

# Лямбда-исчисление

---

Косарев Дмитрий

13 марта 2023 г.

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену

# Введение: $\lambda$ -исчисление



Alonzo Church (1903–1995)

Алонзо Чёрч 1935 открыл  $\lambda$ -исчисление

Аналогичный подход от А. Тьюринга с его машинами Тьюринга

Это разные подходы для формализации понятия “алгоритм”

В принципе, могло быть изобретено уже в 1910-х г.г.

---

Изображение из [Википедии](#)

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену

# Состояние математики в 1910-х

Матан, алгебра, геометрия...

Информатики (computer science) явным образом пока нет, как часть математики

Математическая логика

- 1 Пытается формализовать интуитивно понятные утверждения
- 2 Языки (т.е. синтаксис), чтобы на них можно было правильно сформулировать теоремы
- 3 Различные “семантики” как интерпретации синтаксиса , потому что формулы могут быть верны и не верны в зависимости от семантики
- 4 “Исчисления” – правильные способы доказательств
- 5 Теоремы, которые невозможно ни доказать, ни опровергнуть.

Начинают задумываться, что такое “алгоритм”, “вычисление” и “вычислимая функция”

# Зачем формализовывать то, что и так понятно?

## "Наивная" теория множеств

Множества можно делить на два типа

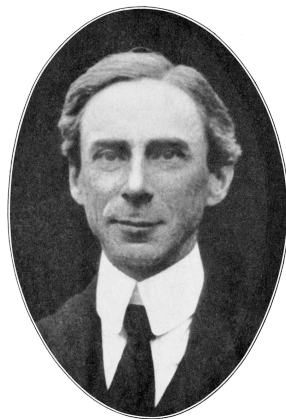
- 1 набор не является элементом самого себя
- 2 Расселовские: набор является элементом самого себя.

Рассмотрим  $P = \{y : y \notin P\}$  и задумаемся про  $P \in P$ ?

- Если формула верна, то нарушается определение
- Если ложна, то не принадлежит, но по определению должна

---

Изображение из [Википедии](#)



Bertrand Russell  
(1872–1970)

# Некоторые известные языки и исчисления из математической логики

- Нулевого порядка (высказываний)
- Первого порядка (предикатов)
- Высших порядков
- Исчисление конструкций (calculus of constructions)

## Важное замечание

То, что нельзя записать в языке, нельзя использовать в исчислениях/доказательствах

# Знакомая вам булева (бинарная) логика

- 1 Логические константы True и False
- 2 Логические переменные  $x, y, z, \dots$
- 3 Бинарные связки  $\vee, \wedge, \Rightarrow$  и т.д.

Правила вывода в исчислении, например:

$$\frac{P \Rightarrow Q \quad P}{Q} \text{ modus ponens}$$

## Теорема (Язык и исчисление “хорошие”)

*Верную формулу можно доказать за конечное число шагов. Ложную можно опровергнуть.*

*Т.е. существует алгоритм, который всегда завершается и говорит да/нет.*

Язык и исчисление “плохие”, потому что не всё можно записать (где кванторы?)



# Разрешимые и неразрешимые задачи

## Определение (Алгоритмически неразрешимая задача)

Задача, которая имеет ответ “да” или “нет”, но для которого невозможно реализовать алгоритм, который *всегда завершается, и выдает правильный ответ*.

## Определение (Полуразрешимая задача)

Неразрешимая задача, для которой можно предъявить алгоритм, который либо дает правильный ответ “да”, либо не завершается. Полуразрешимые<sup>+</sup> умеют говорить “да”, полуразрешимые<sup>-</sup> — “нет”.

Как доказывать неразрешимость

- Разбирать случаи и искать противоречие в каждом
- Сводить каноничную неразрешимую задачу к нашей

# Язык и исчисление 1-го порядка (предикатов)

Термы:

- Предметные константы:  $1.0$ ,  $42$ ,  $\pi$
- Функциональные символы  $n$ -арности  $1 \leq n$  от термов. Например,  $+$ ,  $\times$ ,  $f$ ,  $mod$  и т.д.
- Предметные переменные  $x, y, z, \dots$

Формулы:

- Логические константы  $\text{True}$  и  $\text{False}$
- Бинарные связки  $\vee, \wedge, \Rightarrow$  (и т.д.)
- Предикатные символы (от термов)  $n$ -арности  $1 \leq n$
- Кванторы  $\forall, \exists$  от имени предметной переменной и формулы

Важно

$+$ ,  $\times$ ,  $f$ ,  $mod$  это названия функциональных символов, никто не гарантирует, что  $+$  это сложение чисел

Пример:

$\forall xz \exists y (x < y) \wedge (y < z)$   
верно, если  $x, y, z \in \mathbb{R}$ ,  
неверно, если  $x, y, z \in \mathbb{N}$

# Преимущества и недостатки языка 1го порядка

- Для некоторых формул из синтаксиса можно понять, что они верны (общезначимы). Для них есть алгоритм, который их докажет за конечное число шагов (см. “метод британского музея”)
- Огромное количество формул верны только в некоторой семантике, для них нельзя предъявить, алгоритм, который завершается и выдает вердикт.  
В общем виде проверка формулы на истинность/ложность – неразрешимая задача
- Язык недостаточно богат. Кванторы пробегают только предметные переменные, нельзя выразить “для любой формулы  $P$ , верно...”, например, принцип индукции

$$\forall P. \quad P(0) \Rightarrow (\forall n. P(n) \Rightarrow P(n+1)) \Rightarrow (\forall n. P(n))$$

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов**
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену

# Но можно попробовать вывернуться

Введем специальный синтаксис

$$\lambda P. \text{ phormula}(P)$$

Опишем принцип индукции, и применим его для  $P(n) \equiv 0 + \dots + n = \frac{n \cdot (n + 1)}{2}$

$$\lambda P. \quad P(0) \Rightarrow (\forall n. P(n) \Rightarrow P(n + 1)) \Rightarrow (\forall n. P(n))$$

# Но можно попробовать вывернуться

Введем специальный синтаксис

$\lambda P. \text{ phormula}(P)$

Опишем принцип индукции, и применим его для  $P(n) \equiv 0 + \dots + n = \frac{n \cdot (n + 1)}{2}$

$\lambda P. P(0) \Rightarrow (\forall n. P(n) \Rightarrow P(n + 1)) \Rightarrow (\forall n. P(n))$

применение/подстановка  $\Downarrow \Uparrow$  абстракция

$$\begin{aligned} (0 \equiv 0) \Rightarrow (\forall n. (0 + \dots + n = \frac{n \cdot (n + 1)}{2}) \Rightarrow (0 + \dots + (n + 1) = \frac{(n + 1) \cdot (n + 2)}{2})) \\ \Rightarrow (\forall n. 0 + \dots + n = \frac{n \cdot (n + 1)}{2}) \end{aligned} \quad (1)$$

# Правила работы с новым языком $\lambda$

## $\alpha$ -ЭКВИВАЛЕНТНОСТЬ

При выборе новых имен, они не должны случайно перекрыть старые.

Предложения языка, отличающиеся только переименованием переменных, считаются  $(\alpha)$ эквивалентными

Например: если ни  $P$ , ни  $Q$  не встречаются в  $phormula$ , то

$$\lambda P.phormula(P) \stackrel{\alpha}{\equiv} \lambda Q.phormula(Q)$$

## $\beta$ -ЭКВИВАЛЕНТНОСТЬ

Если у нас встречается  $(\lambda P.phormula(P))X$ , то мы можем продолжить с этим работать совершив подстановку  $X$  вместо  $P$  в  $phormula$  (записывается как  $phormula[P \mapsto X]$ ), т.е. заменив все свободные вхождения  $P$  на  $X$  внутри  $phormula$ .

## Синтаксис:

- Переменные:  $x, y, z, \dots$
- Абстракция  $(\lambda \nu. A)$ , где  $A$  —  $\lambda$ -выражение, а  $\nu$  — произвольное имя переменной
- Применение  $(AB)$ , где  $A$  и  $B$  —  $\lambda$ -выражения

## Определение

Редекс — это  $\lambda$ -выражение вида  $(\lambda \nu. A)B$

## В терминах программирования:

- Переменные
- Объявления 1-аргументных функций
- Вызов функции от одного аргумента

Процесс вычисления — это процесс устранения редексов (возможно, не всех) путём подстановок  $\lambda$ -выражений вместо переменных.



# Каррирование

## Определение

Каррирование — это представление  $n$ -арных функций через 1-арные функции

В  $\lambda$ -исчислении функция  $n$  аргументов представляются как функция одного аргумента, которые возвращает функцию от  $n - 1$  аргумента.

В мире названо в честь Хаскеля Карри. Впервые появилось в 1924 в работе М. И. Шейнфинкеля.

---

Изображение взято с [Википедии](#)



Моисей Исаевич  
Шейнфинкель  
(1888 – 1942)

## Символ $\lambda$ работает как квантор

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



- свободные вхождения
- связанных вхождения и т.д.

# Определения алгоритма

## Теорема (Тезис Чёрча)

*Используя  $\lambda$ -исчисление можно реализовать произвольный алгоритм (с точностью до представления данных).*

## Теорема (Тезис Тьюринга)

*Используя машину Тьюринга можно реализовать произвольный алгоритм (с точностью до представления данных).*

Т.е. теперь под алгоритмом понимается только то, что можно записать в формализме (-ах).

# Как происходят вычисления (редукция) $\lambda$ -исчисления?

Определение (Процесс вычислений регламентирует стратегия)

Ищем редексы  $(\lambda x.P)Q$

- Если редексов нет, то вычисление закончилось
- Если редексы есть, стратегия регламентирует какой на данном шаге редекс стоит  $\beta$ -редуцировать
- Или же, стратегия может сказать, что все редексы нужно оставить как есть, и выдать ответ.

## Две стратегии: Call-by-value и аппликативная

Call-by-value

$$\frac{}{(\lambda x.e) \xrightarrow{cbv} (\lambda x.e)} \text{ Abs}$$

Applicative order

$$\frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')} \text{ Abs}$$

---


$$\frac{e_1 \rightarrow (\lambda x.e) \quad e_2 \rightarrow e'_2 \quad [e'_2/x]e \rightarrow e'}{(e_1 e_2) \rightarrow e'} \text{ App-abs}$$


$$\frac{}{x \rightarrow x} \text{ Var} \qquad \frac{e_1 \rightarrow e' \neq (\lambda x.e) \quad e_2 \rightarrow e'_2}{(e_1 e_2) \rightarrow (e'_1 e'_2)} \text{ App-non-abs}$$

# Две стратегии: Call-by-value и аппликативная

## Call-by-value


$$\frac{}{(\lambda x. e) \xrightarrow{cbv} (\lambda x. e)} \text{ Abs}$$


 Подходит для написания произвольных алгоритмов

 Не считает под абстракцией, поэтому ответ иногда длиннее, чем хотелось бы

## Applicative order

$$\frac{e \xrightarrow{ao} e'}{(\lambda x. e) \xrightarrow{ao} (\lambda x. e')} \text{ Abs}$$

 Нет возможности отложить вычисления на потом, т.е. в if-then-else вычисляется и then, и else. Нельзя дождаться ответа от рекурсивных функций

 Считает под абстракцией, поэтому ответ короче

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления**
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену

# Что нужно для представления алгоритмов?

- Принимать входные данные
- Делать ветвления в зависимости от входных данных
- Совершать некоторое количество однотипных действий в зависимости от входных данных (т.е. должны быть циклы или их аналог – рекурсия)
  - Чтобы понимать, сколько действий уже сделали нужны натуральные числа



$$T \equiv (\lambda x. (\lambda y. x)) \equiv fst$$

$$F \equiv (\lambda x. (\lambda y. y)) \equiv snd$$

$$ite \equiv \lambda c. \lambda th. \lambda el. (c \ th \ el)$$

$$(ite \ T) \equiv \lambda th. \lambda el. (T \ th \ el) \rightsquigarrow th$$

$$(ite \ F) \equiv \lambda th. \lambda el. (F \ th \ el) \rightsquigarrow el$$

# Историческое напоминание: числа Пеано

Первым ввел аксиоматику арифметики в 1889 году. Натуральные числа определяются через “базу” и “следующий”

1. 0 — натуральное число
6. Для любого натурального  $n$ ,  $S(n)$  тоже натуральное. т.е. натуральные числа замкнуты относительно операции  $S(\cdot)$
9. Аксиома индукции.

---

Peano's axioms in their historical context

Изображение взято с [Википедии](#)



Giuseppe Peano (1858 – 1932)

## Представление чисел (нумералы Чёрча)

$$0 \sim (\lambda s. (\lambda x. x))$$

$$1 \sim (\lambda s. (\lambda x. s \ x))$$

$$2 \sim (\lambda s. (\lambda x. s \ (s \ x)))$$

и т.д.

Сложение (один из вариантов): взять два нумерала  $m$  и  $n$ , взять  $f$  и  $x$ , а затем к  $x$  применить  $n$  раз  $f$ , а затем к результату применить  $m$  раз  $f$ .

$$add \equiv \lambda m. \lambda n. \lambda f. \lambda x. (m \ f \ (n \ f \ x))$$

$$\begin{aligned}
& (\lambda m. \lambda n. \lambda f. \lambda x. (m \ f \ (n \ f \ x))) 22 \xrightarrow{cbv} \\
& (\lambda m. \lambda n. \lambda f. \lambda x. (m \ f \ (n \ f \ x))) 22 \xrightarrow{cbv} \\
& (\lambda n. \lambda f. \lambda x. (2 \ f \ (n \ f \ x))) 2 \xrightarrow{cbv} \\
& \lambda f. \lambda x. (2 \ f \ (2 \ f \ x)) \longrightarrow \\
& \lambda f. \lambda x. ((\lambda f. (\lambda x. f(fx))) \ f \ (2 \ f \ x)) \xrightarrow{ao} \\
& \lambda f. \lambda x. ((\lambda x. f(fx)) \ (2 \ f \ x)) \longrightarrow \\
& \lambda f. \lambda x. ((\lambda x. f(fx)) \ (((\lambda f. (\lambda x. f(fx))) \ f \ x))) \xrightarrow{ao} \\
& \lambda f. \lambda x. ((\lambda x. f(fx)) \ ((\lambda x. f(fx)) \ x)) \xrightarrow{ao} \\
& \lambda f. \lambda x. ((\lambda x. f(fx)) \ (f(fx))) \xrightarrow{ao} \\
& \lambda f. \lambda x. f(f(f(fx)))
\end{aligned}$$

$$\begin{aligned}
 & (\lambda m. \lambda n. \lambda f. \lambda x. (m \ f \ (n \ f \ x))) 2 2 \xrightarrow{cbv} \\
 & (\lambda m. \lambda n. \lambda f. \lambda x. (m \ f \ (n \ f \ x))) 2 2 \xrightarrow{cbv} \\
 & \quad (\lambda n. \lambda f. \lambda x. (2 \ f \ (n \ f \ x))) 2 \xrightarrow{cbv} \\
 & \quad \lambda f. \lambda x. (2 \ f \ (2 \ f \ x))
 \end{aligned}$$

Это ответ для  $\xrightarrow{cbv}$ .

Давайте посмотрим, что будет, если мы ответ применим к  $g$  и  $y$

$$\begin{aligned}
 & (\lambda f. \lambda x. (2 \ f \ (2 \ f \ x))) g y \rightarrow \\
 & \quad (\lambda x. (2 \ g \ (2 \ g \ x))) y \rightarrow \\
 & \quad (2 \ g \ (2 \ g \ y)) \rightarrow \\
 & \quad ((\lambda f. (\lambda x. f \ (f x))) \ g \ (2 \ g \ y)) \rightarrow \\
 & \quad ((\lambda x. g \ (g x)) \ (2 \ g \ y)) \rightarrow \\
 & \quad ((\lambda x. g \ (g x)) \ (((\lambda f. (\lambda x. f \ (f x))) \ g \ x))) \rightarrow \\
 & \quad ((\lambda x. g \ (g x)) \ ((\lambda x. g \ (g x)) \ y)) \rightarrow \\
 & \quad ((\lambda x. g \ (g x)) \ (g \ (g y))) \rightarrow \\
 & \quad g \ (g \ (g \ (g y)))
 \end{aligned}$$

# Рекурсия через комбинатор неподвижной точки

англ. `FIXed point combinator`

Не понятно как вызвать самого себя, так как имен нет.

Идея:

- записываем функцию  $f$ , чтобы она принимала первый аргумент, который будет вызываться вместо рекурсивного вызова
- Везде, где надо вызвать эту “рекурсивную” функцию, будем писать  $Yf$  или  $Zf$

FIX для $\xrightarrow{ao}$ и $\xrightarrow{cbv}$ :	$Z \equiv (\lambda f. (\lambda x. f(\lambda v. x \ x \ v)) (\lambda x. f(\lambda v. x \ x \ v))))$
FIX для “ленивых” стратегий:	$Y \equiv (\lambda f. (\lambda x. f(x \ x)) (\lambda x. f(x \ x)))$

Откуда такое название?

$$YR = (\lambda x. R(x \ x)) (\lambda x. R(x \ x)) \rightarrow R((\lambda x. R(x \ x)) (\lambda x. R(x \ x))) = R(YR)$$

# Рекурсия в call-by-value (упрощенно)

Настолько упрощенно, что даже не совсем правильно

$$FIX\ R = R\ (FIX\ R)$$

Факториал:  $fac \equiv (\lambda self.\lambda n.(\text{if } n < 2 \text{ then } 1 \text{ else } n \cdot self(n - 1)))$

$$\begin{aligned} &FIX(\lambda self.\lambda n.(\text{if } n < 2 \text{ then } 1 \text{ else } n \cdot self(n - 1)))2 \rightarrow \\ &\quad (\lambda n.(\text{if } n < 2 \text{ then } 1 \text{ else } n \cdot FIX\ fac(n - 1)))2 \rightarrow \\ &\quad\quad 2 \cdot FIX\ fac\ (2 - 1) \rightarrow \\ &2 \cdot (FIX(\lambda self.\lambda n.(\text{if } n < 2 \text{ then } 1 \text{ else } n \cdot self(n - 1)))\ 1) \rightarrow \\ &\quad 2 \cdot (\text{if } 1 < 2 \text{ then } 1 \text{ else } n \cdot (FIX\ fac\ (1 - 1))) \rightarrow \\ &\quad\quad\quad 2 \cdot 1 \rightarrow 2 \end{aligned}$$

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену



## Дэмка на Си (1/5): представление выражений

```
enum Tag { VAR, APP, ABS };  
struct ulc {  
    Tag tag;  
    union body {  
        struct Var { char* name; } Var;  
        struct App { ulc* f; ulc* arg; } App;  
        struct Abs { char* name; ulc* body; } Abs;  
    } body;  
};
```

## Демка на Си (2/5): объявление стратегии

```
struct Strategy {  
    ulc* (*onvar)(Strategy* self, char* name);  
    ulc* (*onApp)(Strategy* self, struct ulc *f, struct ulc *arg);  
    ulc* (*onAbs)(Strategy* self, char *name, struct ulc *arg);  
};
```

```
struct ulc* applyStrategy(Strategy *self, struct ulc *root) {  
    switch (root→tag) {  
        case VAR: return self→onvar(self, root→body.Var.name);  
        case APP: return self→onApp(self, root→body.App.f, root→body.App.arg);  
        case ABS: return self→onAbs(self, root→body.Abs.name, root→body.Abs.body);  
    }  
    assert(false);    return nullptr; // unreachable  
}
```

## Демка на Си (3/5): тривиальная константная стратегия

```
struct ulc *evalVar(Strategy *this, char *name) {  
    return var(name);  
}  
  
struct ulc *dontReduceUnderAbstraction(Strategy *this, char *name, ulc *body) {  
    return abs(name, body);  
}  
  
struct ulc *dontReduceApplication(Strategy *this, ulc* f, ulc* arg) {  
    return app(f, arg);  
}  
  
struct Strategy NoStrategy = {  
    .onvar = evalVar,  
    .onApp = dontReduceApplication,  
    .onAbs = dontReduceUnderAbstraction,  
};
```

## Дэмка на Си (4/5): Call-by-value

```
struct ulc *evalApplyByValue(strategy *this, ulc *f, ulc *arg) {
    auto f2 = applyStrategy(this, f);
    switch (f2→tag) {
        case VAR:      case APP:      return app(f2, arg);
        case ABS: {
            auto arg2 = applyStrategy(this, arg);
            auto rez = subst(arg2, f2→body.Abs.name, f2→body.Abs.body);
            return applyStrategy(this, rez);
        }
    }
    assert(false); return nullptr; // unreachable
}

struct Strategy CallByValue = {
    .onvar = evalVar,
    .onApp = evalApplyByValue,
    .onAbs = dontReduceUnderAbstraction };
}
```

## Дэмка на Си (5/5): понятие наследования

```
struct ulc *evalApplyByValue(Strategy *this, ulc *f, ulc *arg)
struct ulc *evalVar(Strategy *this, char *name);
struct ulc *dontReduceUnderAbstraction(Strategy *this, char *name, ulc *body);
struct ulc *dontReduceApplication(Strategy *this, ulc *f, ulc *arg);
```

```
struct Strategy NoStrategy = {
    .onvar = evalVar,
    .onAbs = dontReduceUnderAbstraction,
    .onApp = dontReduceApplication,
};

struct Strategy CallByValue = {
    .onvar = evalVar,
    .onAbs = dontReduceUnderAbstraction
    .onApp = evalApplyByValue,
};
```

# Оглавление

- 1 Введение и историческая справка
- 2 Лямбды как апгрейд языка предикатов
- 3 Написание алгоритмов с помощью  $\lambda$ -исчисления
- 4 Как писать интерпретатор на Си?
- 5 Вопросы к экзамену

- Разрешимые и неразрешимые задачи
- $\lambda$ -исчисление.  $\alpha$  и  $\beta$  правила
- Нумералы Чёрча. Сложение
- Рекурсия и факториал на “пальцах”
- Наследование, но оно будет ещё в других билетах



*Реализация интерпретатора на Си*

<https://github.com/Kakadu/kakadu.github.io/tree/master/papers/lambda2023/cpp>



*Реализация интерпретатора на OCaml* <https://gitlab.com/Kakadu/>

[fp2020course-materials/-/blob/master/code/Lambda/lib/lambda.ml](https://gitlab.com/Kakadu/fp2020course-materials/-/blob/master/code/Lambda/lib/lambda.ml)



## 6 Проблема останова

- ## 7 Дополнительные слайды
- Call-By-Name
  - Call-By-Value
  - Аппликативный порядок
  - "Нормальный" порядок

# Проблема останова (1/2)

## Вопрос

Можем ли мы написать алгоритм, который будет брать на вход произвольную  $\lambda$ -абстракцию и аргумент, и говорить посчитается ли для их применения нормальная форма?

# Проблема останова (1/2)

## Вопрос

Можем ли мы написать алгоритм, который будет брать на вход произвольную  $\lambda$ -абстракцию и аргумент, и говорить посчитается ли для их применения нормальная форма?

Положим наши программы либо зависают, либо выдают значение `true`.

Положим существует гипотетическая (*Halting*  $P w$ ), которая всегда завершается, и возвращает `true`, если  $(P w)$  редуцируется в `true`, иначе (*Halting*  $P w$ ) возвращает `false`.

Покажем от противного, что *Halting* не может существовать.

## Проблема останова (2/2)

Вопрос: во что редуцируется  $E$ , в true или в false?

$$E = \text{Halting}((\lambda m. \text{not}(\text{Halting } m \ m)), (\lambda m. \text{not}(\text{Halting } m \ m)))$$

Если  $E$  редуцируется в true, то применим функцию  $(\lambda m. \text{not}(\text{Halting}(m, m)))$  к аргументу  $(\lambda m. \text{not}(\text{Halting}(m, m)))$  и получим

$$\text{not}(\text{Halting}((\lambda m. \text{not}(\text{Halting } m \ m)), (\lambda m. \text{not}(\text{Halting } m \ m)))) = \neg E$$

что является отрицание истинного факта выше.

Если  $E$  редуцируется в false, то это означает,  $\text{Halting}$  иногда зависит, что противоречит определению функции  $\text{Halting}$ . □

## 6 Проблема останова

- ## 7 Дополнительные слайды
- Call-By-Name
  - Call-By-Value
  - Аппликативный порядок
  - "Нормальный" порядок

# Нормальные формы

У нас как минимум четыре возможности

- Редуцируем ли под абстракциями? (да/нет)
- Редуцируем ли аргументы перед подстановкой? (да/нет)

Редуцируем аргументы?	Редуцируем под абстракциями?	
	Да(strong)	Нет(weak)
Да(strict)	Normal form $E ::= (\lambda x.E) \mid xE_1 \dots E_n$	Weak normal form $E ::= (\lambda x.e) \mid xE_1 \dots E_n$
Нет(lazy)	Head normal form $E ::= (\lambda x.E) \mid xe_1 \dots e_n$	Weak head normal form $E ::= (\lambda x.e) \mid xe_1 \dots e_n$

В таблице  $E_j$  – это выражение в соответствующей нормальной форме, а  $e_i$  – произвольный  $\lambda$ -терм.

---

<sup>0</sup>То, что у некоторых  $E$  нет индексов – не опечатка

## Порядков редукции бывает много...[?]

- 1 Call-by-Name
- 2 Normal Order
- 3 Call-by-Value (OCaml)
- 4 Applicative Order
- 5 Hybrid Applicative Order
- 6 Head Spine Reduction
- 7 Hybrid Normal Order

И ещё есть оптимизации связанные с мемоизацией (кешированием) нормальных форм подвыражений.

Так Call-by-Name + кеширование = Call-by-Need (Haskell)

## 6 Проблема останова

- ## 7 Дополнительные слайды
- Call-By-Name
  - Call-By-Value
  - Аппликативный порядок
  - "Нормальный" порядок



## Call-By-Name $\rightsquigarrow$ Weak Head Normal Form

Редуцирует **самый левый внешний** редекс, который **не под абстракцией**. Например,  $(\lambda x.(\lambda y.M)N)$  уже в WHNF, потому что единственный редекс  $(\lambda y.M)N$  под абстракцией.

$$\begin{array}{c} \frac{}{x \xrightarrow{cbn} x} \text{Var} \qquad \frac{}{(\lambda x.e) \xrightarrow{cbn} (\lambda x.e)} \text{Abs} \\[1em] \frac{e_1 \xrightarrow{cbn} (\lambda x.e) \quad [e_2/x]e \xrightarrow{cbn} e'}{(e_1 e_2) \xrightarrow{cbn} e'} \text{App-abs} \\[1em] \frac{e_1 \xrightarrow{cbn} e'_1 \neq (\lambda x.e)}{(e_1 e_2) \xrightarrow{cbn} (e'_1 e_2)} \text{App-non-abs} \end{array}$$

CBN может посчитать 1 аргумент несколько раз по сравнению с CBV.

## 6 Проблема останова

## 7 Дополнительные слайды

- Call-By-Name
- Call-By-Value
- Аппликативный порядок
- "Нормальный" порядок

# Call-by-Value $\rightsquigarrow$ Weak Normal Form

Редуцирует **самый левый внутренний** редекс, который **не под абстракцией**.  
Например, в  $(\lambda x. (\lambda y. U) V) ((\lambda z. M) N)$  самый левый внутренний – это  $(\lambda y. U) V$ , но редуцироваться первым будет  $((\lambda z. M) N)$ .

$$\begin{array}{c} \frac{}{x \xrightarrow{cbv} x} \text{ Var} \qquad \frac{}{(\lambda x. e) \xrightarrow{cbv} (\lambda x. e)} \text{ Abs} \\[2ex] \frac{e_1 \xrightarrow{cbv} (\lambda x. e) \quad e_2 \xrightarrow{cbv} e'_2 \quad [e'_2/x]e \xrightarrow{cbv} e'}{(e_1 e_2) \xrightarrow{cbv} e'} \text{ App-abs} \\[2ex] \frac{e_1 \xrightarrow{cbv} e'_1 \neq (\lambda x. e) \quad e_2 \xrightarrow{cbv} e'_2}{(e_1 e_2) \xrightarrow{cbv} (e'_1 e'_2)} \text{ App-non-abs} \end{array}$$

Стандарт для большинства языков программирования.

# Нормальной формы может не быть!

## Определение

Нормализация – процесс поиска соответствующей нормальной формы с помощью применения  $\beta$ -редукции согласно соответствующей стратегии

Пример: комбинатор  $\Omega = (\lambda x.xx)(\lambda x.xx)$

$$(\lambda x.xx)(\lambda x.xx) \xrightarrow{cbv} [(\lambda x.xx)/x](xx) \xrightarrow{cbv} (\lambda x.xx)(\lambda x.xx) \xrightarrow{cbv} \dots$$

Call-by-Name чаще завершается

$$(\lambda x. (\lambda y. y)) \Omega \xrightarrow{cbv} \text{расходится}$$

$$(\lambda x. (\lambda y. y)) \Omega \xrightarrow{cbn} (\lambda y. y)$$

Но Call-by-Name иногда вычисляет аргументы больше одного раза

$$(\lambda x. (Ax)(Bx))((\lambda y. y) C) \xrightarrow{cbn} (A((\lambda y. y) C)) (B((\lambda y. y) C))$$

$$(\lambda x. (Ax)(Bx))((\lambda y. y) C) \xrightarrow{cbv} (\lambda x. (Ax)(Bx)) C \xrightarrow{cbv} (AC)(BC)$$

## 6 Проблема останова

- ## 7 Дополнительные слайды
- Call-By-Name
  - Call-By-Value
  - **Аппликативный порядок**
  - "Нормальный" порядок

## Applicative Order $\rightsquigarrow$ Normal Form

Редуцирует **самый левый внутренний** редекс, и **под абстракцией тоже**. Например, в  $(\lambda x.(\lambda y.U) V)((\lambda z.M) N)$  самый левый внутренний – это  $(\lambda y.U) V$ .

$$\begin{array}{c} \frac{}{x \xrightarrow{ao} x} \text{Var} \qquad \frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')} \text{Abs} \\[10pt] \frac{e_1 \xrightarrow{ao} (\lambda x.e) \quad e_2 \xrightarrow{ao} e'_2 \quad [e'_2/x]e \xrightarrow{ao} e'}{(e_1 e_2) \xrightarrow{ao} e'} \text{App-abs} \\[10pt] \frac{e_1 \xrightarrow{ao} e' \neq (\lambda x.e) \quad e_2 \xrightarrow{ao} e'_2}{(e_1 e_2) \xrightarrow{ao} (e'_1 e'_2)} \text{App-non-abs} \end{array}$$

N.B. Аппликативный порядок совершает больше редукций и выдает более простой ответ по сравнению с CBV, но не гарантирует, что редукция завершится.

## 6 Проблема останова

- ## 7 Дополнительные слайды
- Call-By-Name
  - Call-By-Value
  - Аппликативный порядок
  - "Нормальный" порядок



## Normal Order $\rightsquigarrow$ Normal Form

Сначала редуцирует **самый левый внешний** редекс. Встретив применение  $(e_1 e_2)$  вначале пытается редуцировать  $e_1$  как CBN. Если не получилась абстракция – принимается за аргументы.

$$\begin{array}{c} \frac{}{x \xrightarrow{nor} x} \text{Var} \qquad \frac{e \xrightarrow{nor} e'}{(\lambda x. e) \xrightarrow{nor} (\lambda x. e')} \text{Abs} \\[2ex] \frac{e_1 \xrightarrow{cbn} (\lambda x. e) \quad [e_2/x]e \xrightarrow{nor} e'}{(e_1 e_2) \xrightarrow{nor} e'} \text{App-abs} \\[2ex] \frac{e_1 \xrightarrow{cbn} e'_1 \neq (\lambda x. e) \quad e'_1 \xrightarrow{nor} e''_1 \quad e_2 \xrightarrow{nor} e'_2}{(e_1 e_2) \xrightarrow{nor} (e''_1 e'_2)} \text{App-non-abs} \end{array}$$

N.B. Нормальный порядок сочетает две стратегии, позволяет получить более простые результаты, чем CBN. Чаще завершается, чем АО.