Функциональное программирование

Михайлов Максим

28 сентября 2021 г.

Оглавление

Лекция 1 3 сентября **2**
0.1 История . . . 2
0.2 Функции в теории множеств . . . 2
0.3 Лямбда-исчисление, базовые определения . . . 3
0.4 Методы редукции . . . 4
0.5 Типы . . . 4
0.6 Типизированное лямбда-исчисление . . . 5
0.7 Полиморфизм . . . 5

Лекция 1

3 сентября

Эта лекция обзорная и рукомахательная. Читать для общего развития.

В рамках этого курса мы будем изучать язык Haskell, названный в честь Хаскелла Карри, американского логика. Последний стандарт этого языка — `Haskell2010` и его основной компилятор — `ghc`. У него открыт исходный код, можно предлагать свои proposal'ы, которые фильтрует сообщество и комитет.

0.1 История

В двадцатых годах XX века рассматривались вопросы основ математики. Алонзо Чёрч предложил альтерантиву теории множеств — **лямбда-исчисление**, которое основывается на понятии функции. Парадокс Клини-Россера показал, что начальная версия лямбда-исчисления противоречива, поэтому Чёрч ввел **типы** в лямбда-исчисление в сороковых годах.

Типы появились раньше[1] в рамках теории типов Бертрана Рассела и Альфреда Норта Уайтхеда. После работ Чёрча, теория типов развивалась Хаскеллом Карри и Уильямом Ховардом как часть теории доказательств в 50-60 годах. В 70-х годах было создано полиморфное лямбда-исчисление, вместе с языком ML.

0.2 Функции в теории множеств

Определение. Пусть X, Y — непустые множества. Бинарное отношение $R \subseteq X \times Y$ называется **функциональным**, если из $(x, y) \in R$ и $(x, z) \in R$ следует $y = z$.

Определение. **Функция** $f : X \to Y$ есть тройка (X, Y, f), где f — функциональное отношение. $f(x) = y$ обозначает $(x, y) \in f$.

[1] в 1910-х годах

Это определение отождествляет функцию с ее графиком, и это определение было предложено Бурбаки — группой математиков. Однако, можно рассматривать функцию как примитив, что приведет нас к Тьюринг-полной модели вычислений.

0.3 Лямбда-исчисление, базовые определения

Определение. Пусть $Var = \{x_0, x_1, x_2 \dots\}$ — счётное множество переменных. Множество **предтермов** есть множество, порожденное следующей контекстно-свободной грамматикой:

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

Примечание.

- $\lambda x.M$ — функция от x, возвращающая частный случай терма M. ???
- MN есть подстановка функций. ???

Определение. α**-конверсия** — отношение, переписывающее связанные переменные. α**-эквивалентность** есть рефлексивное, транзитивное и симметричное замыкание α-конверсии.

Пример. $\lambda x.x + 3$ и $\lambda y.y + 3$ α-эквивалентны.

Определение. **Лямбда-терм** есть претерм с точностью до α-эквивалентности.

Помимо грамматики, нам еще нужно определить операционную семантику.

Определение (β-редукция). Лямбда-терм M β-редуцируется к N, если существует последовательность переписываний по следующим правилам:

$$(\lambda x.M)N \to_\beta M[x := N]$$

$$\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N}$$

$$\frac{M_1 \to_\beta M_2}{N M_1 \to_\beta N M_2}$$

Определение. Терм вида $(\lambda x.M)N$ называется **редексом**.

Определение. Терм в **нормальной форме**, если он не содержит редексов.

Обозначение. $M \twoheadrightarrow_\beta N$ обозначает “M β-редуцируется к N в несколько шагов”.

???

0.4 Методы редукции

Примечание. Применение левоассоциативно.

Пример. Попробуем редуцировать терм $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$

С одной стороны:

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

С другой стороны:

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

И так бесконечно.

Таким образом, одной стратегией мы привели результат к нормальной форме, а другой — нет, мы зациклились. Несложно заметить, что порядок редукции важен. Проблема редукции относится только к применению функций.

Рассмотрим $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$. Есть два порядка редукций:

1. **Аппликативный**: сначала редуцируем N_i для всех $i \in \{1 \dots n\}$
2. **Нормальный**: сначала редуцируем $(\lambda x_1 \dots x_n.M)N_i$ для всех $i \in \{1 \dots n\}$.

Аппликативный порядок называется вызовом **по значению** и используется в традиционных ЯП: C, Java, Python .... Нормальный порядок называется вызовом **по имени**. В Haskell используется вызов **по необходимости**, который близок к вызову по имени с небольшими изменениями для возможности работы в реальном мире.

По следующей теореме нормальный порядок лучше:

Теорема 1. Пусть M — терм с нормальной формой M', тогда M можно редуцировать к M' с помощью нормального порядка редукции.

0.5 Типы

Типы были созданы как альтернатива теории множеств, однако в программировании они служат другим целям:

- Частичная спецификация поведения программы
- Проверка типов позволяет отлавливать простые ошибки, т.е. сложение числа со строкой не скомпилируется, если нет приведения типов, как в JS.

Существуют следующие *(грубые)* классификации систем типов:

- Статическая или динамическая:
  - C, C++, Java, Haskell
  - JavaScript, Ruby, PHP
- Неявная и явная:
  - Ruby, JS
  - Java, C++, C
- Выведенная:
  - Haskell, ML, OCaml

0.6 Типизированное лямбда-исчисление

Это похоже на матлогику.

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

Γ есть конечный набор утверждений вида $x : A$, обозначающих “x имеет тип A”.

Второе правило значит, что написав код *(терм M)*, возвращающий переменную типа B и использующий некоторую переменную x типа A, можно абстрагироваться по этой переменной *(рефакторнуть)* и получить функцию $A \to B$.

Третье правило значит, что если подставить в функцию $A \to B$ переменную типа A, мы получим переменную типа B.

Функции высшего порядка — функции, которые принимают другие функции как аргумент. В нетипизированном лямбда-исчислении все функции высшего порядка. В типизированном лямбда-исчислении функция высшего порядка, если она имеет вид $A \to (B \to C)$.

0.7 Полиморфизм

Пример параметрического полиморфизма на типах в Haskell:

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

Здесь `a` — произвольный тип, т.е. спрятан квантор “$\forall$”.

System F — полиморфное лямбда-исчисление, которое было создано в 1970-х. Хотя оно создавалось для исследование арифметики второго порядка, но для программистов это формализованное представление параметрического полиморфизма. Полиморфизм System F реализован в Haskell через расширение `RankNTypes`.