнера `t` сохраняется, а аппликативные эффекты внутренних `f` объединяются в результирующем `f`. `traverse` — это `map` с эффектами: проезжаем по структуре `t a`, последовательно применяя функцию к элементам типа `a` и монтируем в точности ту же структуру из результатов типа `b`, параллельно «коллекционируя» эффекты.

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

```
instance Traversable [] where
  traverse _ [] = pure []
  traverse g (x:xs) = (:) <$> g x <*> traverse g xs
```

Сравним с реализацией Functor:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap g (Just x) = Just $ g x
instance Functor [] where
  fmap _ [] = []
  fmap g (x:xs) = (:) (g x) (fmap g xs)
```



```
newtype Identity a = Identity {runIdentity :: a}

instance Functor Identity where
  fmap g (Identity x) = Identity (g x)

instance Applicative Identity where
  pure = Identity
  Identity g <*> v = fmap g v
```

(1) identity

```
traverse Identity ≡ Identity
```

```
GHCi> traverse Identity [1,2,3]
Identity [1,2,3]
```

Всякий Traversable — это Functor:

```
fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault g = runIdentity . traverse (Identity . g)
```

- Identity :: b -> Identity b
- (Identity . g) :: a -> Identity b
- traverse (Identity . g) :: Identity (t b)
- runIdentity . traverse (Identity . g) :: t b

Законы Traversable (2) и (3)

(2) composition

```
traverse (Compose . fmap g2 . g1) ≡  
  Compose . fmap (traverse g2) . traverse g1
```

(3) naturality

```
t . traverse g ≡ traverse (t . g)
```

где $t :: (\text{Applicative } f, \text{Applicative } g) \Rightarrow f \ a \rightarrow g \ a$
произвольный аппликативный гомоморфизм, то есть функция
удовлетворяющая требованиям (1) $t \ (\text{pure } x) = \text{pure } x$; (2)
 $t \ (x \langle * \rangle y) = t \ x \langle * \rangle t \ y$.

Законы Traversable дают следующие гарантии:

- Траверсы не пропускают элементов.
- Траверсы посещают элементы не более одного раза.
- `traverse pure ≡ pure`.
- Траверсы не изменяют исходную структуру — она либо сохраняется, либо полностью исчезает.

```
GHCi> traverse Just [1,2,3]
Just [1,2,3]
GHCi> traverse (const Nothing) [1,2,3]
Nothing
```