

Российская Академия наук  
Санкт-Петербургский академический университет —  
Научно-образовательный центр нанотехнологий  
Санкт-Петербургская кафедра философии РАН

## История и методология механизмов элиминации в языках с зависимыми типами

Реферат аспиранта СПБАУ-НОЦНТ  
Шабалина Александра Леонидовича

Научный руководитель  
к.ф.-м.н., доцент  
Москвин Денис Николаевич

Руководитель аспирантской группы  
д.ф.н., проф.  
Мангасарян Владимир Николаевич

Санкт-Петербург  
2015

# Содержание

<b>1</b>	<b>Лямбда-исчисление с индуктивными типами</b>	<b>3</b>
1.1	Лямбда-исчисление . . . . .	3
1.1.1	Нетипизированное лямбда-исчисление . . . . .	3
1.1.2	Просто типизированное лямбда-исчисление . . . . .	4
1.1.3	Другие способы типизирования . . . . .	4
1.2	Сопоставление с образцом . . . . .	5
1.2.1	Расширение лямбда-исчисления дополнительными типами данных . . . . .	5
1.2.2	Механизм сопоставления с образцом . . . . .	8
1.3	Индуктивные типы данных . . . . .	8
<b>2</b>	<b>Зависимые типы в лямбда-исчислении</b>	<b>10</b>
2.1	Зависимые типы . . . . .	10
2.2	Индуктивные семейства типов . . . . .	10
2.3	Сопоставление с образцом для индуктивных семейств типов	11
<b>3</b>	<b>Механизмы элиминации в языках с зависимыми типами</b>	<b>12</b>
3.1	Выразимость сопоставления с образцом через элиминаторы	12
3.2	Расширение сопоставления с образцом views . . . . .	12
3.3	Прочие расширения сопоставления с образцом . . . . .	12
3.4	Проблемы механизма сопоставлением с образцом . . . . .	12
<b>4</b>	<b>Список литературы</b>	<b>13</b>

# 1. Лямбда-исчисление с индуктивными типами

## 1.1. Лямбда-исчисление

### 1.1.1. Нетипизированное лямбда-исчисление

Лямбда-исчисление Чёрча, наряду с машинами Тьюринга и общерекурсивными функциями Гёделя, задает множество эффективно вычисляемых функций [1]. То есть функций, для вычисления которых существует алгоритм. Это, в свою очередь, означает, что любая программа, исполняемая на современных вычислительных устройствах, должна быть выразима на языке лямбда-исчисления. Формально этот язык задается следующим образом [1]:

- Есть некоторое множество *переменных*  $V$ .
- Есть множество *термов*  $T$ , задаваемых индуктивно:

$$T = \begin{cases} x, & x \in V & \text{переменная} \\ M N, & M, N \in T & \text{применение} \\ \lambda x.M, & x \in V, M \in T & \text{лямбда-абстракция} \end{cases}$$

- Множеством свободных переменных терма  $t$  называется подмножество  $V$ , в которое входят все переменные в терме  $t$ , кроме тех, что связаны лямбда-абстракцией (определение аналогично для других связывающих конструкций в математике, к примеру, кванторов существования и всеобщности):

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

- Вычисление выполняется с помощью правила бета-редукции:

$$(\lambda x.M)N \Rightarrow_{\beta} M[x := N]$$

Где нотация  $M[x := N]$  означает замену в  $M$  всех свободных вхождений переменной  $x$  на терм  $N$ .

### 1.1.2. Просто типизированное лямбда-исчисление

Недостаток нетипизированного лямбда-исчисления лежит во «вседозволенности» операции применения  $MN$ : ничто не гарантирует, что  $M$  будет функцией. Чтобы решить эту проблему, Чёрч разработал просто типизированное лямбда-исчисление [2]:

- Множество переменных на уровне термов  $V$  и на уровне типов  $TyV$ .
- Множество типов  $Ty$ :

$$Ty = \begin{cases} x, & x \in TyV \\ \sigma \rightarrow \tau, & \sigma, \tau \in Ty \end{cases}$$

- Множество термов  $T$ :

$$T = \begin{cases} x, & x \in V \\ M N, & M, N \in T \\ \lambda x : \sigma. M, & x \in V, \sigma \in Ty, M \in T \end{cases}$$

- Правила присваивания типов:

$$\frac{}{\Gamma \vdash x : \sigma} \quad (x : \sigma) \in \Gamma$$

$$\frac{\Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

$$\frac{(\Gamma \cup \{x : \sigma\}) \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. M) : \sigma \rightarrow \tau}$$

- Вычисление выполняется также с помощью бэта-редукции, имеющему аналогичный вид нетипизированному случаю. Но определяется только для термов, которым можно присвоить тип.

### 1.1.3. Другие способы типизирования

Недостаток просто типизированного лямбда-исчисления обратный: множество термов, которые хотелось бы написать будут отвергнуты системой типов. Поэтому было придумано множество способов расширить систему типов, чтобы с одной стороны позволить писать больше программ, которые хочется, а с другой — отвергать некорректные. Одной из таких систем типов является System F Жирапа [9]:

- Типы

$$Ty = \begin{cases} x, & x \in TyV \\ \sigma \rightarrow \tau, & \sigma \in Ty, \tau \in Ty \\ \forall x.\sigma, & x \in TyV, \sigma \in Ty \end{cases}$$

- Термы

$$T = \begin{cases} x, & x \in V \\ M N, & M, N \in T \\ \lambda x : \sigma.M, & x \in V, \sigma \in Ty, M \in T \\ \Lambda x.M, & X \in TyV, M \in T & \text{абстракция по типу} \\ M[\sigma], & M \in T, \sigma \in Ty & \text{применение типа} \end{cases}$$

- Типизация. Все правила из просто типизированного плюс:

$$\frac{\Gamma \cup \{x\} \vdash M : \sigma}{\Gamma \vdash \Lambda x.M : \forall X.\sigma}$$

$$\frac{\Gamma \vdash M : \forall x.\tau}{\Gamma \vdash M[\sigma] : \tau[x := \sigma]}$$

Введение абстракции по типам позволяет строить утверждения о программах [11]. Например, существует ровно один терм с типом  $\forall a.\forall b.a \rightarrow b \rightarrow a$  —  $\Lambda a.\Lambda b.\lambda x.\lambda y.x$ .

## 1.2. Сопоставление с образцом

### 1.2.1. Расширение лямбда-исчисления дополнительными типами данных

Чтобы превратить System F в практический язык программирования, требуется ввести дополнительные типы данных, например *Bool* и *List* [9]:

- Расширяем множество типов типами *Bool*, *List T*, где  $T \in Ty$ .
- Расширяем множество термов термами *true*, *false*, *ifThenElse*, *nil*, *cons*, *isnil*, *head*, *tail*.
- Дополняем правила типизации:

$$\Gamma \vdash true : Bool$$

$$\Gamma \vdash \text{false} : \text{Bool}$$

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash t : T \quad \Gamma \vdash f : T}{\Gamma \vdash \text{ifThenElse } e \ t \ f : T}$$

$$\Gamma \vdash \text{nil}[T] : \text{List } T$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash xs : \text{List } T}{\Gamma \vdash \text{cons}[T] \ x \ xs : \text{List } T}$$

$$\frac{\Gamma \vdash xs : \text{List } T}{\Gamma \vdash \text{isnil}[T] \ xs : \text{Bool}}$$

$$\frac{\Gamma \vdash xs : \text{List } T}{\Gamma \vdash \text{head}[T] \ xs : T}$$

$$\frac{\Gamma \vdash xs : \text{List } T}{\Gamma \vdash \text{tail}[T] \ xs : \text{List } T}$$

- И расширяем правила вычислений. Назовем их дельта-редукциями:

$$\frac{e \Rightarrow e'}{\text{ifThenElse } e \ t \ f \Rightarrow \text{ifThenElse } e' \ t \ f}$$

$$\frac{t \Rightarrow t'}{\text{ifThenElse } \text{true} \ t \ f \Rightarrow t'}$$

$$\frac{f \Rightarrow f'}{\text{ifThenElse } \text{false} \ t \ f \Rightarrow f'}$$

$$\frac{x \Rightarrow x' \quad xs \Rightarrow xs'}{\text{cons}[T] \ x \ xs \Rightarrow \text{cons}[T] \ x' \ xs'}$$

$$\frac{xs \Rightarrow xs'}{\text{isnil}[T] \ xs \Rightarrow \text{isnil}[T] \ xs'}$$

$$\text{isnil}[S] \ (\text{nil}[T]) \Rightarrow \text{true}$$

$$\text{isnil}[S] \ (\text{cons}[T] \ x \ xs) \Rightarrow \text{false}$$

$$\frac{xs \Rightarrow xs'}{\text{head}[T] \ xs \Rightarrow \text{head}[T] \ xs'}$$

$$\text{head}[S] \ (\text{cons}[T] \ x \ xs) \Rightarrow x$$

$$\frac{xs \Rightarrow xs'}{\text{tail}[T] \ xs \Rightarrow \text{tail}[T] \ xs'}$$

$$\text{tail}[S] \ (\text{cons}[T] \ x \ xs) \Rightarrow xs$$

Можно заметить, что введение типа в систему состоит из шагов:

1. Введение типа на уровень типов: *Bool*, *List T*.
2. Введение конструкторов — функций (или, в частном случае, констант), которые дают на выходе элемент введенного типа: *true*, *false*, *nil*, *cons*.
3. Введение предикатов, которые проверяют каким конструктором был построен аргумент: *isnil*.
4. Введение деструкторов — функций, действующих обратным конструкторам, то есть получая на вход элемент вводимого типа, они возвращают аргумент соответствующего конструктора: *head*, *tail*

Последние 2 этапа можно объединить в один элиминатор, как в случае с *Bool*: *ifThenElse*. В общем случае эта функция принимает на вход элемент вводимого типа, затем столько же функций, сколько конструкторов и каждая из этих функций имеет число аргументов такое же, как и соответствующий ей конструктор. Все эти функции возвращают какой-то тип *a* и весь элиминатор тоже возвращает этот тип *a*. Вычислительно первый аргумент проверяется на то, каким конструктором и с какими аргументами он был построен и на результат выдается применение соответствующей функции к аргументам конструктора.

### 1.2.2. Механизм сопоставления с образцом

В 1968-м году Бурсталл [3] предложил синтаксическую конструкцию, которая позволяет избавиться от явного написания предикатов с декструкторами, заменив их соответствующими конструкторами:

$$\begin{aligned}\mathbf{let} \text{ } lst &= \mathbf{cons} \ x \ xs && \Leftrightarrow && \mathbf{let} \text{ } lst &= \mathbf{cons} \ x \ xs \\ \mathbf{let} \text{ } (\mathbf{cons} \ x \ xs) &= lst && \Leftrightarrow && \mathbf{let} \text{ } x = \mathbf{head} \ lst; \mathbf{let} \text{ } xs = \mathbf{tail} \ lst \\ lst &\mathbf{is} \ \mathbf{nil} && \Leftrightarrow && \mathbf{isnil} \ lst\end{aligned}$$

И развивая идею чуть дальше, заменять конструкции вида:

$$\begin{aligned}\mathbf{if} &&& e \mathbf{is} \ c_1 \mathbf{then} \mathbf{let} \text{ } (c_1 \ x_1 \ \dots \ x_{k_1}) = e; \phi_1 \ x_1 \ \dots \ x_{k_1} \\ \mathbf{else} \mathbf{if} &&& e \mathbf{is} \ c_2 \mathbf{then} \mathbf{let} \text{ } (c_2 \ x_1 \ \dots \ x_{k_2}) = e; \phi_2 \ x_1 \ \dots \ x_{k_2} \\ &&& \vdots \\ \mathbf{else} \mathbf{if} &&& e \mathbf{is} \ c_n \mathbf{then} \mathbf{let} \text{ } (c_n \ x_1 \ \dots \ x_{k_n}) = e; \phi_n \ x_1 \ \dots \ x_{k_n}\end{aligned}$$

конструкцией

**cases**  $e$  :

$$\begin{aligned}c_1 \ x_1 \ \dots \ x_{k_1} &: \phi_1 \ x_1 \ \dots \ x_{k_1} \\ c_2 \ x_1 \ \dots \ x_{k_2} &: \phi_2 \ x_1 \ \dots \ x_{k_2} \\ &\vdots \\ c_n \ x_1 \ \dots \ x_{k_n} &: \phi_n \ x_1 \ \dots \ x_{k_n}\end{aligned}$$

Механизм, названный позднее сопоставлением с образцом получил реализацию в функциональных языках ML и Haskell. Там он оказался практически удобным и, обрета популярность, начал появляться в новых языках программирования, например, Rust и Swift.

### 1.3. Индуктивные типы данных

Встроенных в систему типов бывает недостаточно и хочется иметь возможность вводить собственные типы данных. Это можно делать, явно предоставляя функции преобразования между новым введенным типом и комбинацией встроенных [4], что позволяет писать программы в терминах нового типа и при этом дает возможность изменить его внутреннее представление, не переписывая сами программы.



Но от ручного написания функций преобразования тоже хочется избавиться, что было впервые сделано в 1980-м году в языке Норе [5]. В нем был введен способ вводить пользовательские типы данных, которые имеют форму деревьев. Например, *Bool*, *List T* принимают вид:

```
data Bool == true ++ false
typevar T
data List(T) == nil ++ cons(T#List(T))
```

## 2. Зависимые типы в лямбда-исчислении

### 2.1. Зависимые типы

Развитие идеи из 1.1.3 о построении утверждений по программам носит название «утверждения как типы» [10]. Это соответствие систем типов системам математической логики. В частности, System F соответствует интуиционистской (конструктивной) пропозициональной логике второго порядка, но без квантора существования. Суть соответствия:

Тип  $\leftrightarrow$  Утверждение

Терм типа  $\leftrightarrow$  Доказательство утверждения

Вычисление терма  $\leftrightarrow$  Упрощение доказательства

Если теперь в качестве логики взять интуиционистскую логику предикатов высших порядков и построить по ней систему типов, то получится интуиционистская теория типов [8], еще известная как система зависимых типов или просто MLTT (Martin-Löf Type Theory, по имени автора). «Зависимостью» здесь называется зависимость типов от термов: в System F мир типов и мир термов были разделены, а в MLTT они объединены. Примером является тип зависимых функций,  $\Pi$ -тип:  $\Pi(x : A)B(x)$ . Здесь объявляется функция с доменом  $A$  и множеством значений, зависящим от переданного аргумента. Также в системе есть тип равенств:  $I(A, a, b)$ , где  $a$  и  $b$  имеют тип  $A$ . Этот тип населен тогда и только тогда, когда  $a$  и  $b$  равны, то есть имеют одну и ту же нормальную форму. Имея  $\Pi$  и  $I$  можно представить, как писать верифицированные программы используя лишь систему типов:

$$idempotent\_sort : \Pi(x : List\ A)I(List\ A, sort\ x, sort\ (sort\ x))$$

Функция с таким типом отвечает утверждению, что для любого списка функция  $sort$  идемпотентна. И написав реализацию  $idempotent\_sort$ , мы получаем доказательство этого утверждения, которое будет автоматически проверено на корректность на этапе компиляции.

### 2.2. Индуктивные семейства типов

Интенциональная теория типов Мартин-Лёфа — закрытая система в том смысле, что у пользователя нет способа добавить еще один тип и необходимо использовать существующие:  $\Pi$ -типы,  $\Sigma$ -типы, размеченные

объединения, тип равенства, тип конечных множеств, натуральные числа и вселенные.

Несмотря на то, что система типов Мартин-Лёфа закрыта, введение типов данных производилось систематически:

1. Ввести константу
2. Ввести правила построения термов этого типа
3. Ввести правила использования термов этого типа (элиминации)
4. Ввести правила вычисления, по которым правила элиминации и построения обратны друг другу

В [7] предлагает схему с семействами типов, рекурсивно зависящими друг от друга.

Альтернативный подход: ввести тип  $W$ , который является самым общим индуктивным типом. Но мы теряем некоторые свойства связанные с равенством.

## **2.3. Сопоставление с образцом для индуктивных семейств типов**

Усложнение системы по сравнению с System F сказывается на реализации сопоставления с образцом. Наличие  $\Pi$ -типов означает, что в функции нескольких аргументов выполняя сопоставление по первому аргументу, меняется тип последующих и, как следствие, набор подходящих для них паттернов. А добавляя к этому еще семейства типов, теряется линейность.

Механизм сопоставления с образцом предпочтительней прямого использования элиминаторов по следующим соображениям [6]:

- Неожидаемое вычислительное поведение
- Читаемость
- В индуктивных семействах в зависимости от того, что считать параметрами, а что индексами меняется элиминатор.

Первое представление сопоставления с образцом для языков с зависимыми типами и индуктивными семействами было дано в [6].

### **3. Механизмы элиминации в языках с зависимыми типами**

**3.1. Выразимость сопоставления с образцом через элиминаторы**

**3.2. Расширение сопоставления с образцом views**

**3.3. Прочие расширения сопоставления с образцом**

**3.4. Проблемы механизма сопоставлением с образцом**

## 4. Список литературы

- [1] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.
- [2] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.
- [3] Rod M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.
- [4] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
- [5] Rod M Burstall, David B MacQueen, and Donald T Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143. ACM, 1980.
- [6] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Program*, pages 66–79. Citeseer, 1992.
- [7] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.
- [8] P. Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium*, 1973.
- [9] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [10] Philip Wadler. Propositions as types.
- [11] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.