

Функциональное программирование

Лекция 1. Лямбда-исчисление

Денис Николаевич Москвин

ВШЭ СПб, СПбШФМиКН,
бакалавриат ПМии

04.09.2020

- 1 Функциональное vs императивное программирование
- 2 Неформальное введение в λ -исчисление
- 3 Чистое λ -исчисление: термы
- 4 Эквивалентность термов

- 1 Функциональное vs императивное программирование
- 2 Неформальное введение в λ -исчисление
- 3 Чистое λ -исчисление: термы
- 4 Эквивалентность термов

Императивное программирование: вычисление описывается в терминах **инструкций**, изменяющих **состояние** вычислителя.

В императивных программах:

- Инструкции исполняются **последовательно** `C1; C2; C3;`
- Состояние изменяется инструкциями **присваивания** значений **изменяемым переменным**: `v := value;`
- Есть механизм **условного исполнения**: инструкции `if, switch;`
- группы инструкций можно повторять с помощью **циклов**: инструкции `while, for;`
- типы данных описываются с оглядкой на их физическое представление в памяти.

Иногда говорят про стиль фон Неймана (термин Джона Бэкуса).

- Факториал числа для фон-неймановского языка

```
long factorial (int n) {  
    long res = 1;  
    for (int i = n; i > 0; i--)  
        res = res * i;  
    return res;  
}
```

- Выполнение программы: переход вычислителя из начального состояния в конечное с помощью последовательных инструкций.
- Часть конечного состояния (память, доступная по адресу, на который ссылается локальная переменная `res`) интерпретируется как результат вычислений.

Функциональная программа — *выражение*, её выполнение — вычисление (*редукция*) этого выражения.

```
factorial n = if n == 0 then 1 else n * factorial (n-1)
```

Выполнение программы: редукция выражения с помощью *подстановки* определений функций в местах их «вызова» с заменой формальных параметров на фактические.

```
factorial 3  
→ if 3 == 0 then 1 else 3 * factorial (3-1)  
→ ...  
→ 3 * factorial (3-1)  
→ 3 * if (3-1) == 0 then 1 else (3-1) * factorial ((3-1)-1)  
→ ...  
→ 3 * 2 * 1 * 1 → 6
```

Реализация может быть эффективнее, но «подстановочная» семантика должна сохраняться.

Модель вычислений в ФП не использует понятия изменяемого состояния.

- Нет состояний — **нет изменяемых переменных**
- Нет переменных — **нет присваивания**
- **Нет циклов**, поскольку нет различий между итерациями
- **Последовательность не важна**, поскольку выражения независимы

Вместо этого есть:

- **Рекурсия** — замена циклов
- **Функции высших порядков (HOF)**
- **Сопоставление с образцом**

Все функции — **чистые**, то есть возвращаемое значение зависит только от значений параметров.

- Регулярный и лаконичный синтаксис.
- Мощная типизация, необременительная благодаря эффективным алгоритмам вывода типов.
- Эффективная доказуемость свойств программ алгебраическими методами.
- Возможность генерации программ по набору свойств.
- Высокоуровневые оптимизации на базе эквивалентных преобразований.

- 1 Функциональное vs императивное программирование
- 2 Неформальное введение в λ -исчисление**
- 3 Чистое λ -исчисление: термы
- 4 Эквивалентность термов

- λ-исчисление — формальная система, лежащая в основе функционального программирования.
- Разработано Алонзо Чёрчем в 1930-х для формализации и анализа понятия вычислимости.
- Имеет бестиповую и множество типизированных версий.
- Дает возможность компактно описывать семантику вычислительных процессов.

В λ-исчислении имеются два способа строить выражения:

- *применение (аппликация, application)*;
- *лямбда-абстракция (abstraction)*.

Нотация *применения* (аппликации) F к X :

$$FX$$

- С точки зрения программиста: F (алгоритм) применяется к X (входные данные).
- Скобки вокруг X не используются. Можно интерпретировать пробел как инфиксный оператор применения. (Иногда используют явный @.)
- Различия между алгоритмами и данными нет. Например, допустимо самоприменение:

$$FF$$

- Пусть M — выражение, возможно содержащее x . Тогда *лямбда-абстракция* или просто *абстракция*

$$\lambda x. M$$

обозначает функцию

$$x \mapsto M[x],$$

то есть каждому x сопоставляется $M[x]$.

- Лямбда-абстракция — способ задать неименованную (анонимную) функцию.
- В лямбда-выражении часть слева от точки называют *абстрактором*, а справа — *телом*.
- Если x в $M[x]$ отсутствует, то $\lambda x. M$ — константная функция со значением M .

- Применение и абстракция работают совместно:

$$\underbrace{(\lambda x. 5x + 2)}_F \underbrace{8}_x = 5 \cdot 8 + 2 (= 42).$$

- То есть выражение

$$(\lambda x. 5x + 2) 8$$

это применение функции $y(x) = 5x + 2$ к аргументу 8, дающее в результате вычисления $5 \cdot 8 + 2$.

- Вычисление определяется как подстановка фактических аргументов вместо формальных.

- Обобщая, вводят понятие *редукции*

$$(\lambda x. M) N \rightarrow [x \mapsto N] M$$

где $[x \mapsto N] M$ обозначает *подстановку* N вместо x в M .

- Аппликацию вида $(\lambda x. M) N$, в которой левый аппликанд является абстракцией, называют *β -редексом*.
- Шаг вычисления по приведенному выше правилу называют *сокращением* редекса.
- В *чистом* λ -исчислении нет ничего кроме применения, абстракции и редукции.

- Если выражение содержит несколько переменных, то абстракция по одной из них вносит асимметрию:

$$\lambda n. 2m + 3n$$

Переменную n называют *связанной*, а m — *свободной*.

- Продолжая абстракцию, получим *замкнутое* выражение

$$\lambda m. (\lambda n. 2m + 3n)$$

- Теперь вычисление требует двух редукций

$$\begin{aligned} ((\lambda m. (\lambda n. 2m + 3n)) 15) 4 &\rightarrow \\ (\lambda n. 2 \cdot 15 + 3n) 4 &\rightarrow 2 \cdot 15 + 3 \cdot 4 \end{aligned}$$

- Функции нескольких аргументов, в которые аргументы передают последовательно, называют *карьерными*.

- 1 Функциональное vs императивное программирование
- 2 Неформальное введение в λ -исчисление
- 3 Чистое λ -исчисление: термы
- 4 Эквивалентность термов

Определение

Множество λ -**термов** Λ индуктивно строится из переменных $V = \{x, y, z, \dots\}$ с помощью применения и абстракции:

$$x \in V \Rightarrow x \in \Lambda$$

$$M, N \in \Lambda \Rightarrow (MN) \in \Lambda$$

$$M \in \Lambda, x \in V \Rightarrow (\lambda x. M) \in \Lambda$$

- В абстрактном синтаксисе

$$\Lambda ::= V \mid (\Lambda \Lambda) \mid (\lambda V. \Lambda)$$

- **Соглашение.** Произвольные термы пишем заглавными буквами, переменные — строчными.

Примеры λ -термов

$$\begin{aligned} &x \\ &(x z) \\ &(\lambda x. (x z)) \\ &((\lambda x. (x z)) y) \\ &(\lambda y. ((\lambda x. (x z)) y)) \\ &((\lambda y. ((\lambda x. (x z)) y)) w) \\ &(\lambda z. (\lambda w. ((\lambda y. ((\lambda x. (x z)) y)) w))) \end{aligned}$$

В этом списке каждый следующий терм содержит предыдущие в качестве *подтерма*.

Общеприняты следующие соглашения:

- Внешние скобки опускаются.
- Применение ассоциативно *влево*:

$FXYZ$ обозначает $((F X) Y) Z$

- Абстракция ассоциативна *вправо*:

$\lambda x y z. M$ обозначает $(\lambda x. (\lambda y. (\lambda z. (M))))$

- Тело абстракции простирается вправо насколько это возможно:

$\lambda x. M N K$ обозначает $\lambda x. (M N K)$

Те же примеры, что и выше, но с использованием соглашений

$$x = x$$

$$(x z) = x z$$

$$(\lambda x. (x z)) = \lambda x. x z$$

$$((\lambda x. (x z)) y) = (\lambda x. x z) y$$

$$(\lambda y. ((\lambda x. (x z)) y)) = \lambda y. (\lambda x. x z) y$$

$$((\lambda y. ((\lambda x. (x z)) y)) w) = (\lambda y. (\lambda x. x z) y) w$$

$$(\lambda z. (\lambda w. ((\lambda y. ((\lambda x. (x z)) y)) w))) = \lambda z w. (\lambda y. (\lambda x. x z) y) w$$

Абстракция $\lambda x. M[x]$ связывает до того свободную переменную x в терме M .

$$(\lambda y. (\lambda w. w z) y) x$$

Переменные y и w — связанные, а z и x — свободные.

Связывание переменной ограничивает ее область видимости телом лямбды. Поэтому вне тела лямбды то же самое имя может связываться повторно.

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

Переменная x — связанная (дважды!) и свободная, а z — свободная.

Определение

Множество $FV(T)$ *свободных (free) переменных* в λ -терме T :

$$\begin{aligned}FV(x) &= \{x\}; \\FV(M N) &= FV(M) \cup FV(N); \\FV(\lambda x. M) &= FV(M) \setminus \{x\}.\end{aligned}$$

Определение

Множество $BV(T)$ *связанных (bound) переменных* в терме T :

$$\begin{aligned}BV(x) &= \emptyset; \\BV(M N) &= BV(M) \cup BV(N); \\BV(\lambda x. M) &= BV(M) \cup \{x\}.\end{aligned}$$

Определение

M — *замкнутый λ -терм* (или *комбинатор*), если $FV(M) = \emptyset$. Множество замкнутых λ -термов обозначается Λ^0 .

Многим комбинаторам даны общепринятые имена:

Классические комбинаторы

$$I = \lambda x. x$$

$$\omega = \lambda x. x x$$

$$\Omega = \omega \omega = (\lambda x. x x)(\lambda x. x x)$$

$$K = \lambda x y. x$$

$$K_* = \lambda x y. y$$

$$C = \lambda f x y. f y x$$

$$B = \lambda f g x. f (g x)$$

$$S = \lambda f g x. f x (g x)$$

Переименование связанных переменных

Имена связанных переменных не важны; их можно (почти) безболезненно переименовывать:

$$\begin{aligned} \mathbf{I} &= \lambda x. x = \lambda y. y = \lambda z. z \\ \mathbf{B} &= \lambda f g x. f (g x) = \lambda u v z. u (v z) \end{aligned}$$

Это верно в том смысле, что термы с переименованиями при вычислениях дают один и тот же результат:

$$\begin{aligned} (\lambda x. x) N &\rightarrow N \\ (\lambda y. y) N &\rightarrow N \\ (\lambda z. z) N &\rightarrow N \end{aligned}$$

Термы, отличающиеся только именами связанных переменных, называют α -эквивалентными.

- Комбинаторы можно определить как примитивы, задав их вычислительное поведение на аргументах-переменных:

Классические комбинаторы

$$I x = x$$

$$\omega x = x x$$

$$K x y = x$$

$$S f g x = f x (g x)$$

- Вычисление опять задается подстановкой определения с заменой формальных аргументов на фактические:

$$\omega I \rightarrow II \rightarrow I.$$

- Выбрав базис (например S, K), получим исчисление, эквивалентное λ -исчислению.

- 1 Функциональное vs императивное программирование
- 2 Неформальное введение в λ -исчисление
- 3 Чистое λ -исчисление: термы
- 4 Эквивалентность термов

Подстановка: нюансы и проблемы

Подстановка выполняется только вместо **свободных** вхождений:

$$[x \mapsto \lambda z. z](x (\lambda x. x y) x) = (\lambda z. z) (\lambda x. x y) (\lambda z. z)$$

Проблема *захвата переменной* (variable capture):

$$[x \mapsto y](\lambda y. x y) = \lambda y. y y$$

Соглашение Барендрегта

Имена связанных переменных всегда будем выбирать так, чтобы они отличались от имён свободных переменных.

Тогда коллизий, связанных с захватом, можно избежать:

$$[x \mapsto y](\lambda y'. x y') = \lambda y'. y y'$$

Однако удобнее встроить переименование в определение подстановки.

Определение подстановки $[x \mapsto N] M$

Подстановка терма N вместо свободных вхождений переменной x в терм M задается индукцией по M :

$$[x \mapsto N] x = N,$$

$$[x \mapsto N] y = y,$$

$$[x \mapsto N] (P Q) = ([x \mapsto N] P) ([x \mapsto N] Q),$$

$$[x \mapsto N] (\lambda x. P) = \lambda x. P,$$

$$[x \mapsto N] (\lambda y. P) = \lambda y. [x \mapsto N] P, \quad \text{если } y \notin FV(N),$$

$$[x \mapsto N] (\lambda y. P) = \lambda z. [x \mapsto N] ([y \mapsto z] P), \quad \text{если } y \in FV(N).$$

Подразумевается, что $x \not\equiv y$, а z — свежая, то есть $z \notin FV(N) \cup FV(P)$.

Основная схема аксиом: для любых $M, N \in \Lambda$

$$(\lambda x. M)N =_{\beta} [x \mapsto N] M \quad (\text{правило } \beta)$$

- Логические аксиомы и правила

$$M =_{\beta} M;$$

$$M =_{\beta} N \quad \Rightarrow \quad N =_{\beta} M;$$

$$M =_{\beta} N, N =_{\beta} L \quad \Rightarrow \quad M =_{\beta} L.$$

- Правила совместимости

$$M =_{\beta} M' \quad \Rightarrow \quad MZ =_{\beta} M'Z;$$

$$M =_{\beta} M' \quad \Rightarrow \quad ZM =_{\beta} ZM';$$

$$M =_{\beta} M' \quad \Rightarrow \quad \lambda x. M =_{\beta} \lambda x. M' \quad (\text{правило } \xi).$$

- Если $M =_{\beta} N$ доказуемо в λ -исчислении, пишут $\lambda \vdash M =_{\beta} N$.

Иногда вводят схему аксиом α -преобразования:

$$\lambda x. M =_{\alpha} \lambda y. [x \mapsto y] M, \text{ если } y \notin FV(M) \quad (\text{правило } \alpha).$$

Для рассуждений достаточно соглашения Барендрегта, но для компьютерной реализации α -преобразование полезно:

Пусть $\omega = \lambda x. x x$ и $\mathbf{1} = \lambda y z. y z$. Тогда

$$\begin{aligned} \omega \mathbf{1} &= (\lambda x. x x) (\lambda y z. y z) \\ &=_{\beta} (\lambda y z. y z) (\lambda y z. y z) \\ &=_{\beta} \lambda z. (\lambda y z. y z) z \\ &=_{\alpha} \lambda z. (\lambda y z'. y z') z \\ &=_{\beta} \lambda z z'. z z' \\ &=_{\alpha} \lambda y z'. y z' \\ &=_{\alpha} \lambda y z. y z = \mathbf{1} \end{aligned}$$

- *Индексы Де Брауна (De Bruijn)* представляют альтернативный способ представления термов.
- Связанные переменные не именовются, а индексируются, индекс показывает, сколько лямбд «назад» переменная была связана:

$$\lambda x. (\lambda y. x y) \leftrightarrow \lambda (\lambda 1 0)$$

$$\lambda x. x (\lambda y. x y y) \leftrightarrow \lambda 0 (\lambda 1 0 0)$$

- Свободные переменные при этом получают индексы, превышающие число лямбд слева:

$$\lambda x. z x y \leftrightarrow \lambda 2 0 1$$

- При таком представлении все α -эквивалентные термы кодируются одинаково, и коллизий захвата переменной не возникает.

- Схема аксиом η -преобразования:

$$\lambda x. M x =_{\eta} M \quad (\text{правило } \eta)$$

в предположении, что $x \notin FV(M)$.

- Для определения отношения эквивалентности надо, конечно, добавить общелогические аксиомы и правила. Разумно также обеспечить совместимость.
- Смысл η -эквивалентности в том, что аппликативное поведение термов слева и справа от знака равенства одинаково; для произвольного N верно

$$(\lambda x. M x) N =_{\beta} M N$$

- η -преобразование обеспечивает принцип *ЭКСТЕНСИОНАЛЬНОСТИ*: две функции считаются экстенционально эквивалентными, если они дают одинаковый результат при любом одинаковом вводе:

$$\forall N : FN = GN.$$

- Выбирая $y \notin FV(F) \cup FV(G)$, получаем (ξ , затем η)

$$\begin{aligned}Fy &= Gy \\ \lambda y. Fy &= \lambda y. Gy \\ F &= G\end{aligned}$$

- В Хаскелле так называемый «бесточечный» стиль записи основан на η -преобразовании.