Лямбда-исчисление

Косарев Дмитрий

20 марта 2023 г.

Оглавление

Введение: λ-исчисление

Alonzo Church (1903–1995)

Алонзо Чёрч 1935 открыл λ-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

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?

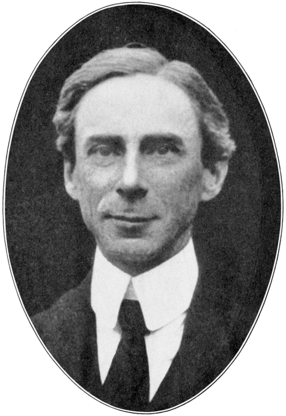
"Наивная" теория множеств

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



Bertrand Russell
(1872–1970)

Изображение из Википедии

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 $x, y, z, \ldots$
3. Бинарные связки $\vee, \wedge, \Rightarrow$ и т.д.

Правила вывода в исчислении, например:

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

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 которое 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$^{+}$ умеют говорить "да", полуразрешимые$^{-}$ — "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, π
- Функциональные символы арности $1 \leqslant n$ от термов. Например, $+, \times, f, mod$ и т.д.
- Предметные переменные $x, y, z, \ldots$

Формулы:

- Логические константы True и False
- Бинарные связки $\vee, \wedge, \Rightarrow$ (и т.д.)
- Предикатные символы (от термов) арности $1 \leqslant n$
- 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

Но можно попробовать вывернуться

Введем специальный синтаксис

$$\lambda P. \quad \mathit{phormula}(P)$$

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

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

Но можно попробовать вывернуться

Введем специальный синтаксис

$$\lambda P. \quad phormula(P)$$

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

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

$$\text{применение/подстановка} \Big\Downarrow \Big\Uparrow \text{абстракция}$$

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

Правила работы с новым языком λ

α-эквивалентность

При выборе новых имен, они не должны случайно перекрыть старые.
Предложения языка, отличающиеся только переименованием переменных, считаются (α)эквивалентными

Например: если ни P, ни Q не встречаются в *phormula*, то
$\lambda P.phormula(P) \stackrel{\alpha}{\equiv} \lambda Q.phormula(Q)$

β-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*.

λ-исчисление

Синтаксис:

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

Определение

Редекс — это λ-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 λ-выражений вместо переменных.

λ-исчисление

Синтаксис:

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

Определение

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

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

Каррирование

Определение

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

В λ-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.

Моисей Исаевич Шейнфинкель (1888 – 1942)

Изображение взято с Википедии

Символ λ работает как квантор

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

- свободные вхождения
- связанных вхождения и т.д.

[0] TODO: сказать про скобочки

Подстановка

Редекс $(\lambda x.(\lambda x.x)x)y$ вида $(\lambda \nu.A)B$, где

- $A \equiv (\lambda x.x)x$
- $B \equiv y$
- $\nu \equiv x$

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

Подстановка “x вместо A в B” в лит-ре обозначается по-разному:

- $[x \mapsto A]B$
- $[A/x]B$

Определения алгоритма

Теорема (Тезис Чёрча)

Используя λ-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) λ-исчислении?

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 β-редуцировать
- Или же, стратегия может сказать, что все редексы нужно оставить как есть, и выдать ответ.

Дэмка на C++ (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
}
```

Две стратегии: Call-by-value и аппликативная

Call-by-value

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

Applicative order

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

$$\frac{f_1 \to (\lambda x.e) \qquad a_1 \to a_2 \qquad [x \mapsto a_2]e \to r}{(f_1 a_1) \to r}\ \text{App-abs}$$

$$\frac{}{x \to x}\ \text{Var} \qquad \frac{f_1 \to f_2 \neq (\lambda x.e) \qquad a_1 \to a_2}{(f_1 a_1) \to (f_2 a_2)}\ \text{App-non-abs}$$

Две стратегии: Call-by-value и аппликативная

Call-by-value	Applicative order
$$\frac{}{(\lambda x.e) \xrightarrow{cbv} (\lambda x.e)}\ \text{Abs}$$	$$\frac{e \xrightarrow{ao} e'}{(\lambda x.e) \xrightarrow{ao} (\lambda x.e')}\ \text{Abs}$$
👍 Подходит для написания произвольных алгоритмов	👎 Нет возможности отложить вычисления на потом,т.е. в if-then-else вычисляется и then, и else. Нельзя дождаться ответа от рекурсивных функций
👎 Не считает под абстракцией, поэтому ответ иногда длиннее, чем хотелось бы	👍 Считает под абстракцией, поэтому ответ короче

$$\frac{f_1 \to (\lambda x.e) \qquad a_1 \to a_2 \qquad [x \mapsto a_2]e \to r}{(f_1 a_1) \to r}\ \text{App-abs}$$

$$\frac{f_1 \to f_2 \neq (\lambda x.e) \qquad a_1 \to a_2}{(f_1 a_1) \to (f_2 a_2)}\ \text{App-non-abs}$$

```
struct ulc *evalApplyByValue(Strategy *self, ulc *f, ulc *a1) {
  auto f2 = applyStrategy(self, f);
  switch (f2->tag) {
    case VAR:    case APP:      return app(f2, a1);
    case ABS: {
      auto a2 = applyStrategy(self, a1);
      auto r = subst(a2, f2->body.Abs.name, f2->body.Abs.body);
      return applyStrategy(self, r);
    }
  }
  assert(false); return nullptr; // unreachable
```

Оглавление

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

Ветвления

$$T \equiv (\lambda x.(\lambda y.x)) \equiv fst \qquad F \equiv (\lambda x.(\lambda y.y)) \equiv snd$$

$$\begin{aligned} ite &\equiv \lambda c.\lambda th.\lambda el.(c\ th\ el) \\ (ite\ T) &\equiv \lambda th.\lambda el.(T\ th\ el) \xrightarrow{*} th \\ (ite\ F) &\equiv \lambda th.\lambda el.(F\ th\ el) \xrightarrow{*} el \end{aligned}$$

[0] Здесь $\xrightarrow{*}$ означает редукцию за несколько шагов

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.

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

2+2

$$(\lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x)))22 \xrightarrow{cbv}$$

$$(\lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x)))22 \xrightarrow{cbv}$$

$$(\lambda n.\lambda f.\lambda x.(2\ f\ (n\ f\ x)))2 \xrightarrow{cbv}$$

$$\lambda f.\lambda x.(2\ f\ (2\ f\ x)) \longrightarrow$$

$$\lambda f.\lambda x.((\lambda f.(\lambda x.f(fx)))\ f\ (2\ f\ x)) \xrightarrow{ao}$$

$$\lambda f.\lambda x.((\lambda x.f(fx))\ \ (2\ f\ x)) \longrightarrow$$

$$\lambda f.\lambda x.((\lambda x.f(fx))\ \ (((\lambda f.(\lambda x.f(fx)))\ f\ x))) \xrightarrow{ao}$$

$$\lambda f.\lambda x.((\lambda x.f(fx))((\lambda x.f(fx))x)) \xrightarrow{ao}$$

$$\lambda f.\lambda x.((\lambda x.f(fx))(f(fx))) \xrightarrow{ao}$$

$$\lambda f.\lambda x.f(f(f(fx)))$$

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

$$\text{FIX для } \xrightarrow{ao} \text{ и } \xrightarrow{cbv}: \qquad Z \equiv (\lambda f.(\lambda x.f(\lambda v.\ x\ x\ v))(\lambda x.f(\lambda v.\ x\ x\ v)))$$

$$\text{FIX для "ленивых" стратегий:} \qquad Y \equiv (\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))$$

Откуда такое название?

$$YR = (\lambda x.R(x\ x))(\lambda x.R(x\ x)) \to R\big((\lambda x.R(x\ x))(\lambda x.R(x\ x))\big) = R(YR)$$

Рекурсия в call-by-value (упрощенно)

Настолько упрощенно, что даже не совсем правильно

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

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

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

Оглавление

Дэмка на C++ (1/5): представление выражений

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

Дэмка на C++ (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
}
```

Дэмка на C++ (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 C++ (4/5): Call-by-value

```
struct ulc *evalApplyByValue(Strategy *self, ulc *f, ulc *a1) {
  auto f2 = applyStrategy(self, f);
  switch (f2->tag) {
    case VAR:     case APP:        return app(f2, a1);
    case ABS: {
      auto a2 = applyStrategy(self, a1);
      auto r = subst(a2, f2->body.Abs.name, f2->body.Abs.body);
      return applyStrategy(self, r);
    }
  }
  assert(false); return nullptr; // unreachable
}
struct Strategy CallByValue = {
  .onvar = evalVar,
  .onApp = evalApplyByValue,
  .onAbs = dontReduceUnderAbstraction };
```

Дэмка на C++ (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

Вопросы к экзамену

- Разрешимые и неразрешимые задачи
- λ-исчисление. α и β правила
- Нумералы Чёрча. Сложение
- Рекурсия и факториал на "пальцах"
- Наследование, но оно будет ещё в других билетах

Ссылки

- *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

Оглавление

Проблема останова (1/2)

Вопрос

Можем ли мы написать алгоритм, который будет брать на вход произвольную λ-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 λ-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`?

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

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

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

что является отрицание истинного факта выше.

Если E редуцируется в `false`, то это означает, *Halting* иногда зависает, что противоречит определению функции *Halting*. $\square$

Оглавление

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 $E ::= (\lambda x.E) \mid xE_1 \ldots E_n$	Weak normal form $E ::= (\lambda x.e) \mid xE_1 \ldots E_n$
Нет(lazy)	Head normal form $E ::= (\lambda x.E) \mid xe_1 \ldots e_n$	Weak head normal form $E ::= (\lambda x.e) \mid xe_1 \ldots e_n$

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

[0]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)

Оглавление

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.

$$\frac{}{x \xrightarrow{cbn} x}\ \text{Var} \qquad \frac{}{(\lambda x.e) \xrightarrow{cbn} (\lambda x.e)}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{cbn} (\lambda x.e) \qquad [e_2/x]e \xrightarrow{cbn} e'}{(e_1\, e_2) \xrightarrow{cbn} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbn} e_1' \neq (\lambda x.e)}{(e_1\, e_2) \xrightarrow{cbn} (e_1'\, e_2)}\ \text{App-non-abs}$$

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

Оглавление

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 $((\lambda z.M)N)$.

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

$$\frac{e_1 \xrightarrow{cbv} (\lambda x.e) \qquad e_2 \xrightarrow{cbv} e_2' \qquad [e_2'/x]e \xrightarrow{cbv} e'}{(e_1\,e_2) \xrightarrow{cbv} e'}\,\text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbv} e_1' \neq (\lambda x.e) \qquad e_2 \xrightarrow{cbv} e_2'}{(e_1\,e_2) \xrightarrow{cbv} (e_1'\,e_2')}\,\text{App-non-abs}$$

Стандарт для большинства языков программирования.

Нормальной формы может не быть!

Определение

Нормализация – процесс поиска соответствующей нормальной формы с помощью применения β-редукции согласно соответствующей стратегии

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

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

CBN vs. CBV

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

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

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

Оглавление

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 $(\lambda y.U)\,V$.

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

$$\frac{e_1 \xrightarrow{ao} (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2' \qquad [e_2'/x]e \xrightarrow{ao} e'}{(e_1\, e_2) \xrightarrow{ao} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{ao} e' \neq (\lambda x.e) \qquad e_2 \xrightarrow{ao} e_2'}{(e_1\, e_2) \xrightarrow{ao} (e_1'\, e_2')}\ \text{App-non-abs}$$

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

Оглавление

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.

$$\frac{}{x \xrightarrow{nor} x}\ \text{Var} \qquad\qquad \frac{e \xrightarrow{nor} e'}{(\lambda x.e) \xrightarrow{nor} (\lambda x.e')}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{cbn} (\lambda x.e) \qquad [e_2/x]e \xrightarrow{nor} e'}{(e_1\, e_2) \xrightarrow{nor} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbn} e_1' \neq (\lambda x.e) \qquad e_1' \xrightarrow{nor} e_1'' \qquad e_2 \xrightarrow{nor} e_2'}{(e_1\, e_2) \xrightarrow{nor} (e_1''\, e_2')}\ \text{App-non-abs}$$

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