

Функциональное программирование

Лекция 1. Лямбда-исчисление

Денис Николаевич Москвин

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

10.02.2012

- 1 Функциональное vs императивное программирование
- 2 Введение в λ -исчисление
- 3 Подстановка и преобразования
- 4 Расширения чистого λ -исчисления

- 1 Функциональное vs императивное программирование
- 2 Введение в λ -исчисление
- 3 Подстановка и преобразования
- 4 Расширения чистого λ -исчисления

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

В императивных программах:

- Состояние изменяется инструкциями **присваивания**: $v = E$
- Есть механизм **условного исполнения**: инструкции `if`, `switch`
- Есть механизм **циклов**: инструкции `while`, `for`
- Инструкции исполняются **последовательно** `C1; C2; C3`

Иногда говорят про стиль фон Неймана (Джон Бэкус)

Скалярное произведение двух n -мерных векторов для фон-нейманновского языка

```
res = 0;
for (i = 0; i < n; ++i)
    res = res + a[i] * b[i];
```

Как здесь формируется состояние и какие изменения с ним происходят?

Скалярное произведение двух n -мерных векторов для фон-нейманновского языка

```
res = 0;
for (i = 0; i < n; ++i)
    res = res + a[i] * b[i];
```

Как здесь формируется состояние и какие изменения с ним происходят?

Абстрактное представление (пока забываем про I/O):

$$\sigma_0 \Longrightarrow \sigma_1 \Longrightarrow \sigma_2 \Longrightarrow \dots \Longrightarrow \sigma_n$$

Выполнение программы: переход *вычислителя* из начального состояния в конечное с помощью *инструкций*.

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

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

Вместо этого есть:

- **Рекурсия** — вместо циклов
- **Функции высших порядков (HOF)**

Скалярное произведение для функционального языка

```
innerProduct = (sum .) . zipWith (*)
```

Абстрактное представление:

$$f_n(f_{n-1}(\dots f_2(f_1(\sigma_0))\dots))$$

Выполнение программы: *вычисление выражения.*

```
innerProduct as bs -> ((sum .) . zipWith (*)) as bs  
                   -> (sum .) (zipWith (*) as) bs  
                   -> sum (zipWith (*) as bs)  
                   -> ...
```


Функциональное программирование: преимущества и недостатки

Преимущества:

- Более ясная семантика
- Большая свобода при исполнении (например, поддержка параллельности)
- Большая выразительность
- Лучшая параметризация и модульность
- Удобство при работе с «бесконечными» структурами данных

Недостатки:

- Ввод-вывод: нужен специальный инструментарий
- Интерактивные программы: нужен специальный инструментарий
- Быстродействие (исполнение в чуждой архитектуре)

Чистота и побочные эффекты

Рассмотрим императивный код:

```
bool flag;
int f (int n) {
    int retVal;
    if flag then retVal = 2 * n;
        else retVal = n;
    flag = ! flag;
    return retVal;
}
void test() {
    flag = true;
    printf("f(1) + f(2) = %d\n", f(1) + f(2));
    printf("f(2) + f(1) = %d\n", f(2) + f(1));
}
//    f(1) + f(2) = 4
//    f(2) + f(1) = 5
```

Чистота и побочные эффекты (2)

- Функция f не является **чистой (pure)** математической функцией. Говорят о нарушении *ссылочной прозрачности (reference transparency)*.
- Её результат зависит от внешних факторов, а исполнение приводит к возникновению **побочного эффекта (side effect)**. Причина: глобально доступное состояние + разрушающее присваивание.
- В *чистом* функциональном программировании такие функции не используются (в Хаскелле живут в гетто IO, в других — допустимы, но не рекомендуются).

- 1 Функциональное vs императивное программирование
- 2 Введение в λ -исчисление**
- 3 Подстановка и преобразования
- 4 Расширения чистого λ -исчисления

- 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.
- Позволяет простым образом описывать семантику вычислительных процессов.

- В λ -исчислении имеются две операции:
 - *применение (аппликация, application)*;
 - *абстракция (abstraction)*.
- Нотация **применения** F к X :

$F X$

- С точки зрения программиста: F (алгоритм) применяется к X (входные данные).
- Однако явного различия между алгоритмами и данными нет, в частности допустимо самоприменение:

$F F$

- Пусть $M \equiv M[x]$ — выражение, (возможно) содержащее x .
Тогда **абстракция**

$$\lambda x. M$$

обозначает функцию

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

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

- Если x в $M[x]$ отсутствует, то $\lambda x. M$ — константная функция со значением M .

- Применение и абстракция работают совместно:

$$\underbrace{(\lambda x. 2 \times x + 1)}_F \underbrace{42}_x = 2 \times 42 + 1 \quad (= 85).$$

- То есть $(\lambda x. 2 \times x + 1) 42$ — применение функции $x \mapsto 2 \times x + 1$ к аргументу 42, дающее в результате $2 \times 42 + 1$.
- В общем случае имеем β -преобразование

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

где $M[x := N]$ обозначает подстановку N вместо x в M .

Определение

Множество λ -**термов** Λ 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

x

$(x z)$

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

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

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

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

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

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

$$x \equiv x$$

$$(x z) \equiv x z$$

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

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

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

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

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

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

Пример 1

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

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

Пример 2

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

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

Свободные и связанные переменные (2)

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 .

Классические комбинаторы:

$$I \equiv \lambda x. x$$

$$\omega \equiv \lambda x. x x$$

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

$$K \equiv \lambda x y. x$$

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

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

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

- Шонфинкель (1924): функции нескольких переменных могут быть описаны последовательным применением.
- Пусть $\varphi(x, y)$ — терм, содержащий свободные x и y . Введём, путём последовательных абстракций

$$\Phi_x = \lambda y. \varphi(x, y)$$

$$\Phi = \lambda x. \Phi_x = \lambda x. (\lambda y. \varphi(x, y)) = \lambda x y. \varphi(x, y)$$

- Тогда применение этого Φ к произвольным X и Y может быть выполнена последовательно

$$\Phi X Y = (\Phi X) Y = \Phi_x Y = (\lambda y. \varphi(X, y)) Y = \varphi(X, Y).$$

- Переход от функции нескольких аргументов к функции, принимающей аргументы «по одному» называется *каррированием*.

Тождественное равенство термов

- Имена *связанных* переменных не важны. Переименуем x в y :

$$\lambda x. M[x], \quad \lambda y. M[y]$$

Они ведут себя (при подстановках) одинаково:

$$(\lambda x. M[x]) N = M[x := N], \quad (\lambda y. M[y]) N = M[y := N]$$

- Поэтому $P \equiv Q$ обозначает, что P и Q – это один и тот же терм с точностью до переименования связанных переменных. Например,

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

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

- Иногда такое переименование называют α -преобразованием и пишут $P \equiv_{\alpha} Q$.

- 1 Функциональное vs императивное программирование
- 2 Введение в λ -исчисление
- 3 Подстановка и преобразования
- 4 Расширения чистого λ -исчисления

Определение

$M[x := N]$ обозначает *подстановку* N вместо **свободных** вхождений x в M . Правила подстановки:

$$x[x := N] \equiv N;$$

$$y[x := N] \equiv y;$$

$$(P Q)[x := N] \equiv (P[x := N]) (Q[x := N]);$$

$$(\lambda y. P)[x := N] \equiv \lambda y. (P[x := N]), \quad y \notin FV(N);$$

$$(\lambda x. P)[x := N] \equiv (\lambda x. P).$$

Подразумевается, что $x \neq y$.

Пример

$$((\lambda x. (\lambda x. x z) x) x)[x := N] \equiv (\lambda x. (\lambda x. x z) x) N$$

Проблема захвата переменной

Неприятность: $(\lambda y. x y)[x := y]$ ($y \in FV(N)$ в четвёртом правиле).

Соглашение Барендрегта

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

Пример

Вместо

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

будем писать

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

Тогда можно использовать подстановку без оговорки о свободных и связанных переменных.

Лемма подстановки

Пусть $M, N, L \in \Lambda$. Предположим $x \neq y$ и $x \notin FV(L)$. Тогда

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Доказательство

Нудная индукция по всем 5 случаям. ■

Основная схема аксиом для λ -исчисления

Для любых $M, N \in \Lambda$

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

- Логические аксиомы и правила:

$$M = M;$$

$$M = N \Rightarrow N = M;$$

$$M = N, N = L \Rightarrow M = L;$$

$$M = M' \Rightarrow MZ = M'Z;$$

$$M = M' \Rightarrow ZM = ZM';$$

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

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

Иногда вводят:

- схему аксиом α -преобразования:

$$\lambda x . M = \lambda y . M[x := y] \quad (\alpha)$$

в предположении, что $y \notin FV(M)$;

- схему аксиом η -преобразования:

$$\lambda x . M x = M \quad (\eta)$$

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

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

Пример

Пусть $\omega \equiv \lambda x. x x$ и $1 \equiv \lambda y z. y z$. Тогда

$$\begin{aligned}\omega 1 &\equiv (\lambda x. x x)(\lambda y z. y z) \\ &= (\lambda y z. y z)(\lambda y z. y z) \\ &= \lambda z. (\lambda y z. y z) z \\ &\equiv \lambda z. (\lambda y z'. y z') z \\ &= \lambda z z'. z z' \\ &\equiv \lambda y z. y z \\ &\equiv 1\end{aligned}$$

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

$$\lambda x. (\lambda y. x y) \leftrightarrow \lambda (\lambda 2 1)$$

$$\lambda x. x (\lambda y. x y y) \leftrightarrow \lambda 1 (\lambda 2 1 1)$$

- При таком представлении проблемы захвата переменной нет.

Преобразования (конверсии): η

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

$$\forall x : F x = G x.$$

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

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

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

- 1 Функциональное vs императивное программирование
- 2 Введение в λ -исчисление
- 3 Подстановка и преобразования
- 4 Расширения чистого λ -исчисления

- Можно расширить множество λ -термов константами:

$$\Lambda(\mathbb{C}) ::= \mathbb{C} \mid V \mid \Lambda(\mathbb{C}) \Lambda(\mathbb{C}) \mid \lambda V. \Lambda(\mathbb{C})$$

- Например, $\mathbb{C} = \{\mathbf{true}, \mathbf{false}\}$.
- Но нам ещё нужно уметь их использовать. Поэтому лучше

$$\mathbb{C} = \{\mathbf{true}, \mathbf{false}, \mathbf{not}, \mathbf{and}, \mathbf{or}\}$$

- И всё равно, помимо констант нужны дополнительные правила, описывающие работу с ними. Какие?

- Всем известные:

not true $=_{\delta}$ **false**

not false $=_{\delta}$ **true**

and true true $=_{\delta}$ **true**

and true false $=_{\delta}$ **false**

and false true $=_{\delta}$ **false**

and false false $=_{\delta}$ **false**

...

- «Внешние» функции над константами порождают новые правила преобразований.

δ -преобразование: обобщение

- Если на множестве термов X (обычно $X \subseteq \mathbb{C}$) задана «внешняя» функция $f : X^k \rightarrow \Lambda(\mathbb{C})$, то для неё добавляем δ -правило:
 - выбираем незанятую константу δ_f ;
 - для $M_1, \dots, M_k \in X$ добавляем правило сокращения

$$\delta_f M_1 \dots M_k =_{\delta} f(M_1, \dots, M_k)$$

- Для одной f — не одно правило, а целая схема правил.
- Например, для $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ схемы правил:

$$\begin{aligned} \text{plus } m \ n &=_{\delta} m + n \\ \text{mult } m \ n &=_{\delta} m \times n \\ \text{equal } n \ n &=_{\delta} \text{true} \\ \text{equal } m \ n &=_{\delta} \text{false, если } m \neq n \end{aligned}$$