CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 6. Классы типов для описания абстракций над вычислительными контекстами

В. Н. Брагилевский

14 марта 2018 г.

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

Понятие вычислительного контекста

- `v :: a` — значение типа `a`
- `w :: C a` — значение типа `a` в контексте `C`
- `f :: a1 -> a2 -> ... -> C a` — функция, результатом которой является значение в контексте

Таким образом, контекст — это тип сорта `* -> *`.

Примеры вычислительных контекстов

- **`IO`** — вычисление с потенциально возможными побочными эффектами (пример: `getLine :: IO String`).
- **`Maybe`** — вычисление с возможной неудачей (пример: `Just 5` или **`Nothing`**).
- **`Either a`** — вычисление с возможной неудачей и сообщением об ошибке типа `a` (пример: `Right 'x'` или `Left "Incorrect File Format"`).
- **`[]`** — вычисление с недетерминированным результатом (пример: `[1,3,5]`).

Зачем нужно учитывать вычислительный контекст?

- Контроль
- Выразительность

Зачем нужно абстрагировать понятие вычислительного контекста?

- Вычислительные контексты поддерживают операции, общие для всех контекстов.
- Можно описывать процессы, работающие независимо от контекста, и применять их к любым контекстам.

Основной инструмент абстракции в Haskell — классы типов.

Класс Functor

Определение класса Functor

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

- f — это вычислительный контекст.
- Функтор — преобразование значения с сохранением контекста.

Функтор для списка — функция map

```
fmap :: Functor f => (a -> b) -> f a -> f b
map :: (a -> b) -> [a] -> [b]

instance Functor [] where
    fmap = map

ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Функтор для Maybe

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

ghci> fmap (++"!!!") (Just "Hello")
"Hello!!!"
ghci> fmap (++"!!!") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Функтор для Either a

```haskell
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left y) = Left y
```

```
ghci> fmap (+2) (Right 5)
Right 7
ghci> fmap (+2) (Left "Сообщение об ошибке")
Left "Сообщение об ошибке"
```

Загадка

```
ghci> fmap (+1) (1,2)
(1,3)
```

Функтор для IO

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

main = do
   par1 <- fmap head getArgs
   putStrLn par1

main = do
   line <- fmap (++"!") getLine
   putStrLn line
```

Функтор для IO

```
main = do
    line <- fmap (reverse . map toUpper) getLine
    putStrLn line
```

Функтор позволяет применять функцию к результату вычисления без изменения контекста.

Примеры обобщённых вычислений с функторами

```haskell
inc :: (Functor t, Num a) => t a -> t a
inc v = fmap (+1) v
```

```
ghci> inc (Just 10)
Just 11
ghci> inc (Right 10)
Right 11
ghci> inc [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

```haskell
inc :: (Functor t, Num a) => t a -> t a
inc v = fmap (+1) v

readInt :: IO Int
readInt = do
  s <- getLine
  return $ read s
```

```
ghci> inc readInt
10
11
```

Идеи функторов

- Функтор — это обобщение некоторого поведения — преобразование значения с сохранением особенностей вычислительного контекста.
- Одинаковый код для преобразования значения в контексте вне зависимости от контекста.
- С учётом каррирования fmap можно рассматривать как «подъём функции» (lift):

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

Законы функторов

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

- Эти законы гарантируют, что структура контейнера или вычислительный контекст не будут затронуты функтором.

Пример нарушения законов

```
instance Functor [] where
  fmap _ [] = []
  fmap g (x:xs) = g x : g x : fmap g xs
```

Проблема многопараметрических функций

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

- В результате применения `fmap` функция оказывается внутри контекста.
- Применить её дальше средствами функторов не удаётся.

Класс Applicative

Определение аппликативного функтора

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

- Две функции:
  - помещение значения в контекст;
  - преобразование значения в контексте с помощью функции, находящейся в контексте.

На что похоже <*>?

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

Вспомогательные операции из Applicative

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

Экземпляр для Maybe

- Помещение в контекст (pure).
- Применение функции из контекста (<*>).

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

Примеры

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"!!!") <*> Nothing
Nothing
ghci> Nothing <*> Just "ololo"
Nothing
```

Многопараметрические функции (аппликативный стиль)

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

Аппликативный стиль

Функция <$>

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

```
ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing
ghci> (+) <$> Nothing <*> Just 5
Nothing
```

Экземпляр для списков

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

```
instance Applicative [] where
   pure :: a -> [a]
   (<*>) :: [a->b] -> [a] -> [b]
   ...
```

- Как можно реализовать эти функции?
- Возможны различные реализации, например: склейка списков, недетерминированные вычисления.

Экземпляр для списка: недетерминированные вычисления

- Простейший контекст: одноэлементный список.
- Список функций $\to$ список значений $\to$ применение каждой функции к каждому значению.

```
instance Applicative [] where
  pure x = [x]
  gs <*> xs = [ g x | g <- gs, x <- xs ]
```

Пример: недетерминированные вычисления

```
ghci> (*) <$> [1,2,3] <*> [4,5]
[4,5,8,10,12,15]
ghci> :t (*) <$> [1,2,3]
(*) <$> [1,2,3] :: Num a => [a -> a]
ghci> length $  (*) <$> [1,2,3]
3
ghci> (++) <$> ["abc", "def"] <*> pure "!!!"
["abc!!!","def!!!"]
```

Обёртка для типа списка (Control.Applicative)

```haskell
newtype ZipList a = ZipList { getZipList :: [a] }
```

```
ghci> :t ZipList [1,2,3]
ZipList [1,2,3] :: Num a => ZipList a
ghci> getZipList $ ZipList [1,2,3]
[1,2,3]
```

Экземпляр для списка: склейка списков

```haskell
instance Applicative ZipList where
  pure = undefined
  (ZipList gs) <*> (ZipList xs) =
        ZipList (zipWith ($) gs xs)
```

- Что делать с pure?
- Нужен бесконечный список!

```haskell
  pure x = ZipList (repeat x)
```

Примеры: склейка списков

```
ghci> getZipList $ (+) <$> ZipList [1,2,3]
                       <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> pure 100
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3]
                       <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "foo"
                        <*> ZipList "bar"
                        <*> ZipList "tre"
[('f','b','t'),('o','a','r'),('o','r','e')]
ghci> :t (,,)
(,,) :: a -> b -> c -> (a, b, c)
```

Экземпляр для IO

```haskell
instance Applicative IO where
    pure = return
    a <*> b = do
          f <- a
          x <- b
          return (f x)
```

Примеры: IO как аппликативный функтор

```haskell
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b

myAction :: IO String
myAction = (++) <$> getLine <*> getLine

main = do
    a <- myAction
    putStrLn $ "Две строки, соединённые вместе: "
               ++ a
```

Примеры: IO как аппликативный функтор

Последовательное выполнение действий

```
ghci> putStr "Your name: " *> getLine
Your name: John
"John"

(*>) :: Applicative f => f a -> f b -> f b


ghci> (,) <$> (putStr "Your name: " *> getLine)
          <*> (putStr "Your age: " *> getLine)
Your name: John
Your age: 20
("John","20")
```

Проблема

```haskell
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
```

- Как ввести строку, а затем вывести её на консоль?
- Наблюдение: второе действие зависит от результата первого.
- Вывод: возможностей аппликативных функторов недостаточно.

Класс Monad

Проблема и её решение: класс типов Monad

- Как ввести строку (`getLine`), а затем её напечатать (`putStr`)?
- Наблюдение: необходимо скомбинировать два действия, причём второе зависит от результата первого.

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- Операция `>>=` называется монадическим связыванием (bind)
- `(>>)` игнорирует результат первого вычисления
- `return` — это то же самое, что `pure`

Примеры

```
ghci> getLine >>= putStrLn
abc
abc
ghci> getLine >>= putStr >> putStrLn "!"
hello
hello!

ghci> Just 7 >>= (\x -> Just $ x+1)
Just 8
ghci> Nothing >>= (\x -> Just $ x+1)
Nothing
ghci> [1,2] >>= (\x -> [x-1, x+1])
[0,2,1,3]
```

Синтаксис монадических операций: do-блоки

```
do
  action1
  action2
```

```
action1 >> action2
```

```
do
  pat <- expr
  morelines
```

```
expr >>= (\pat -> do
                  morelines)
```

- Например:

```
do
  x <- action
  process x
```

```
action >>= process
```

```
do
  x <- action
  return x
```

```
action
```

Монада IO

- Область применения: ввод-вывод.
- Эффект: наличие побочных эффектов во время вычислений.

Пример

Вычислить количество строк в файле, имя которого задано в параметрах командной строки.

Решение

```
main = do
  fname <- fmap head getArgs
  content <- readFile fname
  print $ length $ lines content
```

Решение без do-блока

```
main = head <$> getArgs >>= readFile
                        >>= print.length.lines
```

Приоритеты (`:info`)

```
infixl 4 <$>
infixl 1 >>=
infixr 9 .
```

Решение с явными приоритетами

```
main = ((head <$> getArgs) >>= readFile)
                   >>= (print.(length.lines))
```

Maybe: экземпляр класса типов Monad

```haskell
instance Monad Maybe where
    return x = Just x

    Nothing >>= f = Nothing
    Just x  >>= f = f x
```

- Область применения: вычисления с возможной неудачей.
- Эффект: не нужно проверять предыдущую операцию на неудачу.

```
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

Список как монада

```
instance Monad [] where
    return :: a -> [a]
    return x = [x]
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

- Недетерминированные вычисления — функция (возвращающая список) применяется к каждому элементу исходного списка.

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
ghci> [1,2] >>= \n -> ['a','b']
            >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации для списков

```
ghci> [1,2] >>= \n -> ['a','b']
              >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

```
ghci> listOfTuples
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Монады: выхода нет

```haskell
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
```

- Поместить в монаду значение можно, а извлечь нельзя.
- Для конкретных монад можно написать соответствующую функцию (`fromJust`).

Примеры контекстов

- `IO` — вычисление с потенциально возможными побочными эффектами.
- `Maybe` — вычисление с возможной неудачей.
- `Either a` — вычисление с возможной неудачей и сообщением об ошибке типа `a`.
- `[]` — вычисление с недетерминированным результатом.

Примеры общего поведения

- `Functor` — изменение значения в контексте без изменения контекста.
- `Applicative` — применение функции в контексте к значению в контексте.
- `Monad` — зависимость вычисления в контексте от результата предыдущего вычисления в контексте.