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

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 вычислимых функций [2]. 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 [2]:

- 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 бэта-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 [3]:

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

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

- 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:

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

$$\frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

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

- 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 Жирара [13]:

- Типы

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

- Термы

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

- 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 [16]. Например, существует ровно один терм с типом $\forall a.\forall b.a \to b \to a$ — $\Lambda a.\Lambda b.\lambda x.\lambda y.x$.

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 $List$ [13]:

- 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:

$$\Gamma \vdash true : Bool$$

$$\Gamma \vdash false : Bool$$

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

$$\Gamma \vdash nil[T] : List\ T$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 обратно 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 [4] предложил синтаксическую конструкцию, которая позволяет избавиться от явного написания предикатов с декструкторами, заменив их соответствующими конструкторами:

$$
\begin{array}{lll}
\textbf{let}\ lst = cons\ x\ xs & \Leftrightarrow & \textbf{let}\ lst = cons\ x\ xs \\
\textbf{let}\ (cons\ x\ xs) = lst & \Leftrightarrow & \textbf{let}\ x = head\ lst; \textbf{let}\ xs = tail\ lst \\
lst\ \textbf{is}\ nil & \Leftrightarrow & isnil\ lst
\end{array}
$$

И развивая идею чуть дальше, заменять конструкции вида:

$$
\begin{array}{ll}
\textbf{if} & e\ \textbf{is}\ c_1\ \textbf{then let}\ (c_1\ x_1\ \ldots\ x_{k_1}) = e; \phi_1\ x_1\ \ldots\ x_{k_1} \\
\textbf{else if} & e\ \textbf{is}\ c_2\ \textbf{then let}\ (c_2\ x_1\ \ldots\ x_{k_2}) = e; \phi_2\ x_1\ \ldots\ x_{k_2} \\
 & \vdots \\
\textbf{else if} & e\ \textbf{is}\ c_n\ \textbf{then let}\ (c_n\ x_1\ \ldots\ x_{k_n}) = e; \phi_n\ x_1\ \ldots\ x_{k_n}
\end{array}
$$

конструкцией

$$
\begin{array}{l}
\textbf{cases}\ e: \\
\quad c_1\ x_1\ \ldots\ x_{k_1} : \phi_1\ x_1\ \ldots\ x_{k_1} \\
\quad c_2\ x_1\ \ldots\ x_{k_2} : \phi_2\ x_1\ \ldots\ x_{k_2} \\
\quad \vdots \\
\quad c_n\ x_1\ \ldots\ x_{k_n} : \phi_n\ x_1\ \ldots\ x_{k_n}
\end{array}
$$

Механизм, названный позднее сопоставлением с образцом получил реализацию в функциональных языках ML и Haskell. Там он оказался практически удобным и, обретя 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 [5], 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 Hope [6]. В нем был введен способ вводить пользовательские типы данных, которые имеют форму деревьев. Например, $Bool$, $List\ T$ принимают вид:

$$\begin{aligned}
&\textbf{data } Bool == true ++ false\\
&\textbf{typevar } T\\
&\textbf{data } List(T) == nil ++ cons(T \# List(T))
\end{aligned}$$

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» [15]. Это соответствие систем типов системам математической логики. В частности, System F соответствует интуиционистской (конструктивной) пропозициональной логике второго порядка, но без квантора существования. Суть соответствия:

$$
\begin{aligned}
\text{Тип} &\leftrightarrow \text{Утверждение}\\
\text{Терм типа} &\leftrightarrow \text{Доказательство утверждения}\\
\text{Вычисление терма} &\leftrightarrow \text{Упрощение доказательства}
\end{aligned}
$$

Если теперь в качестве логики взять интуиционисткую логику предикатов высших порядков и построить по ней систему типов, то получится интуиционистская теория типов [12], 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, П-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 П 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 утверждение, которое будет автоматически проверено на корректность на этапе компиляции.

Нужно заметить, что поскольку в типах теперь могут встречаться термы, то требуется гарантия, что проверка типов разрешима. Для этого достаточно, чтобы все термы редуцировались до нормальной формы за конечное число шагов.

2.2. Индуктивные семейства типов

MLTT — закрытая система в том смысле, что у пользователя нет способа добавить еще один тип и необходимо использовать существующие: П-типы, Σ-типы, размеченные объединения, тип равенства, тип конечных множеств, натуральные числа и вселенные. Преимущество закрытой системы в её простоте [1]: легче доказать непротиворечивость и прочие свойства. Но как отмечалось в 1.3, встроенных типов бывает недостаточно.

Один из подходов — расширить систему одним типом W, через который выразимы все индуктивные типы [10]. Но он работает только в системах с экстенциональным равенством, а рассматриваемая здесь система обладает интенциональным. Интенциональное равенство — равенство по определению с точностью до нормальной формы и, вследствие редуцируемости всех термов, является разрешимым. Экстенциональное равенство — равенство по наблюдению. В частности, функции экстенционально равны, когда они на всех аргументах дают одинаковый результат, а интенционально равны, когда их определения совпадают. Недостаток эксенциональности в неразрешимости [1].

Другой подход заключается в открытии наружу схемы введения новых типов. Несмотря на закрытость системы, Мартин-Лёф вводил каждый новый тип по одной и той же схеме, которая напоминает используемую в 1.2.1:

1. Ввести константу, отвечающую типу
2. Ввести правила построения термов этого типа
3. Ввести правила использования термов этого типа (элиминации)
4. Ввести правила вычисления, по которым правило элиминации обратно правилу построения

Например, равенство определяется:

1. $I(A, a, b)$, где $a : A$, $b : A$.
2. $r : \Pi(x : A) I(A, x, x)$
3. $I\_elim : \Pi(x : A)\Pi(c : C(x, x, r(x)))\Pi(y : A)\Pi(z : I(A, x, y))C(x, y, z)$

4. $I\_elim(x, c, x, r(x)) = c$.

Равенство еще является примером семейств индуктивных типов: в $I(A, a, b)$ у типа три параметра: A, a и b. У конструктора r результирующий тип $I(A, x, x)$, где x — аргумент r. В такой ситуации A называется параметром типа, потому что во всех конструкторах он получен из определения типа, а a и b называются индексами, потому что они зависят от конструктора и его аргументов. Типы с индексами называются семействами типов.

Дибиер [10] расширяет эту схему семействами типов, которые могут рекурсивно зависеть друг от друга.

2.3. Сопоставление с образцом для индуктивных семейств типов

Усложнение системы по сравнению с System F сказывается на реализации сопоставления с образцом. Наличие П-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. Линейным называется паттерн, в котором все переменные различны. Наличие типов с индексами может наложить ограничения на две различные переменные в паттерне, требуя чтобы они были одинаковы. Например, $f : \Pi(n : Nat)\Pi(k : Nat)\Pi(pr : I(Nat, n+1, k))Nat$. Выполняя сопоставление с образцом на третьем аргументе, единственный подходящий конструктор — $r(n+1)$. Его тип $I(Nat, n+1, n+1)$ и таким образом на месте второго аргумента обязано стоять выражение $n+1$.

Первое представление сопоставления с образцом для языков с зависимыми типами и индуктивными семействами было дано в [9].

Идея состоит в разбиении контекста. Имея функцию $f(x_1 : A_1, \ldots, x_n : A_n) : A$ контекстом является набор $x_i : A_i$ и A. Сопоставление с одним из x_i разбивает контекст: уточняется значение x_i, тип A_i (если в нем есть индексы), все A_j для $j > i$ и A из-за возможной зависимости от x_i, все x_j для $j < i$ из-за возможной зависимости по индексам в A_i. Уточнение A_j приводит к тому, что множество их значений может измениться и в том числе стать пустым. В таком случае тело функции для данного образца можно не реализовывать, потому что все равно невозможно было бы вызвать функцию с такими аргументами.

Поскольку в MLTT все функции обязаны завершаться на всех входах, то для данного способа введения новых функциональных констант в систему нужно ввести ограничения: для каждого входа должен быть соответствующий образец и никакой рекурсивный вызов не должен привести к бесконечной рекурсии. Обе эти задачи в общем случае неразрешимы [11]. И поэтому в обоих случаях используются неточные алгоритмы.

3. Механизмы элиминации в языках с зависимыми типами

3.1. Выразимость сопоставления с образцом через элиминаторы

В 2.3 сопоставление с образцом приводилось как альтернатива элиминаторам. С точки зрения использования оно удобнее, потому что напоминает математическое задание функции и из определения сразу видно вычислительную составляющую [9]. Но с точки зрения теории, механизм усложняет доказательство корректности ядра системы.

Чтобы разрешать сопоставление с образцом, но при этом оставить ядро простым для понимания в [11] предлагается способ трансляции функции, заданной с помощью сопоставления с образцом в функцию, заданную с помощью элиминаторов, сохраняя вычислительные свойства.

Еще один результат этой работы — доказательство, что для введения в MLTT сопоставления с образцом необходимо и достаточно обогатить систему аксиомой K. Аксиома K — это альтернативный элиминатор для типа равенства. Он имеет тип: $K : P(r(x)) \to \Pi(h : Id(A, x, x))P(h)$. Отсюда в том числе следует $UIP : \Pi(p_1 : Id(A, x, y))\Pi(p_2 : Id(A, x, y))Id(Id(A, x, y), p_1, p_2)$. Другими словами, что все доказательства равенства x и y равны между собой. Такое свойство совместимо с MLTT, но вступает в противоречие с набирающей популярность гомотопической теорией типов (HoTT) [14]. HoTT можно рассматривать как расширение MLTT добавлением аксиомы унивалентности и высших индуктивных типов. Отсюда следует, что в HoTT не может быть использован метод сопоставления с образцом из [9]. Тем не менее, чтобы не терять механизм полностью, предлагаются варианты ограничения механизма, так чтобы он не требовал K [8].

3.2. Расширение сопоставления с образцом views

3.3. Прочие расширения сопоставления с образцом

3.3.1. Механизм «smartcase»

3.3.2. Сопоставление с образцом без упорядочивания уравнений

Механизмы в 1.2.2 и 2.3 используют упорядоченный набор уравнений: алгоритм идет сверху вниз по уравнениям, пока не найдет первый подходящий набор образцов. Преимущества такого способа в том, что он позволяет в некоторых случаях уменьшить количество требуемых уравнений [7] и в том, что написанные таким образом функции подлежат трансляции в элиминаторы как описано в [11].

С другой стороны, при таком определении отсутствует интересное свойство: уравнения нельзя считать равенствами по определению. Уравнение для набора образцов является равенством в системе только если все предыдущие наборы образцов не подошли. Например, определение сложения натуральных чисел:

$$
\begin{aligned}
&add : Nat \to Nat \to Nat \\
&add\ 0\ n = n \\
&add\ (succ\ n)\ m = succ\ (add\ n\ m) \\
&add\ n\ 0 = n \\
&add\ n\ (succ\ m) = succ\ (add\ n\ m)
\end{aligned}
$$

Из этого определения не следует, что $add\ n\ 0 =_{def} n$. Этот факт придется доказывать отдельно и система не сможет использовать его автоматически.

В [7] предлагается использовать подход, когда каждое уравнение порождает равенство по определению.

3.4. Проблемы механизма сопоставлением с образцом

4. Список литературы

[1] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

[2] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[3] 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.

[4] Rod M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[5] Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[6] 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.

[7] Jesper Cockx. *Overlapping and order-independent patterns in type theory.* PhD thesis, Master thesis, KU Leuven, 2013.

[8] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without k. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 257–268. ACM, 2014.

[9] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Program*, pages 66–79. Citeseer, 1992.

[10] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[11] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

[12] P. Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium*, 1973.

[13] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[14] VA Voevodsky et al. Homotopy type theory: Univalent foundations of mathematics. *Institute for Advanced Study (Princeton), The Univalent Foundations Program*, 2013.

[15] Philip Wadler. Propositions as types.

[16] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.