Лекции по Функциональному Програмированию 5 семестр

Ilya Yaroshevskiy

13 мая 2023 г.

Оглавление

Лекция 0

1.1 Лямбда Исчисление

Определение. Пусть X, Y — непустые множества. Отношение $R \subseteq X \times Y$ — **функциональное** отношение если Доделать

Определение. Пусть $Var = \{x_0, x_1, x_2\}$ — счетное множество переменных. Множество **предтермов** производится следующей грамматикой:

$$M, N \coloneqq x \mid (\lambda x.M) \mid (MN)$$

Примечание. Хотим отождествлять функции вообще говоря одинаковых, но различных графически(имена переменных).

Пусть α-конверсия — переименование связанных переменных. Введем отношение α-эквивалентности — рефлексивное, транзитивное, симметричное замыкание α-конверсии, которое отождествляет такие функции.

Определение. Лямбда терм — предтерм с точностью до α-эквивалентности

Определение. Лямбда трем M **β-редуцируется** к N, если есть способ переписать его по следующим правилам:

- $(\lambda x.M)N \to_\beta M[x \coloneqq N]$
- $\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N}$
- $\frac{M_1 \to_\beta M_2}{N M_1 \to_\beta N M_2}$

Примечание. Терм $(\lambda x.M)N$ называется редуцируемым.

Определение. Терм называется **слабо нормализуемым**, если существует путь который заканчивается(приходим к терму который не редуцируется). **Сильно нормализуемым**, если любой путь заканчивается.

Пример.

$$\lambda xy.x(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$$

Можем проредуцировать его двумя способами:

1.

$$
\begin{aligned}
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&(\lambda y.[x \coloneqq (\lambda z.z)]((\lambda x.xx)(\lambda x.xx))) \to_\beta \\
&(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&(\lambda z.z)[y \coloneqq (\lambda x.xx)(\lambda x.xx)] \to_\beta \\
&\lambda z.z
\end{aligned}
$$

2.

$$
\begin{aligned}
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&(\lambda xy.x)(\lambda z.z)(xx)[x \coloneqq [\lambda x.xx]] \to_\beta \\
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&\dots
\end{aligned}
$$

Примечание. Два способа редуцировать:

- $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$ — **аппликативный** порядок. Сначала редуцируем $(N_i)_{i \in \{1,\dots,n\}}$
- $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$ — **нормальный** порядок. Сначала редуцируем $(\lambda x_1 \dots x_n.M)N_1$, и т.д.

Теорема 1.1.1. Пусть M — терм, такой что есть нормальная форма M'
Тогда M может быть редуцирован к M' с помощью нормального порядка редукции.

1.1.1 Типы

Примечание. Система типов — синтаксический формализм который позволяет доказывать наличие поведенческий особенностей программы.

Определение. Просто типизированное лямбда-исчисление

- $\frac{}{\Gamma, x:A \vdash x:A}$ — аксиома
- $\frac{\Gamma, x:A M:B}{\Gamma \lambda x.M:A \to B}$
- $\frac{\Gamma \vdash M:A \to B \quad \Gamma \vdash N:A}{\Gamma \vdash MN:B}$

Примечание. $x : A$ — x принадлежит типу A.

Примечание. Γ — конечный набор утверждений, которые говорят что какая-то переменная имеет какой-то тип.

Примечание. $\Gamma \vdash N : A$ — С использованием переменных из Γ можем построить терм N имеющий тип A.

Примечание. В типизированном лямбда-исчислении функция типа $A \to B$ — функция высшего порядка, если $A : C \to D$

```
changeTwiceBy :: (Int -> Int) -> Int -> Int
changeTwiceBy operation value = operation (operation value)
```

```
changeTwiceBy :: (String -> String) -> String -> String
changeTwiceBy operation value = operation (operation value)
```

Можем написать одну функцию для абстрактного типа `a`:

```
changeTwiceBy :: (a -> a) -> a -> a
changeTwiceBy operation value = operation (operation value)
```

1.1.2 Система F

Примечание. Версия Curry:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : p.A} \quad p \notin rng(\Gamma)$$

$$\frac{\Gamma \vdash M : \forall p.A}{\Gamma M : A[p \coloneqq B]}$$

Примечание. Версия Church: Исправить

$$\frac{\Gamma, p \vdash M : A}{\Gamma \vdash \forall p.M : \forall p.A}$$

$$\frac{\Gamma \forall p.A}{\Gamma \vdash M[B] : A[p \coloneqq B]}$$

Примечание. Система типов Хиндли-Миллера

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \vec{p}.A} \quad \vec{p} \notin rng(\Gamma)$$

$$\frac{\Gamma \vdash M : \forall p.A}{\Gamma \vdash M : A[p \coloneqq B]}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \lambda x.M : A \to B}$$

, где A — свободная от кванторов.

Лекция 4

2.1 Semigroup

```haskell
class Semigroup m where
  (<>):: m -> m -> m
```

Ассоциатвиный закон для `Semigroup`:

1. $(x$ `<>` $y)$ `<>` $z \equiv x$ `<>` $(y$ `<>` $z)$

Пример.

```haskell
instance Semigroup [a] where
  (<>) = (++)
```

2.2 Monoid

```haskell
class Semigroup m => Monoid m where
  mempty :: m
```

`Identity` Закон для `Monoid`:

1. x `<>` mempty $\equiv x$
2. mempty `<>` $x \equiv x$

2.3 Foldable

```haskell
class Foldable t where
  {-# MINIMAL foldMap | foldr #-}

  fold    :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr   :: (a -> b -> b) -> b -> t a -> b
```

Примечание. `length` определен через `Foldable`

Лекция 5

3.1 Functor

Определение.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Примечание. (<$>) = $fmap$ — инфиксный оператор

Пример.

```
Prelude> :kind (->)
(->) :: * -> * -> *
Prelude> :kind (->) Int
(->) Int :: * -> *
```

```
instance Functor ((->) r)  where
  fmap :: (a -> b) -> (r -> a) -> r -> b
  fmap = (.)
```

3.1.1 Законы

1. `fmap id` ≡ `id`
2. `fmap (f . g)` ≡ `fmap f . fmap g`

3.2 Applicative

Определение.

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

Пример.

```haskell
instance Applicative ((->) r) where
  pure :: a -> r -> a
  pure x = \_ -> x

  (<*>) :: (r -> a -> b) -> (r -> a) -> r -> b
  f <*> g = \x -> f x (g x)
```

3.2.1 Законы

1. `pure id <*> v ≡ b`
2. `pure ( . ) <*> u <*> v <*> w ≡ u <*> (v <*> w)`
3. `pure f <*> pure x ≡ pure (f x)`
4. `u <*> pure y ≡ pure($ y) <*> u`

3.3 Alternative

Определение.

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

Пример.

```haskell
instance Alternative Maybe where
  empty :: Maybe a
  empty = Nothing

  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> r = r
  l       <|> _ = l
```

```haskell
instance Alternative [] where
  empty :: [a]
  empty = []

  (<|>) :: [a] -> [a] -> {a}
  <|> = (++)
```

3.4 Traversable

Определение.

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

Пример.

```haskell
instance Traversable Maybe where
  traverse :: Applicative f => (a -> f b) -> Maybe a -> f (Maybe b)
  traverse _ Nothing = pure Nothing
  traverse f (Just x) = Just <$> f x
```

```haskell
instance Traversable [] where
  traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
  traverse f = foldr consF (pure [])
    where
      consF x ys = (:) <$> f x <*> ys
```