

Правительство Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение высшего образования  
«Санкт-Петербургский государственный университет»

На правах рукописи

Березун Даниил Андреевич

## **ТРАССИРУЮЩАЯ НОРМАЛИЗАЦИЯ**

Специальность 05.13.11

«Математическое и программное обеспечение вычислительных машин,  
комплексов и компьютерных сетей»

Диссертация на соискание учёной степени  
кандидата физико-математических наук

Научный руководитель:  
доктор технических наук, доцент, профессор кафедры системного  
программирования  
КОЗНОВ Дмитрий Владимирович

Санкт-Петербург — 2017

# Содержание

<b>Введение</b> . . . . .	<b>5</b>
<b>1 Обзор области исследования</b> . . . . .	<b>13</b>
1.1 Лямбда-исчисление . . . . .	13
1.2 Простое типизированное лямбда-исчисление . . . . .	17
1.3 Стратегии вычислений . . . . .	19
1.3.1 Слабые порядки редукции . . . . .	21
1.3.2 Сильные порядки редукции . . . . .	22
1.4 Головная нормальная форма и дерево Бёма . . . . .	23
1.5 $\eta$ -длинная форма терма . . . . .	24
1.6 Головная линейная редукция . . . . .	26
1.6.1 Головная линейная редукция: классическое определение . . . . .	27
1.6.2 Трассирующая нормализация . . . . .	29
1.7 Системы переходов . . . . .	31
1.8 Выводы . . . . .	31
<b>2 Полная головная линейная редукция</b> . . . . .	<b>33</b>
2.1 Модель головной линейной редукции . . . . .	33
2.1.1 Головная линейная редукция как система переходов . . . . .	33
2.1.2 Согласованность головной и головной линейной стратегий редукции . . . . .	37
2.2 Модель полной головной линейной редукции . . . . .	41
2.2.1 Система переходов для полной головной линейной редукции . . . . .	41
2.2.2 Корректность модели полной головной линейной редукции . . . . .	43
<b>3 Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления</b> . . . . .	<b>46</b>

3.1	Семантика, основанная на подстановке . . . . .	47
3.2	Вычисление с окружением . . . . .	47
3.3	Хвосто-рекурсивная семантика . . . . .	50
3.4	Семантика, основанная на истории и окружении . . . . .	50
3.5	Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления или семантика, основанная только на истории .	52
<b>4</b>	<b>Корректность алгоритма трассирующей нормализации для нетипи- зированного лямбда-исчисления . . . . .</b>	<b>57</b>
4.1	Головная линейная редукция и ограниченная версия алгоритма трассирующей нормализации . . . . .	59
4.1.1	Система переходов для BUNP . . . . .	59
4.1.2	Соответствие между головной линейной редукцией и огра- ниченной версией алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления . . . . .	61
4.2	Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления и полная головная линейная редукция . . . . .	64
4.2.1	UNP в виде системы переходов . . . . .	65
4.2.2	Соответствие между CHLR и UNP . . . . .	67
<b>5</b>	<b>Компиляция путём специализации . . . . .</b>	<b>71</b>
5.1	Частичные вычисления . . . . .	71
5.2	Проекция Футамуры или проекция Футамуры–Ершова–Турчина .	73
5.3	Компиляция нетипизированного лямбда-исчисления путём специ- ализации алгоритма трассирующей нормализации . . . . .	76
5.3.1	Преобразование нормализующей процедуры в компилятор	76
5.3.2	Специализация UNP на входной лямбда-терм . . . . .	77
<b>6</b>	<b>Трассирующая нормализация и стратегии вычислений . . . . .</b>	<b>84</b>
6.1	Алгоритм трассирующей нормализации, соответствующий аппли- кативному порядку редукции . . . . .	85
6.2	Об адаптации алгоритма трассирующей нормализации для страте- гии вызова по значению . . . . .	90
6.3	Трассирующая нормализация для PCF-подобного языка . . . . .	93

<b>Заключение</b> . . . . .	<b>96</b>
<b>Литература</b> . . . . .	<b>98</b>
<b>Список рисунков</b> . . . . .	<b>110</b>
<b>Список таблиц</b> . . . . .	<b>112</b>
<b>Приложение А. Пример исполнения программ на LLL</b> . . . . .	<b>113</b>
<b>Приложение Б. Пример работы системы переходов для алгоритма трас- сирующей нормализации, соответствующего аппликативному по- рядку редукции</b> . . . . .	<b>120</b>

# Введение

## Актуальность работы

Лямбда-исчисление, наряду с машинами Тьюринга, теорией частично-рекурсивных функций, формальными алгорифмами Маркова и Поста, является одной из основных формальных моделей вычислений [1–3]. Первая модель лямбда-исчисления, предложенная А. Чёрчем (A. Church) в 30-х годах 20 века, оказалась противоречивой: в 1935 году С. Клини (S. Kleene) и Д. Б. Россер (J. V. Rosser) выявили в ней парадокс, названный в их честь. В 1936 году Чёрч предложил первую непротиворечивую систему, ныне известную как *нетипизированное, бестиповое* или *чистое* лямбда-исчисление, а в 1940 году он же предложил модель простейшего типизированного исчисления — *простое типизированное лямбда-исчисление* [4]. С тех пор лямбда-исчисление играет особую роль в теории вычислимости и стало основой для целой парадигмы программирования — функционального программирования.

Тем не менее, различные языки, основанные на лямбда-исчислении, долгое время не имели *полностью абстрактных моделей* вычислений, т.е. таких денотационных моделей, что существует изоморфизм между терминами языков и их представлениями в этих моделях. Впервые задача построения полностью абстрактной модели вычислений была поставлена в 1975 году Г. Плоткиным (G. Plotkin) для языка программирования вычислимых функций PCF (Programming Computable Functions) [5]. Первым подходом, предложившим решение поставленной задачи, стала *игровая семантика*, на основе которой были построены полностью абстрактные модели для целого спектра языков программирования.

Игровая семантика определяет как *денотационную*, так и *операционную* модели языка как некоторое взаимодействие, игру, между программой и её окружением. Недавние работы Л. Онга и В. Блюма (С.-Н. L. Ong и W. Blum) показали, что игровая семантика способна нормализовывать лямбда-термы просто-

го типизированного лямбда-исчисления без использования стандартных подходов, таких как  $\beta$ -редукции и окружения [6, 7]. Стратегия вычислений, основанная на данном подходе, получила название *головная линейная редукция* (Head Linear Reduction) [8–10], а сам подход — *трассирующая нормализация* (Traversal-Based Normalization) [6, 11, 12]. Головная линейная редукция играет особую роль в различных подходах к вычислениям, таких как оптимальные редукции [13–16], геометрия взаимодействия [17, 18], сети доказательств [19, 20] и др.

Тем не менее, до сих пор оставался открытым вопрос расширения подхода трассирующей нормализации до исчисления, полного по Тьюрингу. Такое расширение определило бы свежий взгляд на функциональные языки программирования, позволяя исследовать их свойства с нового ракурса.

### Степень разработанности темы исследования

Существует множество работ, посвящённых лямбда-исчислению и его свойствам, начиная с классических работ А. Чёрча [4, 21, 22], А. М. Тьюринга (A. M. Turing) [23–25] и Х. Б. Карри (H. B. Curry) [26–29]. Эти работы были посвящены основам математики и математической логики, и, в частности, теориям сложности и вычислимости. Значительный вклад в теорию лямбда-исчисления внёс голландский математик Х. П. Барендрегт (H. P. Barendregt), предложив так называемый лямбда-куб [30, 31]. Множество отечественных и зарубежных учёных изучали свойства языков, основанных на лямбда-исчислении, в том числе Д. С. Скотта (D. S. Scott) и К. Страчей (C. S. Strachey) построили денотационные модели для целого ряда языков программирования [32–35]. Тем не менее, денотационные модели некоторых языков программирования не обладали *свойством полной абстракции* (Full-Abstraction Property), задачу построения которых сформулировал G. Plotkin [5, 36].

В 50-х годах 20 века П. Лоренсен (P. Logenzen) предложил так называемую игровую семантику для логики [37], которая получила развитие в работах немецкого философа К. Лоренца (K. Lorenz) [38]. Дальнейшее развитие этого подхода такими учёными как С. Абрамски (S. Abramsky), Л. Онг (C.-H. L. Ong), П. Малакария (P. Malacaria), Р. Джагадесан (R. Jagadeesan) и Г. МакКускер (G. McCusker) [39–41], позволило в начале 90-х годов решить задачу, сформулированную Плоткиным. Дальнейшее развитие игровая семантика получила в работах Д. М. Э. Хуланда (J.

М. Е. Hyland), А. Кера (A. D. Ker), Х. Никау (H. Nickau), В. Блюма (W. Blum), Д. Гика (D. Ghica) и др [10, 39, 42–49].

В последние годы исследование операционных особенностей игрой семантики стало актуальной темой исследований, о чём свидетельствуют работы Д. Р. Гика (D. R. Ghica), Л. Онга и В. Блюма [10, 46–50]. В. Данос (V. Danos) и Л. Ренье (L. Regnier) описывали игровую семантику программ с помощью абстрактных машин КАМ (Krivine Abstract Machine) и ПАМ (Pointer Abstract Machine) [8, 9, 51], а также предложили связать её с линейной редукцией. Л. Онг впервые определил трассирующую нормализацию для простого типизированного лямбда-исчисления [6], а В. Блюм распространил этот подход до безопасного лямбда-исчисления (Safe Lambda-Calculus) [44, 52]. Тем не менее, вопрос распространения подхода трассирующей нормализации до полного по Тьюрингу исчисления оставался открытым.

**Целью** данной работы является обобщение подхода трассирующей нормализации до полного по Тьюрингу исчисления, обоснование корректности трассирующей нормализации и исследование возможности представления различных языковых конструкций в рамках предложенного подхода при компиляции функциональных языков программирования.

Для достижения вышеупомянутой цели были поставлены следующие **задачи**.

1. Разработать в рамках стратегии вычисления нормального порядка редукции алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления.
2. Формально доказать корректность предложенного алгоритма.
3. Исследовать возможность адаптации трассирующей нормализации для различных стратегий вычислений.
4. Исследовать возможность представления различных языковых конструкций в рамках предложенного подхода с точки зрения компиляции основных конструкций функциональных языков программирования.

Постановка цели и задач исследования соответствует следующим пунктам паспорта специальности 05.13.11: методы, модели и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований,

верификации и тестирования (пункт 1); языки программирования и системы программирования, семантика программ (пункт 2).

### Методология и методы исследования

Методология исследования основана на идеях и подходах информатики к описанию и анализу понятия вычислимости. Используются также операционная и денотационная концепции описания семантик языков программирования.

В работе использовались методы синтаксического преобразования лямбда-термов [30, 53–56]. Для формализации моделей использовался подход к описанию семантик в виде систем переходов [57, 58], а для доказательства их корректности — метод симуляции одной системы переходов другой посредством определения отношения, связывающего соответствующие состояния этих систем переходов. Были использованы стратегии вычислений вызова по имени, нормального порядка, вызова по значению, аппликативного порядка, головной редукции, линейной редукции и вызова по необходимости. Для автоматического преобразования программ и семантик использовались методы частичных вычислений и дефункционализации по Рейнольдсу (Reynolds) [59–61]. Программная реализация предложенных в диссертации результатов была выполнена с помощью языков программирования Haskell [62] и Racket [63].

### Основные положения, выносимые на защиту

1. Разработан алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления, соответствующий нормальному порядку редукций.
2. Представлена модель полной головной линейной редукции, являющаяся расширением известной модели головной линейной редукции. Предложенная модель формализована в виде системы переходов, доказана корректность этой модели относительно головной редукции.
3. Доказана корректность представленного алгоритма трассирующей нормализации относительно предложенной модели полной головной линейной редукции путём его формализации в виде системы переходов и дальнейшей симуляции системы переходов для полной головной линейной редукции. Таким образом, было доказано, что предложенная процедура трассирующей нормализации действительно является нормализующей.



4. Предложенный алгоритм адаптирован для других, отличных от нормального порядка, стратегий вычислений: аппликативного порядка редукций и вызова по необходимости.
5. Предложен и реализован на примере нетипизированного лямбда-исчисления новый метод компиляции функциональных языков программирования в низкоуровневое представление путём специализации представленного алгоритма трассирующей нормализации на входной терм.

**Научная новизна** полученных в ходе исследования результатов заключается в следующем.

1. Впервые было введено и формально описано понятие полной головной линейной редукции и доказана её корректность.
2. Алгоритм трассирующей нормализации, представленный в работе, отличается от аналогов (работы Л. Онга и В. Блюма [6,7,52]) тем, что он не накладывает каких-либо ограничений на лямбда-термы, что позволяет корректно нормализовывать произвольный лямбда-терм. Кроме того, алгоритмы, предложенные Л. Онгом и В. Блюмом, применяются лишь к частным случаям, а именно, к простому типизированному лямбда-исчислению и безопасному лямбда-исчислению.
3. Предложен новый подход к компиляции функциональных языков программирования путём специализации процедуры трассирующей нормализации на входной терм, и исследованы её свойства.

**Теоретическая значимость** диссертационного исследования заключается в формализации понятий головной и полной головной линейной редукций, разработке формального алгоритма, решающего задачу трассирующей нормализации, соответствующей полной головной линейной редукции, для бестипового лямбда-исчисления, а также в формальном доказательстве корректности представленного алгоритма.

### **Практическая значимость**

Трассирующая нормализация представляет собой новый подход к вычислениям, позволяя исследовать ряд свойства языков программирования с нового ракурса. Как показано в диссертации, применение частичных вычислений [64–72]

к предложенному в диссертации алгоритму трассирующей нормализации позволяет производить трансляцию лямбда-термов в низкоуровневое представление, сохраняя исходную семантику программы [73]. Такой подход к компиляции является перспективным направлением в области *автоматической генерации компиляторов на основе семантики* (Semantics-Directed Compiler Generation) [74–79]. Описание известных языковых конструкций и программных трансформаций при помощи нового подхода открывает перспективы к проектированию языка промежуточного представления, являющегося базой для описания семантик широкого класса языков программирования и позволяющего автоматически генерировать компиляторы соответствующих языков.

### **Степень достоверности результатов**

Достоверность и обоснованность результатов исследования опирается на использование формальных методов исследуемой области, выполнение формальных доказательств и инженерные эксперименты.

Основные результаты работы были представлены на следующих научных конференциях и семинарах: семинар “Математические вопросы информатики” (27 октября 2017, МГУ им. М.В. Ломоносова, мех.-мат. ф-т, Москва, Россия), семинар в ИПС им. А.К. Айламазяна РАН (26 октября 2017, Переславль-Залесский, Россия), семинар в ИПМ им. М.В. Келдыша РАН (25 октября 2017, Москва, Россия), конференция “Языки программирования и компиляторы” (PLC 2017, 3–5 апреля 2017, Ростов-на-Дону, Россия), Games for Logic and Programming Languages XII (GaLoP 2017, 22-23 апреля 2017, Уппсала, Швеция), PERM 2017 Workshop on Partial Evaluation and Program Manipulation (16 – 17 января, 2017, Париж, Франция), внутренние семинары Department of Computer Science of Oxford University (февраль 2016, апрель 2016, март 2017, Оксфорд, Великобритания), Fifth International Valentin Turchin Workshop on Metacomputation (Meta 2016, 27 июня – 1 июля, 2016, Переславль-Залесский, Россия), Games for Logic and Programming Languages XI (GaLoP 2016, 2-3 апреля 2016, Эйнховен, Нидерланды) и внутренние семинары Department of Computer Science of DIKU (октябрь-ноябрь 2015 и ноябрь 2017, Копенгаген, Дания).

### **Публикации по теме диссертации**

Все результаты диссертации опубликованы в пяти печатных работах, зарегистрированных в РИНЦ. Две одиночные статьи изданы в журналах из “Перечня

российских рецензируемых научных журналов, в которых должны быть опубликованы основные научные результаты диссертаций на соискание учёных степеней доктора и кандидата наук”. Две статьи опубликованы в изданиях, входящих в базы цитирования SCOPUS и Web of Science.

Вклад автора в публикациях, написанных в соавторстве, распределён следующим образом. В работе [73] автору принадлежит формализация метода трассирующей нормализации, реализация предложенных семантических преобразований, идея и формализация метода трассирующей нормализации для языка PCF. Соавторы предложили схему представления нормализующей процедуры, описали применение частичных вычислений к процедуре нормализации с целью компиляции термов лямбда-исчисления, сформулировали направления дальнейших исследований. В работе [80] автору принадлежит реализация инструментальных средств, разработка и реализация модели представления программной кучи и проведение экспериментов.

### **Объем и структура работы**

Диссертация состоит из введения, шести глав, заключения и двух приложений. Полный объём диссертации составляет 130 страниц с 34 рисунками и 1 таблицей. Список литературы содержит 107 наименований. В первой главе приводится обзор области исследования. Рассматриваются существующие подходы к редукции термов, а также описываются существующие подходы к трассирующей нормализации. Во второй главе приводится модель головной линейной редукции, согласованная с классическим определением В. Даноса и Л. Ренье [8], в виде системы переходов, а также модель полной головной линейной редукции, являющаяся расширением модели головной линейной редукции и устанавливается корректность предложенных моделей относительно головной редукции. В третьей главе представлен разработанный автором алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления. В четвёртой главе представлена формализация предложенного алгоритма трассирующей нормализации в виде системы переходов, а также установлена её корректность путём определения отношения симуляции между системами переходов для полной головной линейной редукции и трассирующей нормализации. В пятой главе показано, как алгоритм трассирующей нормализации может быть использован для генерации компилятора нетипизированного лямбда-исчисления в низкоуровневое представление с помощью

известных методов специализации программ. Наконец, в шестой главе показано, как предложенный алгоритм трассирующей нормализации может быть адаптирован для различных стратегий редукции и поддержки различных языковых конструкций на примере языка РСФ.

### **Благодарности**

Автор выражает благодарность Д. Ю. Булычеву и Н. Д. Джонсу (N. D. Jones) за создание условий для изучения данной темы и руководство на начальных этапах, Л. Онгу (L. Ong), С. А. Романенко, Н. Н. Непейводе и А. М. Миронову за ценные вопросы и комментарии к работе, позволившие уточнить ряд формулировок и улучшить изложение результатов, научному руководителю Д. В. Кознову за неоценимый вклад в работу над диссертацией и помощь по ходу исследований, компании ООО “ИнтелиДжей Лабс” и А. Иванову за поддержку исследований. Отдельную благодарность хочется выразить моей семье, особенно маме, которая является моей опорой и верным спутником на протяжении всей моей жизни.

# Глава 1

## Обзор области исследования

В данной главе рассмотрены существующие подходы к нормализации лямбда-термов и, в том числе, к трассирующей нормализации. Описываются алгоритмы трассирующей нормализации для простого типизированного лямбда-исчисления и безопасного лямбда-исчисления, предложенные Л. Онгом и В. Блюмом соответственно.

### 1.1. Лямбда-исчисление

Лямбда-исчисление было предложено А. Чёрчем в 30-х годах 20 века. В 1935 году С. Клини и Д. Б. Россер установили противоречивость первой модели исчисления. Спустя непродолжительное время А. Чёрч предложил новую модель, известную сейчас как *чистое, нетипизированное или беспитовое лямбда-исчисление*. Спустя 4 года, в 1940, он создал первую модель типизированного исчисления, ныне известную как *простое-типизированное лямбда-исчисление* [4]. В 1960-х годах была установлена связь между лямбда-исчислением и языками программирования, а само лямбда-исчисление стало одной из основных моделей вычислений наряду с машинами Тьюринга, моделью частично-рекурсивных функций, алгоритмами Маркова и Поста и другими.

Объекты модели лямбда-исчисления, называемые *лямбда-термам*, строятся с помощью трёх синтаксических конструкций: переменной, функциональной абстракции по переменной, т.е. анонимной функции, и применения одного лямбда-терма к другому — функции к аргументу. Формально, *лямбда-выражением*

(лямбда-термом,  $\lambda$ -термом или просто термом,  $\lambda$ -term) называется выражение, удовлетворяющее следующей грамматике

$$\Lambda^1 ::= V \mid \Lambda @^2 \Lambda \mid \lambda V. \lambda,$$

где  $V$  — множество конечных строк, называемых *переменными*, над некоторым фиксированным алфавитом  $\Sigma \setminus \{_, ., @, \lambda\}$ , выражение вида  $\lambda x. e$  называется *лямбда-абстракцией* (или просто *абстракцией*,  $\lambda$ -abstraction) по переменной  $x$ , а выражение вида  $e_1 @ e_2$ <sup>3</sup> — *применением* (application) терма  $e_1$  к *аргументу* — терму  $e_2$ .

Выделяют два вида вхождения переменной в терм: свободное и связанное. *Связанными* называются такие вхождения переменных, по которым уже была произведена абстракция в дереве синтаксического разбора терма, все остальные вхождения переменных называются *свободными*. Так, например, в терме  $\lambda y. x @ y @ (\lambda x. x)$  первое вхождение переменной  $x$  является свободным, а второе — связанным, также как и единственное вхождение переменной  $y$ .

Основной формой эквивалентности лямбда-термов является так называемая  $\alpha$ -эквивалентность ( $\alpha$ -конверсия,  $\alpha$ -conversion), согласно которой термы, получаемые друг из друга посредством переименования одной или нескольких связанных переменных, являются  $\alpha$ -эквивалентными. Так, термы  $\lambda x. x$  и  $\lambda y. y$  —  $\alpha$ -эквивалентны. Отношение  $\alpha$ -эквивалентности обозначается знаком  $\sim_\alpha$ , например,  $\lambda x. x \sim_\alpha \lambda y. y$ . Как правило, термы в лямбда-исчислении рассматриваются с точностью до  $\alpha$ -эквивалентности, поэтому вместо терминов связанное и свободное вхождение переменной часто используют термины *связанная* и *свободная* переменная, соответственно, подразумевая, что различные вхождения одной и той же переменной не могут быть и связанными, и свободными одновременно. Далее мы будем придерживаться *соглашения Барендрегта* (Barendregt's convention), подразумевающего, что различные переменные имеют различные имена. Формальные определения множеств свободных ( $FV$ ) и связанных ( $BV$ ) переменных приведены на рисунке 1.

<sup>1</sup>Мы также будем использовать скобки (“(”, “)”) как незначащие символы грамматики в тех ситуациях, когда из конкретного синтаксиса невозможно однозначное восстановление абстрактного.

<sup>2</sup>Символ @ иногда опускается:  $\Lambda @ \Lambda \equiv \Lambda \Lambda \equiv \Lambda \Lambda$ .

<sup>3</sup>Мы будем считать, что приоритет оператора “@” выше чем у “.”, т.е.  $\lambda x. x @ y \equiv \lambda x. (x @ y)$ , а ассоциативность у оператора “@” — левая, т.е.  $x @ y @ z = (x @ y) @ z$ .

Основной аксиомой лямбда-исчисления является аксиома  $\beta$ -редукции, которая производит применение функции (лямбда-абстракции) к её аргументу путём синтаксической замены всех вхождений переменной, связанной этой абстракцией, на тело аргумента:

$$(\lambda x.e_1) e_2 =_{\beta} e_1[x/e_2].$$

Такая замена называется *подстановкой* аргумента  $e_1$  в терм  $e_2$ , а выражение вида  $(\lambda x.e_1) e_2$  — *редексом*. Снабжённое аксиомой  $\beta$ -редукции лямбда-исчисление является *полным по Тьюрингу*, определяя тем самым одну из простейших моделей вычислений. Термы, получаемые друг из друга по правилу  $\beta$ -редукции, называются  $\beta$ -эквивалентными. Запись  $s \rightarrow_{\beta} t$  означает, что терм  $t$  получается из терма  $s$  за один шаг  $\beta$ -редукции, а запись  $s \rightarrow_{\beta}^* t$  означает, что терм  $t$  получается из терма  $s$  за ноль или более шагов  $\beta$ -редукции.

Заметим, что подстановка не является тривиальной операцией. Например, рассмотрим терм  $(\lambda x.x y) x$ . Согласно правилу  $\alpha$ -эквивалентности он эквивалентен терму  $(\lambda z.z y) x$ , который, в свою очередь,  $\beta$ -эквивалентен терму  $\lambda z.z x$ . Если произвести подстановку  $[x/y]$  непосредственно в терме  $\lambda x.x y$ , то результатом будет терм  $\lambda x.x x$ , который уже не эквивалентен терму  $\lambda z.z x$ .

Таким образом, для подстановки важно, чтобы свободные переменные терма не связывались при подстановке. Такая подстановка называется *свободной от связывания* (Capture-Avoiding Substitution), далее будем называть ее просто подстановкой. Формальное определение подстановки приведено на рисунке 1.

В начале 70-х годов Н. де Брауном (N. de Bruijn<sup>4</sup>) была предложена нотация, получившая название *представление де Брауна* или неименованное представление [81], позволяющая полностью избавиться от имён в лямбда-термах: вхождения переменных заменяются натуральными числами, называемыми *индексами де Брауна* (de Bruijn indexes), а абстракции становятся анонимными. Таким образом, термы в представлении де Брауна представляют собой классы эквивалентности именованных лямбда-термов, факторизованных по правилу  $\alpha$ -эквивалентности. *Индекс де Брауна* — это натуральное число, которое обозначает вхождение переменной; оно равно количеству абстракций между этим вхождением и абстракцией, связывающей переменную, включительно. Например, терм

<sup>4</sup>В русскоязычной литературе также встречаются и другие варианты перевода этой фамилии: “де Брюйн”, “де Брейн”, “де Бройн”.



## Синтаксис

$$\Lambda ::= V \mid \Lambda @ \Lambda \mid \lambda V. \Lambda$$

$$V ::= \{x, y, \dots\}$$

## Подстановка

$$x[x/r] = r$$

$$y[x/r] = y, \text{ если } x \neq y$$

$$(e_1 e_2)[x/r] = (e_1[x/r])(e_2[x/r])$$

$$(\lambda x. e)[x/r] = \lambda x. e$$

$$(\lambda y. e)[x/r] = \lambda y. e[x/r], \text{ если } x \neq y \text{ и } y \notin FV(r)$$

 $\beta$ -редукция

$$(\lambda x. e_1) e_2 =_{\beta} e_1[x/e_2]$$

## Множество свободных переменных

$$FV(x) = \{x\}$$

$$FV(\lambda x. e) = FV(e) \setminus \{x\}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

## Рисунок 1. Именованное представление чистого лямбда-исчисления

$\lambda x. \lambda y. \lambda z. x z (y z)$ , известный как  $S$  комбинатор, в нотации де Брауна имеет следующий вид:  $\lambda \lambda \lambda 3 1 (2 1)$ . Заметим, что в представлении де Брауна при осуществлении подстановки может потребоваться пересчёт индексов. Формальное определение неименованного представления чистого лямбда-исчисления приведено на рисунке 2.

Ещё одно преобразование иногда считается стандартным для лямбда-исчисления —  $\eta$ -конверсия (эта-конверсия,  $\eta$ -conversion), отождествляющее функцию и абстракцию этой функции, применённой к переменной, по которой осуществлена абстракция. Преобразование функции  $f$  к виду  $\lambda x. f x$  получило название  $\eta$ -расширения ( $\eta$ -expansion), а преобразование, ему обратное, —  $\eta$ -редукции ( $\eta$ -reduction). Определения  $\eta$ -расширения и  $\eta$ -редукции приведены на рисунке 3.



## Синтаксис

$$\Lambda^d ::= \mathbb{N} \mid \Lambda^d \Lambda^d \mid \lambda \Lambda^d$$

Сдвиг в  $\Lambda^d$ 

$$\begin{aligned} \uparrow_c^d(k) &= \begin{cases} k, & k < c \\ k + d, & k \geq c \end{cases} \\ \uparrow_c^d(\lambda e) &= \lambda \uparrow_{c+1}^d(e) \\ \uparrow_c^d(e_1 e_2) &= \uparrow_c^d(e_1) \uparrow_c^d(e_2) \end{aligned}$$

Подстановка в  $\Lambda^d$ 

$$\begin{aligned} k[j/r] &= \begin{cases} r, & k = j \\ k, & \text{иначе} \end{cases} \\ (\lambda e)[j/r] &= \lambda e[j + 1 / \uparrow_0^1 r] \\ (e_1 e_2)[j/r] &= e_1[j/r] e_2[j/r] \end{aligned}$$

 $\beta$ -редукция в  $\Lambda^d$ 

$$(\lambda e_1)e_2 \rightarrow \uparrow_0^{-1}(e_1[0 / \uparrow_0^1(e_2)])$$

Множество связанных переменных

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.e) &= BV(e) \cup \{x\} \\ BV(e_1 e_2) &= BV(e_1) \cup BV(e_2) \end{aligned}$$

 $\alpha$ -эквивалентность

$$\lambda x.e \sim_\alpha \lambda y.e[x/y], \text{ если } y \notin FV(e)$$

Рисунок 2. Неименованное представление чистого лямбда-исчисления

 $\eta$ -конверсия

$$f \sim_\eta \lambda x.f x, \text{ если } x \notin FV(f)$$

 $\alpha$ -эквивалентность

$$\lambda x.e \sim_\alpha \lambda y.e[x/y], \text{ если } y \notin FV(e)$$

Рисунок 3.  $\eta$ -конверсия

## 1.2. Простое типизированное лямбда-исчисление

*Простое типизированное лямбда-исчисление* (Simply-Typed Lambda-Calculus, STLC) имеет такой же синтаксис, как и бестиповое. Тем не менее, не все термы бестипового исчисления являются *допустимыми* (Valid) термами простого типизированного. Типы в простом типизированном лямбда-исчислении описываются следующей грамматикой:  $\tau ::= \iota \mid \tau \rightarrow \tau$ , где  $\iota$  представляет собой некоторый *базовый тип* (Ground Type), а  $\tau_1 \rightarrow \tau_2$  — *функциональный тип* (стрелочный тип, Arrow Type), ассоциативность у оператора *стрелка*  $\rightarrow$  правая. Принято вы-

Синтаксис	Типизация
$\Lambda^{Cu} ::= \Lambda$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{Cu} x : \tau}$
Типы	
$\tau ::= \iota \mid \tau \rightarrow \tau$	$\frac{\Gamma, x : \tau \vdash_{Cu} e : \sigma}{\Gamma \vdash_{Cu} \lambda x. e : \tau \rightarrow \sigma}$
Контексты	
$\Gamma ::= \emptyset \mid \Gamma, V : \tau$	$\frac{\Gamma \vdash_{Cu} e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{Cu} e_2 : \tau_2}{\Gamma \vdash_{Cu} e_1 e_2 : \tau_2}$

Рисунок 4. Простое типизированное лямбда-исчисление в стиле Карри

делять два похода, два стиля типизации: стиль *Карри* (типизация по Карри, á la Curry) и стиль *Чёрча* (типизация по Чёрчу, á la Church). Эти два подхода являются принципиально разными: при типизации по Карри сначала задаётся грамматика термов — синтаксис, затем определяется их поведение — семантика, и наконец вводится система типов — типизация, отвергающая термы, обладающие нежелательным поведением, в то время как в стиле Чёрча типизация предшествует семантике<sup>5</sup> [23, 31, 82–85]. Иными словами, при типизации по Чёрчу семантикой — смыслом — наделены исключительно *правильно типизированные* (Well-Formed, Well-Typed) термы, а типизация по Карри даёт возможность рассуждать о поведении лямбда-термов вне зависимости от того, являются ли они правильно типизированными или нет. Синтаксис и типизации по Чёрчу и по Карри простого типизированного лямбда-исчисления в виде правил вывода (inference rules) приведены на рисунках 5 и 4 соответственно. Суждение вида  $\Gamma \vdash \Lambda : \tau$  называется *термом в контексте* (Term-in-Context), где *контекст типизации*  $\Gamma$  (Typing Context) является множеством текущих предположений о типах термов.

Одним из основных отличий двух систем типизации является свойство единственности типа. А именно, при типизации по Чёрчу терм имеет не более одного типа, в то время как при типизации по Карри он может иметь один или несколько типов, или быть нетипизируемым вовсе. Так, терм  $\lambda x : \tau. x$  (при типизации по Чёрчу) имеет единственный тип  $\tau \rightarrow \tau$ , где  $\tau$  — некоторый конкретный тип,

<sup>5</sup>Иногда под типизацией по Чёрчу понимают *явно* типизированные системы (типы переменных явно указываются в синтаксисе языка), а под типизацией по Карри — *неявно* типизированные (типы присваиваются в процессе компиляции или интерпретации программы). Путаница возникла в виду того, что сам Чёрч описывал своё исчисление в явном стиле, в то время как Карри использовал неявный стиль. Несмотря на то, что такой взгляд на стили типизации является исторически сложившимся: явно типизированные системы часто описывались и описываются в стиле Чёрча, а неявно типизированные — в стиле Карри, — он всё же является ошибочным.

Синтаксис	Типизация
$\Lambda^{Ch} ::= V \mid \Lambda^{Ch} \Lambda^{Ch}$ $\quad \mid \lambda V^\tau. \Lambda^{Ch}$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{Ch} x : \tau}$
<p style="text-align: center;">Типы</p> $\tau ::= \iota \mid \tau \rightarrow \tau$	$\frac{\Gamma, x : \tau \vdash_{Ch} e : \sigma}{\Gamma \vdash_{Ch} \lambda x^\tau. e : \tau \rightarrow \sigma}$
<p style="text-align: center;">Контексты</p> $\Gamma ::= \emptyset \mid \Gamma, V : \tau$	$\frac{\Gamma \vdash_{Ch} e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_{Ch} e_2 : \tau_2}{\Gamma \vdash_{Ch} e_1 e_2 : \tau_2}$

Рисунок 5. Простое типизированное лямбда-исчисление в стиле Чёрча

указанный в синтаксисе, например,  $\iota$  или  $(\iota \rightarrow \iota \rightarrow \iota) \rightarrow \iota \rightarrow \iota \rightarrow \iota$  и т.д. Аналогичному терму —  $\lambda x.x$  — в представлении Карри соответствует целое множество типов  $\{\alpha \rightarrow \alpha \mid \alpha \in \tau\}$ .

Как уже упоминалось, не все термы бестипового исчисления являются термами простого типизированного, более того, в отличие от бестипового лямбда-исчисления, простое типизированное является *строго нормализуемым* (Strongly Normalized). Это означает, что каждый допустимый терм имеет нормальную форму, а процедура его нормализации всегда завершается. Так например, терм бестипового лямбда-исчисления  $\omega = (\lambda x.xx)(\lambda x.xx)$  не имеет нормальной формы и не является допустимым термом простого типизированного лямбда-исчисления. Исчерпывающее описание простого типизированного лямбда-исчисления и его свойств, а также и описание более богатых систем типов читатель может найти в [30, 31, 83].

### 1.3. Стратегии вычислений

Говорят, что терм находится в  *$\beta$ -нормальной форме* (далее — просто в *нормальной форме*), если он не содержит редексов, т.е. к нему не применима аксиома  $\beta$ -редукции. Два терма называются *равными* с точностью до  $\alpha$ -конверсии, если они имеют одну и ту же нормальную форму. Вычисление в лямбда-исчислении есть вычисление нормальной формы терма — *нормализация*. Конечно, ввиду Тьюринг-полноты  $\lambda$ -исчисления, не все термы имеют нормальную форму, например, вычисление терма, также известного как терм *омега* ( $\omega$ , омега-комбинатор,

Редукция аргументов	Редукция под абстракцией	
	Да	Нет
Да	(Сильная) нормальная форма $S ::= \lambda x.N \mid x N_1 \dots N_n$	Слабая нормальная форма $W ::= \lambda x.e \mid x W_1 \dots W_n$
Нет	Головная нормальная форма $H ::= \lambda x.H \mid x e_1 \dots e_n$	Слабая головная нормальная форма $E ::= \lambda x.e \mid x e_1 \dots e_n$

где  $\forall i \in \mathbb{N}$ ,  $e_i$  — произвольный лямбда-терм

Таблица 1: Нормальные формы

Omega-Combinator),  $\omega = \sigma\sigma = (\lambda x.x x) (\lambda x.x x)$  расходится. Важным свойством в контексте нормализации является *свойство Чёрча–Россера* (Church–Rosser), также известное как свойство ромба: если термы  $s$  и  $t$  получены из терма  $n$  посредством  $\beta$ -редукции,  $n \rightarrow_{\beta}^* s$  и  $n \rightarrow_{\beta}^* t$ , то существует такой терм  $m$ , что  $s \rightarrow_{\beta}^* m$  и  $t \rightarrow_{\beta}^* m$ . Прямым следствием свойства ромба является единственность нормальной формы терма, если таковая существует. Тем не менее, в аксиоме  $\beta$ -редукции ничего не сказано про порядок, в котором должна выполняться  $\beta$ -редукция терма. Далее в этой главе мы рассмотрим существующие *порядки* (стратегии, Reduction Strategies) *редукции* и их свойства.

Разнообразие стратегий вычислений в лямбда-исчислении обусловлено, в том числе, тем, что разные языки программирования создавались для различных целей и, соответственно, имеют разные свойства и реализации. Несмотря на то, что не все порядки редукции приводят терм к нормальной форме, они зачастую завершаются в некоторой “нормальной”<sup>6</sup> для этого порядка форме, которая с точки зрения конкретного порядка редукции считается *значением*, т.е. термом, который не редуцируется дальше, даже если содержит редексы. Основными свойствами стратегий редукции являются произведение редукции под абстракцией и редукция аргументов. В таблице 1<sup>7</sup> приведены “нормальные” формы, получаемые при всех комбинациях этих свойств, описанные в терминах контекстно-свободных грамматик.

<sup>6</sup> Данная форма не является нормальной формой в том смысле, что терм, находящийся в ней, может содержать редексы. Для того, чтобы такая терминология не вводила в заблуждение, нормальную форму часто называют *сильной* нормальной формой, а остальные “нормальные” формы имеют некоторый дополнительный префикс, описывающий “какая именно это нормальная форма”, например, головная нормальная форма (см. раздел 1.4 и таблицу 1).

<sup>7</sup> Таблица заимствована из статьи П. Сестофта (P. Sestoft) [86].

### 1.3.1. Слабые порядки редукции

Слабые порядки редукции характеризуются тем, что они рассматривают абстракцию как значение и, соответственно, не производят редукцию под абстракцией.

**Вызов по имени (Call-by-Name).** Стратегия вычисления по имени сначала вычисляет функцию до значения, лямбда-абстракции, после чего редуцирует её применение к аргументу путём подстановки тела аргумента вместо каждого вхождения переменной, по которой эта абстракция произведена. Это может привести к повторным вычислениям аргументов, если имеется несколько вхождений переменной, по которой производилась абстракция функции. Формально, на каждом шаге раскрывается самый левый самый внешний редекс, который не находится под лямбда-абстракцией. Таким образом, вызов по имени завершается в слабой головной нормальной форме.

**Вызов по значению (Call-by-Value)** в отличие от вызова по имени сначала производит редукцию аргументов и лишь потом применяет к ним функции. Формально, на каждом шаге раскрывается самый левый самый внутренний редекс, который не находится под лямбда-абстракцией. Таким образом, вызов по значению завершается в слабой нормальной форме.

**Вызов по необходимости (Call-by-Need)** является вызовом по имени<sup>8</sup>, в который добавлена  *мемоизация*. Это означает, что при вычислении аргумента его значение сохраняется и не вычисляется повторно при последующем использовании аргумента. В *чистых функциональных языках*, где нет побочных эффектов, результат вызова по имени и вызова по необходимости будет одинаковым, а значит, вызов по необходимости завершается в слабой головной нормальной форме. Одним из наиболее популярных языков, использующих вызов по необходимости, является язык **Haskell**.

---

<sup>8</sup>Иногда стратегию вызова по необходимости определяют как вызов по значению, вычисление аргумента в котором отложено до момента его реального использования. Не составляет труда показать, что это определение эквивалентно приведённому выше.

### 1.3.2. Сильные порядки редукции

**Нормальный порядок** отличается от стратегии вызова по имени тем, что он производит *сильную* редукцию, т.е. редукцию под абстракцией, а также редукцию аргументов. Первым на каждом шаге раскрывается самый левый самый внешний редекс. В отличие от других порядков редукции, нормальный порядок редукции является *нормализующим* (Normalizing), т.е. завершается в сильной нормальной форме терма тогда и только тогда, когда последняя существует. С другой стороны, как и вызов по имени, нормальный порядок редукции может проделывать одну и ту же работу несколько раз. Например,  $(\lambda x.xx)((\lambda y.y)a) \rightarrow ((\lambda y.y)a)((\lambda y.y)a) \rightarrow a((\lambda y.y)a) \rightarrow aa$  дважды редуцирует редекс  $(\lambda y.y)a$ .

**Апplikативный порядок.** В то время как нормальный порядок является сильной версией вызова по имени, аппликативный порядок является сильной версией вызова по значению. Иными словами, на каждом шаге первым раскрывается самый левый самый глубокий редекс. Например,  $(\lambda x.xx)((\lambda y.y)a) \rightarrow (\lambda x.xx)a \rightarrow aa$ . Аппликативный порядок, если завершается, приводит терм в сильную нормальную форму. Тем не менее, в отличие от нормального порядка редукции, аппликативный порядок может не завершаться даже в том случае, если терм имеет нормальную форму. Зачастую, это происходит тогда, когда какой-нибудь аргумент функции не имеет нормальной формы и при этом не используется в её теле. Например, терм  $(\lambda x.\lambda y.x)a((\lambda x.xx)(\lambda x.xx))$  имеет нормальную форму  $a$ , в то время как его аргумент  $((\lambda x.xx)(\lambda x.xx))$ , уже известный нам терм  $\omega$ , нормальной формы не имеет, и его вычисление расходится. Более того, аппликативный порядок редукции не способен редуцировать терм, являющийся применением рекурсивной функции к аргументу, даже если комбинатор рекурсии, используемый для определения рекурсивной функции, является специальным комбинатором для вызова по значению.

**Головная редукция (Head Reduction, Leftmost Head Reduction)** производит редукцию только головных редексов. Редекс называется *головным*, если его предками в абстрактном синтаксическом дерева терма являются лишь абстракции. Например, редекс  $(\lambda y.e_1)e_2$  является головным в терме  $\lambda x_1. \dots \lambda x_n. (\lambda y.e_1)e_2 \dots e_m$ . Завершается головная редукция в головной нормальной форме. Заметим, что по-

вторное применение головной редукции к аргументам получившейся головной нормальной формы нормализует терм. Более того, такая стратегия редукции является нормализующей.

Существуют и другие порядки редукции, зачастую являющиеся лишь вариациями и разнообразными комбинациями вышеописанных порядков редукции. Можно упомянуть такие порядки как *гибридный* (Hybrid) аппликативный порядок, гибридный нормальный порядок или *спинальная головная редукция* (Spine Head Reduction), и другие (см. [30, 86]).

## 1.4. Головная нормальная форма и дерево Бёма

Произвольный лямбда-терм  $M$  можно записать в следующей форме<sup>9</sup>:

$$M = \lambda \vec{v}. UV_1 \dots V_n, \text{ где } U = \begin{cases} y \\ (\lambda y.P)Q \end{cases} \quad (1)$$

В первом случае говорят, что терм  $M$  находится в *головной нормальной форме* (Head Normal Form), во втором случае  $(\lambda y.P)Q$  образует головной редекс. Заметим, что по определению терм находится в головной нормальной форме тогда и только тогда, когда он не содержит головных редексов. Важным следствием этого определения является то, что головная нормальная форма не обладает свойством единственности: один и тот же терм может иметь более одной головной нормальной формы — например, головной нормальной формой терма  $y((\lambda x.x)z)$  является как терм сам по себе, так и его нормальная форма  $yz$ . Среди множества головных нормальных форм терма выделяют *основную* головную нормальную форму (Principal Head Normal Form, phnf), которая получается посредством головной редукции терма, записанного в виде (1), если редукция завершается.

*Деревом Бёма* (Böhm Tree)  $BT(M)$  терма  $M$  называется дерево, которое строится по следующим правилам

<sup>9</sup>Утверждение может быть легко доказано индукцией по структуре терма  $M$ .

$$BT(I^a x \omega(Ix)) = \begin{array}{c} x \\ / \quad \backslash \\ \omega \quad x \end{array}$$

а) Конечное дерево

<sup>a</sup>Терм  $I = \lambda x.x$  является один из основных комбинаторов исчисления комбинаторов

$$BT(Y) = BT(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))) = \begin{array}{c} \lambda f.f \\ | \\ f \\ | \\ f \\ | \\ \dots \end{array}$$

б) Бесконечное дерево

Рисунок 6. Пример: деревья Бёма для термов, не имеющих нормальной формы

$$BT(M) := \begin{cases} \omega & , \text{ если } M \text{ не имеет hnf} \\ \begin{array}{c} \lambda \vec{x}.y \\ / \quad | \quad \backslash \\ BT(V_1) \quad \dots \quad BT(V_n) \end{array} & , \text{ если } \text{phnf}(M) = \lambda \vec{x}.yV_1 \dots V_n \end{cases}$$

Как уже отмечалось ранее, повторное применение головной редукции к аргументам головной нормальной формы нормализует терм. Таким образом, если терм имеет нормальную форму, то дерево Бёма, представляющее этот терм, вычислимо и конечно, и наоборот [30, 43, 87]. Также отметим, что если терм нормальной формы не имеет, соответствующее ему дерево Бёма либо конечно, а процедура построения его следующего уровня расходится (см. рис. 6а)), либо является бесконечным (см. рис. 6б)).

## 1.5. $\eta$ -длинная форма терма

$\eta$ -длинной форма ( $\eta$ -long Form) терма получает путём его полного  $\eta$ -расширения и заменой бинарного оператора применения на оператор *длинного применения*  $@_{long}$  (Long Application). Заметим, что каждая переменная, непосредственным предком которой не является абстракция, также подлежит  $\eta$ -расширению путём введения *анонимной* (Dummy) абстракции ( $x \mapsto \lambda.x$ ). Например,  $\eta$ -длинной формой терма  $\lambda x^{\iota \rightarrow \iota \rightarrow \iota}.x$  является терм  $\lambda x^{\iota \rightarrow \iota \rightarrow \iota}.a^{\iota}b^{\iota}.x(\lambda.b)(\lambda.a)$ . Заметим, что понятие  $\eta$ -длинной форм имеет смысл только в типизированном исчислении, т.к. в нём  $\eta$ -расширение ограничено хотя бы размером типа терма, в то время как в бестиповом случае  $\eta$ -расширять терм можно до бесконечности.



$\beta$ -нормальной  $\eta$ -длинной является форма терма, не содержащая  $\beta$ -редексов, но являющаяся  $\eta$ -длинной. Например,  $\beta$ -нормальной  $\eta$ -длинной формой терма  $(\lambda x^{\iota \rightarrow \iota}.x)$   $y$  является терм  $\lambda a^{\iota}.y(\lambda.a)$ ,  $\eta$ -редуцирующийся к терму  $y^{\iota \rightarrow \iota}$ , являющемуся, в свою очередь, нормальной формой исходного терма.

Заметим, что абстрактный синтаксис, используемый в определении  $\eta$ -длинной формы терма, отличается от стандартного абстрактного синтаксиса  $\lambda$ -исчисления, введённого в начале раздела 1.1. А именно, он описывается нижеследующей грамматикой.

$$\begin{aligned} \Lambda_{odd}^{lf} &::= \lambda x_1^{\tau_1} \dots x_p^{\tau_p}. \Lambda_{even}^{lf} \\ \Lambda_{even}^{lf} &::= x \Lambda_{1_{odd}}^{lf} \dots \Lambda_{n_{odd}}^{lf} \mid \Lambda_{0_{odd}}^{lf} @_{long} \Lambda_{1_{odd}}^{lf} \dots \Lambda_{m_{odd}}^{lf}, \text{ где } m \in \mathbb{N}, n, p \in \mathbb{N}_0 \end{aligned}$$

Иными словами, нечётные уровни в абстрактном синтаксическом дереве являются абстракциями по произвольному числу аргументов, а чётные — применением переменной к нулю или более аргументам, либо же применение другого  $\eta$ -длинного терма к одному или более аргументам. Последнее применение получило название оператора длинного применения  $@_{long}$ .

**Определение.** *Абстрактным синтаксическим деревом эта-длинной формы терма*, АСД<sup>10</sup>, называется  $\Sigma$ -помеченное дерево ( $\Sigma$ -Labelled Tree), где

$$\Sigma ::= \underbrace{\Lambda^{odd}}_{\text{лямбда-вершины}} \cup \underbrace{\Lambda^{even}}_{\text{другие вершины}} \equiv \underbrace{\{\lambda \vec{\xi} : \vec{\xi} \subseteq Var\}}_{\text{лямбда-вершины}} \cup \underbrace{Var \cup \{@_{long}^A : A \in Types\}}_{\text{другие вершины}}$$

является ранжированным алфавитом таким, что  $ar(x^A) := ar(A)$ ,  $ar(\lambda \vec{x}) := 1$ ,  $ar(@^A) := ar(A) + 1$ ,  $Var$  является бесконечным множеством переменных,  $Types ::= \{\tau : \tau = o \mid \tau \rightarrow \tau\}$  — множество простых типов,  $\vec{\xi}$  — вектор переменных,  $@_{long}^A$  — оператор длинного применения соответствующий типу  $A$ . По построению в дереве эта-длинной формы вершинами нечётных уровней дерева являются элементы множества  $\Lambda^{odd}$ , а элементами чётных уровней — элементы множества  $\Lambda^{even}$ . Так, например, на рис. 7 приведено абстрактное синтаксическое дерево  $\eta$ -длинной формы терма  $mul \vec{2} \vec{2}$ , которое задает умножение двух нумеролов Чёрча  $\vec{2} \equiv \lambda s.\lambda z.s@(s@z)$  в кодировке Чёрча.

<sup>10</sup>Под аббревиатурой АСД мы будем понимать, в зависимости от контекста, как абстрактное синтаксическое дерево  $\eta$ -длинной формы терма, так и его стандартное абстрактное синтаксическое дерево.

$$\begin{aligned}
mul \vec{2} \vec{2} &\equiv \lambda S. \lambda Z. (((\lambda s1. \lambda z1. s1 @ (s1 @ z1)) @ ((\lambda s2. \lambda z2. s2 @ (s2 @ z2)) @ S))) @ Z \\
&\equiv^{\eta\text{-long}} \lambda SZ. @_{long} (\lambda s1 z1. s1 (\lambda. s1 (\lambda. z1))) (\lambda q. @_{long} (\lambda s2 z2. s2 (\lambda. s2 (\lambda. z2)))) (\lambda q. S (\lambda P))) (\lambda. Z)
\end{aligned}$$

Рисунок 7. Терм  $mul \vec{2} \vec{2}$ , его  $\eta$ -длинная форма и её АСД

## 1.6. Головная линейная редукция

Как уже отмечалось ранее, лямбда-исчисление в первую очередь представляет собой модель вычислений. Соответственно, закономерным является вопрос о *сложности* (Complexity) вычислений в лямбда-исчислении. Иными словами, пусть дан некоторый терм и зафиксирована стратегия вычислений. Какова сложность вычислений (нормализации) данного терма? Простейшей мерой сложности вычисления является количество шагов  $\beta$ -редукции, необходимых для нормали-

Терм $T$	Условие	Список головных абстракций $\lambda_h(T)$	Список простых редексов $pr(T)$
$x$		$[\ ]$	$[\ ]$
$UV$	$\lambda_h(U) = [\ ]$	$[\ ]$	$pr(U)$
$UV$	$\lambda_h(U) = \lambda x : l$	$l$	$(\lambda x, V) : pr(U)$
$\lambda x.U$		$\lambda x : \lambda_h(U)$	$pr(U)$

Рисунок 8. Определение списков головных абстракций и простых редексов

зации терма. К сожалению, такой подход не соответствует сложности вычислений на реальных вычислителях, поскольку сложность операции подстановки находится в нелинейной зависимости от длины терма. Работы [13–15, 88–90] представляют различные подходы к описанию сложности вычислений в лямбда-исчислении и определению оптимальных стратегий редукции, основанных на той или иной мере сложности вычислений. Одним из таких подходов является *головная линейная редукция* (Head Linear Reduction) [8, 9], когда на каждом шаге вместо обычных подстановок производятся так называемые *линейные подстановки* (Linear Substitution) — подстановки только одного вхождения переменной в терм. В этом разделе мы рассмотрим классические определения головной линейной редукции, трассирующей нормализации, также обсудим связь между ними.

### 1.6.1. Головная линейная редукция: классическое определение

Для формального определения понятия головной линейной редукции нам понадобится ввести некоторое количество дополнительных определений. *Хребтными* или *спинальными подтермами* (Spine Sub-Terms) терма  $T$  называются сам терм  $T$ , а также все спинные подтермы терма  $U$ , если  $T = UV$  или  $T = \lambda x.U$  соответственно. Заметим, что любой лямбда-терм имеет ровно один спинальный подтерм-переменную. Такое вхождение переменной, так же как и сам подтерм, называется *головным* (Head Occurrence, hoc). Ещё два понятия: *список головных абстракций* (Head  $\lambda$  List,  $\lambda_h(T)$ ) и *простые редексы* (Prime Redexes) — определяются индукцией по структуре терма  $T$ . Простой редекс представляет собой пару  $(\lambda x, N)$ , первый и второй элементы которой называются *абстракцией* и *аргументом простого редекса* соответственно. Индуктивное определение списков простых редексов и головных абстракций приведены на рис. 8, где  $pr(T)$  обозначает список простых редексов терма  $T$ .

*Редексом головного вхождения переменной* (hoc Redex) терма  $T$  называется простой редекс  $(\lambda x, V)$ , где  $x$  является головной переменной, если такой редекс существует.

Головная линейная редукция линеаризует последовательность подстановок путём замены на каждом шаге лишь одного вхождения переменной — головного, оставляя сам головной редекс нетронутым. Если же он не определён, то головная линейная редукция завершается в так называемой *псевдоголовной нормальной форме* (Quasi-Head-Normal Form, qhn).

Пусть  $r = (\lambda x, \dots)$  и  $s = (\lambda y, \dots)$  — два простых редекса терма  $T$ . Говорят, что некоторый простой редекс  $r$  *содержит* другой простой редекс  $s$ , если  $\lambda y$  является вершиной поддерева, непосредственным предком которого является вершина  $\lambda x$ . Более того, простые редексы  $r$  и  $s$  называются последовательными, если  $r$  содержит  $s$ , и не существует такого простого редекса  $t$ , который содержится в  $r$ , но не содержится в  $s$ .

**Пример.** Пусть  $T = \lambda s.(\lambda x.(\lambda y.(\lambda w.w@s)@y)@x)@(\lambda z.z)$ , тогда

$$\begin{cases} \lambda_h(T) = [\lambda s] \\ pr(T) = [(\lambda x, (\lambda z.z)), (\lambda y, x), (\lambda z, y)] \rightleftharpoons [r, s, t] \end{cases}$$

Пары простых редексов  $r$  и  $s$ , и  $s$  и  $t$  образуют последовательные простые редексы.

**Теорема 1.** Пусть  $N$  — произвольный терм, а терм  $M$  является результатом применения к нему головной линейной редукции. Тогда:

1.  $M \equiv_{\beta} N$ ;
2. Головная линейная редукция завершается тогда и только тогда, когда завершается головная редукция.

Первое утверждение может быть показано индукцией по числу шагов головной линейной редукции.

Что же касается второго утверждения, то его необходимость устанавливается в силу того, что головная редукция терма  $N$  завершится за число шагов, равное количеству простых редексов терма  $M$ . Действительно, никакая  $\beta$ -редукция терма  $M$  не способна произвести подстановку головного вхождения переменной, а

значит и не способна создать новый простой редекс. Таким образом, терм имеет головную нормальную норму, что в свою очередь гарантирует завершаемость головной редукции [30].

Тем не менее, доказать достаточность этого утверждения не так просто. Для этого мы введём формальное определение головной линейной редукции в виде системы переходов, с помощью которой формально и докажем утверждение теоремы (см. раздел 2.1).

## 1.6.2. Трассирующая нормализация

В 70-ых годах 20 века Г. Плоткин [5] сформулировал проблему построения *полностью абстрактной модели вычислений* (Fully Abstract Model of Computations) для языка программирования вычислимых функций PCF (Programming Computable Functions), заключающуюся в построении абстрактной модели языка, являющейся одновременно *полной* (Complete) — каждый терм языка представим в его модели — и *согласованной* (Sound) — каждый элемент абстрактной модели имеет прообраз в языке. Эта проблема была решена в начале 90-ых годов независимо двумя группами исследователей: М. Хуландом, Л. Онгом [39] и С. Абрамски, Г. МакКаскером [40, 41]. Это было сделано посредством использования *игровой семантики* (Game Semantics). Последняя является одним из способов задания формальной семантики языков программирования и рассматривает вычисления как *игру* (Game) между *игроками*: *пропонентом* (-ами) (Proponent, Player) и *оппонентом*(-ами) (Opponent) — программой и её окружением. Семантикой программы в данном случае является *стратегия* (Strategy), которой пропонент должен придерживаться. Подробнее с игровой семантикой читатель может ознакомиться в следующих работах [39, 41, 42, 44, 45, 91]. В 2015 году Л. Онг заметил, что из игровой семантики программ для простого типизированного лямбда-исчисления естественным образом вытекает нестандартная процедура нормализации [6]. Отличительной особенностью данной процедуры является то, что вместо изменения терма посредством  $\beta$ -редукции, используя стандартные приёмы, такие как окружения или замыкания<sup>11</sup>, *трассирующая нормализация*

<sup>11</sup>*Замыканием* (Closure) называется функция первого порядка, определяющая значения переменных, определённых вне тела функции; в случае лямбда-исчисления эти переменные являются свободными. Использование замыканий является одним из стандартных способов реализации функциональных языков программирования.

ция (Traversal-Based Normalization, Normalization by Traversals, [6, 11, 12]) оставляет входной терм нетронутым, производя его нормализацию путём обхода его абстрактного синтаксического дерева терма, запоминая историю этого обхода.

Трассирующая нормализация в стиле Онга, ONP— оксфордская процедура нормализации (Oxford Normalization Procedure, ONP), определена для термов простого типизированного лямбда-исчисления, находящихся в  $\eta$ -длинной форме [6]. Заметим, что поскольку рассматриваемое исчисление является простым типизированным, каждый терм имеет уникальную  $\eta$ -длинную форму, а процедура его нормализации, предложенная Онгом, всегда завершается.

**Определение.** *Обоснованной последовательностью* (Justified Sequence) называется последовательность, каждый элемент которой может быть *обоснован* (Sustified), т.е. снабжён указателем на ранний элемент последовательности (Backpointer).

**Определение.** *Обходом* (Traversal) называется обоснованная последовательность вершин абстрактного синтаксического дерева  $\eta$ -длинной формы терма.

Каждому дереву некоторого корректного просто типизированного терма соответствует некоторое непустое множество обходов  $\mathcal{T}_{\text{trav}}$ , каждый из которых соответствует пути в абстрактном синтаксическом дереве  $\eta$ -длинной  $\beta$ -нормальной формы исходного терма. Таким образом, ONP строит всё множество обходов данного терма, которое и определяет  $\eta$ -длинную  $\beta$ -нормальную форму терма, сводимую к нормальной форме путём  $\eta$ -редукции, которая, в свою очередь, всегда завершается.

Такой подход к нормализации термов оказался согласован с головной линейной редукцией. Заметим, что важной составляющей частью трассирующей нормализации в том виде, в котором она была предложена Онгом, является преобразование терма в  $\eta$ -длинную форму, которое, в свою очередь, существенно опирается на типы термов и не может быть непосредственно распространено до нетипизированного лямбда-исчисления. Последнее утверждение справедливо в силу того, что ввиду отсутствия типизации, неконтролируемое  $\eta$ -расширение терма, очевидно, расходится. Тем не менее, последние работы Блюма показывают, что трассирующая нормализация в стиле Онга может быть распространена до нетипизированного лямбда-исчисления посредством *эта-расширения на лету* (On-the-Fly Eta-Expansion). Последнее вместо построения эта-длинной формы терма и даль-

нейшей её нормализации производит эта-расширение терма по ходу самой нормализации тогда и только тогда, когда это необходимо [92].

## 1.7. Системы переходов

Одним из подходов к изучению поведения дискретных систем являются *системы переходов* (Transition System, [57]) — суть четвёрка  $(S, I, F, \rightarrow)$ , где  $S$  — множество состояний системы,  $I \subseteq S$  — множество исходных состояний системы,  $F \subseteq S$  — множество конечных состояний системы,  $\rightarrow \subseteq S \times S$  — отношение, означающее дискретный переход системы из одного состояния в другое, обозначаемое  $(s_1, s_2) \in \rightarrow$  или, для удобства чтения,  $s_1 \rightarrow s_2$ , где  $s_1, s_2 \in S$ .

Система переходов называется *детерминированной*, если её текущее состояние однозначно определяет последующее, в противном случае, она называется *недетерминированной*.

Например, алгоритм Евклида может быть записан нижеследующей системой переходов.

$$S = \mathbb{N} \times \mathbb{N}$$

$$I = S$$

$$F = \{(n, n) \mid n \in \mathbb{N}\}$$

$$\rightarrow = \{((m, n), (m - n, n)) \mid m > n\} \cup \{((m, n), (m, n - m)) \mid m < n\}$$

## 1.8. Выводы

На основе проделанного обзора сделаны следующие выводы.

1. Существующие алгоритмы трассирующей нормализации определены лишь для некоторых неполных по Тьюрингу исчислений. При этом ориентированность этих алгоритмов на системы типов и статические ограничения на структуру исходного терма осложняют их обобщение для поддержки исчислений, полных по Тьюрингу.
2. Задача построения алгоритмов трассирующей нормализации для нетипизированного лямбда-исчисления для стратегий вычислений нормального и ап-

пликативного порядков, а также стратегии вызова по необходимости, является актуальной задачей в области анализа языков программирования.

3. Алгоритмы трассирующей нормализации требуют формального операционного доказательства корректности.



## Глава 2

# Полная головная линейная редукция

В главе представлена модель головной линейной редукции (см. раздел 2.1), согласованная с известной оригинальной моделью, предложенной В. Даносом и Л. Ренье [8], а также предлагается модель *полной головной линейной редукции* (см. раздел 2.2), являющаяся расширением этой модели. Приводится формальное представление обеих моделей в виде систем переходов (см. рисунки 9 и 13), причём системы переходов для головной линейной редукции является частным случаем системы переходов для полной головной линейной редукции. Наконец, приводится доказательство корректности обеих моделей, а именно устанавливается их согласованность с головной редукцией лямбда-термов (см. теоремы 2 и 3).

### 2.1. Модель головной линейной редукции

Ниже приведена формальная модель головной линейной редукции в виде системы переходов, а также установлена корректность предложенной модели относительно модели головной редукции.

#### 2.1.1. Головная линейная редукция как система переходов

Для представления головной линейной редукции в виду системы переходов (см. раздел 1.7) мы определим каждую из компонент этой системы (множество

состояний, начальное состояние, множество конечных состояний, а также правила переходов) по отдельности.

**Множество состояний системы переходов.** Состоянием системы переходов для головной линейной редукции является тройка  $\langle A[\underline{B}]; \Gamma; \Delta \rangle$ , где:

- $A[\underline{B}]$  — это  $\lambda$ -терм, в котором вершина, являющаяся корнем поддерева  $B$ , выделена<sup>12</sup>;
- $\Gamma := (var \mapsto (t, \Gamma_1)) : \Gamma \mid [ ]$  является *окружением*, иными словами, списком связываний переменных, где  $var$  — переменная,  $t$  —  $\lambda$ -терм,  $\Gamma_1$  — сохранённое окружение, соответствующее терму  $t$ ;
- $\Delta := (t, \Gamma) : \Delta \mid [ ]$  является стеком *висячих аргументов* и представляет собой список пар вида  $(t, \Gamma)$ , где  $t$  —  $\lambda$ -терм, а  $\Gamma$  — соответствующее ему окружение.

*Начальным* является состояние  $\langle A[\underline{A}]; [ ]; [ ] \rangle$ , где первое вхождение  $[ ]$  обозначает пустое окружение, второе — пустой стек висячих аргументов,  $A[\underline{A}]$  обозначает входной терм с выделенным корнем.

*Конечным* является состояние вида  $\langle A[\underline{x}]; \Gamma; \Delta \rangle$ , где  $A[\underline{x}]$  — исходный  $\lambda$ -терм с выделенной переменной  $x$ ,  $x \notin \Gamma$ .

**Правила переходов.** Формальное определение правил переходов приведено на рис. 9. Ниже дано описание каждого из этих правил, а также обозначений, используемых в определении на рис. 9.

- Если выделенной вершиной в текущем состоянии является применение, (правило [App]), то выделенной вершиной следующего состояния системы переходов является левый аргумент этого применения, а в стек висячих аргументов помещается аргумент с текущим окружением. В дальнейшем этот аргумент может быть использован для формирования простого редекса.

<sup>12</sup> Выделение является синтаксической пометкой вершины абстрактного синтаксического дерева терма и обозначается подчёркиванием в примерах и правилах.

## Обозначения

$A[\underline{B}]$	$\lambda$ -терм с выделенным поддеревом $B$
$\Gamma := (var \mapsto (t, \Gamma_1)) : \Gamma$	окружение (список)
$\Delta := (t, \Gamma) : \Delta$	стек висячих аргументов
$[]$	пустой список либо пустой стек

Состояние	Начальное состояние	Конечное состояние
$\langle A[\underline{B}]; \Gamma; \Delta \rangle$	$\langle A[\underline{A}]; []; [] \rangle$	$\langle A[\underline{x}]; \Gamma; \Delta \rangle$

## Правила переходов

$\langle A [e_1 @ e_2]; \Gamma; \Delta \rangle$	$\rightarrow \langle A [\underline{e_1} @ e_2]; \Gamma; (e_2, \Gamma) : \Delta \rangle$	[App]
$\langle A [\underline{\lambda x. e}]; \Gamma; [] \rangle$	$\rightarrow \langle A [\lambda x. \underline{e}]; \Gamma; [] \rangle$	[Lam-Non-Elim]
$\langle A [\underline{\lambda x. e}]; \Gamma; (B, \Gamma') : \Delta \rangle$	$\rightarrow \langle A_{\cancel{x}} [\cancel{\lambda x. e}]; (x \mapsto (B, \Gamma')) : \Gamma; \Delta \rangle$	[Lam-Elim]
$\langle A [\underline{x}]; [\dots, (x \mapsto (B, \Gamma')), \dots]; \Delta \rangle$	$\rightarrow \langle A [\underline{B}]; \Gamma'; \Delta \rangle$	[BVar]

Рисунок 9. Система переходов для головной линейной редукции

- Если выделенной вершиной является абстракция, то следующей выделенной вершиной становится корень подтерма, представляющего тело этой абстракции. Если стек висячих аргументов пуст, то переменная абстракции не связывается ни с каким аргументом (правило [Lam-Non-Elim]). Если же стек висячих аргументов не пуст, эта абстракция формирует простой редекс с его вершиной (правило [Lam-Elim]). В последнем случае, абстракция и соответствующие применение и его аргумент вычёркиваются<sup>13</sup> из текущего дерева (первого компонента текущего состояния системы переходов), а простой редекс сохраняется в текущем окружении. Очевидно, что правила напрямую согласованы с определением простого редекса.
- Если же выделенной вершиной текущего состояния является переменная, то, если она является свободной, система переходов достигла своего конечного состояния, ежели переменная является связанной, то либо существует простой редекс, аргумент которого может быть подставлен вместо вхождения этой головной переменной (правило [BVar]), либо такого редекса нет, и опять же система переходов достигла своего конечного состояния.

Заметим, что система переходов является детерминированной и синтаксически–управляемой, а выбор правила зависит лишь от первого эле-

<sup>13</sup>Вычёркивание является синтаксической пометкой вершин в дереве, а не полным удалением подтерма. С этого момента мы будем использовать обозначение  $A_{\cancel{x}}[\cancel{\lambda x. e}]$  для терма  $A$ , у которого подтерм  $B$ , вершина “ $\lambda x$ ” и соответствующая вершина оператора применения помечены (вычеркнуты).

$$\langle (\lambda x . x) @ (\lambda y . y); []; [] \rangle \rightarrow^{[App]} \quad (2.2)$$

$$\langle (\lambda x . x) @ (\lambda y . y); []; [((\lambda y . y), [])] \rangle \rightarrow^{[Lam-Elim]} \quad (2.3)$$

$$\langle (\lambda x . x) @ (\lambda y . y); [x \mapsto ((\lambda y . y), [])]; [] \rangle \rightarrow^{[BVar]} \quad (2.4)$$

$$\langle (\lambda x . \lambda y . y) @ (\lambda y . y); []; [] \rangle \rightarrow^{[Lam-Non-Elim]} \quad (2.5)$$

$$\langle (\lambda x . \lambda y . y) @ (\lambda y . y); []; [] \rangle \not\rightarrow \quad (2.6)$$

Рисунок 10. Поведение системы переходов для головной линейной редукции на примере термина  $(\lambda x . x) @ (\lambda y . y)$

мента состояния (формально, выбор между правилами [Lam-\*] зависит от текущего состояния стека висячих аргументов, но это состояние также зависит от входного термина). Единственным случаем, когда система переходов может достичь своего конечного состояния, является случай выделенной переменной, которая либо свободна, либо не связана никаким простым редексом. Более того, правила гарантируют, что выделенной может быть лишь вершина, находящаяся на самом левом пути абстрактного синтаксического дерева термина. Иными словами, если система переходов достигает конечного состояния, вычисления завершаются, головное вхождение переменной (нос-переменная) оказывается выделенным, а сама переменная не связана никаким простым редексом.

Пример поведения системы переходов для головной линейной редукции на терме  $(\lambda x . x) @ (\lambda y . y)$  приведён на рисунке 10. Как и ожидалось, первым элементом конечного состояния (игнорируя вычеркивание вершин) является псевдо-головная нормальная форма входного термина, а именно терм  $(\lambda x . \lambda y . y) @ (\lambda y . y)$ . Головной же нормальной формой рассматриваемого термина является терм  $\lambda y . y$ , получаемый в данном случае простым “выбрасыванием” вычеркнутых вершин из первого элемента конечного состояния системы переходов. В общем случае нам потребуется определить специальную функцию *расширения expr*, которая будет возвращать соответствующий терм из конечного состояния, а также необходимо будет установить согласованность этой функции с головной редукцией (см. раздел 2.1.2). ■

## 2.1.2. Согласованность головной и головной линейной стратегий редукции

В данном разделе приводится доказательство согласованности головной и головной линейной стратегий редукции. Сначала определяется дополнительная функция *расширения* — *exp*.

**Функция расширения.** По данному состоянию системы переходов функция *exp* возвращает  $\lambda$ -терм. Эта функция производит последовательную редукцию всех простых редексов, накопившихся на данный момент времени. Поскольку эта последовательность совпадает с последовательностью головных редексов, головная и линейная головная редукции оказываются согласованы. Формально это выражается следующим образом<sup>14</sup>:

$$\text{exp} \langle M[\underline{A}]; \Gamma \bullet [(x \mapsto (B, \Gamma'))]; \Delta \rangle = \text{exp} \langle M[\underline{A}[x/B[\Gamma']]]; \Gamma; \Delta \rangle \quad (2.7)$$

$$\text{exp} \langle M_B[\underline{A}]; []; (B, \Gamma') : \Delta \rangle = \text{exp} \langle M_{B[\Gamma']}[\underline{A}]; []; \Delta \rangle \quad (2.8)$$

$$\text{exp} \langle M; []; [] \rangle = M', \quad (2.9)$$

где  $B[\Gamma'] = \text{exp} \langle \underline{B}; \Gamma'; [] \rangle$ , а терм  $M'$  получается из терма  $M$  удалением всех вычеркнутых вершин.

Согласно определению, сначала функция *exp* производит последовательность подстановок согласно информации, накопленной в текущем окружении  $\Gamma$  (см. определение (2.7)). А именно, функция расширения производит подстановку терма  $B[\Gamma']$  вместо всех вхождений переменной  $x$  в такое поддерево терма  $M$ , которое имеет корнем выделенную вершину (т.е. поддерево  $A$ ). Далее функция *exp* производит последовательность подстановок в аргументах из стека висячих аргументов  $\Delta$ , согласно сохранённым вместе с ними окружениям. А именно, каждый рекурсивный вызов, соответствующий определению (2.8), производит подстановку всех переменных в соответствии с контекстом  $\Gamma'$  исключительно в висячем аргументе  $B$ . Напомним, мы придерживаемся соглашения Барендрегта (см. главу 1), т.е. предполагается, что все переменные имеют уникальные имена. Следование

<sup>14</sup> $\alpha \bullet \beta$  означает конкатенацию контейнеров  $\alpha$  и  $\beta$ .

этой концепции в данном случае<sup>15</sup> позволяет избежать *захвата имён* переменных при подстановке (см. главу 1). Далее мы будем называть *расширением* состояния системы переходов результат применения функции  $exp$  к этому состоянию.

Отметим некоторые важные свойства системы переходов головной линейной редукции.

1. По определению функции  $exp$  (уравнения (2.7) и (2.8)), единственное правило системы переходов, при применении которого расширения получаемого состояния не равны расширению исходного состояния, из которого оно получено, — это [Lam-Elim]; все остальные правила оставляют расширение неизменным.
2. Количество переходов системы без применения правила [Lam-Elim] конечно. Данный факт является прямым следствием определения контекста, конечности размера входного терма и того, что в силу определения только правило [Lam-Elim] может изменить контекст. С этого момента мы будем обозначать последовательное применение правил без применения правила [Lam-Elim] через  $\rightarrow^*$ .

**Пример.** Ниже показан результат применения функции  $exp$  к каждому состоянию системы переходов для примера, приведённого на рисунке 10.

$$exp((2.2)) = exp(\langle (\lambda x . x) @ (\lambda y . y); [], [] \rangle) = (\lambda x . x) @ (\lambda y . y)$$

$$exp((2.3)) = exp(\langle (\lambda x . x) @ (\lambda y . y); [], [(\lambda y . y), []] \rangle)$$

$$= exp(\langle (\lambda x . x) @ (\lambda y . y); [], [] \rangle)$$

$$= (\lambda x . x) @ (\lambda y . y)$$

$$exp((2.4)) = exp(\langle (\cancel{\lambda x} . x) @ (\cancel{\lambda y} . y); [(x \mapsto ((\lambda y . y), []))]; [] \rangle)$$

$$= exp(\langle (\cancel{\lambda x} . \lambda y . y) @ (\cancel{\lambda y} . y); [], [] \rangle) = \lambda y . y$$

$$exp((2.5)) = exp(\langle (\cancel{\lambda x} . \lambda y . y) @ (\cancel{\lambda y} . y); [], [] \rangle) = \lambda y . y$$

$$exp((2.6)) = exp(\langle (\cancel{\lambda x} . \lambda y . y) @ (\cancel{\lambda y} . y); [], [] \rangle) = \lambda y . y$$

<sup>15</sup>Заметим, что следование соглашению Барендрегта не избавляет от неприятностей, связанных с переименованием переменных, при подстановке в общем случае. Тем не менее, при использовании окружений эта проблема откладывается до момента конструирования результата вычислений (Read-Out Function). Для краткости изложения, мы будем игнорировать эту проблему. Заметим, что она может быть решена с помощью отдельного обхода по АСД результата вычислений, генерации новых имён и переименовании переменных на лету.



Идея доказательства согласованности головной и линейной головной стратегий редукции состоит в следующем. Расширение не может быть изменено никаким правилом, кроме [Lam-Elim], следовательно, после применения функции расширения к состояниям системы переходов каждое применение правила [Lam-Elim] соответствует одному шагу головной редукции. Так, в примере выше  $(\lambda x . x) @ (\lambda y . y)$  соответствует первому шагу системы переходов, а  $\lambda y . y$  соответствует трём оставшимся шагам.

**Теорема 2.** Пусть  $\langle \dots \rangle$  — некоторое состояние системы переходов, расширение которого обозначено троеточием  $\dots$  (см. иллюстрацию к теореме, рисунок 11), такое, что на следующем шаге должно быть применено правило [Lam-Elim]. При этом результатом его применения является состояние  $\langle M_i; \Gamma_i; \Delta_i \rangle$ , расширением которого является некоторый терм  $M'_i$ . Далее система переходов делает некоторое конечное число шагов до следующего применения правила [Lam-Elim]. Обозначим  $\langle M_i; \Gamma_i; \Delta_i \rangle$  состояние, расширением которого является терм  $M'_i$ . Тогда если может быть применено правило [Lam-Elim], расширяющее окружение  $\Gamma$  новым связыванием  $(x \mapsto (B, \Gamma'))$ , и расширением результата применения которого является некоторый терм  $M'_{i+1}$ , то терм  $M'_{i+1}$  получается из терма  $M_i$  за одним шаг головной редукции.

*Доказательство.* Теорема доказывается индукцией по количеству применений правила [Lam-Elim].

База индукции. Первый элемент, добавляемый в  $\Gamma$ , является головным редексом по определению, поэтому база индукции очевидна.

Индукционный переход. Согласно индукционному предположению, после  $i$ -го применения правила [Lam-Elim], текущим состоянием будет  $\langle M_i; \Gamma_i; \Delta_i \rangle$ , расширением которого является некоторый терм  $M'_i$ . Как уже отмечалось ранее,  $\xrightarrow{*}$  не изменяет расширения и число шагов  $\xrightarrow{*}$  конечно. Таким образом, существуют следующие возможные случаи.

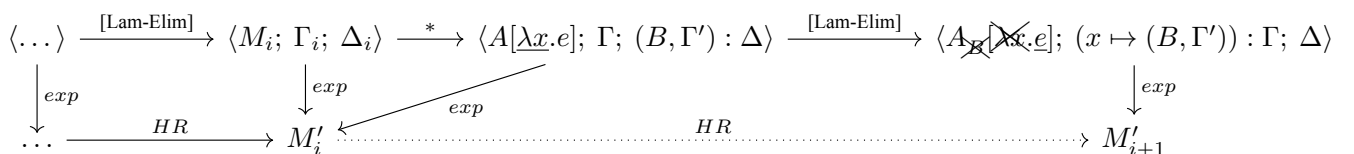


Рисунок 11. Иллюстрация к теореме 2

1. Система переходов достигла конечного состояния, и доказывать нечего.
2. Система не достигла конечного состояния, и может быть применено правило [Lam-Elim]. В данном случае, согласно определению правила [Lam-Elim], мы знаем вид исходного состояния:  $\langle A[\underline{\lambda x}.e]; \Gamma; (B, \Gamma') : \Delta \rangle$ . Обозначим это состояние как (i), а результат применения к нему правила [Lam-Elim] обозначим как  $\langle A_{\cancel{x}}[\cancel{\lambda x}.e]; (x \mapsto (B, \Gamma')) : \Gamma; \Delta \rangle$  или сокращенно (ii). Нам надо показать, что результат применения функции  $exp$  к состоянию (ii) совпадает с результатом одного шага головной редукции терма  $M'_i$ , т.е. с термом  $M'_{i+1}$ . Применим функцию  $exp$  к состоянию (i):

$$exp \langle A[\underline{\lambda x}.e]; \Gamma; (B, \Gamma') : \Delta \rangle \quad (2.10)$$

$$\xrightarrow{* exp} exp \langle A[\underline{\lambda x}.e[\Gamma]]; []; (B, \Gamma') : \Delta \rangle \quad \text{по (2.7)} \quad (2.11)$$

$$\xrightarrow{exp} exp \langle A_{B[\Gamma']}[\underline{\lambda x}.e[\Gamma]]; []; \Delta \rangle \quad \text{за один шаг (2.8)} \quad (2.12)$$

$$\xrightarrow{exp} \dots \quad (2.13)$$

Теперь применим функцию  $exp$  к состоянию (ii):

$$exp \langle A_{\cancel{x}}[\cancel{\lambda x}.e]; (x \mapsto (B, \Gamma')) : \Gamma; \Delta \rangle \quad (2.14)$$

$$\xrightarrow{* exp} exp \langle A_{\cancel{x}}[\cancel{\lambda x}.e[\Gamma]]; [(x \mapsto (B, \Gamma'))]; \Delta \rangle \quad \text{согласно (2.7)} \quad (2.15)$$

$$\xrightarrow{exp} exp \langle A_{\cancel{x}}[\cancel{\lambda x}.e[\Gamma]][x/B[\Gamma']]; []; \Delta \rangle \quad \text{за шаг (2.7)} \quad (2.16)$$

$$\xrightarrow{exp} \dots \quad (2.17)$$

Легко заметить, что терм, являющийся первым компонентом состояния (2.12),  $A_{B[\Gamma']}[\underline{\lambda x}.e[\Gamma]]$ , имеет головной редекс  $(\lambda x, B)$ , поскольку контекст уже является пустым. Таким образом, если применить шаг головной редукции к данному терму, то результатом будет терм  $A_{\cancel{x}}[\cancel{\lambda x}.e[\Gamma]][x/B[\Gamma']]$ . По определению головной редукции этот терм соответствует первой компоненте состояния (2.16). Согласно определениям (2.7) и (2.8), если продолжить применение функции  $exp$  к состояниям (2.12) и (2.16), то в обоих случаях один и тот же стек висячих аргументов  $\Delta$  не произведёт никаких изменений ни в аргументе  $B$ , ни в выделенном поддереве  $e$ . Следовательно, редекс



$(\lambda x, B)$  является головным для терма  $M'_i$ , а значит, расширением состояния  $(ii)$  является терм  $M'_{i+1}$ . ■

Теорема 2 сопоставляет каждому шагу головной редукции шаг системы переходов, соответствующий применению правила [Lam-Elim]. Таким образом, прямым следствием теоремы является тот факт, что *головная линейная редукция завершается тогда и только тогда, когда завершается головная редукция*.

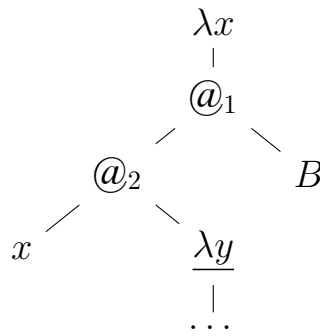
Более того, поскольку функция расширения не способна изменить путь от корня дерева терма до корня его выделенного поддерева, первая компонента конечного состояния системы переходов для головной линейной редукции содержит такой терм, что самый левый путь в представляющем его дереве является частью головной нормальной формы, а значит, расширение конечного состояния является вершиной дерева Бёма. Эти рассуждения являются неформальным обоснованием того, что рекурсивное применение головной линейной редукции ко всем оставшимся аргументам, т.е. полная головная линейная редукция (см. раздел 2.2), нормализует терм. Теорема 3 является формальным доказательством это утверждения.

## 2.2. Модель полной головной линейной редукции

*Полная головная линейная редукция* (Complete Head Linear Reduction, CHLR) является расширением головной линейной редукции, рекурсивно применяющим последнюю к аргументам при достижении конечного состояния. В данном разделе приводится разработанное автором формальное определение полной головной линейной редукции в виде системы переходов, являющийся расширением системы переходов для головной линейной редукции, приведённой в разделе 2.1.

### 2.2.1. Система переходов для полной головной линейной редукции

Система переходов для CHLR является расширением системы переходов для HLR тремя новыми правилами [FVar-\*]. Эти правила предназначены для обра-

Рисунок 12. Терм  $M[\lambda y]$ 

ботки ситуации, когда система переходов для HLR достигла своего конечного состояния. А именно, если система переходов для HLR достигла своего конечного состояния, то его первой компонентой является некоторый терм, выделенным поддеревом которого является вхождение головной переменной, которая либо является свободной, либо динамически несвязанной ни с каким аргументом. Под *динамически несвязанная с аргументом* переменной понимается такая связанная переменная, чья абстракция не применена ни к какому аргументу. Другими словами, абстракция *динамически несвязанной с аргументом* связанной переменной не участвует в формировании какого-либо простого редекса. Итак, система переходов для полной головной линейной редукции обладает нижеследующими изменениями по сравнению с системой переходов для головной линейной редукции.

- Стек висячих аргументов  $\Delta$  расширяется специальным символом  $\$$ . Этот символ играет роль *разделителя*, запрещающего связывать абстракцию с аргументом, если путь от этого аргумента до соответствующей абстракции не является левым путём. Мы будем именовать символ  $\$$  *символом разделителем*. Символ разделитель гарантирует, что все редексы в текущем контексте (окружении  $\Gamma$ ) являются простыми для данного подтерма. Например, пусть дан некоторый входной терм, и в ходе полной головной линейной редукции получено состояние  $\langle M[\lambda y]; \Gamma; [ \$, (B, \Gamma_0) ] \rangle$ , где терм  $M[\lambda y]$  изображён на рисунке 12. В данном примере разделитель  $\$$  запрещает связывание вершины  $\lambda y$  с висячим аргументом  $B$ , поскольку они не образуют простого редекса.
- Конечное состояния системы переходов также претерпевает изменения: теперь стек висячих аргументов в нём должен быть пустым. *Конечным* стано-

вится состояние следующего вида:  $\langle M[x]; \Gamma; [] \rangle$ , где  $x \notin \text{dom}(\Gamma)$ . Начальное же состояние совпадает с начальным состоянием системы переходов головной линейной редукции  $\langle \lambda_1; []; [] \rangle$ , где  $\lambda_1$  является входным термом с выделенным корнем.

- Результатом CHLR является терм в нормальной форме. В терминах системы переходов это означает, что он может быть получен из первой компоненты конечного состояния системы переходов путём удаления из неё всех вычеркнутых вершин.
- На рисунке 13 приведены правила для переходов системы переходов для CHLR. Для удобства чтения новые/измененные правила по сравнению с системой переходов для HLR выделены с помощью прямоугольников: выделение.

## 2.2.2. Корректность модели полной головной линейной редукции

Как и в случае HLR, определим функцию расширения состояний системы переходов  $exp$  до соответствующего лямбда-терма. По сравнению с функцией расширения, определённой для системы переходов для HLR, новая функция расширения распространяется на ещё один случай — (2.20). А именно, этот случай обрабатывает ситуацию, при которой на вершине стека висячих аргументов оказался символ разделитель.

$$exp \langle M[\underline{A}]; \Gamma \bullet (x \mapsto (B, \Gamma')); \Delta \rangle = exp \langle M[\underline{A}[x/B[\Gamma']]]; \Gamma; \Delta \rangle \quad (2.18)$$

$$exp \langle M_B[\underline{A}]; []; (B, \Gamma') : \Delta \rangle = exp \langle M_{B[\Gamma']}[\underline{A}]; []; \Delta \rangle \quad (2.19)$$

$$\boxed{exp \langle M_B[\underline{A}]; []; \$ : \Delta \rangle} = \boxed{exp \langle M_B[\underline{A}]; []; \Delta \rangle} \quad (2.20)$$

$$exp \langle M; []; [] \rangle = M' \quad (2.21)$$

Здесь  $B[\Gamma'] = exp \langle \underline{B}; \Gamma'; [] \rangle$ , а  $M'$  получается из  $M$  удалением вычеркнутых вершин.

Обозначения	
$A[\underline{B}]$	$\lambda$ -терм с выделенным поддеревом $B$
$\Gamma := (var \mapsto (t, \Gamma_1)) : \Gamma$	окружение (список
$\Delta := (t, \Gamma) : \Delta$	стек висячих аргументов
$[]$	пустой список либо пустой стек
$\$$	символ-разделитель

Состояние	Начальное состояние	Конечное состояние
$\langle A[\underline{B}]; \Gamma; \Delta \rangle$	$\langle A[\underline{A}]; []; [] \rangle$	$\langle A[\underline{x}]; \Gamma; [] \rangle$

Правила переходов		
$\langle A[e_1 @ e_2]; \Gamma; \Delta \rangle$	$\rightarrow \langle A[e_1 @ e_2]; \Gamma; (e_2, \Gamma) : \Delta \rangle$	[App]
$\langle A[\lambda x.e]; \Gamma; \boxed{\$ : \Delta} \rangle$	$\rightarrow \langle A[\lambda x.e]; \Gamma; \boxed{\$ : \Delta} \rangle$	[Lam-Non-Elim]
$\langle A[\lambda x.e]; \Gamma; (B, \Gamma') : \Delta \rangle$	$\rightarrow \langle A[\lambda x.e]; (x \mapsto (B, \Gamma')) : \Gamma; \Delta \rangle$	[Lam-Elim]
$\langle A[\underline{x}]; [\dots, (x \mapsto (B, \Gamma')), \dots]; \Delta \rangle$	$\rightarrow \langle A[\underline{B}]; \Gamma'; \Delta \rangle$	[BVar]
$\langle A[M[\underline{x}] @ B]; \Gamma; (B, \Gamma') : \$ : \Delta \rangle$	$\rightarrow \langle A[M[x] @ \underline{B}]; \Gamma'; \$ : \Delta \rangle,$ $x \notin dom(\Gamma)$	[FVar-0]

NB: здесь  $B$ , как и во всех [FVar-\*] правилах, является компонентом списка висячих аргументов. Более того,  $B$  должен быть аргументом некоторого вышестоящего в контексте  $A$  применения.

$\langle A[M[\underline{x}] @ B]; \Gamma; (B, \Gamma') : C : \Delta \rangle$	$\rightarrow \langle A[M[x] @ \underline{B}]; \Gamma'; \$ : C : \Delta \rangle,$ $C \neq \$, x \notin dom(\Gamma)$	[FVar-1]
$\langle A[M[\underline{x}] @ B]; \Gamma; \$ : (B, \Gamma') : \Delta \rangle$	$\rightarrow \langle A[M[x] @ \underline{B}]; \Gamma'; \Delta_1 \rangle,$ $x \notin dom(\Gamma), \Delta_1 = \begin{cases} \Delta, & \text{если } \Delta = \$ : \Delta_2 \\ \$ : \Delta, & \text{иначе} \end{cases}$	[FVar-2]

Рисунок 13. Система переходов для полной головной линейной редукции

**Теорема 3.** Полная головная линейная редукция завершается тогда и только тогда, когда терм имеет нормальную форму. Более того, если  $M$  — некоторый терм,  $s_{final} := \langle M'; \Gamma'; \Delta' \rangle$  — конечное состояние системы переходов полной головной линейной редукции для терма  $M$ , и  $M_{exp}$  — расширение состояния  $s_{final}$ , тогда:

- $M_{exp}$  есть  $M'$  за исключением вычеркнутых поддеревьев;
- $M_{exp}$  не содержит редексов;
- $M_{exp}$  является нормальной формой терма  $M$ .

*Доказательство.* Заметим, что правила [FVar-\*] не могут изменить расширение состояния системы переходов. Следовательно, доказательство корректности полной головной линейной редукции является прямым следствием корректности головной линейной редукции (см. следствия к теореме 2). Это значит, полная головная линейная редукция завершается тогда и только тогда, когда завершается полная головная редукция терма, что, в свою очередь, эквивалентно конечности дерева Бёма, представляющего входной терм, а значит эквивалентно и существованию нормальной формы входного терма. Иными словами, полная головная линейная редукция завершается тогда и только тогда, когда у входного терма существует нормальная форма. ■

## Глава 3

# Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления

В главе представлен разработанный автором алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления. Алгоритм согласован с нормальным порядком редукций и нормализует произвольный лямбда-терм, нормальная форма которого существует. Вместо изменения термов по правилам  $\beta$ -редукции алгоритм строит *обход* абстрактного синтаксического дерева терма, оставляя сам терм неизменным.

**Определение.** *Обоснованной последовательностью* (Justified Sequence) называется упорядоченная последовательность, каждый элемент которой может быть снабжён одним или несколькими указателями на более ранние элементы.

**Определение.** *Обходом* (Traversal) абстрактного синтаксического дерева лямбда-терма называется обоснованная последовательность его вершин, именуемых далее *токенами*.

Алгоритм трассирующей нормализации будет представлен как результат ряда последовательных трансформаций стандартной семантики нетипизированного лямбда-исчисления, основанной на подстановке. Данная глава завершается примером работы алгоритмы трассирующей нормализации на терме  $mul \vec{2} \vec{2} = \vec{2}(\vec{2}S)Z$ .

### 3.1. Семантика, основанная на подстановке

По определению (см. главу 1), терм находится в нормальной форме тогда и только тогда, когда он не содержит редексов — подвыражений вида  $e_1@e_2$ , где  $e_1 \equiv \lambda x.e$ . Стандартная процедура нормализации сначала редуцирует выражение  $e_1$  до функциональной абстракции, используя слабую редукцию, а затем применяет аксиому  $\beta$ -редукции. Семантика слабой редукции [86] приведена на рисунке 14.

$$\frac{\begin{array}{c} \text{wnf} \\ x \Downarrow x \end{array} \quad \frac{\begin{array}{c} \text{wnf} \\ e_1 \Downarrow \lambda x.e \end{array} \quad \frac{\begin{array}{c} \text{wnf} \\ e[e_2/x] \Downarrow e' \end{array}}{\begin{array}{c} \text{wnf} \\ e_1@e_2 \Downarrow e' \end{array}}}{\begin{array}{c} \text{wnf} \\ \lambda x.e \Downarrow \lambda x.e \end{array}} \quad \frac{\begin{array}{c} \text{wnf} \\ e_1 \Downarrow e' \neq \lambda x.e \end{array}}{\begin{array}{c} \text{wnf} \\ e_1@e_2 \Downarrow e'_1@e_2 \end{array}}$$

Рисунок 14. Слабая редукция

Сильная редукция (нормальный порядок), приводящая терм в нормальную форму, если и только если последняя существует, может быть представлена в виде правил, приведённых на рисунке 15 [86]. Эта процедура нормализации является *эффективной редуцирующей стратегией* (в терминологии [30]). Отметим, что процедура  $snf$  не является *хвосто-рекурсивной*, и, кроме того, использует имена переменных во время исполнения.

$$\frac{\frac{\begin{array}{c} \text{snf} \\ x \Downarrow x \end{array}}{\begin{array}{c} \text{snf} \\ e \Downarrow e' \end{array}} \quad \frac{\frac{\begin{array}{c} \text{wnf} \\ e_1 \Downarrow \lambda x.e \end{array} \quad \frac{\begin{array}{c} \text{snf} \\ e[e_2/x] \Downarrow e' \end{array}}{\begin{array}{c} \text{snf} \\ e_1@e_2 \Downarrow e' \end{array}}}{\begin{array}{c} \text{snf} \\ \lambda x.e \Downarrow \lambda x.e' \end{array}} \quad \frac{\begin{array}{c} \text{snf} \\ e_1 \Downarrow e_1' \neq \lambda x.e \end{array} \quad \frac{\begin{array}{c} \text{snf} \\ e_1' \Downarrow e_1'' \end{array} \quad \frac{\begin{array}{c} \text{snf} \\ e_2 \Downarrow e_2' \end{array}}{\begin{array}{c} \text{snf} \\ e_1@e_2 \Downarrow e_1''@e_2' \end{array}}}{\begin{array}{c} \text{snf} \\ e_1@e_2 \Downarrow e_1''@e_2' \end{array}}$$

Рисунок 15. Строгая редукция

### 3.2. Вычисление с окружением

Операция подстановки, используемая в разделе 3.1 для определения обеих стратегий редукции, как отмечалось в главе 1, является нетривиальной и дорого-

**Домены:**

$$\begin{aligned}
e \in Exp &= \lambda\text{-термы} \\
EE &= Exp \times Env \\
\rho \in Env &= Variable \rightarrow EE \cup \{Free\} \\
\rho_0 &= \lambda x. Free \\
\alpha \in Flag &= \{T, F\} \text{ (контекст применения)}
\end{aligned}$$

**Семантические функции:**

$$\begin{aligned}
\mathcal{R} : Exp &\rightarrow EE \\
[[ \ ]] : Exp &\rightarrow Flag \rightarrow Env \rightarrow EE
\end{aligned}$$

**Семантические равенства:**

$$\begin{aligned}
\mathcal{R}[[e]] &= [[e]] F \rho_0 \\
[[x]] \alpha \rho &= \mathbf{case} \rho x \mathbf{of} (e', \rho') \Rightarrow [[e']] \alpha \rho' \\
&\quad Free \Rightarrow (x, \rho_0)
\end{aligned}$$

$$\begin{aligned}
[[\lambda x.e]] T \rho &= (\lambda x.e, \rho) \\
[[\lambda x.e]] F \rho &= [[e]] F \rho[x \mapsto Free]
\end{aligned}$$

$$\begin{aligned}
[[e_1 @ e_2]] \alpha \rho &= \mathbf{let} (e'_1, \rho') = [[e_1]] T \rho \mathbf{in} \\
&\quad \mathbf{case} e'_1 \mathbf{of} \\
&\quad \lambda x.e_0 \Rightarrow [[e_0]] \alpha \rho'[x \mapsto (e_2, \rho)] \\
&\quad v \Rightarrow [[e_2]] F \rho
\end{aligned}$$

Рисунок 16. Строгая редукция, основанная на окружении

стоящей. Стандартным приёмом является использование *окружений*, вместо явной подстановки. На рисунке 16 приведена семантика строгой редукции лямбда-исчисления, основанная на вычислении с окружением.  $[[ \ ]]$  обозначает семантическую функцию,  $\rho$  — окружение, а  $A \rightarrow B$  — частичную функцию, областью определения которой является домен  $A$ , а областью значений — домен  $B$ .

Аргумент  $\alpha$  (контекст применения) является логическим флагом, значение которого равно  $T$  в случае слабой редукции и  $F$  в случае строгой.

Следует отметить, что семантика, приведённая на рисунке 16 завершается тогда и только тогда, когда нормальная форма терма существует. Для краткости и наглядности изложения, она симулирует все шаги вычислений, не возвращая реального результата. Для построения нормальной формы необходимо преобразовать результат  $(e, \rho)$  в лямбда-терм, раскрывая окружения и производя переименование переменных в случае необходимости.



**Домены:**

$$\begin{aligned}
e \in Exp &= \lambda\text{-термы} \\
EE &= Exp \times Env \\
\rho \in Env &= Variable \rightarrow EE \cup \{Free\} \\
\rho_0 &= \lambda x. Free \\
\alpha \in Flag &= \{T, F\} && \text{(контекст применения)} \\
k \in K &= \{\langle Kend \rangle\} && \text{(продолжения)} \\
&\cup \{\langle Kapp e \alpha \rho k \rangle \mid e \in Exp, \rho \in Env, k \in K\}
\end{aligned}$$

**Семантические функции:**

$$\begin{aligned}
\mathcal{R} : Exp &\rightarrow \{\mathbf{Succeed}\} \\
[[\ ]] : Exp &\rightarrow Flag \rightarrow Env \rightarrow K \rightarrow \{\mathbf{Succeed}\} \\
apk : Exp &\rightarrow K \rightarrow Env \rightarrow \{\mathbf{Succeed}\}
\end{aligned}$$

**Семантические равенства:**

$$\begin{aligned}
\mathcal{R}[[e]] &= [[e]] F \rho_0 \langle Kend \rangle \\
[[x]] \alpha \rho k &= \mathbf{case} \rho x \text{ of } Free \Rightarrow apk \ x \ k \ \rho_0 \\
&\quad (e', \rho') \Rightarrow [[e']] \alpha \rho' k
\end{aligned}$$

$$[[e_1 @ e_2]] \alpha \rho \not\equiv [[e_1]] T \rho \langle Kapp e_2 \alpha \rho k \rangle$$

$$\begin{aligned}
[[\lambda x.e]] F \rho k &= [[e]] F \rho[x \mapsto Free] k \\
[[\lambda x.e]] T \rho \langle Kapp e' \alpha \rho' \not\equiv [[e]] \alpha \rho[x \mapsto (e', \rho')] k \\
[[\lambda x.e]] T \rho \langle Kend \rangle &= \mathbf{Succeed}
\end{aligned}$$

$$\begin{aligned}
apk \ e \ \langle Kend \rangle \ \rho &= \mathbf{Succeed} \\
apk \ e \ \langle Kapp e' \alpha \rho' k \rangle \ \rho &= \mathbf{case} \ e \ \mathbf{of} \\
\lambda x.e'' &\Rightarrow [[e'']] \alpha \rho[x \mapsto (e', \rho')] k \\
\_ &\Rightarrow [[e']] F \rho' k
\end{aligned}$$

Рисунок 17. Строгая хвосто-рекурсивная редукция

Приведённая семантика обладает свойством *полукомпозициональности*. Это свойство является важным для эффективной компиляции лямбда-термов в целевой код. Согласно терминологии из [93], функция  $[[\ ]]$  применяется только к аргументам, которые являются синтаксическими подвыражениями исходного термина, что означает, что семантическая переменная является таковой “со статически ограниченным множеством значений” (Bounded Static Variation) (см. раздел 5.3.2 и [68]).

### 3.3. Хвосто-рекурсивная семантика

Следующим шагом является преобразование семантики в хвосто-рекурсивную. Для этого необходимо устранить все вложенные вызовы семантической функции, что возможно произвести в два этапа. Первый шаг — введение дополнительной функции *продолжения*, на вызов которой будут заменены все вложенные вызовы семантической функции. Затем, применяется подход дефункционализации по Рейнолдсу [59], позволяющий заменить функцию *продолжения* на структуру данных. На рисунке 17 приведён результат применения вышеописанных шагов к семантике, приведённой на рисунке 16. На рисунке 17  $\kappa \in K$  является дефункционализированным продолжением, функция  $ark$  применяет  $\kappa$  к выражению. Результатом семантической функции является либо значение “Succeed”, если вычисления завершились, либо семантическая функция расходится, и нормальной формы у исходного терма не существует.

Данная семантика является одновременно хвосто-рекурсивной и обладает свойством полуконпозициональности. Таким образом, согласно правилам, приведённым на рисунке 17, семантическая функция, будучи применённой ко входному лямбда-терму  $M$ , произведут следующую последовательность вызовов:

$$\llbracket e_1 \rrbracket \alpha_1 \rho_1 \kappa_1 \rightarrow \llbracket e_2 \rrbracket \alpha_2 \rho_2 \kappa_2 \rightarrow \dots \rightarrow \llbracket e_i \rrbracket \alpha_i \rho_i \kappa_i \rightarrow \dots,$$

где  $\forall i. e_i$  является подтермом терма  $M$ .

### 3.4. Семантика, основанная на истории и окружении

Следующим преобразованием является замена продолжения  $\kappa$  на указатель в историю  $h$ . Подобно хвосто-рекурсивной семантике, семантика, основанная на истории и окружении, осуществляет в процессе вычисления следующую последовательность вызовов:

$$\llbracket e_1 \rrbracket h_1 \rightarrow \llbracket e_2 \rrbracket h_2 \rightarrow \dots \rightarrow \llbracket e_i \rrbracket h_i \rightarrow \dots$$

История  $h$  является кумулятивным обходом, накапливающим последовательности вызовов семантических функций совместно с их аргументами. Каждая исто-

**Домены:**

$e \in Exp$	$=$	$\lambda - Expression$	
$EE$	$=$	$Exp \times Env$	
$\rho \in Env$	$=$	$Variable \rightarrow EE \cup \{Free\}$	
$\rho_0$	$=$	$\lambda x . Free$	
$\alpha \in Flag$	$=$	$\{T, F\}$	
$h, \in H, ch \in CH$	$=$	$[Item]$	(История)
$it \in Item$	$=$	$\langle Exp Flag Env CH \rangle$	

**Семантические функции:**

$\mathcal{R} : Exp \rightarrow H$
$eval : H \rightarrow H$
$apk : Exp \rightarrow Env \rightarrow CH \rightarrow H \rightarrow H$

**Семантические равенства:**

$$\mathcal{R}[[e]] = eval [ \langle e F \rho_0 [] \rangle ]$$

 **$eval h = \text{let } it : \_ = h \text{ in case } it \text{ of}$** 

$\langle x \quad \alpha \rho ch \rangle \Rightarrow apk x \rho_0 ch h$	<b>if</b>	$\rho x = Free$
$\langle x \quad \alpha \rho ch \rangle \Rightarrow eval \langle e' \alpha \rho' ch \rangle : h$	<b>if</b>	$\rho x = (e', \rho')$
$\langle \lambda x.e \quad T \rho ch \rangle \Rightarrow apk \lambda x.e \rho ch h$		
$\langle \lambda x.e \quad F \rho ch \rangle \Rightarrow eval \langle e F \rho[x \mapsto Free] ch \rangle : h$		
$\langle e_1 @ e_2 \alpha \rho ch \rangle \Rightarrow eval \langle e_1 T \rho h \rangle : h$		

 **$apk e \rho ch h = \text{case } ch \text{ of}$** 

$[]$	$\Rightarrow h$
$\langle (e_1 @ e_2 \alpha \rho_0 ch') : \_ \rangle$	$\Rightarrow eval (f e) : h$
<b>where</b>	
$f (\lambda x.e_0) = \langle e_0 \alpha \rho[x \mapsto (e_2, \rho)] ch' \rangle$	
$f e = \langle e_2 F \rho_0 ch' \rangle$	

Рисунок 18. Семантика, основанная на истории и окружении

рия  $h_i$  представляет собой список, элементами которого являются вызовами семантической функции  $[[e_j]]$ , где  $j = 1, \dots, i$ .  $\forall i > 0$  элемент истории  $h_i$  имеет следующий вид:  $h_i = \langle e_i \alpha_i \rho_i ch_i \rangle : h_{i-1}$ , где  $ch_i$  представляет  $\kappa_i$ .

Суть рассматриваемого преобразования заключается в замене окружения из хвосто-рекурсивной семантики на структуру данных  $\langle (Karr e_2) \alpha \rho k \rangle$  на момент

времени  $t_i$ , в который это продолжение было создано. Время  $t_i$  представляется указателем на соответствующий компонент  $ch_i$ , являющийся головой истории  $h_i$ .

Семантика, основанная на истории и окружении, отличается от хвосторекурсивной семантики тем, что, во-первых, история явно представляется кумулятивной записью, во-вторых, каждое значение  $\kappa_i \in K$  заменяется на *историю управления*  $ch_i$ , являющуюся префиксом текущей истории, т.е. указателем на ранний её элемент. Наконец, семантика, основанная на истории, не создаёт структур, представляющих продолжения, поскольку информация о продолжении содержится в каждом элементе истории  $\langle e \ \alpha \ \rho \ ch \rangle$ . Семантика, основанная на истории и окружении, приведена на рисунке 18.

### 3.5. Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления или семантика, основанная только на истории

Заключительным шагом является преобразование окружения в историю, являющуюся префиксом текущей истории и обозначаемую  $bh$ . Появляется новая семантическая функция *lookup*, реализующая ту же функциональность, что и окружение  $\rho$ , путём поиска соответствующего значения в текущей истории  $bh$ . В данной главе предполагается, что входной лямбда-терм снабжён индексами де Брауна. Как правило, индексы де Брауна используются вместо имён переменных. Для удобства изложения и компактности графического представления мы будем считать, что переменные представлены и их именами, и индексами одновременно. А именно, индексы де Брауна будут использоваться в определении семантических равенств, в то время как имена переменных будут использоваться для обозначения переменных в текущем обходе.

Алгоритм трассирующей нормализации приведён на рисунке 19.  $BV \ x \ i$  обозначает вождение связанной переменной  $x$  с индексом де Брауна, равным  $i$ , а  $FV \ x$  обозначает свободное вождение переменной  $x$ .

**Домены:**

$e \in Expr$	$=$	$\lambda - Expression$
$\alpha \in Flag$	$=$	$\{T, F\}$
$h \in H, ch \in CH, bh \in BH$	$=$	$[Item]$ (История)
$it \in Item$	$=$	$\langle Expr Flag BH CH \rangle$

**Семантические функции:**

$\mathcal{R}$	$:$	$Expr \rightarrow H$
$eval$	$:$	$H \rightarrow H$
$evoperand$	$:$	$Item \rightarrow H \rightarrow H$
$apk$	$:$	$Expr \rightarrow CH \rightarrow H \rightarrow H$
$lookup$	$:$	$Int \rightarrow Flag \rightarrow BH \rightarrow CH \rightarrow H \rightarrow H$

**Семантические равенства:**

$$\mathcal{R}[e] = eval [ \langle e F [] [] \rangle ]$$

$eval h = \mathbf{let} it : \_ = h \mathbf{ in case} it \mathbf{ of}$

$$\begin{aligned} \langle (FV x) \alpha bh ch \rangle &\Rightarrow apk (FV x) ch h \\ \langle (BV x i) \alpha bh ch \rangle &\Rightarrow lookup i \alpha bh ch h \\ \langle \lambda x.e \quad T bh ch \rangle &\Rightarrow apk \lambda x.e ch h \\ \langle \lambda x.e \quad F bh ch \rangle &\Rightarrow eval \langle e F bh ch \rangle : h \\ \langle e_1 @ e_2 \quad \alpha bh ch \rangle &\Rightarrow eval \langle e_1 T bh h \rangle : h \end{aligned}$$

$lookup 0 \alpha (\langle \_ T \_ ch' \rangle : \_) ch h = \mathbf{case} ch' \mathbf{ of}$

$$\langle e \_ bh \_ \rangle : \_ \Rightarrow evoperand \langle e \alpha bh ch \rangle h$$

$\_ \Rightarrow \mathbf{case} ch \mathbf{ of}$

$$[] \Rightarrow h$$

$$\langle ap \_ bh'' ch'' \rangle : \_ \Rightarrow evoperand \langle ap F bh'' ch'' \rangle h$$

$lookup 0 \_ (\langle \_ F \_ ch' \rangle : h') ch h = apk (BV \_ 0) ch h$

$lookup i \alpha (\langle \_ \_ bh' \_ \rangle : \_) ch h = lookup (i - 1) \alpha bh' ch h$

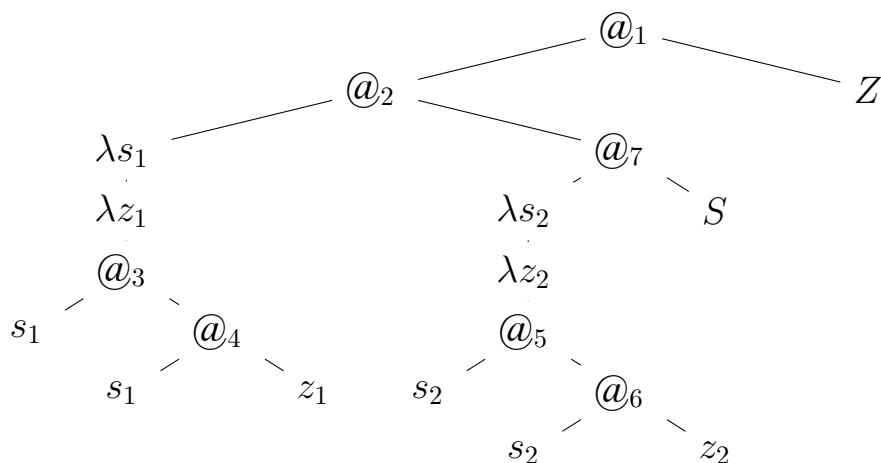
$apk \_ [] h = h$

$apk \lambda x.e (\langle \_ \alpha \_ ch \rangle : \_) h = eval \langle e \alpha h ch \rangle : h$

$apk \_ (\langle e \alpha bh ch \rangle : \_) h = evoperand \langle e F bh ch \rangle h$

$evoperand \langle e_1 @ e_2 \alpha bh ch \rangle h = eval \langle e_2 \alpha bh ch \rangle : h$

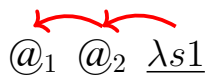
Рисунок 19. Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления

Рисунок 20. АСД терма  $mul \vec{z} \vec{z}$ 

## Пример

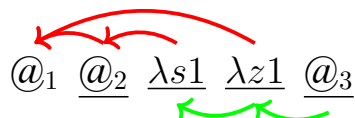
Приведём пример работы алгоритма трассирующей нормализации на терме  $mul \vec{z} \vec{z} = ((\lambda s_1. \lambda z_1. s_1 @_3 (s_1 @_4 z_1)) @_2 ((\lambda s_2. \lambda z_2. s_2 @_5 (s_2 @_6 z_2)) @_7 S)) @_1 Z$ , абстрактное синтаксическое дерево разбора которого приведено на рисунке 20. *Обход* представляет собой *историю*, каждый элемент которой может быть снабжён двумя видами указателей: *ch* для истории управления и *bh* для истории связываний переменных. На диаграммах, представляющих *обходы*, *bh* изображается зелёным указателем под *обходом*, а *ch* красным указателем над ним же. *Токен* выделен подчёркиванием, если его флаг  $\alpha$  имеет значение  $T$ . Начальным состоянием (исходной историей  $h_0$ ) является корень абстрактного синтаксического дерева входного терма с пустым множеством указателей,  $h = @_1$ .

Первые два шага алгоритма похожи друг на друга: к текущему состоянию применяется функция *eval*, следующая по самому левому пути в абстрактном синтаксическом дереве терма, *bh* не изменяется (остаётся пустым), а *ch* выставляется равным текущей истории.

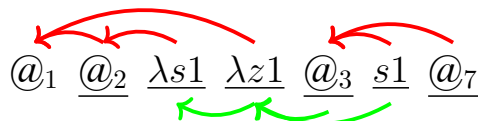


Следующие два шага также похожи: два токена–абстракции связываются с соответствующими токенами–применениями посредством указателя *ch*, образуя тем самым спинальные редексы. Получается, что функция применения продол-

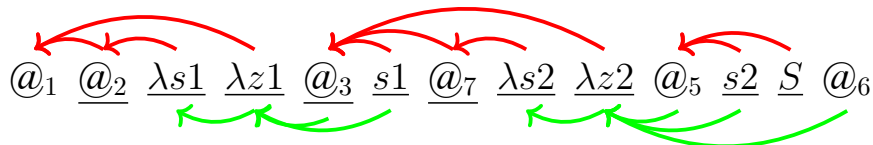
жения,  $apk$ , вызывается в обоих случаях. Функция  $apk$  расширяет текущий *обход*, снабжая новые токены указателями истории связывания, выставленными на самый поздний токен-абстракцию. Вершина  $bh$  снимается, а функция  $eval$  вызывается с не изменённым флагом  $\alpha$ .



Далее алгоритм выполняет шаг, аналогичных первым двум, после чего последним токеном становится токен-переменная  $s1$ . Эта переменная является связанной, а соответствующая ей абстракция может быть найдена в истории по цепочке указателей  $bh$ , что и достигается посредством вызова функции  $lookup$ . Результатом вызова функции  $lookup$  будет токен  $@_7$ , являющийся аргументом спинального редекса, образованного абстракцией  $\lambda s1$  и применением  $@_2$  (абстракция образует спинальный редекс, если она связана указателем  $ch$  с токеном-применением). После этого функция  $eval$  вызывается с аргументом  $@_7$ .

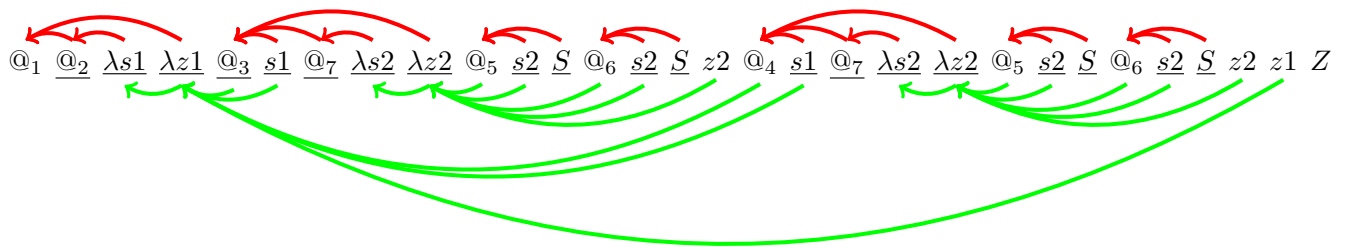


Следующим шагом построения обхода, заслуживающим внимания, является двенадцатый шаг. На этом шаге функция  $eval$  вызывает функцию  $lookup$  на аргументах  $h_{1..12}$   $S$ . Переменная  $S$  является свободной, следовательно, вызывается функция применения продолжения  $apk$ . Поскольку  $ch$  не пуст, вызывается функция  $eval$  на аргументе последнего применения, который ещё не был просмотрен и не образует спинального редекса. В данном случае таковым является токен  $@_6$ .

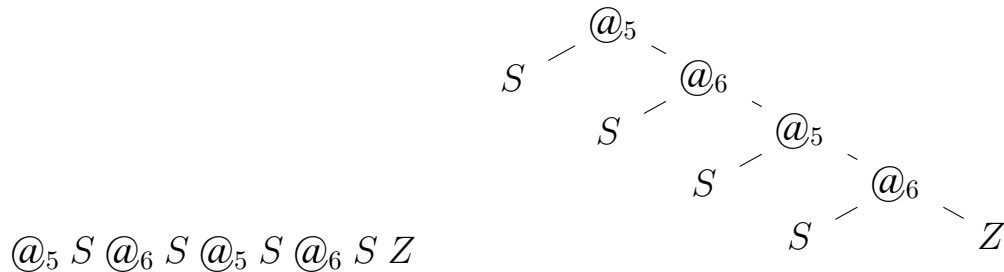


Наконец, *обход* завершится за тридцать шагов. Итоговый обход представлен на рисунке 21а).

Нормальная форма может быть получена из итогового обхода терма удалением из него всех токенов, участвующих в формировании спинальных редексов, а также всех вхождений связанных переменных, вместо которых была произведе-



а) Итоговый обход



б) Обход после “очистки”

в) АСД терма соответствующего обходу после “очистки”

Рисунок 21. Результат работы UNP на терме  $mul \vec{2} \vec{2}$ 

на подстановку. Результатом такой “очистки” обхода, полученного посредством алгоритма трассирующей нормализации, будет обход абстрактного синтаксического дерева нормальной формы терма в глубину. Оба они приведены на рисунках<sup>16</sup> 21б) и 21в) соответственно. Как и ожидалось, им соответствует терм  $S(S(S(S Z)))$ , являющийся нормальной формой исходного терма,  $mul \vec{2} \vec{2}$ .

<sup>16</sup>На рисунке видно, что некоторые токены (вершины АСД исходного терма  $mul \vec{2} \vec{2}$ ) могут встречаться в результате несколько раз, хотя входить в исходный терм многие из них могли лишь один раз. Например, оба токена применения,  $@_5$  и  $@_6$ , встречаются дважды, а свободная переменная  $S$  встречается аж четыре раза. В данной ситуации нет никакого противоречия: токены применения рассматриваются, игнорируя нумерацию, которая была нужна лишь для большей наглядности графических представлений обходов, также игнорируются и имена переменных. Напомним, что мы рассматриваем переменные сразу в двух представлениях: именованном и анонимном. Таким образом, после уничтожения имён, переменные представляются индексами де Брауна, а абстракции становятся анонимными.

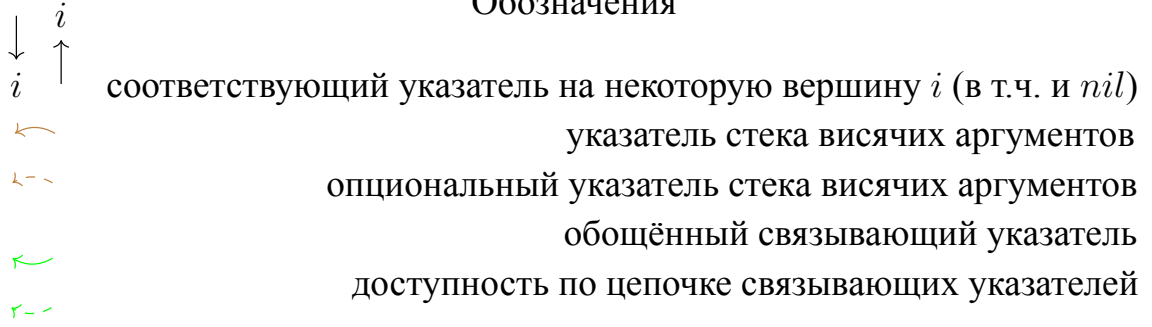


## Глава 4

# Корректность алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления

Данная глава посвящена доказательству<sup>17</sup> корректности алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления, приведённого в разделе 3.5. Для этого вводится ограниченная версия алгоритма трассирующей нормализации, BUNP, в виде системы переходов (см. раздел 4.1), и устанавливается соответствие между ней и системой переходов для головной линейной редукцией (см. теорему 4). Затем в раздел 4.1.1 и сам алгоритм трассирующей нормализации представляется в виде системы переходов, которая в свою очередь является расширением системы переходов для BUNP, а также устанавливается соответствие между этой системой и системой переходов для полной головной линейной редукции — см. теорему 5.

Обозначения

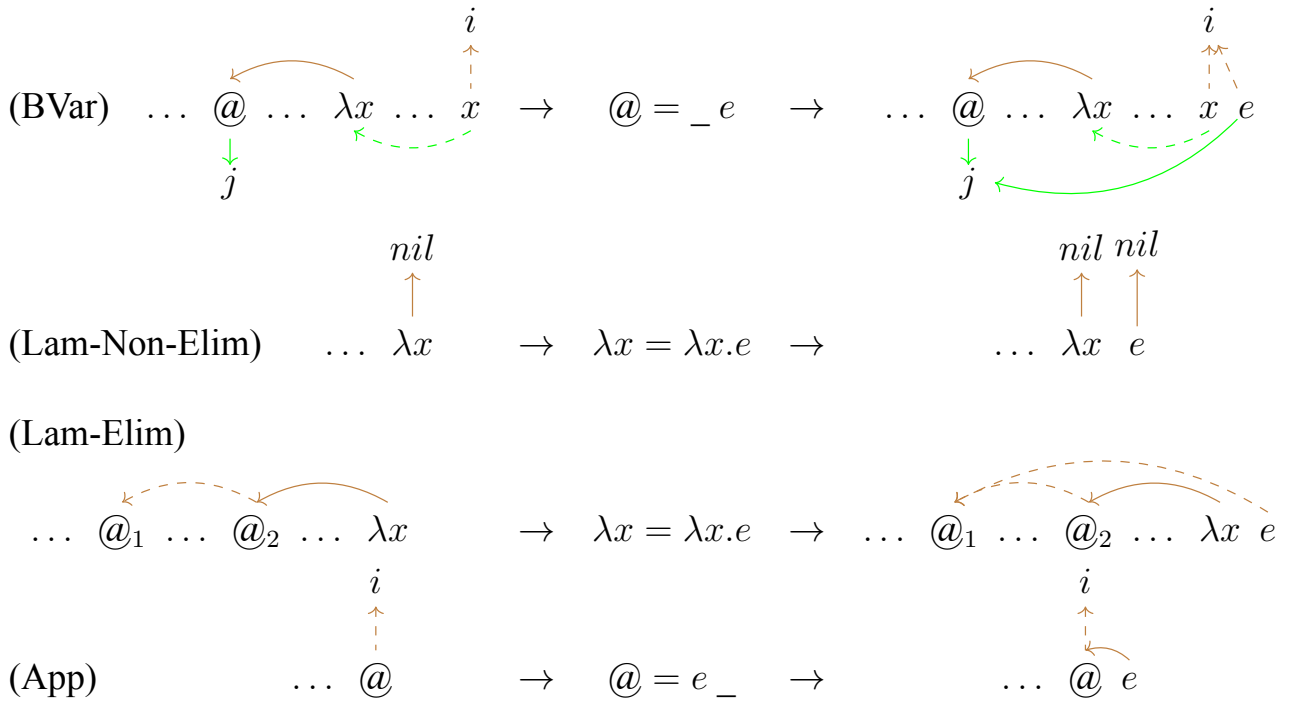


Состояние обход<sup>a</sup>

Начальное состояние корень AST входного терма

Конечное состояние обход<sup>b</sup>

Правила переходов



<sup>a</sup>Для удобства формального изложения мы будем считать, что состоянием системы переходов для BUNP является тройка  $\langle t; \beta; \alpha \rangle$ , где  $t : \Sigma \rightarrow [1 .. |t|]$  является упорядоченной последовательностью,  $|t|$  — длина обхода  $t$ , а частичные функции  $\alpha, \beta : [1 .. |t|] \rightarrow [1 .. |t| - 1]$  определяют множества указателей стека висячих аргументов и связывающих указателей соответственно.

<sup>b</sup>Конечным состоянием является обход, такой, что его последний токен является токеном-переменной, не имеющим указателя висячих аргументов,  $t = \_ \bullet x : \alpha(x) = \perp$ , являющейся либо свободной,  $\exists k \in \mathbb{N} : \alpha \circ \beta^k(x) = \lambda x$ , либо связанной, чья абстракция не связана никаким простым редексом,  $\nexists k \in \mathbb{N} : \beta^k(x) = \lambda x \Rightarrow \beta^k(x) = \lambda x \notin \mathfrak{D}(\alpha)$ .

Рисунок 22. Система переходов для BUNP

## 4.1. Головная линейная редукция и ограниченная версия алгоритма трассирующей нормализации

В этом разделе представлена ограниченная версия алгоритма трассирующей нормализации в виде системы переходов, и установлено соответствие между приведёнными системами переходов для головной линейной редукции (см. рис. 9) и ограниченной версии алгоритма трассирующей нормализации (см. рис. 22).

### 4.1.1. Система переходов для BUNP

BUNP, или базовый UNP, является ограниченной версией UNP (см. раздел 3.5, [73]). BUNP соответствует головной линейной редукции, являющейся основой для полной головной линейной редукции. Далее мы установим соответствие между системами переходов для BUNP и HLR, определив функцию преобразования состояний одной системы переходов в состояния другой. Более того, правила переходов рассматриваемых систем переходов напрямую отображаются друг в друга. В отличие от HLR, состояние системы переходов для BUNP является обоснованной последовательностью вершин входного лямбда-терма, снабжённых указателями на ранние вершины последовательности. В BUNP существует два следующих рода указателей.

- *Обобщённый связывающий указатель* (generalised binder backpointer). Для краткости мы часто будем упускать слово “обобщённый” и говорить просто — *связывающий указатель*. На диаграммах, представляющих обходы, он будет обозначаться зелёной стрелкой под последовательностью токенов. Связывающий указатель является BUNP-эквивалентом окружения: он позволяет построить соответствующее окружение на каждом шаге алгоритма. Для каждого токена он указывает на токен, представляющий последнюю абстракцию, которую необходимо добавить в окружение при его формировании, если эта абстракция участвует в образовании какого-либо простого редекса.

---

<sup>17</sup>Доказательство опубликовано в [94].

- *Указатель стека висячих аргументов*. На диаграммах он будет представлен коричневой стрелкой, изображаемой над последовательностью токенов. Указатель стека висячих аргументов является BUNP-эквивалентом стека висячих указателей  $\sigma$  системы переходов для HLR. Инвариантом этого указателя является то, что он всегда указывает либо на токен, представляющий применение термов, либо отсутствует вовсе. Указатель стека висячих аргументов позволяет конструировать список спинальных аргументов терма на каждом шаге алгоритма. Заметим, что любое вхождение нижеследующего шаблона в обход формирует простой редекс  $(\lambda x, A)$ , где  $A$  — спинальный аргумент (аргумент  $@$  в АСД терма), а  $\lambda x$  — головная абстракция.



Таким образом, указатель стека висячих аргументов для токенов-абстракций всегда указывает на токен-применение, с аргументом которого он формирует простой редекс, а для остальных токенов — на применение, являющиеся непосредственным предком последнего спинального аргумента. В некоторых случаях, чтобы подчеркнуть, что некоторый токен не имеет указателя стека висячих аргументов, мы будем вводить специальную вершину *nil* за пределами обхода, на которую соответствующий указатель и будет указывать.

Система переходов для BUNP приведена на рис. 22<sup>18</sup>. Для краткости изложения и ясности графического представления, а также в виду их простоты, правила установки обобщённого связывающего указателя опущены во всех правилах, кроме (BVar). Для всех остальных правил применяется следующее правило установки обобщённого связывающего указателя: если последний токен обхода был абстракцией, то у добавляемого в обход токена обобщённый связывающий указатель выставляется на эту самую абстракцию, в противном случае он выставляется такой же, как у предыдущего токена.

<sup>18</sup>Заметим, что правила (Lam-Elim) и (Lam-Non-Elim), вообще говоря, могут быть объединены в одно. Тем не менее, для более наглядного представления и взаимной-однозначности соответствия между одноимёнными правилами систем переходов для HLR и BUNP мы вводим правила именно в таком виде.

## 4.1.2. Соответствие между головной линейной редукцией и ограниченной версией алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления

Мы установим соответствие между приведёнными в предыдущих разделах системами переходов для HLR (см. рис. 9) и BUNP (см. рис. 22). Мы покажем, что как состояния систем переходов, так и их шаги находятся во взаимно-однозначном соответствии.

Для того, чтобы восстановить состояние системы переходов для HLR из текущего обхода, необходимо определить каждую из его компонент: текущий терм с выделенной вершиной, текущее окружение  $\rho$  и корректный стек висячих аргументов  $\sigma$ . Итак, пусть дан обход  $t$ , тогда терм, окружение и стек висячих аргументов могут быть восстановлены из него нижеследующим образом.

- *Терм.* Обход  $t$  представляет собой подпоследовательность самого левого пути в абстрактном синтаксическом дереве терма. Таким образом, для восстановления соответствующего первого компонента системы переходов для HLR необходимо снабдить токены-применения соответствующими висячими аргументами, игнорируя вхождения связанных переменных, и выделить вершину, соответствующую последнему токenu обхода.
- *Окружение.* Напомним, что каждое вхождение шаблона

$$\dots @ \dots \lambda x \dots$$

определяет простой редекс  $(\lambda x, A)$ , где  $A$  является аргументом вершины-применения  $@$ . Если последним токеном текущего обхода является абстракция, то при восстановлении окружения обход рассматривается без учёта этой абстракции. Пусть  $\mathcal{R}$  — множество всех простых редексов обхода  $t$ ,  $\Lambda$  — список лямбда-абстракций, доступных из последнего токена по цепочке связывающих указателей, тогда текущее окружение  $\rho(t) = \{x \mapsto (A, \rho') \mid (\lambda x, A) \in \mathcal{R}\} \uparrow \Lambda$ , где  $\rho'$  определено рекурсивно. Заметим, что мы подразумеваем, что множество  $\mathcal{R}$  упорядочено: пусть  $r1 = (\lambda x, A), r2 = (\lambda y, B) \in \mathcal{R}$ , тогда  $r1 > r2$  тогда и только тогда, когда токен  $\lambda x$  входит в обход до токена  $\lambda y$ .

- *Висячие аргументы.* Стек висячих аргументов получается из обхода ограничением последнего на список токенов, доступных из него по цепочке указателей висячих аргументов, и снабжением их соответствующими окружениями, согласно предыдущему пункту.

$$\begin{array}{ccc}
 T = \langle t; \beta; \alpha \rangle & \xrightarrow{\text{rule}_T} & T' = \langle t'; \beta'; \alpha' \rangle \\
 \downarrow & & \vdots \\
 S = \langle M; \rho; \sigma \rangle & \xrightarrow{\text{rule}_S} & S' = \langle M'; \rho'; \sigma' \rangle
 \end{array}$$

Рисунок 23. Иллюстрация к Теореме 4

**Теорема 4** (Согласованность систем переходов для BUNP и HLR). Приведённый выше алгоритм восстановления состояния системы переходов для HLR из соответствующего состояния системы переходов для BUNP корректен.

Пусть  $S$  и  $T$  — соответствующие состояния систем переходов для HLR и BUNP, а  $S'$  и  $T'$  получаются из  $S$  и  $T$  применением правил  $\text{rule}_S$  и  $\text{rule}_T$  соответственно, тогда правила  $\text{rule}_S$  и  $\text{rule}_T$  имеют одинаковые имена, а состояние  $S'$  получается из состояния  $T'$  согласно вышеописанному алгоритму. Иными словами диаграмма, приведённая на рис. 23, является коммутативной.

*Доказательство.* Доказательство выполним индукцией по количеству применений правила  $\text{rule}_T$ .

База индукции. Тривиально.

Индукционный переход. Рассмотрим все возможные случаи для правила  $\text{rule}_T$ .

- (App):  $\dots @ \longrightarrow @ = e_ \longrightarrow \dots @ \overset{\curvearrowright}{e}$

Согласно индукционному предположению,  $S$  соответствует  $T$ . Более того, первой компонентой  $S$  является терм  $M[@]$ . Согласно правилам системы переходов для HLR, правило [App] может быть применено к состоянию  $S$ , и никакое другое правило в данном случае применено быть не может.

1. В обеих системах переходов правила требуют перейти к первому аргументу (левому ребёнку в смысле АСД терма) применения, таким об-

разом первый компонент  $S'$  в точности соответствует тому, который будет восстановлен из  $T' — M[e]$ .

2. Согласно правилам восстановления состояния системы переходов для HLR из обхода, окружение, восстановленное из состояния  $T'$ , совпадает с окружением, восстановленным из состояния  $T$ . Действительно, если  $\exists x : e = \lambda x$ , то по определению токен  $\lambda x$  исключается из рассмотрения при восстановлении окружения. Если же  $\forall x. e \neq \lambda x$ , то, очевидно, список простых редексов обхода  $T$  и  $T'$  совпадают. Согласно же правилу [App], окружение состояния  $S'$  является тем же самым, что и у состояния  $S$ . Таким образом,  $\rho_{S'} = \rho_S \stackrel{\text{ИП}}{=} \rho_T = \rho_{T'}$ .
3. В обоих случаях стек висячих аргументов расширяется аргументом рассматриваемых применений, которые совпадают согласно индукционному предположению.

$$\text{– (Lam-Non-Elim):} \quad \dots \overset{\text{nil}}{\uparrow} \lambda x \longrightarrow \lambda x = \lambda x.e \longrightarrow \dots \overset{\text{nil}}{\uparrow} \lambda x \quad \overset{\text{nil}}{\uparrow} e$$

Поскольку указатель стека висячих аргументов у токена  $\lambda x$  является нулевым, то согласно индукционному предположению стек висячих указателей у  $S$  пуст. Следовательно, только правило [Lam-Non-Elim] может быть применено к  $S$ . По тем же соображениям, что и в случае (App), терм и окружение, восстанавливаемые из  $T'$  в точности совпадают с первыми двумя компонентами  $S'$ , а стек висячих аргументов в обоих случаях остаётся пустым.

– (Lam-Elim):

$$\dots \overset{\curvearrowright}{@_1} \dots \overset{\curvearrowright}{@_2} \dots \lambda x \longrightarrow \lambda x = \lambda x.e \longrightarrow \dots \overset{\curvearrowright}{@_1} \dots \overset{\curvearrowright}{@_2} \dots \lambda x \quad e$$

Согласно индукционному предположению, окружение в состоянии  $S$  не пусто. Более того, его первым элементом является  $(A, \rho')$ , где  $A$  — аргумент  $@_2$ , а  $\rho'$  — соответствующее окружение. Таким образом правило [Lam-Elim] должно быть применено к состоянию  $S$ . Как и в случае с правилом (App), терм, построенный из состояния  $T'$ , является первым компонентом состояния  $S'$ . Окружения в обоих случаях расширяются простым редексом

$(\lambda x, A)$ , где окружение терма  $A$  одинаково в обеих системах переходов. Также в обоих случаях ровно один элемент был снят со стека висячих аргументов.

$$- \text{ (BVar): } \dots @ \dots \lambda x \dots x \xrightarrow{\quad} @ = \_e \xrightarrow{\quad} \dots @ \dots \lambda x \dots x e$$

Если правило (BVar) может быть применено, то, во-первых, токен  $\lambda x$  должен быть доступен по цепочке связывающих указателей из токена  $x$ , во-вторых, токены  $\lambda x$  и  $@$  должны быть связаны указателем стека висячих аргументов (т.е.  $\alpha(\lambda x) = @$ ). Согласно индукционному предположению  $S$  восстанавливается из  $T$ , а значит,  $(x, e) \in \rho_S$ , и правило [BVar] может быть применено к состоянию  $S$ , и никакое другое правило к  $S$  применено быть не может, следовательно, термы, подставляемые вместо  $x$  в обоих случаях, совпадают между собой. Таким образом терм, восстанавливаемый из  $T'$  совпадает с первой компонентой состояния  $S'$ , а его окружение — с окружением  $S'$ . Наконец, стек висячих аргументов остаётся неизменным в обоих случаях.

■

**Следствие 1.** Первой компонентой состояния, получающегося восстановлением из конечного состояния системы переходов для BUNP, согласно приведённым правилам, является терм в псевдо-головной нормальной форме.

## 4.2. Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления и полная головная линейная редукция

Система переходов для полной головной линейной редукции (CHLR) приведена на рисунке 13. Для удобства чтения изменения по сравнению правилами система переходов для HLR выделены с помощью прямоугольников: выделение. Система переходов для полной головной линейной редукции отличается от таковой для головной линейной редукции наличием новых правил [FVar-\*], отвечающих



за рекурсивное применение к аргументам, а также нового символа-разделителя  $\$$ , запрещающего связывать абстракции и применения, не находящиеся на одном пути в абстрактном синтаксическом дереве терма, соответствующего текущему состоянию системы переходов. Более того, конечным состоянием может быть лишь такое, в котором стек висячих аргументов пуст, а текущей выделенной вершиной является переменная, не определённая в текущем окружении.

Далее приводится система переходов для UNP, согласованная с оригинальной формулировкой алгоритма трассирующей нормализации (см. рис. 19 и [73]), и устанавливается соответствие между системами переходов для CHLR и UNP подобно тому, как это было сделано с соответствующими системами переходов для HLR и BUNP в предыдущем разделе.

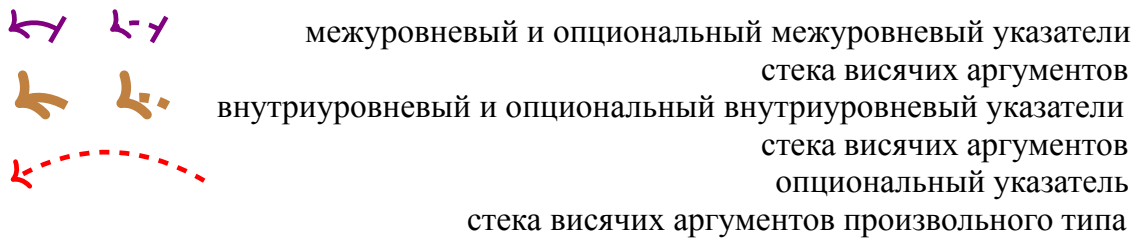
#### 4.2.1. UNP в виде системы переходов

В данном разделе приводится формальное представление UNP в виде системы переходов. Заметим, что ниже представленная система переходов имеет в точности столько же правил переходов, сколько и система переходов для полной головной линейной редукции, предложенная в [95] и приведённая на рисунке 9. Более того, мы покажем, как состояния этих систем переходов связаны отношением симуляции.

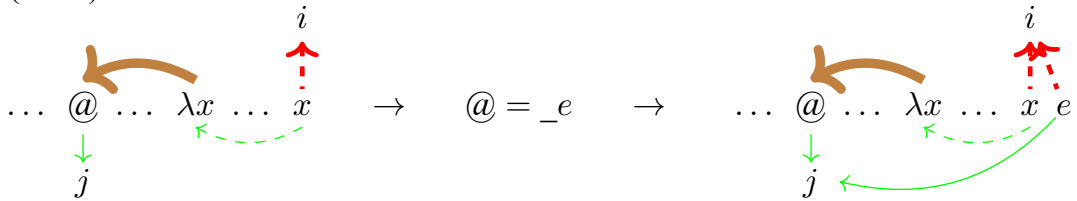
Подобно тому, как система переходов для полной головной линейной редукции является обобщением системы переходов для головной линейной редукции, так и система переходов для алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления является обобщением системы переходов для ограниченной версии алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления. Как и в случае BUNP, в UNP каждый токен может быть снабжён указателями нижеследующих родов.

- *Указатель стека висячих аргументов.* В отличие от BUNP, в UNP указатель стека висячих аргументов бывает двух нижеследующих типов.
  - *Внутриуровневый указатель.* Изображается жирной коричневой стрелкой над обходом и имеет тот же смысл, что и указатель стека висячих аргументов в BUNP.

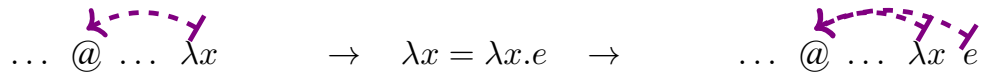
## Новые обозначения



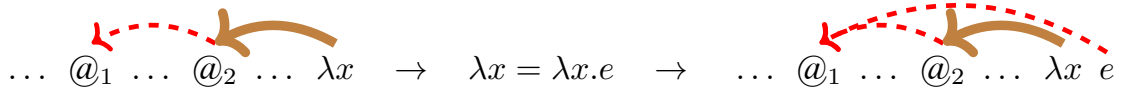
(BVar)



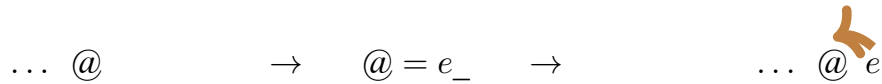
(Lam-Non-Elim)



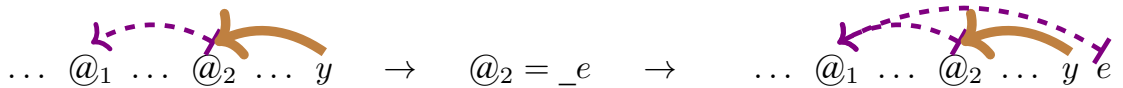
(Lam-Elim)



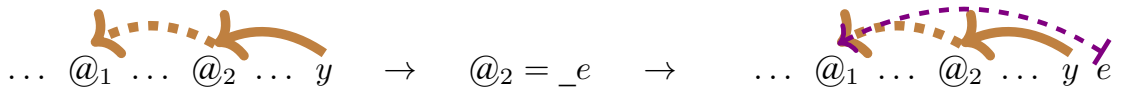
(App)



(FVar-0)



(FVar-1)



(FVar-2)

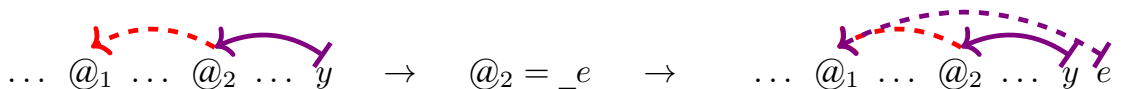


Рисунок 24. Правила переходов для UNP

- *Межуровневый* указатель. Изображается фиолетовой стрелкой с плоским началом над обходом. В то время как межуровневый указатель всё ещё указывает на последний висячий аргумент, он запрещает формирование простого редекса. Интуитивно, он является аналогом символа-разделителя  $\$$  системы переходов для CHLR.

Мы также будем использовать пунктирный красный указатель над обходом для обозначения указателя стека висячих аргументов в тех случаях, когда нас не интересует его тип. При формулировке правил переходов красные пунктирные указатели должны быть согласованы: добавляемый к новому токену указатель стека висячих аргументов, будучи изображённым красной пунктирной стрелкой, имеет тот же тип, что и аналогичный ему указатель, уже представленный в левой части правила.

- *Обобщённый связывающий* указатель. Как и в случае с BUNP, связывающий указатель изображается на диаграммах зелёной стрелкой под обходом. Более того, его смысл и правила установки остаются неизменными.

Правила переходов для UNP приведены на рисунке 24. Для обеспечения детерминированности системы переходов, мы будем считать, что правила упорядочены и в случае, когда несколько правил может быть применено, применяется то, что приведено раньше на рисунке 24. В действительности, в конфликте могут находиться лишь правило (BVar) и любое из правил (FVar-\*), а наш порядок означает, что последние могут быть применены тогда и только тогда, когда не может быть применено первое, иными словами, правила для свободных переменных применяются, только если текущая переменная не связана никаким простым редексом.

#### 4.2.2. Соответствие между CHLR и UNP

Для восстановления состояния системы переходов для полной головной линейной редукции из соответствующего состояния системы переходов для алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления мы определим способ восстановления каждой из его компонент по отдельности.

- *Терм с выделенной вершиной*. Напомним, что *обход* (Traversal) представляет собой обход абстрактного синтаксического дерева в глубину. Таким об-

разом для восстановления соответствующего терма необходимо поочерёдно добавлять токены, соблюдая арность соответствующих конструкторов, при этом опуская связанные переменные, вместо которых была произведена линейная подстановка<sup>19</sup>, за исключением тех случаев, когда последним токеном обхода является токен, представляющий связную переменную, и вычёркивая простые редексы — пары токенов  $(@, \lambda x) : \alpha(\lambda x) = @$ <sup>20</sup>. Последний токен текущего обхода определяет выделенную вершину, а стек висячих аргументов является упорядоченным список аргументов, которые необходимо подставить, как соответствующие аргументы применений.

- *Окружение*. Окружение восстанавливается так же, как и в случае BUNP. Построение текущего окружения завершается при появлении первого межуровневого указателя.
- *Стек висячих аргументов* восстанавливается напрямую из текущей цепочки указателей стека висячих аргументов: внутриуровневые указатели определяют текущий аргумент, окружение которого восстанавливается согласно предыдущему пункту, а межуровневые, помимо вышеописанного, соответствуют добавлению в стек символа-разделителя \$.

**Теорема 5** (Согласованность алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления с полной головной линейной редукцией). Приведённый выше алгоритм восстановления состояния системы переходов для CHLR из соответствующего состояния системы переходов для UNP корректен.

Пусть  $S$  и  $T$  — соответствующие состояния систем переходов для CHLR и UNP, а  $S'$  и  $T'$  получаются из  $S$  и  $T$  применением правил  $rule_S$  и  $rule_T$  соответственно, тогда правила  $rule_S$  и  $rule_T$  имеют одинаковые имена, а состояние  $S'$  получается из состояния  $T'$  согласно вышеописанному алгоритму. Иными словами, диаграмма, приведённая на рисунке 25, является коммутативной.

<sup>19</sup>Заметим, что связанные переменные, вместо которых не была произведена линейная подстановка, т.е. было применено одно из правил (FVar-\*), а не правило (BVar), не исключаются из обхода при восстановлении терма с выделенной вершиной.

<sup>20</sup>Мы будем придерживаться следующих обозначений:  $\alpha(\lambda x) = @$ для межуровневых и  $\alpha(\lambda x) = @$  для внутриуровневых указателей стека висячих аргументов.$

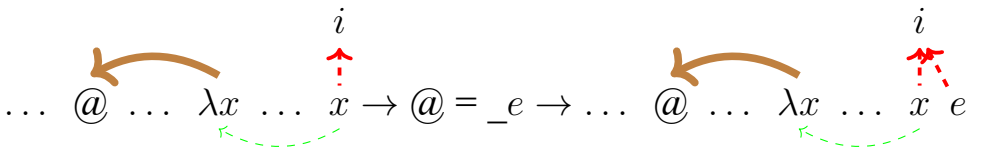
$$\begin{array}{ccc}
T = \langle t; \beta; \alpha \rangle & \xrightarrow{\text{rule}_T} & T' = \langle t'; \beta'; \alpha' \rangle \\
\downarrow & & \vdots \\
S = \langle M; \rho; \sigma \rangle & \xrightarrow{\text{rule}_S} & S' = \langle M'; \rho'; \sigma' \rangle
\end{array}$$

Рисунок 25. Иллюстрация к Теореме 5

*Доказательство.* Доказательство проводится индукцией по количеству применений правила  $\text{rule}_T$ .

База индукции. Тривиально.

Индукционный переход. Рассмотрим все случаи для  $\text{rule}_T$ .

(BVar) 

Итак, согласно индукционному предположению состояние  $S$  восстанавливается из состояния  $T$ ,  $T'$  получается из  $T$  посредством правила (BVar). Докажем, что  $S'$  получается из  $S$  применением правила [BVar] и восстанавливается из  $T'$ . Поскольку  $S$  восстанавливается из  $T$ , то первой его компонентой является терм с выделенной вершиной  $x$ ,  $M[x]$ . Более того,  $x \in \rho_S$  поскольку согласно индукционному предположению, т.к.  $\exists k \in \mathbb{N} : \beta^k(x) = \lambda x \wedge \alpha(\lambda x) = @ = \_e$ . Таким образом, правило [BVar] может быть применено к состоянию  $S$ , а поскольку система переходов для CHLR является детерминированной, то применено может быть только оно. Согласно индукционному предположению, окружения всяческого аргумента  $e$ , которое становится текущим после применения соответствующего правила, равны в обеих системах переходов, а стек всячих аргументов состояния  $S$  равен тому, который восстанавливается из состояния  $T$ , и в обеих системах переходов он остаётся неизменным.

(Lam-Non-Elim) 

Согласно индукционному предположению,  $S = \langle M[\underline{\lambda x}.e]; \rho; \$ : \sigma \rangle$ , следовательно,  $S \xrightarrow{[\text{Lam-Non-Elim}]} S' = \langle M[\lambda x.\underline{e}]; \rho; \$ : \sigma \rangle$ . Согласно же правилам восстановления состояния системы переходов для CHLR из  $T'$ , как окружение, так и стек всячих аргументов остаётся неизменным, а выделенной вершиной

терма, очевидно, становится корень  $e$ .

$$\text{(Lam-Elim)} \quad \dots @_1 \dots @_2 \dots \lambda x \rightarrow \lambda x = \lambda x.e \rightarrow \dots @_1 \dots @_2 \dots \lambda x e$$

По индукционному предположению,  $S = \langle M[\underline{\lambda x}.e]; \rho; (e_1, \rho') : (e_2, \rho'') : \sigma \rangle$ , тогда  $S \xrightarrow{[\text{Lam-Elim}]} S' = \langle M[\underline{\lambda x}.e]; (e_1, \rho') : \rho; (e_2, \rho'') : \sigma \rangle$ . Согласно правилам восстановления состояния системы переходов для CHLR из соответствующего состояния системы переходов для UNP, состояние, восстанавливаемое из состояния  $T'$ , отличается от  $S$  переносом выделения на вершину  $e$ , расширением окружения элементом  $(e_1, \rho')$  и отсутствием последнего на стеке висячих аргументов.

(App) Аналогично BUNP (см. теорему 4).

$$\text{(FVar-0)} \quad \dots @_1 \dots @_2 \dots y \rightarrow @_2 = \_e \rightarrow \dots @_1 \dots @_2 \dots y e$$

$$S \stackrel{\text{ИП}}{=} \langle A[M[x]@B]; \rho; (B, \rho') : \$ : \sigma \rangle \xrightarrow{[\text{FVar-0}]} S' = \langle A[M[x]@B]; \rho'; \$ : \sigma \rangle.$$

Пусть состояние, восстанавливаемое из  $T'$ ,  $S_{T'} \Leftarrow \langle M_{T'}; \rho_{T'}; \sigma_{T'} \rangle$ . Тогда, очевидно,  $M_{T'} = A[M[x]@B]$ ,  $\rho_{T'} = \rho'$  по ИП для токена  $@_2$ ,  $\sigma_{T'}$  отличается от  $\sigma_T = (B, \rho') : \$ : \sigma$  снятием вершины последнего, т.е.  $\sigma_{T'} = \$ : \sigma$ .

$$\text{(FVar-1)} \quad \dots @_1 \dots @_2 \dots y \rightarrow @_2 = \_e \rightarrow \dots @_1 \dots @_2 \dots y e$$

$S = \langle A[M[x]@B]; \rho; (B, \rho') : C : \sigma \rangle$ ,  $C \neq \$$ ,  $x \notin \mathcal{D}(\rho)$  соответствует состоянию, восстанавливаемому из  $T \Leftarrow S_T$ , а  $S \xrightarrow{[\text{FVar-1}]} S' = \langle A[M[x]@B]; \rho'; \$ : C : \sigma \rangle$ . Тогда, очевидно,  $M_{T'} = A[M[x]@B]$ ,  $\rho_{T'} = \rho'$  по ИП для токена  $@_2$ ,  $\sigma_{T'}$  отличается от  $\sigma_T = (B, \rho') : C : \sigma$  снятием вершины последнего и добавлением токена-разделителя ввиду изменения типа указателя на межуровневый, т.е.  $\sigma_{T'} = \$ : C : \sigma$ .

(FVar-2) Аналогично двум предыдущим случаям. ■

**Следствие 2** (UNP является эффективной нормализующей процедурой). UNP завершается *тогда и только тогда*, когда нормальная форма входного терма существует, и она же является первой компонентой состояния, восстановленного из конечного состояния системы переходов для UNP.

## Глава 5

# Компиляция путём специализации

В данной главе мы покажем, как алгоритм трассирующей нормализации может быть использован для генерации компилятора путём его специализации. Известно, что компилятор из языка  $S$  в язык  $T$  (как и генератор компиляторов) может быть автоматически сгенерирован на основе интерпретатора языка  $S$ , реализованного на языке  $T$ , при наличии самоприменимого смешанного вычислителя *mix* для языка  $T$  [64, 67, 68]. Существование смешанного вычислителя следует из так называемых *проекций Футамуры-Ершова-Турчина* (см. раздел 5.2).

### 5.1. Частичные вычисления

*Частичным вычислителем* или *специализатором* называется мета-программа, которая по некоторой другой программе и подмножеству её входных данных порождает *специализированную* версию этой программы (см. [68]). При этом результат исполнения специализированной программы на оставшихся входных данных должен совпадать с результатом исполнения исходной программы на всём множестве входных данных. Обоснованием возможности специализации и существования специализаторов является широко известная *лемма о трансляции*, или чаще  *$s - m - n$  теорема Клини*, доказанная С. Клини в 1943 году. Частичные вычисления могут быть определены, как специальный случай  *$s - m - n$  теоремы*, а именно,  $s - 1 - 1$ .

**Определение** (*Частичный вычислитель*). Частичным вычислителем или специализатором называется программа  $mix$ , удовлетворяющая нижеследующему уравнению.

$$\forall p \in \text{Programs} . \forall s, d \in \text{Data} . \llbracket [mix](p, s) \rrbracket (d) = \llbracket p \rrbracket (s, d)$$

В результате, в то время как классические вычисления  $\llbracket p \rrbracket (s, d)$  представляют собой одношаговое вычисление, при частичных вычислениях исполнение программы разбивается на две стадии,  $\llbracket [mix](p, s) \rrbracket$ .

В случае лямбда-исчисления частичным вычислителем можно называть такую программу  $mix$ , что для любых лямбда-термов  $p$  и  $x$  (программы и её данных) существует специализированная программа  $p_x$  такая, что справедливы следующие утверждения.

1. Если  $mix \bar{p} \bar{x}$  имеет  $\beta$ -нормальную форму, то она равна  $\bar{p}_x$ .
2.  $\forall y . p x \equiv_{\beta} p x y$ .

При этом  $\bar{p}$  является представлением (кодировкой) терма  $p$  согласно некоторой фиксированной схеме представления (кодировки) данных. Согласно первому утверждению, частичный вычислитель может быть не всюду определенным — вычисления могут не заканчиваться, если нормальной формы у специализируемой на данных программы не существует. Если вычисления завершаются, то результатом является представление специализированной программы  $p_x$ . Второе условие является “критерием успеха”, т.е. условием корректности специализатора: на любых данных  $y$  специализированная программа  $p_x$  должна иметь то же поведение, что и программа  $p$  на данных  $x$  и  $y$ . Очевидно, определение частичного вычислителя может быть напрямую распространено до произвольного числа аргументов. Аргументы, по которым производится специализация, называются *статическими*, а остальные аргументы — *динамическими*. Частичный вычислитель принимает представления статических аргументов программы, поскольку они могут стать частью специализированной программы. Это свойство является обязательным в случае лямбда-исчисления [96].

Важным в контексте специализации программ является понятие *точками специализации*. Идея соответствующего подхода к специализации заключается в построение программных точек специализированной программы на основе программных точек исходной (специализируемой) программы, объединённых со



*статическими* значениями в этой точке. Это означает, что каждая логическая единица специализируемой программы — программная точка — может быть специализирована несколько раз с учётом *статических* данных, доступных в данной программной точке. Результат каждой такой специализации программной точки образует новую программную точку уже в специализированной программе. При таком подходе программные точки исходной программы, подлежащие специализации, называют *точками специализации*, а получаемые на их основе с учётом значений статических данных их специализированные версии называют *специализированными программными точками*. Примером точек специализации могут служить базовые блоки низкоуровневого императивного языка программирования, а также объявления функций в функциональных языках программирования (см. [68]). Построение оптимального множества точек специализации является нетривиальной задачей, от конкретного решения которой существенно зависит результат самой специализации (см. [68]).

## 5.2. Проекция Футамуры или проекция Футамуры–Ершова–Турчина

Частичный вычислитель называется *самоприменимым*, если он способен специализировать себя самого на некоторую программу. Естественно, это подразумевает, что язык реализации частичного вычислителя не отличается от языка, для которого он написан. Классическим применением самоприменимых частичных вычислителей являются проекции Футамуры, использующие специализацию для компиляции программ, генерации компиляторов и генераторов компиляторов [69]. Пусть даны произвольные языки  $S, T, L_0, L_1, L_2$ , программа  $s$  на языке  $S$ <sup>21</sup>, интерпретатор  $int$  языка  $S$  на языке  $T$ <sup>22</sup> и частичные вычислители  $mix_{L_0}^T$ ,  $mix_{L_1}^{L_0}$  и  $mix_{L_2}^{L_1}$ <sup>23</sup>, тогда<sup>24</sup>:

<sup>21</sup>Мы будем обозначать  $s_S$  программу  $s$ , написанную на языке  $S$ .

<sup>22</sup>Мы будем обозначать  $int_T^S$  интерпретатор  $int$  языка  $s$ , написанный на языке  $T$ .

<sup>23</sup>Мы будем обозначать  $mix_T^S$  частичный вычислитель  $mix$  языка  $s$ , написанный на языке  $T$ .

<sup>24</sup>Запись  $\llbracket s \rrbracket_S$  означает семантику программы  $s$  в языке  $S$ .

1. Первая проекция Футамурь: *специализация интерпретатора на программу суть программа на языке реализации интерпретатора.*

$$\llbracket mix_{L_0}^T(int_T^S, s_S) \rrbracket_T = t_T : \forall x . \llbracket t_T \rrbracket_T x = \llbracket int_T^S \rrbracket_T(s_S, x)$$

Первая проекция Футамурь производит компиляцию программы  $s$ , реализованной на языке  $S$ , в программу  $t$  на языке  $T$  путём специализации интерпретатора для языка  $S$  на языке  $T$ . Сгенерированная и исходная программы  $t$  и  $s$  реализованы на разных языках, но имеют одинаковое поведение на всех входных данных.

2. Вторая проекция Футамурь.

$$\begin{aligned} t_T &= \llbracket mix_{L_0}^T \rrbracket_{L_0} [int_T^S, s_S] && \text{I проекция Футамурь} \\ &= \llbracket \llbracket mix_{L_1}^{L_0} \rrbracket_{L_1} [mix_{L_0}^T, int_T^S] \rrbracket_{L_0} s_S && \text{по определению } mix \\ &= \llbracket comp_{L_0}^{S \rightarrow T} \rrbracket_{L_0} s_S && \text{по определению компилятора} \end{aligned}$$

Вторая проекция Футамурь генерирует компилятор <sup>25</sup> из языка  $S$  в язык  $T$  путём специализации частичного вычислителя  $mix$  на интерпретатор. Для любой программы на языке  $S$  сгенерированный компилятор генерирует программу, обладающую тем же поведением, что программа, сгенерированная первой проекцией Футамурь. Таким образом, вторая проекция Футамурь может быть сформулирована следующим образом: *специализация специализатора на интерпретатор суть компилятор из исходного языка в язык реализации интерпретатора.*

$$comp = \llbracket mix \rrbracket [mix, int]$$

3. Третья проекция Футамурь.

$$\begin{aligned} comp_{L_0}^{S \rightarrow T} &= \llbracket mix_{L_1}^{L_0} \rrbracket_{L_1} [mix_{L_0}^T, int_T^S] && \text{II проекция Футамурь} \\ &= \llbracket \llbracket mix_{L_2}^{L_1} \rrbracket_{L_2} [mix_{L_1}^{L_0}, mix_{L_0}^T] \rrbracket_{L_1} int_T^S && \text{по определению } mix \\ &= \llbracket cogen_{L_1}^T \rrbracket_{L_1} int_T^S && \text{по определению} \end{aligned}$$

<sup>25</sup>По определению компилятором называется такая программа  $comp$ , что  $\llbracket comp_{L_0}^{S \rightarrow T} \rrbracket_{L_0} [s_S] = t_T : \forall d \in \mathfrak{D} . \llbracket s_S \rrbracket_S [d] = \llbracket t_T \rrbracket_T [d]$

Третья проекция Футамуры генерирует генератор компиляторов *cogen* путём специализации частичного вычислителя на другой частичный вычислитель. Тогда по интерпретатору произвольного языка  $S$ , реализованного на языке  $T$ , *cogen* генерирует компилятор из языка  $S$  в язык  $T$ . Таким образом, третья проекция Футамуры может быть сформулирована следующим образом: *специализация специализатора на специализатор суть генератор компиляторов, который по интерпретатору языка генерирует компилятор из интерпретируемого языка в язык реализации интерпретатора.*

$$cogen = \llbracket mix \rrbracket [mix, mix]$$

#### 4. Четвёртая проекция Футамуры.

Изначально Футамура сформулировал только первые две проекции [67]. Ещё одна проекция была сформулирована А.П. Ершовым, а её авторство отдавалось В.Ф. Турчину. В 2009 году R. Glück сформулировал и четвёртую проекцию Футамуры [97]. В действительности четвёртая проекция утверждает, что генератор компиляторов, порождённый оптимальным самоприменимым частичным вычислителем, должен быть способен сгенерировать сам себя.

$$\llbracket cogen \rrbracket mix = cogen$$

Одним из основных свойств частичного вычислителя является то, что оптимальный частичный вычислитель способен устранить накладные расходы, появляющиеся при интерпретации [68]. А именно, частичный вычислитель называется *оптимальным по Джонсу* (Jones optimal), если  $\forall p, \forall d \in \mathcal{D}, t_{p'}(d) \leq t_p(d)$ , где *sint* — самоинтерпретатор<sup>26</sup>,  $p' = \llbracket mix \rrbracket sint p$ , а  $t_p(d)$  обозначает время исполнения программы  $p$  на данных  $d$ .

<sup>26</sup>Самоинтерпретатором называется интерпретатор языка, на котором он сам реализован.

### 5.3. Компиляция нетипизированного лямбда-исчисления путём специализации алгоритма трассирующей нормализации

Итак, частичные вычисления могут быть использованы для специализации алгоритма нормализации на входное лямбда-выражение (см. [68]). Будучи применённым к алгоритму трассирующей нормализации для нетипизированного лямбда-исчисления UNP, частичный вычислитель произведёт компиляцию этого выражения в эквивалентный код на языке низкого уровня, не содержащий следов синтаксиса лямбда-исчисления.

#### 5.3.1. Преобразование нормализующей процедуры в компилятор

Частичные вычисления способны преобразовать нормализующую процедуру  $NP$  нетипизированного лямбда-исчисления (далее ULC) равно как и простого типизированного лямбда-исчисления (далее STLC) в программу, реализующую функцию, сохраняющую семантику:

$$f : ULC \rightarrow LLL \text{ (or } f : STLC \rightarrow LLL).$$

Это утверждение является прямым следствием второй проекции Футамуры, которое может быть записано с помощью диаграмм следующим образом (см. [68]):

$$\text{Если } NP \in \begin{array}{|c|} \hline LC \\ \hline L \\ \hline \end{array}, \text{ то } \llbracket spec \rrbracket (spec, NP) \in \begin{array}{|c|} \hline LC \rightarrow LLL \\ \hline L \\ \hline \end{array}.$$

Здесь LC означает либо ULC, либо STLC, L — язык реализации частичного вычислителя и процедуры нормализации, а LLL — под-язык языка L такой, что его выразительной силы достаточно для представления всех динамических (в смысле частичных вычислений) операций, содержащихся в  $NP$ .

Развитие этого направления и реализация вышеописанного подхода для ряда существующих языков программирования может привести к созданию универсального языка промежуточного представления, предназначенного для описания

семантик языков программирования и автоматической генерации компиляторов, основанной на семантике (semantics-directed compiler generator, дальнейшие детали см. [74, 75]).

### 5.3.2. Специализация UNP на входной лямбда-терм

В процессе специализации нормализующей процедуры UNP на входной лямбда-терм  $M$  частичный вычислитель должен устранить все операции, зависящие только от  $M$ . Таким образом, результат такой специализации,  $UNP_M$ , на входных данных  $d$  не произведёт операций, зависящих от  $M$ . Он будет содержать лишь операции по расширению текущего обхода, а также проверки токенов на синтаксическое равенство, чтобы следовать правилам построения указателей. Заметим, что токены в данном случае используются как метки (имена) и не имеют никакой связи с лямбда-исчислением. Иными словами, подвыражения исходного термина не встречаются в специализированной программе. Множество же токенов является конечным ввиду конечности числа точек специализации, а сами токены используются как метки, и для проверки токенов на равенство, они могут быть заменены, например, на числовые метки.

Для успешной специализации сначала необходимо произвести *анализ времени связывания* (binding time analysis). А именно, разбить всё множество операций нормализующей процедуры UNP, приведённой на рисунке 19, на два вида: статические и динамические. Анализ времени связывания не зависит от входного лямбда-терма  $M$ . Стандартным способом построения такого разбиения является аннотирование кода специализируемой программы (детали могут быть найдены в [68]). Вычисления в UNP помечаются как *динамические*, также называемые *остаточными*, т.е. такие для которых будет сгенерирован остаточный код в  $UNP_M$ , или *статические*, т.е. подлежащие *развёртке* (unfold) во время специализации.

В процессе специализации некоторые динамические по определению переменные могут быть рассмотрены как статические, если множество значений, которые они могут принимать, является конечным. Такие переменные называются *переменными со статически ограниченным множеством значений* (variables of bounded static variation в терминологии из [68]). Например, в случае процедуры

трассирующей нормализации, приведённой на рисунке 19, переменная  $e$  является переменной со статически ограниченным множеством значений: она может принимать только значения подвыражений исходного терма  $M$ , которых для каждого данного терма, конечно, конечное число. Также все логические флаги, очевидно, являются переменными со статически ограниченным множеством значений. Для переменных со статически ограниченным множеством значений в целях улучшения результата специализации в её процессе может быть применён так называемый “The Trick” (см. [68]).

Суть приёма “The Trick” заключается в следующем. Каждое возможное значения переменной со статически ограниченным множеством значений рассматривается как отдельная точка специализации. С целью уменьшения множества возможных значений переменной со статически ограниченным множеством значений в общем случае используются различные эвристики или методы статического анализа. Таким образом, для всех возможных значений такой переменной в специализированной программе будет создана специализированная версия вычислений, в которые эта переменная вовлечена.

*Динамическими* же в случае UNP являются переменные  $bh$  и  $ch$ , представляющие связывающий указатель и указатель потока управления. По определению, каждая переменная, зависящая от других *динамических* переменных, также классифицируется как *динамическая*. Таким образом, *обход* (т.е. история  $h$  с рисунка 19) также является *динамическим*. Также как *динамические* классифицируются и все значения, получаемые на основе *обхода*, например, его размер.

Все рекурсивные вызовы функций в UNP, аргументами и результатом которых не являются подвыражения исходного лямбда-терма  $M$ , также классифицируются как *динамические*. Такой подход, возможно, не является наиболее точным с точки зрения результата специализации, но является общепринятым (см. [68]) и *безопасным* — он позволяет избежать заикливания процесса специализации из-за бесконечного раскрытия вызовов функций. Все остальные вызовы функций раскрываются в процессе специализации.

Результат *анализа времени связывания* алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления приведён на рисунке 26. В “коробочки” (прямоугольники,  $\square$ ) заключена вся информация, классифицируемая как *статическая*, например:

- $\boxed{Exp}$  обозначает статический тип  $Exp$ ;
- Для каждой связанной переменной  $\boxed{lookup\ i\ \alpha}$  обозначает *остаточный* вызов функции, который может быть специализирован для любых значений статических переменных  $i$  и  $\alpha$ ; например, для  $i = 2$  и  $\alpha = T$  *остаточным* вызовом будет  $lookup_{2,T}\ bh\ ch\ h$ ;
- При вызове функции  $lookup$  выражение  $\boxed{i - 1}$  является статическим, а рекурсивные вызовы  $\boxed{lookup\ i\ \alpha}$  внутри функции  $lookup$  являются статическими вызовами функции с двумя статическими переменными; эти вызовы будут раскрыты во время специализации.

Пусть функция  $f$  имеет тип  $\mathcal{D}_1 \rightarrow \mathcal{D}_2 \rightarrow \dots \rightarrow \mathcal{D}_i \rightarrow \mathcal{D}_{i+1} \rightarrow \dots \rightarrow \mathcal{D}_n$ , где  $n$  — общее число аргументов, а  $i$  — количество статических аргументов функции  $f$ . В таком случае тип функции  $f$  будет аннотирован следующим образом:

$$f : \boxed{\mathcal{D}_1 \rightarrow \dots \mathcal{D}_i \rightarrow} \mathcal{D}_{i+1} \rightarrow \dots \rightarrow \mathcal{D}_n.$$

Заметим, что у функций алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления статические параметры всегда образуют некоторый префикс списка всех параметров функции.

В телах определений функций статические вызовы вида  $f\ e_1 \dots e_m$  будут также заключены в прямоугольники, например  $\boxed{f\ e_1 \dots e_m}$ . Если такой вызов функции является ещё и *остаточным*, то вдобавок к выделению прямоугольником он будет подчёркнут, например,  $\underline{\boxed{f\ e_1 \dots e_m}}$ . В случае, если вызов заключён в прямоугольник, но не подчёркнут, он будет раскрыт частичным вычислителем во время специализации.

На рисунке 26 приведена аннотированная версия алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления. Помимо аннотации кода, приведенная версия содержит ещё одно изменение для лучшего разделения времён связывания — тип истории  $H$  развёртывается, а тип  $Item$  местами разбивается на компоненты. До аннотации функция  $eval$  имела тип  $H \multimap H$ , где  $H = [Item]$ . Ниже приведено вышеописанное преобразование типа функции  $eval$



по шагам:

$$\begin{array}{ll}
 eval : H \rightarrow H & \text{за шаг развёртки типа } H \\
 eval : (Item : H) \rightarrow H & \text{применяя каррирование} \\
 eval : Item \rightarrow H \rightarrow H & \text{разбивая тип } Item \text{ на компоненты} \\
 eval : Expr \rightarrow Flag \rightarrow BH \rightarrow CH \rightarrow H \rightarrow H & 
 \end{array}$$

Подобное преобразование позволяет отделить аргументы, классифицируемые как *статические*, т.е.  $e$  и  $\alpha$ , от остальных, *динамических*, аргументов функции. Заметим, что *история*  $H$  всегда является непустым списком, поэтому вышеописанное преобразование является корректным и не требует отдельного рассмотрения случая  $H = []$ .

Приведённая на рисунке 26 аннотированная версия UNP не содержит среди семантических равенств функции *evaloperand*. Аргументы этой функции возникают из *истории*, которая является *динамическим* параметром, и поэтому должны классифицироваться как *динамические*. Тем не менее, аргумент  $e$  является аргументом со *статически ограниченным множеством значений*, а именно, он является подвыражением исходного лямбда-терма  $M$ . Поэтому к функции *evaloperand* применяется “The Trick”. Пусть  $e_1^1 @ e_2^1, \dots, e_1^m @ e_2^m$  — список всех синтаксических применений лямбда-термов терма  $M$ . На рисунке 27 приведён результат применения “The Trick” к функции *evaloperand* и статическому аргументу  $e$ .

$$\begin{array}{l}
 \boxed{evaloperand} \ e_1^1 @ e_2^1 \ \alpha \ bh \ ch \ h = \\
 \quad \boxed{eval \ e_2^1 \ \alpha} \ bh \ ch \ (\langle e_2^1 \ \alpha \ bh \ ch \rangle : h) \\
 \dots \\
 \boxed{evaloperand} \ e_1^m @ e_2^m \ \alpha \ bh \ ch \ h = \\
 \quad \boxed{eval \ e_2^m \ \alpha} \ bh \ ch \ (\langle e_2^m \ \alpha \ bh \ ch \rangle : h)
 \end{array}$$

Рисунок 27. Результат применения “The Trick” к функции *evaloperand* и статическому аргументу  $e$



**Домены:**

$e \in Exp$	= $\lambda$ – выражение
$\alpha \in Flag$	= $\{T, F\}$
$h \in H, ch \in CH, bh \in BH$	= $[Item]$ (История)
$it \in Item$	= $\langle Exp\ Flag\ BH\ CH \rangle$

**Семантические функции:**

$traversal$	: $Exp \rightarrow H$
$eval$	: $Exp \rightarrow Flag \rightarrow BH \rightarrow CH \rightarrow H \rightarrow H$
$apk$	: $Exp \rightarrow CH \rightarrow H \rightarrow H$
$evoperand$	: $Exp \rightarrow Flag \rightarrow H \rightarrow H \rightarrow H \rightarrow H$
$lookup$	: $Int \rightarrow Flag \rightarrow BH \rightarrow CH \rightarrow H \rightarrow H$

**Семантические равенства :** ( $M$  — входной  $\lambda$ -терм)

$$traversal = eval\ M\ F\ []\ []\ [\langle M\ F\ []\ [] \rangle]$$

$$eval\ (FV\ x)\ \_ \_ \_ ch\ h = apk\ (FV\ x)\ ch\ h$$

$$eval\ (BV\ x\ i)\ \alpha\ bh\ ch\ h = lookup\ i\ \alpha\ bh\ ch\ h$$

$$eval\ \lambda x.e\ T\ bh\ ch\ h = apk\ \lambda x.e\ ch\ h$$

$$eval\ \lambda x.e\ F\ bh\ ch\ h = eval\ e\ F\ bh\ ch\ \langle e\ F\ bh\ ch \rangle : h$$

$$eval\ (e_1@e_2)\ \alpha\ bh\ ch\ h = eval\ e_1\ T\ bh\ ch\ \langle e_1\ T\ bh\ ch \rangle : h$$

$$apk\ \_ \ [] \quad h = h$$

$$apk\ \lambda x.e\ (\langle \_ \ \alpha\ bh\ ch' \rangle : \_) \ h = eval\ e\ h\ ch' \langle e\ \alpha\ h\ ch' \rangle : h$$

$$apk\ \_ \ (\langle e\ \alpha\ bh\ ch' \rangle : \_) \ h = evoperand\ e\ F\ bh\ ch' \ h$$

$$lookup\ 0\ \alpha\ (\langle \_ \ T \ \_ \ ch' \rangle : \_) \ ch\ h = \mathbf{case\ ch' \ of}$$

$$\langle e \ \_ \ bh \ \_ \rangle : \_ \Rightarrow evoperand\ e\ \alpha\ bh\ ch\ h$$

$$\_ \Rightarrow$$

**case ch of**

$$[] \Rightarrow h$$

$$\langle ap \ \_ \ bh'' \ ch'' \rangle : \_ \Rightarrow evoperand\ ap\ F\ bh'' \ ch'' \ h$$

$$lookup\ 0 \ \_ \ (\langle \_ \ F \ \_ \ ch' \rangle : h') \ ch\ h = apk\ (BV \ \_ \ 0) \ ch\ h$$

$$lookup\ i\ \alpha\ (\langle \_ \ \_ \ bh' \ \_ \rangle : \_) \ ch\ h = lookup\ (i - 1)\ \alpha\ bh' \ ch\ h$$

Рисунок 26. Аннотированная версия алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления

Таким образом, специализированная программа  $UNP_M$  будет содержать столько специализированных версий функции  $evaloperand$ , сколько операций синтаксического применения термов встречается в исходном терме  $M$ . Соответствующим образом аннотированная версия функции  $evaloperand$  приведена на рисунке 28.

$$\begin{aligned}
 & \boxed{evaloperand} e \alpha bh ch h = \boxed{f M} e \mathbf{where} \\
 & \boxed{f (e' : es)} e = \mathbf{case} e' \mathbf{of} \\
 & \boxed{e_1 @ e_2} \Rightarrow \mathbf{if} e = \boxed{e'} \mathbf{then} \boxed{eval e_2 \alpha} bh ch h \\
 & \quad \quad \quad \mathbf{else} \boxed{f es} e \\
 & \boxed{=} \Rightarrow \boxed{f es} e
 \end{aligned}$$

Рисунок 28. Аннотированная версия функции  $evaloperand$

**Результат специализация UNP на входной лямбда-терм.** Как было описано в предыдущих разделах, при наличии самоприменимого частичного вычислителя на основе интерпретатора языка программирования возможно автоматически сгенерировать его компилятор. Интерпретатором в данном случае является процедура трассирующей нормализации для нетипизированного лямбда-исчисления, реализованная на языке Haskell. Одним из распространённых способов достижения эффекта специализации является использование *генерирующих расширений* вместо частичного вычислителя [65, 68, 98–100]. Именно этот подход и был использован для компиляции лямбда-термов в низкоуровневое представление, LLL. Поскольку алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления был реализован на языке Haskell, то LLL является крошечным подмножеством этого языка, эквивалентным по своей выразительной силе языку F из [2].

Результат специализации процедуры трассирующей нормализации на терм  $mul \vec{2} \vec{2}$  приведён на рисунке 29. Каждая специализированная версия алгоритма трассирующей нормализации на некоторый входной терм  $M$ ,  $UNP_M$ , содержит множество операторов  $apply$ ,  $bind$ ,  $getV$ ,  $\dots$ , которые не зависят от входного терма. Эти функции манипулируют списками указателей *висячих аргументов*  $ch$  и *динамических связываний*  $bh$ . Они могут быть рассмотрены как встроенные операторы языка LLL. Заметим, что в общем случае размер обхода лямбда-

$$\begin{aligned} mul \vec{2} \vec{2} = & (\lambda m. \lambda n. \lambda S. \lambda Z. m @_2 (n @_3 S) @_1 Z) \\ & @_{01} (\lambda s_1. \lambda z_1. s_1 @_4 (s_1 @_5 z_1)) \\ & @_{02} (\lambda s_2. \lambda z_2. s_2 @_6 (s_2 @_7 z_2)) \end{aligned}$$

а) Терм  $mul \vec{2} \vec{2}$

$mul$	$= lm$	$ls1$	$= bind \ lz1$
$lm$	$= bind \ ln$	$lz1$	$= bind \ @4$
$ln$	$= bind \ lS$	$@4$	$= apply \ s1 \ @5$
$lS$	$= bind \ lZ$	$s1$	$= getV \ 1$
$lZ$	$= bind \ @1$	$@5$	$= apply \ s1 \ z1$
$@1$	$= apply \ @2 \ Z$	$z1$	$= getV \ 0$
$@2$	$= apply \ m \ @3$	$ls2$	$= bind \ lz2$
$@3$	$= apply \ n \ S$	$lz2$	$= bind \ @6$
$Z$	$= getV \ 0$	$@6$	$= apply \ s2 \ @7$
$S$	$= getV \ 1$	$s2$	$= getV \ 1$
$n$	$= getV \ 2$	$@7$	$= apply \ s2 \ z2$
$m$	$= getV \ 3$	$z2$	$= getV \ 0$

б) Результат компиляции терма  $mul \vec{2} \vec{2}$  в LLL

Рисунок 29. Результат компиляции программы

терма  $M$  может быть сильно больше его размера [101]. Тем не менее, используя частичные вычисления, результат специализации  $UNP_M$  будет иметь размер  $|UNP_M| = O(|M|)$ . Иными словами, результат компиляции терма  $M$  в LLL имеет размер, линейно зависящий от размера самого терма  $M$ . Исполнение скомпилированной программы с рисунка 29 приведено в приложении А.

## Глава 6

# Трассирующая нормализация и стратегии вычислений

Сложность вычислений в лямбда-исчислении напрямую зависит от выбранной стратегии редукции. Основные стратегии вычислений были описаны в разделе 1.3. Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления в той форме, в которой он был разработан и представлен в разделе 3.5, соответствует стратегии нормального порядка редукции. Стратегия нормального порядка редукции, также как и стратегия вызова по имени, часто проделывает одну и ту же работу несколько раз [102]. В этой главе мы покажем, как алгоритм трассирующей нормализации может быть адаптирован для стратегии аппликативного порядка редукции, а также покажем, как семантика некоторых распространённых функциональных конструкции может быть представлена с помощью подхода трассирующей нормализации на примере языка PCF.

Заметим, что трассирующая нормализация может быть также адаптирована и для иных стратегий вычислений. Так например, в разделе 6.2 описываются особенности адаптации алгоритма трассирующей нормализации для стратегии вызова по значению. Схожим образом<sup>27</sup> трассирующая нормализация может быть определена и для стратегии вызова по необходимости.

---

<sup>27</sup>Для этого потребуется распространить определение обобщённого связывающего указателя с целью перемещения вычисления значения аргументов в место первого вхождения соответствующей связанной переменной.

## 6.1. Алгоритм трассирующей нормализации, соответствующий аппликативному порядку редукции

При аппликативном порядке редукции, также как и при стратегии вызова по значению, редукция функционального аргумента каждого применения редуцируется до абстракции, после чего происходит редукция аргумента, значение которого сохраняется и не перевычисляется в дальнейшем, даже при неоднократном использовании. Более того, результатом вычисления аргумента может быть терм, который уже не является подтермом исходного терма. Таким образом, для адаптации алгоритма трассирующей нормализации к аппликативному порядку редукции необходимо сохранять значения в *обходе* и переиспользовать их в случае необходимости. Также необходим некоторый новый способ ссылаться на токены значений аргументов в процессе переиспользования. Для этого будут введены дополнительные указатели, а также специальные символы  $\boxed{i}^{\uparrow}$  и  $\boxed{i}^{\downarrow}$ , обозначающие начало и конец вычисления значения аргумента с уровнем вложенности  $i$ .

Заметим, что *обход* может содержать множество избыточной информации, которая не является необходимой при повторном использовании аргумента. Например, спинальные редексы могут быть опущены, поскольку они не участвуют в формировании результата (и в том числе значений). Для этого, чтобы избавить обход от излишней информации в тот момент, когда она более не является необходимой, вводится функция *cleanup*, вызываемая в момент окончания вычисления очередного аргумента. Понятие *токена* и *обхода* в случае аппликативного порядка усложняются.

**Определение.** *Токеном* называется либо символ  $\boxed{i}^{\downarrow}$ , либо символ  $\boxed{i}^{\uparrow}$ , либо кортеж  $\langle e, ap, lp, bp, pp, underline, \alpha \rangle$ , где:

- $e$  — некоторая вершина абстрактного синтаксического дерева терма;
- $ap$  — указатель стека висячих аргументов (на диаграммах обозначаются красными округлыми стрелками над обходом);

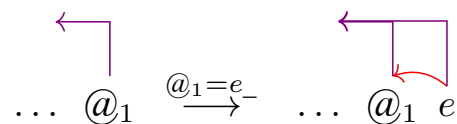
- $lb$  — указатель на последнюю абстракцию перед последним токеном  $\boxed{i}$  (новый; на диаграммах обозначается фиолетовой ломаной стрелкой над обходом);
- $bp$  — связывающий указатель (на диаграммах изображается зелёной круглой стрелкой под обходом);
- $lb$  — указатель на текущий токен аргумента, при его использовании (новый; на диаграммах обозначается чёрной ломаной стрелкой под обходом);
- *underline* — логический флаг, имеющий значением “истина” (*True*), если  $i$  токен не будет участвовать в формировании результата (значения), “ложь” (*False*) — иначе;
- $\alpha$  — логический флаг, имеющий значением “истина” (*True*) если, и только если в текущий момент происходит вычисление некоторого аргумента;
- $\boxed{i}$  и  $\boxed{i}$  обозначают начало и конец вычисления значения аргумента соответственно, а  $i \in \mathbb{N}$  указывает на уровень вложенности.

На диаграммах ниже пунктирным указателем обозначается таковой, значение которого может быть равно *nil*. Иными словами, наличие этого указателя у соответствующего токена не обязательно, но если соответствующий указатель всё же присутствует, то он должен быть согласован с пунктирным указателем.

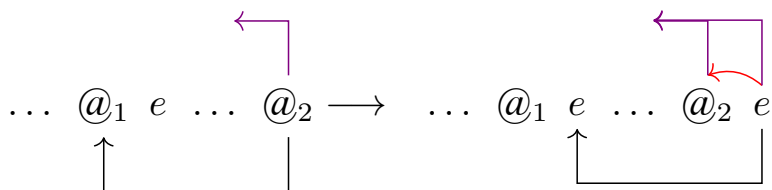
**Определение.** *Обходом* называется список токенов, первым токеном которого является  $\boxed{0}$ , а вторым —  $\langle e, 0, 0, 0, 0, False, False \rangle$ , где  $e$  — корень абстрактного синтаксического дерева исходного терма. Оставшаяся часть обхода строится согласно нижеследующим правилам. Если ни одно из правил не может быть применено, то токен  $\boxed{0}$  добавляется в конец обхода, после чего вызывается функция *cleanup*, которая по данному обходу строит новый обход, являющийся обходом абстрактного синтаксического дерева нормальной формы исходного терма. Напомним, что стратегия аппликативного порядка редукции не является эффективной нормализующей стратегией. Иными словами, как и все алгоритмы нормализации, соответствующие стратегии аппликативного порядка редукции, алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления,

## Применение

- (App-Tree)

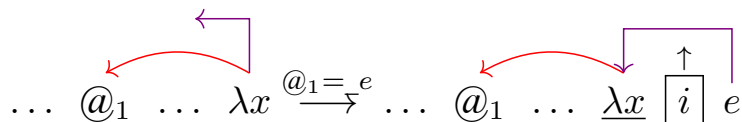


- (App-Argument)

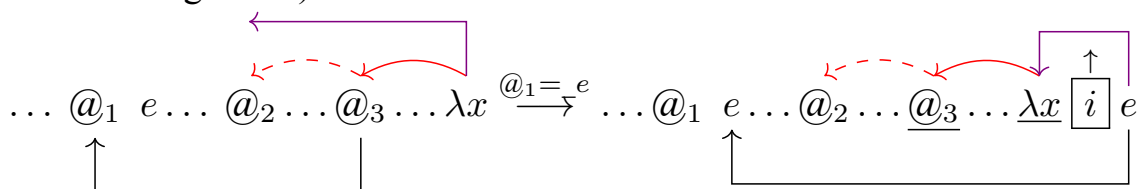


## Абстракция

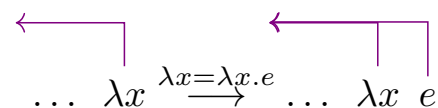
- (Lam-Elim-Tree)



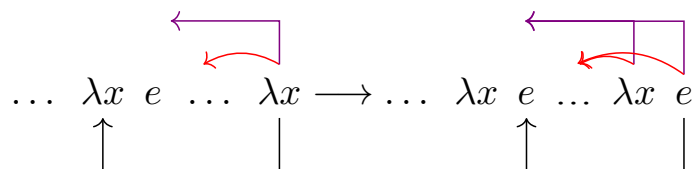
- (Lam-Elim-Argument)



- (Lam-Non-Elim-Tree)



- (Lam-Non-Elim-Argument)



## Связанная переменная

- (BVar)

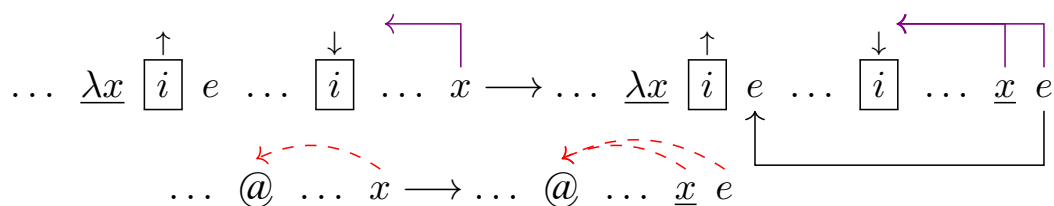


Рисунок 30. Правила переходов для случаев применения, абстракции и связанной переменной

ей соответствующий, может расходиться на некоторых термах, нормальная форма которых существует.

Рассмотрим теперь новые правила построения обхода. Напомним, что после каждого добавления токена  $\boxed{i}$  в обход вызывается функция *cleanup*. Сами правила

## Свободная переменная

- (FVar-I)  $\dots \lambda x \boxed{i} \dots S \rightarrow \dots \lambda x \boxed{i} \dots S \boxed{i} e$
- (Free-II)  $\dots @_1 \dots @_2 \dots \lambda x \boxed{i} \dots S \rightarrow \dots @_1 \dots @_2 \dots \lambda x \boxed{i} \dots S \boxed{i} e$
- (Free-III)  $\dots @_1 e \dots S \boxed{i} \dots @_2 \dots @_3 \dots S \rightarrow \dots @_1 e \dots S \boxed{i} \dots @_2 \dots @_3 \dots S \boxed{i} e$
- (Free-IV)  $\dots \lambda x e \dots S \boxed{i} \dots \lambda x \boxed{j} \dots S \rightarrow \dots \lambda x e \dots S \boxed{i} \dots \lambda x \boxed{j} \dots S \boxed{j} e$
- (Free-V)  $\dots \boxed{i} \dots S e \dots \boxed{i} \dots S \rightarrow \dots \boxed{i} \dots S e \dots \boxed{i} \dots S e$
- (Free-VI)  $\dots @_1 \dots @_2 \dots S \xrightarrow{@_2=e} \dots @_1 \dots @_2 \dots S e$

Рисунок 31. Правила переходов для случая свободной переменной

построения обхода для стратегии аппликативного порядка редукции приведены на рисунках 30 и 31. Для удобства, иногда правила выставления новых указателей разделяются на два.

В случае, если последним токеном обхода является токен-применение, то либо указатель на текущий токен аргумента у этого токена равен *nil*, и в обход добав-



ляется непосредственный первый (левый) аргумент этого токена, либо, если указатель на текущий токен аргумента у него отличен от  $nil$ , обозначим последний за  $e$ , то в обход добавляется новая копия токена  $e$ , а указатель на текущий токен аргумента у новой копии токена  $e$  выставляется на исходный токен  $e$ .

Если текущим последним токеном обхода является токен-абстракция  $\lambda x$ , то существует четыре возможных дальнейших варианта построения обхода. Если указатель на текущий токен аргумента у токена  $@_1$  равен  $nil$  (см. правило (Lam-Elim-Tree) на рис. 30), то в обход добавляется токен  $\boxed{i}$ , где  $i$  является текущим уровнем вложенности, а непосредственно за ним добавляется и токен  $e$ , представляющий аргумент (правого ребёнка) применения  $@_2$ . Токены  $@_1$  и  $\lambda x$  выделяются подчёркиванием, как образующие постой редекс. Если же у токена  $\lambda x$  существует ненулевой указатель стека висячих аргументов, указывающий на токен-применение  $@_3$ , указатель на текущий токен аргумента у которого указывает на некоторый ненулевой токен-применение  $@_1$  (см. правило (Lam-Elim-Argument) на рис. 30), то добавляются токены  $\boxed{i}$  и  $e$ , где  $e$  является токеном, следующим за токеном  $e_1$  в текущем обходе. Токены  $@_3$  и  $\lambda x$  опять же выделяются подчёркиванием, как образующие постой редекс. Если же как указатель на текущий токен аргумента, так и указатель стека висячих аргументов у рассматриваемого токена-абстракции являются нулевыми, то в обход просто добавляется токен, представляющий тело абстракции (его непосредственного ребёнка в смысле АСД; см. правило (Lam-Non-Elim) на рис. 30). Наконец, если указатель на текущий токен аргумента у рассматриваемого токена-абстракции  $\lambda x$  не равен  $nil$ , а указывает на некоторое другое вхождение токена  $\lambda x$  в текущий обход, а также либо  $\alpha(\lambda x) = True$ , либо  $ap(\lambda x) = nil$ , то в обход добавляется токен  $e$ , являющийся копией токена, следующего за этим вхождением токена  $\lambda x$ , а также на него выставляется указатель на текущий токен аргумента у добавляемого в обход токена (см. правило (Lam-Non-Elim-Argument) на рис. 30).

В случае, если последним токеном текущего обхода является токен  $x$ , представляющий вхождение связанной переменной, то применяется правило (BVar), приведённое на рисунке 30. Заметим, что если токен вида  $\lambda x$  недоступен по цепочке связывающих указателей из токена  $x$ , применяются правила для свободной переменной, приведённые на рисунке 31.

Наконец, если последний токен текущего обхода является токеном, представляющим свободную переменную (или такую связанную переменную, токен-абстракция которой не найдена в цепочке связывающих указателей), то применяется одно из шести правил, приведённых на рисунке 31. Неформально говоря, все правила для свободной переменной отвечают за продолжение построения обхода по завершению построения под-обхода, соответствующего вычислению какого-то аргумента, либо какой-то ветки абстрактного синтаксического дерева термина. Отметим, также предполагается, что у токена  $S$  в правиле (FVar-VI) указатель на текущий токен аргумента равен  $nil$ , а в правиле (FVar-I) помимо этого нулевым является ещё и указатель стека висячих аргументов.

**Функция** *cleanup* сжимает *обход*, удаляя из него информацию, которая более не потребуется. В действительности, функция *cleanup* удаляет из обхода все подчёркнутые токены, расположенный между последними парными токенами  $\boxed{i}$  и  $\boxed{i}$ , вместе со всеми под-обходами, заключёнными между токенами  $\boxed{j}$  и  $\boxed{j}$ , включая последние, для всех  $j > i$ . В результате, под-обход, заключённый между токенами  $\boxed{i}$  и  $\boxed{i}$ , образует обход абстрактного синтаксического дерева значения вычисляемого аргумента.

Пример работы системы переходов для алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления, соответствующего аппликативному порядку редукции, приведён в приложении Б.

## 6.2. Об адаптации алгоритма трассирующей нормализации для стратегии вызова по значению

Как уже отмечалось в обзор области исследования (глава 1), стратегия вызова по значению является *слабой* версией стратегии аппликативного порядка редукции. Напомним, что под *слабой стратегией* редукции подразумевается такая стратегия, которая рассматривает абстракцию как значение и не редуцирует её тело. Семантика большого шага нетипизированного лямбда-исчисления, основанная на окружении и соответствующая аппликативному порядку редукции,

приведена на рисунке 32<sup>28</sup>. Соответствующая семантика для стратегии вызова по значению отличается лишь заменой всех  $\Downarrow^{ao}$  на  $\Downarrow^{cbv}$ , а также правилом вычисления значения абстракции — Lam. Соответствующее правило для случая вызова по значению имеет ниже следующий вид<sup>29</sup>.

$$\frac{}{\lambda x.e : \rho \Downarrow^{cbv} \lambda x.e : \rho} \text{ Lam}$$

Заметим, что *значением* с точки зрения описанной семантики вызова по значению является терм в некотором окружении,  $e : \rho$ . Произведение всех необходимых подстановок согласно окружению  $\rho$  приведёт его к слабую нормальную форму исходного терма.

Приведённые семантики большого шага для аппликативного порядка редукции и стратегии вызова по значению, основанные на семантике, не являются *полукомпозиционными*, в смысле определения из раздела 3.2. Именно ввиду отсутствия полукомпозиционности и требуется такое количество дополнительных накладных расходов (указателей в историю) при определении алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления, соответствующего стратегии аппликативного порядка редукции, приведённого в предыдущем разделе.

Свойство полукомпозиционности нарушается из-за правила App — R, приведённого на рисунке 32. Как консеквент применения этого правила может быть сформировано новое выражение  $e'_1 @ e'_2$ , не являющееся подвыражением терма-антецедента  $e'_1 @ e'_2$ . Тем не менее, известно, что семантика вызова по значению, в отличие от семантики аппликативного порядка редукции, в некоторых случаях обладает свойством полукомпозиционности [102, 106].

**Теорема 6.** Семантика вызова по значению для замкнутых<sup>30</sup> строго нормализуемых термов является полукомпозиционной.

*Доказательство.* Теорема может быть доказана индукцией по глубине дерева вывода. Для случаев BVar, FVar, Lam и App — L утверждение теоремы следует

<sup>28</sup>  $\Downarrow^{ao}$  означает шаг редукции согласно стратегии аппликативного порядка редукции.

*Естественная семантика* также известна как *натуральная* или *семантика большого шага* (Natural Semantics, Big-Step Semantics, детали см. в [31, 83, 103–105]).

<sup>29</sup>  $\Downarrow^{cbv}$  означает шаг редукции согласно стратегии вызова по значению.

<sup>30</sup> *Замкнутым* называется лямбда-терм, не содержащий свободных переменных.

$$\begin{array}{c}
\frac{x \in \mathfrak{D}(\rho) \quad e : \rho' = \rho(x)}{x : \rho \Downarrow^{ao} e : \rho'} \text{BVar} \qquad \frac{x \notin \mathfrak{D}(\rho)}{x : \rho \Downarrow^{ao} x : \rho} \text{FVar} \\
\frac{e : \rho \Downarrow^{ao} e' : \rho'}{\lambda x.e : \rho \Downarrow^{ao} \lambda x.e' : \rho'} \text{Lam} \\
\frac{e_1 : \rho' \Downarrow^{ao} \lambda x.e : \rho'' \quad e_2 : \rho'' \Downarrow^{ao} e'_2 : \rho''' \quad e : \rho''[x \mapsto e'_2 : \rho'''] \Downarrow^{ao} e' : \rho'}{e_1 @ e_2 : \rho \Downarrow^{ao} e' : \rho'} \text{App} - \text{L} \\
\frac{e_1 : \rho \Downarrow^{ao} e'_1 : \rho'' \quad e' \not\equiv \lambda x.e \quad e_2 : \rho'' \Downarrow^{ao} e_2' : \rho'}{e_1 @ e_2 : \rho \Downarrow^{ao} e'_1 @ e_2' : \rho'} \text{App} - \text{R}
\end{array}$$

Рисунок 32. Естественная семантика нетипизированного лямбда-исчисления, основанная на окружении и соответствующая аппликативному порядку редукции

напрямую из индукционного предположения. Единственный случай, заслуживающий отдельного внимания, — случай App – R. Но применение этого правила невозможно, если исходный терм замкнут. Базовым значением с точки зрения семантики вызова по значению является либо абстракция, что соответствует правилу App – L, либо переменная, окружение на которой не определено. Такая переменная может быть только свободной, поскольку если переменная является связанной, то либо её абстракция образует редекс, сохранённый в окружении, либо семантика вызова по значению рассматривает саму абстракцию как значение и не редуцирует её тело. Таким образом, единственное правило, нарушающее полуконпозициональность — App – R — неприменимо при редукции замкнутых термов. ■

Теорема 6 позволяет эффективно компилировать процедуру трассирующей нормализации для нетипизированного лямбда-исчисления, соответствующую семантике вызова по значению, для замкнутых лямбда-термов. Отметим, что такая версия алгоритма трассирующей нормализации отличается от приведённой в разделе 6.1 лишь отсутствием правил, отвечающих за редукцию под абстракцией, указателя на текущий токен аргумента, а также правил следования этому указателю.

### 6.3. Трассирующая нормализация для РСФ-подобного языка

Лямбда-исчисление является одним из простейших Тьюринг-полных языков программирования. Тем не менее реализация даже простых функций может вызвать немало трудностей у программиста. Например, все объекты лямбда-исчисления представляют собой термы, поэтому все данные должны быть закодированы в термы согласно некоторому заранее выбранному представлению, а результат должен быть интерпретирован соответствующим образом. Далее мы покажем, как трассирующая нормализация может быть распространена на основные языковые конструкции языка типа РСФ. Рассматриваемый далее язык является надмножеством лямбда-исчисления, расширенного оператором потока управления *if – then – else*, комбинатором неподвижной точки  $Y$ , числовыми константами и бинарными операциями.

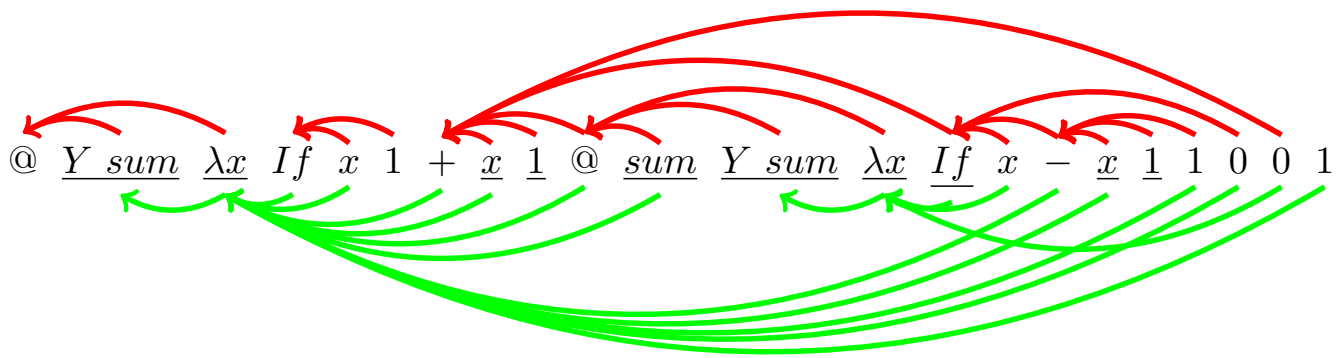
Для обработки дополнительных языковых конструкций функция продолжения *apk* будет принимать ещё один аргумент.

Сначала рассмотрим один из простейших случаев — числовую константу. Подобно случаю свободной переменной, её трассирующая семантика есть вызов функции продолжения *apk*.

Во-вторых, рассмотрим случай комбинатора неподвижной точки  $Y$ . В процессе вычисления этот комбинатор раскрывает тело функции, обновляя окружение (связывающий указатель *bh*) и оставляя текущее продолжение *ch* неизменным. Функция *lookup* также претерпевает изменения: для имён функций она возвращает комбинатор неподвижной точки  $Y$  с исходным окружением, а для имён переменных действует прежним образом. Описанные изменения, очевидно, соответствуют классической семантике комбинатора неподвижной точки:  $\llbracket Y M \rrbracket = \llbracket M(Y M) \rrbracket$ .

Оператор потока управления *if – then – else* также хорошо согласуется с подходом трассирующей нормализации: происходит вычисление тела условия до константы, а затем в функции применения окружения *apk* выбирается соответствующая ветвь вычислений.

Наконец, рассмотрим случай бинарного оператора над значениями. Для простоты мы рассмотрим только функцию сложения, трассирующая семан-

Рисунок 33. Обход терма  $sum\ 1$ 

тика остальных бинарных операторов определяется схожим образом. Существует несколько способов определить семантику бинарных операторов. Первый способ — “синтаксический сахар” над соответствующим лямбда-термом. А именно, сложение может быть определено как псевдоним терма  $\lambda m.\lambda n.\lambda x.(m@f)@((n@f)@x)$ . В случае определения новых синтаксических конструкций таким образом, трассирующая нормализация сохраняет все свои свойства, и в том числе, полукомпозициональность. С другой стороны, такой подход влечёт за собой неэффективность вычислений за счёт того, что результат вычислений не будет сохранён и будет каждый раз перевычисляться в процессе выполнения программы. Другой способ определения трассирующей семантики бинарных операторов — сохранение (простых) значений в обходе подобно тому, как это было сделано в случае аппликативного алгоритма трассирующей нормализации. В данном случае токен, представляющий простое значение (число), может быть *динамическим*, т.е. неизвестным на момент компиляции. Более того, мы позволим функции продолжения изменять аргумент бинарных операций в процессе исполнения, что сделает возможным сохранять промежуточные значения внутри обхода. Заметим, что, сказанное выше применимо лишь для одиночных токенов-значений, представляющих числа, поэтому никаких дополнительных указателей, как в случае аппликативного порядка редукции, не требуется.

Изменения по сравнению с трассирующей семантикой, представленной в разделе 3.5, приведены на рисунке 34. Обход функции  $sum(x) = \sum_{i=1}^x i$ , применённой к аргументу 1:  $Y\ sum\ (\lambda x.\ \underline{if}\ x\ \underline{then}\ x\ +\ sum(x - 1)\ \underline{else}\ 0)$  — приведён на рисунке 33<sup>31</sup>.

<sup>31</sup>Заметим, что  $Y\ sum$  является одним токеном, означающим вызов рекурсивной функции  $sum$ .

```

eval h = let ⟨ e bh α ch ⟩ : _ = h in
  case e of
    ...
    Const n ⇒ apk ch e h α
    Y f e₁ ⇒ eval ⟨ e₁ h α ch ⟩ : h
    If b e₁ e₂ ⇒ eval ⟨ b bh α ch ⟩ : h
    Add e₁ e₂ ⇒ eval ⟨ e₁ bh T h ⟩ : h
apk ch e h α = case ch of
  ...
  ⟨ (If _ e₁ e₂) bh' α ch' ⟩ : _ ⇒
    if e == 0
      then eval ⟨ e₂ bh' α ch' ⟩ : h
      else eval ⟨ e₁ bh' α ch' ⟩ :
  ⟨ (Add e₁ e₂) bh' α ch' ⟩ : ch'' ⇒
    let Const n = e in if α
      then eval ⟨ e₂ bh' F ⟨ (Add e e₂) bh' α ch' ⟩
                  : ch'' ⟩ : h
      else let Const n₁ = e₁
            in eval ⟨ (Const (n₁ + n)) bh' α ch' ⟩ : h
  ⟨ (Pred _ e) bh' α ch' ⟩ : ch'' ⇒
    if e == 0
      then eval ⟨ e₂ bh' F ch'' ⟩ : h
      else eval ⟨ (Const (e₁ - e)) bh' α ch' ⟩ : h
  where ch''' = ⟨ (Sub e e₂) bh α ch' ⟩ : ch''

```

Рисунок 34. Трассирующая семантика новых языковых конструкций

# Заключение

В ходе выполнения данной диссертационной работы были достигнуты следующие результаты.

1. Разработан алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления, соответствующий нормальному порядку редукции.
2. Представлена модель полной головной линейной редукции, являющаяся расширением известной модели головной линейной редукции. Предложенная модель формализована в виде системы переходов, доказана её корректность относительно головной редукции.
3. Доказана корректность представленного алгоритма трассирующей нормализации относительно предложенной модели полной головной линейной редукции путём его формализации в виде системы переходов и дальнейшей симуляции системы переходов для полной головной линейной редукции. Таким образом, доказано, что процедура трассирующей нормализации является нормализующей.
4. Предложенный алгоритм адаптирован для других, отличных от нормального порядка, стратегий вычислений: аппликативного порядка редукции и вызова по необходимости.
5. Предложен новый метод компиляции функциональных языков программирования в низкоуровневое представление путём специализации представленного алгоритма трассирующей нормализации на входной терм.
6. Разработана экспериментальная реализация алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления [107], а также ге-



нерирующего расширения, на ней основанного, для компиляции лямбда-термов в низкоуровневое представление, на языках `Haskell` и `Racket`.

В качестве **рекомендации по применению результатов работы** в индустрии и научных исследованиях указывается, что предложенный алгоритм трассирующей нормализации не преобразует исходный терм, а только посещает его в конечном числе точек, благодаря чему он успешно поддается специализации с помощью хорошо известных методов специализации программ. В частности, применяя вторую проекцию Футамуры–Ершова–Турчина (специализируя специализатор на алгоритм трассирующей нормализации) становится возможной генерация компиляторов для языков программирования.

В качестве **перспектив дальнейшей разработки тематики** может быть отмечено следующее: (1) описание известных языковых конструкций и программных трансформаций с помощью подхода трассирующей нормализации; (2) разработка, на основе предложенного подхода, универсального языка промежуточного представления, который может служить базой для описания семантик широкого класса языков программирования, позволяя автоматически генерировать компиляторы соответствующих языков, а также (3) исследование свойств вычислимости и сложности программ в рамках подхода трассирующей нормализации.

# Литература

1. Мартыненко, Б. К. Языки и трансляции / Б. К. Мартыненко. — Изд-во С.-Петербург. ун-та СПб., 2004.
2. Jones, N. D. Computability and complexity - from a programming perspective / N. D. Jones. Foundations of computing series. — MIT Press, 1997.
3. Michaelson, G. J. An introduction to functional programming through lambda calculus / G. J. Michaelson. International computer science series. — Addison-Wesley, 1989.
4. Church, A. A Formulation of the Simple Theory of Types / A. Church // J. Symb. Log. — 1940. — Vol. 5, no. 2. — P. 56–68.
5. Plotkin, G. D. LCF Considered as a Programming Language / G. D. Plotkin // Theor. Comput. Sci. — 1977. — Vol. 5, no. 3. — P. 223–255.
6. Ong, C.-H. L. Normalisation by Traversals / C.-H. L. Ong // CoRR. — 2015. — T. abs/1511.02629. — URL: <http://arxiv.org/abs/1511.02629> (дата обращения: 07.12.2017).
7. Blum, W. The safe lambda calculus / W. Blum // Ph.D. thesis, University of Oxford, UK. — 2009.
8. Danos, V. Head linear reduction / V. Danos, L. Regnier // unpublished. — 2004. — URL: <https://pdfs.semanticscholar.org/e2ce/424ff63ac58b7b1737b13166fda9a96f3dda.pdf> (дата обращения: 07.12.2017).
9. Danos, V. Game Semantics & Abstract Machines / V. Danos, H. Herbelin, L. Regnier // Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science,

- New Brunswick, New Jersey, USA, July 27-30, 1996. — IEEE Computer Society, 1996. — P. 394–405.
10. Fredriksson, O. Abstract Machines for Game Semantics, Revisited / O. Fredriksson, D. R. Ghica // 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. — IEEE Computer Society, 2013. — P. 560–569.
  11. Ong, C.-H. L. On Model-Checking Trees Generated by Higher-Order Recursion Schemes / C.-H. L. Ong // 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings. — IEEE Computer Society, 2006. — P. 81–90.
  12. Neatherway, R. P. A traversal-based algorithm for higher-order model checking / R. P. Neatherway, S. J. Ramsay, C.-H. L. Ong // ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012 / Ed. by P. Thiemann, R. B. Findler. — ACM, 2012. — P. 353–364.
  13. Lévy, J.-J. Optimal reductions in the lambda-calculus / J.-J. Lévy // In To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. — Academic Press, 1980.
  14. Lamping, J. An Algorithm for Optimal Lambda Calculus Reduction / J. Lamping // Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990. — 1990. — P. 16–30.
  15. Shivers, O. Bottom-Up beta-Reduction: Uplinks and lambda-DAGs / O. Shivers, M. Wand // Lecture Notes in Computer Science, Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings / Ed. by S. Sagiv. — Vol. 3444. — Springer, 2005. — P. 217–232.

16. van Oostrom, V. Lambdascope another optimal implementation of the lambda-calculus / V. van Oostrom, K.-J. van de Looij, M. Zwitserlood // In Workshop on Algebra and Logic on Programming Systems (ALPS). — 2004.
17. Baillot, P. Elementary Complexity and Geometry of Interaction / P. Baillot, M. Pedicini // Fundam. Inform. — 2001. — Vol. 45, no. 1-2. — P. 1–31.
18. Lafont, Y. Interaction Nets / Y. Lafont // Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990. — 1990. — P. 95–108.
19. Mascari, G. Head Linear Reduction and Pure Proof Net Extraction / G. Mascari, M. Pedicini // Theor. Comput. Sci. — 1994. — Vol. 135, no. 1. — P. 111–137.
20. Heijltjes, W. Proof Nets for Additive Linear Logic with Units / W. Heijltjes // Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada. — 2011. — P. 207–216.
21. Church, A. Correction to *A Note on the Entscheidungsproblem* / A. Church // J. Symb. Log. — 1936. — Vol. 1, no. 3. — P. 101–102.
22. Church, A. A Note on the Entscheidungsproblem / A. Church // J. Symb. Log. — 1936. — Vol. 1, no. 1. — P. 40–41.
23. Newman, M. H. A. A Formal Theorem in Church's Theory of Types / M. H. A. Newman, A. M. Turing // J. Symb. Log. — 1942. — Vol. 7, no. 1. — P. 28–33.
24. Turing, A. M. Computability and  $\lambda$ -Definability / A. M. Turing // J. Symb. Log. — 1937. — Vol. 2, no. 4. — P. 153–163.
25. Turing, A. M. The  $p$ -Function in  $\lambda$ -K-Conversion / A. M. Turing // J. Symb. Log. — 1937. — Vol. 2, no. 4. — P. 164.
26. Curry, H. B. Consistency and Completeness of the Theory of Combinators / H. B. Curry // J. Symb. Log. — 1941. — Vol. 6, no. 2. — P. 54–61.
27. Curry, H. B. The Inconsistency of Certain Formal Logic / H. B. Curry // J. Symb. Log. — 1942. — Vol. 7, no. 3. — P. 115–117.

28. Curry, H. B. The Combinatory Foundations of Mathematical Logic / H. B. Curry // J. Symb. Log. — 1942. — Vol. 7, no. 2. — P. 49–64.
29. Curry, H. B. The Permutability of Rules in the Classical Inferential Calculus / H. B. Curry // J. Symb. Log. — 1952. — Vol. 17, no. 4. — P. 245–248.
30. Barendregt, H.P. The Lambda Calculus: Its Syntax and Semantics / H.P. Barendregt. Studies in Logic and the Foundations of Mathematics. — Elsevier Science, 2013.
31. Pierce, B. C. Advanced Topics in Types and Programming Languages / B. C. Pierce. — The MIT Press, 2004.
32. Scott, D. S. Domains for Denotational Semantics / D. S. Scott // Lecture Notes in Computer Science, Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings / Ed. by M. Nielsen, E. M. Schmidt. — Vol. 140. — Springer, 1982. — P. 577–613.
33. Scott, D. S. Logic and Programming Languages / D. S. Scott // Commun. ACM. — 1977. — Vol. 20, no. 9. — P. 634–641.
34. Scott, D. S. Data Types as Lattices / D. S. Scott // SIAM J. Comput. — 1976. — Vol. 5, no. 3. — P. 522–587.
35. Scott, D. S. Combinators and classes / D. S. Scott // Lecture Notes in Computer Science, Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975 / Ed. by C. Böhm. — Vol. 37. — Springer, 1975. — P. 1–26.
36. Hennessy, M. Full Abstraction for a Simple Parallel Programming Language / M. Hennessy, G. D. Plotkin // Lecture Notes in Computer Science, Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979 / Ed. by J. Becvár. — Vol. 74. — Springer, 1979. — P. 108–120.
37. Lorenzen, P. Algebraische und Logistische Untersuchungen Über Freie Verbände / P. Lorenzen // J. Symb. Log. — 1951. — Vol. 16, no. 2. — P. 81–106.

38. Lorenz, K. Rules versus theorems / K. Lorenz // *J. Philosophical Logic*. — 1973. — Vol. 2, no. 3. — P. 352–369.
39. Hyland, J. M. E. On Full Abstraction for PCF: I, II, and III / J. M. E. Hyland, C.-H. L. Ong // *Inf. Comput.* — 2000. — Vol. 163, no. 2. — P. 285–408.
40. Abramsky, S. Full Abstraction for PCF / S. Abramsky, P. Malacaria, R. Jagadeesan // *Lecture Notes in Computer Science, Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings* / Ed. by M. H., J. C. Mitchell. — Vol. 789. — Springer, 1994. — P. 1–15.
41. Abramsky, S. Game semantics / S. Abramsky, G. McCusker // *In Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*. — Springer, 1999. — P. 1–56.
42. Ker, A. D. Innocent game models of untyped lambda-calculus / A. D. Ker, H. Nickau, C.-H. L. Ong // *Theor. Comput. Sci.* — 2002. — Vol. 272, no. 1-2. — P. 247–292.
43. Ker, A. D. A Universal Innocent Game Model for the Böhm Tree Lambda Theory / A. D. Ker, H. Nickau, C.-H. L. Ong // *Lecture Notes in Computer Science, Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings* / Ed. by J. Flum, M. Rodríguez-Artalejo. — Vol. 1683. — Springer, 1999. — P. 405–419.
44. Blum, W. A concrete presentation of game semantics / W. Blum, C.-H. L. Ong // *In Galop 2008: Games for Logic and Programming Languages*. — 2008.
45. Ghica, D. R. The regular-language semantics of second-order idealized  $\Lambda_{\text{LGOL}}$  / D. R. Ghica, G. McCusker // *Theor. Comput. Sci.* — 2003. — Vol. 309, no. 1-3. — P. 469–502.
46. Ghica, D. R. Semantical Analysis of Specification Logic, 3: An Operational Approach / D. R. Ghica // *Lecture Notes in Computer Science, Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*,

- Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings / Ed. by D. A. Schmidt. — Vol. 2986. — Springer, 2004. — P. 264–278.
47. Ghica, D. R. A System-Level Game Semantics / D. R. Ghica, N. Tzevelekos // *Electr. Notes Theor. Comput. Sci.* — 2012. — Vol. 286. — P. 191–211.
  48. Ghica, D. R. Applications of Game Semantics: From Program Analysis to Hardware Synthesis / D. R. Ghica // *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA.* — IEEE Computer Society, 2009. — P. 17–26.
  49. Gabbay, M. Game Semantics in the Nominal Model / M. Gabbay, D. R. Ghica // *Electr. Notes Theor. Comput. Sci.* — 2012. — Vol. 286. — P. 173–189.
  50. Blum, W. The Safe Lambda Calculus / W. Blum, C.-H. L. Ong // *Lecture Notes in Computer Science, Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings* / Ed. by S. R. D. Rocca. — Vol. 4583. — Springer, 2007. — P. 39–53.
  51. Danos, V. Directed Virtual Reductions / V. Danos, M. Pedicini, L. Regnier // *Lecture Notes in Computer Science, Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers* / Ed. by D. van Dalen, M. Bezem. — Vol. 1258. — Springer, 1996. — P. 76–88.
  52. Blum, W. The Safe Lambda Calculus / W. Blum, C.-H. L. Ong // *Logical Methods in Computer Science.* — 2009. — Vol. 5, no. 1.
  53. Barendregt, H. P. Lambda Calculus with Types / H. P. Barendregt, W. Dekkers, R. Statman. *Perspectives in logic.* — Cambridge University Press, 2013.
  54. Barendregt, H. Self-Interpretations in lambda Calculus / H. Barendregt // *J. Funct. Program.* — 1991. — Vol. 1, no. 2. — P. 229–233.
  55. Barendregt, H. P. Functional Programming and Lambda Calculus / H. P. Barendregt // *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B).* — 1990. — P. 321–363.

56. Barendregt, H. Types in Lambda Calculi and Programming Languages / H. Barendregt, K. Hemerik // Lecture Notes in Computer Science, ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings / Ed. by N. D. Jones. — Vol. 432. — Springer, 1990. — P. 1–35.
57. Keller, R. M. Formal Verification of Parallel Programs / R. M. Keller // Commun. ACM. — 1976. — Vol. 19, no. 7. — P. 371–384.
58. Cabodi, G. Computing Timed Transition Relations for Sequential Cycle-Based Simulation / G. Cabodi, P. Camurati, C. Passerone, S. Quer // 1999 Design, Automation and Test in Europe (DATE '99), 9-12 March 1999, Munich, Germany. — IEEE Computer Society / ACM, 1999. — P. 8–12.
59. Reynolds, J. C. Definitional Interpreters for Higher-Order Programming Languages / J. C. Reynolds // Higher-Order and Symbolic Computation. — 1998. — Vol. 11, no. 4. — P. 363–397.
60. Reynolds, J. C. Definitional Interpreters Revisited / J. C. Reynolds // Higher-Order and Symbolic Computation. — 1998. — Vol. 11, no. 4. — P. 355–361.
61. Reynolds, J. C. The design, definition and implementation of programming languages / J. C. Reynolds // ACM SIGSOFT Software Engineering Notes. — 2000. — Vol. 25, no. 1. — P. 75.
62. Haskell language documentation [Электронный ресурс]. — URL: <https://www.haskell.org/documentation> (дата обращения: 03.12.2017).
63. Racket language documentation [Электронный ресурс]. — URL: <https://docs.racket-lang.org/> (дата обращения: 03.12.2017).
64. Romanenko, S. A. A compiler generator produced by a self-applicable specializer can have a suprisingly natural and understandable structure / S. A. Romanenko // A.P. Ershov, D. Bjorner and N.D. Jones, editors, Partial Evaluation and Mixed Computation. — 1988. — P. 445–463.
65. Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998 / Ed. by J. Hatcliff, T. Æ. Mogensen, P. Thiemann. — Vol. 1706, Springer, 1999.



66. Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009 / Ed. by G. Puebla, G. Vidal. — ACM, 2009.
67. Futamura, Y. Partial evaluation of computation process – an approach to a compiler-compiler / Y. Futamura // *Systems, Computers, Controls*. — 1971. — Vol. 2(5). — P. 45–50.
68. Jones, N. D. Partial evaluation and automatic program generation / N. D. Jones, C. K. Gomard, P. Sestoft. Prentice Hall international series in computer science. — Prentice Hall, 1993.
69. Futamura, Y. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler / Y. Futamura // *Higher-Order and Symbolic Computation*. — 1999. — Vol. 12, no. 4. — P. 381–391.
70. Kahn, K. M. Partial Evaluation, Programming Methodology, and Artificial Intelligence / K. M. Kahn // *AI Magazine*. — 1984. — Vol. 5, no. 1. — P. 53–57.
71. Jones, N. D. An Experiment in Partial Evaluation: The Generation of a Compiler Generator / N. D. Jones, P. Sestoft, H. Søndergaard // *Lecture Notes in Computer Science, Rewriting Techniques and Applications, First International Conference, RTA-85, Dijon, France, May 20-22, 1985, Proceedings* / Ed. by J.-P. Jouannaud. — Vol. 202. — Springer, 1985. — P. 124–140.
72. Mahr, B. Term Evaluation in Partial Algebras / B. Mahr // *ADT*. — 1986.
73. Berezun, D. Compiling untyped lambda calculus to lower-level code by game semantics and partial evaluation (invited paper) / D. Berezun, N. D. Jones // *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017* / Ed. by U. P. Schultz, J. Yallop. — ACM, 2017. — P. 1–11.
74. Schmidt, D. A. State transition machines for lambda calculus expressions / D. A. Schmidt // *Lecture Notes in Computer Science, Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980* / Ed. by N. D. Jones. — Vol. 94. — Springer, 1980. — P. 415–440.

75. Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980 / Ed. by N. D. Jones. — Vol. 94, Springer, 1980.
76. Diehl, S. Natural Semantics-Directed Generation of Compilers and Abstract Machines / S. Diehl // Formal Asp. Comput. — 2000. — Vol. 12, no. 2. — P. 71–99.
77. Diehl, S. Semantics-directed generation of compilers and abstract machines / S. Diehl // Ph.D. thesis, Saarland University, Saarbrücken, Germany. — 1996.
78. Hannan, J. Operational Semantics-Directed Compilers and Machine Architectures / J. Hannan // ACM Trans. Program. Lang. Syst. — 1994. — Vol. 16, no. 4. — P. 1215–1247.
79. Paulson, L. C. A Semantics-Directed Compiler Generator / L. C. Paulson // Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982 / Ed. by R. A. DeMillo. — ACM Press, 1982. — P. 224–233.
80. Berezun, D. Precise Garbage Collection for C++ with a Non-cooperative Compiler / D. Berezun, D. Boulytchev // CEE-SECR '14, Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — CEE-SECR '14. — New York, NY, USA: ACM, 2014. — P. 15:1–15:8.
81. de Bruijn, N. G. Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem / N. G. de Bruijn // Indagationes Mathematicae, Elsevier. — 1972. — P. 34: 381–392.
82. Kamareddine, F. Bridging Curry and Church's typing style / F. Kamareddine, J. P. Seldin, J. B. Wells // J. Applied Logic. — 2016. — Vol. 18. — P. 42–70.
83. Pierce, B. C. Types and programming languages / B. C. Pierce. — MIT Press, 2002.
84. Liquori, L. Intersection-types à la Church / L. Liquori, S. R. D. Rocca // Inf. Comput. — 2007. — Vol. 205, no. 9. — P. 1371–1386.

85. Farmer, W. M. A Partial Functions Version of Church's Simple Theory of Types / W. M. Farmer // *J. Symb. Log.* — 1990. — Vol. 55, no. 3. — P. 1269–1291.
86. Sestoft, P. Demonstrating Lambda Calculus Reduction / P. Sestoft // *Lecture Notes in Computer Science, The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]* / Ed. by T. Æ. Mogensen, D. A. Schmidt, I. H. Sudborough. — Vol. 2566. — Springer, 2002. — P. 420–435.
87. Huet, G. P. Regular Böhm trees / G. P. Huet // *Mathematical Structures in Computer Science.* — 1998. — Vol. 8, no. 6. — P. 671–680.
88. Wadsworth, C. P. Semantics and Pragmatics of the Lambda-Calculus / C. P. Wadsworth // Ph.D. thesis, University of Oxford, UK. — 1971.
89. Sinot, F.-R. Complete Laziness: a Natural Semantics / F.-R. Sinot // *Electr. Notes Theor. Comput. Sci.* — 2008. — Vol. 204. — P. 129–145.
90. Launchbury, J. A Natural Semantics for Lazy Evaluation / J. Launchbury // *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993.* — 1993. — P. 144–154.
91. Abramsky, S. Full Abstraction in the Lazy Lambda Calculus / S. Abramsky, C.-H. L. Ong // *Inf. Comput.* — 1993. — Vol. 105, no. 2. — P. 159–267.
92. Blum, W. Imaginary Traversals for the Untyped Lambda Calculus (ongoing work) / W. Blum. — 2017.
93. Jones, N. D. Transformation by interpreter specialisation / N. D. Jones // *Sci. Comput. Program.* — 2004. — Vol. 52. — P. 307–339.
94. Березун, Д. А. Трассирующая нормализация нетипизированного лямбда-исчисления / Д. А. Березун // *Известия вузов. Северо Кавказский регион. Технические науки.* — 2017. — № 4. — С. 5–12.

95. Березун, Д. А. Полная головная линейная редукция / Д. А. Березун // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. — 2017. — Т. 10, № 3. — С. 59–82.
96. Launchbury, J. A Strongly-Typed Self-Applicable Partial Evaluator / J. Launchbury // Lecture Notes in Computer Science, Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings / Ed. by J. Hughes. — Vol. 523. — Springer, 1991. — P. 145–164.
97. Glück, R. Is there a fourth Futamura projection? / R. Glück // Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009 / Ed. by G. Puebla, G. Vidal. — ACM, 2009. — P. 51–60.
98. Gomard, C. K. Partial Type Inference for Untyped Functional Programs / C. K. Gomard // LISP and Functional Programming. — 1990. — P. 282–287.
99. Bondorf, A. Improving Binding Times Without Explicit CPS-Conversion / A. Bondorf // LISP and Functional Programming. — 1992. — P. 1–10.
100. Birkedal, L. Hand-Writing Program Generator Generators / L. Birkedal, M. Welinder // Lecture Notes in Computer Science, Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94, Madrid, Spain, September 14-16, 1994, Proceedings / Ed. by M. V. Hermenegildo, J. Penjam. — Vol. 844. — Springer, 1994. — P. 198–214.
101. Statman, R. The Typed lambda-Calculus Is not Elementary Recursive / R. Statman // 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. — IEEE Computer Society, 1977. — P. 90–94.
102. Jones, N. D. Call-by-Value Termination in the Untyped lambda-Calculus / N. D. Jones, N. Bohr // Logical Methods in Computer Science. — 2008. — Vol. 4, no. 1.

103. Kahn, G. Natural Semantics / G. Kahn // Lecture Notes in Computer Science, STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings / Ed. by F.-J. Brandenburg, G. Vidal-Naquet, M. Wirsing. — Vol. 247. — Springer, 1987. — P. 22–39.
104. Plotkin, G. D. A structural approach to operational semantics / G. D. Plotkin // J. Log. Algebr. Program. — 2004. — Vol. 60-61. — P. 17–139.
105. Hennessy, M. Semantics of programming languages - an elementary introduction using structural operational semantics / M. Hennessy. — Wiley, 1990.
106. Grégoire, B. A compiled implementation of strong reduction / B. Grégoire, X. Leroy // Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002. / Ed. by M. Wand, S. L. P. Jones. — ACM, 2002. — P. 235–246.
107. Экспериментальная реализация алгоритма трассирующей нормализации нетипизированного лямбда-исчисления [Электронный ресурс]. — URL: <https://github.com/danyaberezun/traversal-Based-Normalization> (дата обращения: 10.12.2017).

# Список рисунков

1	Именованное представление чистого лямбда-исчисления . . . . .	16
2	Неименованное представление чистого лямбда-исчисления . . . . .	17
3	$\eta$ -конверсия . . . . .	17
4	Простое типизированное лямбда-исчисление в стиле Карри . . . . .	18
5	Простое типизированное лямбда-исчисление в стиле Чёрча . . . . .	19
6	Пример: деревья Бёма для термов, не имеющих нормальной формы	24
7	Терм $mul \vec{2} \vec{2}$ , его $\eta$ -длинная форма и её АСД . . . . .	26
8	Определение списков головных абстракций и простых редексов . . .	27
9	Система переходов для головной линейной редукции . . . . .	35
10	Поведение системы переходов для головной линейной редукции на примере терма $(\lambda x . x) @ (\lambda y . y)$ . . . . .	36
11	Иллюстрация к теореме 2 . . . . .	39
12	Терм $M[\lambda y]$ . . . . .	42
13	Система переходов для полной головной линейной редукции . . . . .	44
14	Слабая редукция . . . . .	47
15	Строгая редукция . . . . .	47
16	Строгая редукция, основанная на окружении . . . . .	48
17	Строгая хвосто-рекурсивная редукция . . . . .	49
18	Семантика, основанная на истории и окружении . . . . .	51
19	Алгоритм трассирующей нормализации для нетипизированного лямбда-исчисления . . . . .	53
20	АСД терма $mul \vec{2} \vec{2}$ . . . . .	54
21	Результат работы UNP на терме $mul \vec{2} \vec{2}$ . . . . .	56
22	Система переходов для BUNP . . . . .	58
23	Иллюстрация к Теореме 4 . . . . .	62

24	Правила переходов для UNP . . . . .	66
25	Иллюстрация к Теореме 5 . . . . .	69
27	Результат применения “The Trick” к функции <i>evaloperand</i> и статическому аргументу <i>e</i> . . . . .	80
26	Аннотированная версия алгоритма трассирующей нормализации для нетипизированного лямбда-исчисления . . . . .	81
28	Аннотированная версия функции <i>evaloperand</i> . . . . .	82
29	Результат компиляции программы . . . . .	83
30	Правила переходов для случаев применения, абстракции и связанной переменной . . . . .	87
31	Правила переходов для случая свободной переменной . . . . .	88
32	Естественная семантика нетипизированного лямбда-исчисления, основанная на окружении и соответствующая аппликативному порядку редукции . . . . .	92
33	Обход терма <i>sum 1</i> . . . . .	94
34	Трассирующая семантика новых языковых конструкций . . . . .	95

# Список таблиц

1	Нормальные формы . . . . .	20
---	----------------------------	----



# Приложение А. Пример исполнения программ на LLL

В приложении представлено исполнение программы  $mul \vec{2} \vec{2}$  на языке LLL. Исполнение состоит из вызовов функций (например,  $lz1 = bind @4$ ), между которыми приведены текущие значения динамических стеков, представляющих соответствующие указатели алгоритма трассирующей нормализации.

```

    []
    []
@01 = apply @02 data2
    [(data2, [])]
    []
@02 = apply mul data1
    [(data2, []), (data1, [])]
    []
lm = bind ln
    [(data2, [])]
    [(data1, [])]
ln = bind lS
    []
    [(data2, []), (data1, [])]
lS = bind lZ
    []
    [bot, (data2, []), (data1, [])]
lZ = bind @1
    []

```

```

    [bot, bot, (data2, []), (data1, [])]
@1 = apply @2 Z
    [(Z, [bot, bot, (data2, []), (data1, [])])]
    [bot, bot, (data2, []), (data1, [])]
@2 = apply m @3
    [(@3, [bot, bot, (data2, []), (data1, [])]), (Z, [bot,
    ↪ bot, (data2, []), (data1, [])])]
    [bot, bot, (data2, []), (data1, [])]
m = getV 3
    [(@3, [bot, bot, (data2, []), (data1, [])]), (Z, [bot,
    ↪ bot, (data2, []), (data1, [])])]
    []
data1 = ls1 = bind lz1
    [(Z, [bot, bot, (data2, []), (data1, [])])]
    [(@3, [bot, bot, (data2, []), (data1, [])])]
lz1 = bind @4
    []
    [(Z, [bot, bot, (data2, []), (data1, [])]), (@3, [bot,
    ↪ bot, (data2, []), (data1, [])])]
@4 = apply s1 @5
    [(@5, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
    ↪ , [bot, bot, (data2, []), (data1, [])])])]
    [(Z, [bot, bot, (data2, []), (data1, [])]), (@3, [bot,
    ↪ bot, (data2, []), (data1, [])])]
s1 = getV 1
    [(@5, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
    ↪ , [bot, bot, (data2, []), (data1, [])])])]
    [bot, bot, (data2, []), (data1, [])]
@3 = apply n S
    [(S, [bot, bot, (data2, []), (data1, [])]),
    (@5, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
    ↪ , [bot, bot, (data2, []), (data1, [])])])]
    [bot, bot, (data2, []), (data1, [])]

```

```

n = getV 2
  [(S,[bot, bot, (data2,[]), (data1,[])])],
  (@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
    ↪ , [bot, bot, (data2,[]), (data1,[])])])])
  []
data2 = ls2 = bind lz2
  [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
    ↪ , [bot, bot, (data2,[]), (data1,[])])])])
  [(S,[bot, bot, (data2,[]), (data1,[])])]
lz2 = bind @6
  []
  [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
    ↪ , [bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])]
@6 = apply s2 @7
  [(@7, [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])])
    ↪ , (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])])
  [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
    ↪ , [bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])]
s2 = getV 1
  [(@7, [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])])
    ↪ , (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])])
  [bot, bot, (data2,[]), (data1,[])]
S = getV 1
  [(@7, [(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])])
    ↪ , (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])])
  []
bot = continue
  []

```

```

[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
  ↪ ,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])]
@7 = apply s2 z2
[(z2,[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])],
  ↪ (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])]
[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
  ↪ ,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])]
s2 = getV 1
[(z2,[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])],
  ↪ (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])]
[bot, bot, (data2,[]), (data1,[])]
S = getV 1
[(z2,[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])],
  ↪ (@3,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])])]
[]
bot = continue
[]
[(@5, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
  ↪ ,[bot, bot, (data2,[]), (data1,[])])]),
  (S,[bot, bot, (data2,[]), (data1,[])])]
z2 = getV 0
[]
[(Z,[bot, bot, (data2,[]), (data1,[])])], (@3,[bot,
  ↪ bot, (data2,[]), (data1,[])])]
@5 = apply s1 z1
[(z1, [(Z,[bot, bot, (data2,[]), (data1,[])])], (@3
  ↪ ,[bot, bot, (data2,[]), (data1,[])])])]

```

```

    [(Z, [bot, bot, (data2, []), (data1, [])]), (@3, [bot,
      ↪ bot, (data2, []), (data1, [])])]
s1 = getV 1
    [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])])],
    [bot, bot, (data2, []), (data1, [])]
@3 = apply n S
    [(S, [bot, bot, (data2, []), (data1, [])]),
    (z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])])],
    [bot, bot, (data2, []), (data1, [])]
n = getV 2
    [(S, [bot, bot, (data2, []), (data1, [])]),
    (z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])])],
    []
data2 = ls1 = bind lz1
    [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])])],
    [(S, [bot, bot, (data2, []), (data1, [])])]
lz1 = bind @4
    []
    [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])]),
    (S, [bot, bot, (data2, []), (data1, [])])]
@4 = apply s1 @5
    [(@5, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]),
      ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])],
    [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
      ↪ , [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])]
s1 = getV 1

```

```

    [(@5, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])])
      ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])])
  [bot, bot, (data2, []), (data1, [])]
S = getV 1
  [(@5, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])])
    ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])])
  []
bot = continue
  []
  [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
    ↪ , [bot, bot, (data2, []), (data1, [])])]),
    (S, [bot, bot, (data2, []), (data1, [])])])
@5 = apply s1 z1
  [(z1, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])])
    ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])])
  [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
    ↪ , [bot, bot, (data2, []), (data1, [])])]),
    (S, [bot, bot, (data2, []), (data1, [])])])
s1 = getV 1
  [(z1, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])])
    ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])])
  [bot, bot, (data2, []), (data1, [])]
S = getV 1
  [(z1, [(z1, [(Z, [bot, bot, (data2, []), (data1, [])])
    ↪ , (@3, [bot, bot, (data2, []), (data1, [])])]),
      (S, [bot, bot, (data2, []), (data1, [])])])])
  []
bot = continue
  []

```

```

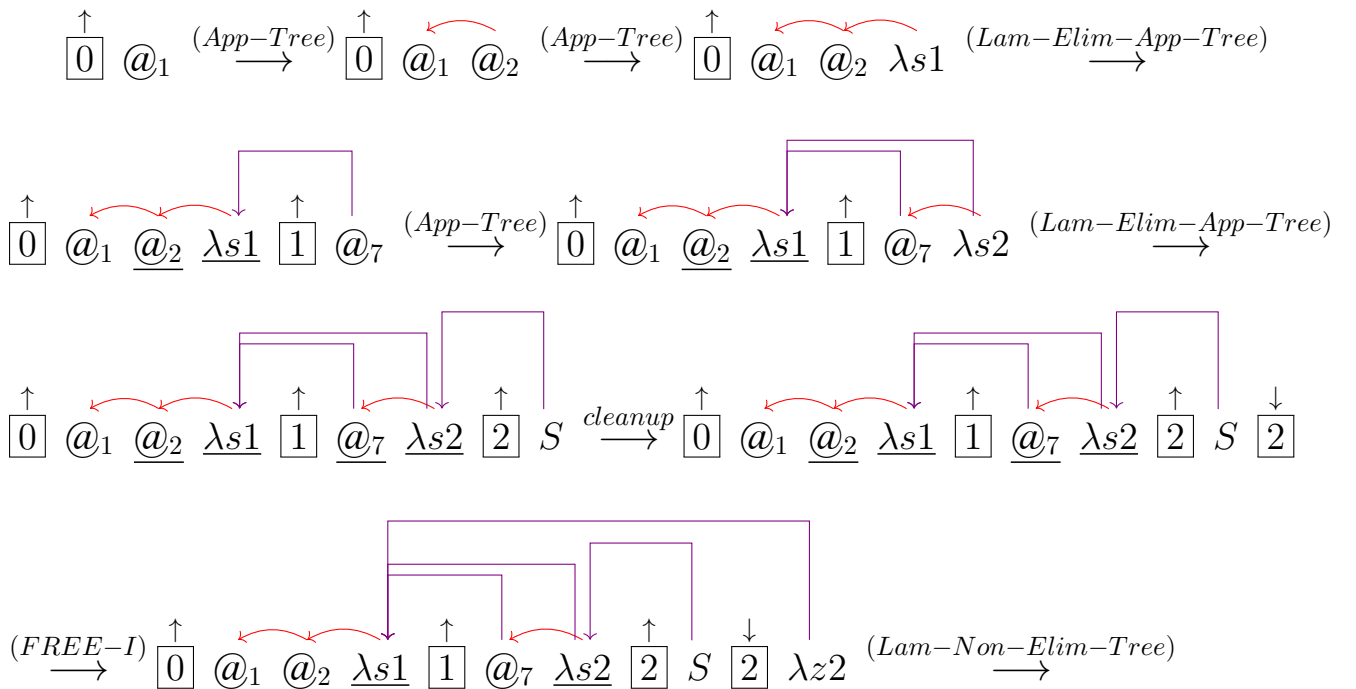
      [(z1, [(Z, [bot, bot, (data2, []), (data1, [])]), (@3
        ↪ , [bot, bot, (data2, []), (data1, [])])]),
        (S, [bot, bot, (data2, []), (data1, [])])]
z1 = getV 0
      []
      [(Z, [bot, bot, (data2, []), (data1, [])]), (@3, [bot,
        ↪ bot, (data2, []), (data1, [])])]
Z = getV 0
      []
      [bot, bot, (data2, []), (data1, [])]
bot = THE END

```

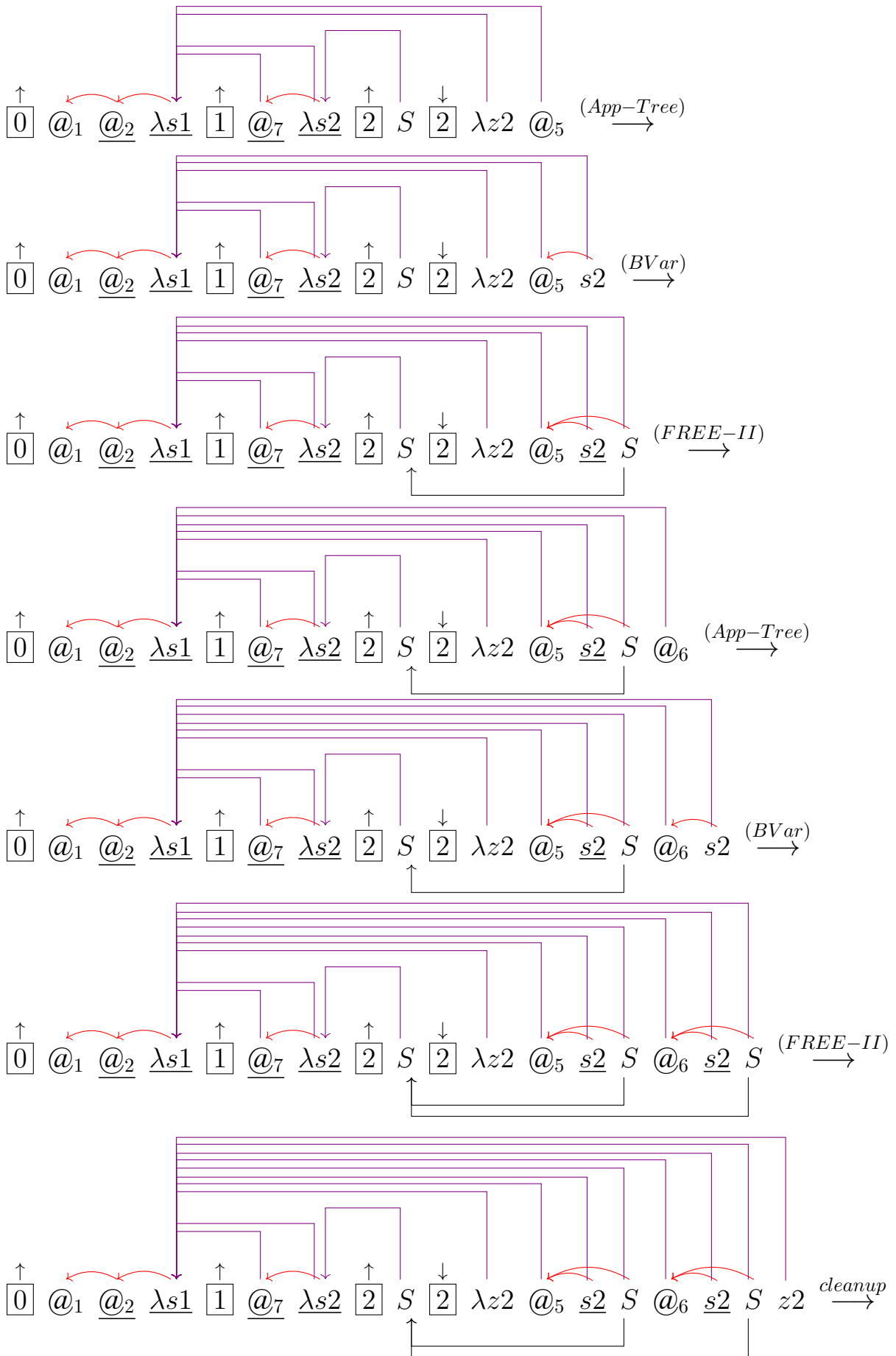
# Приложение Б.

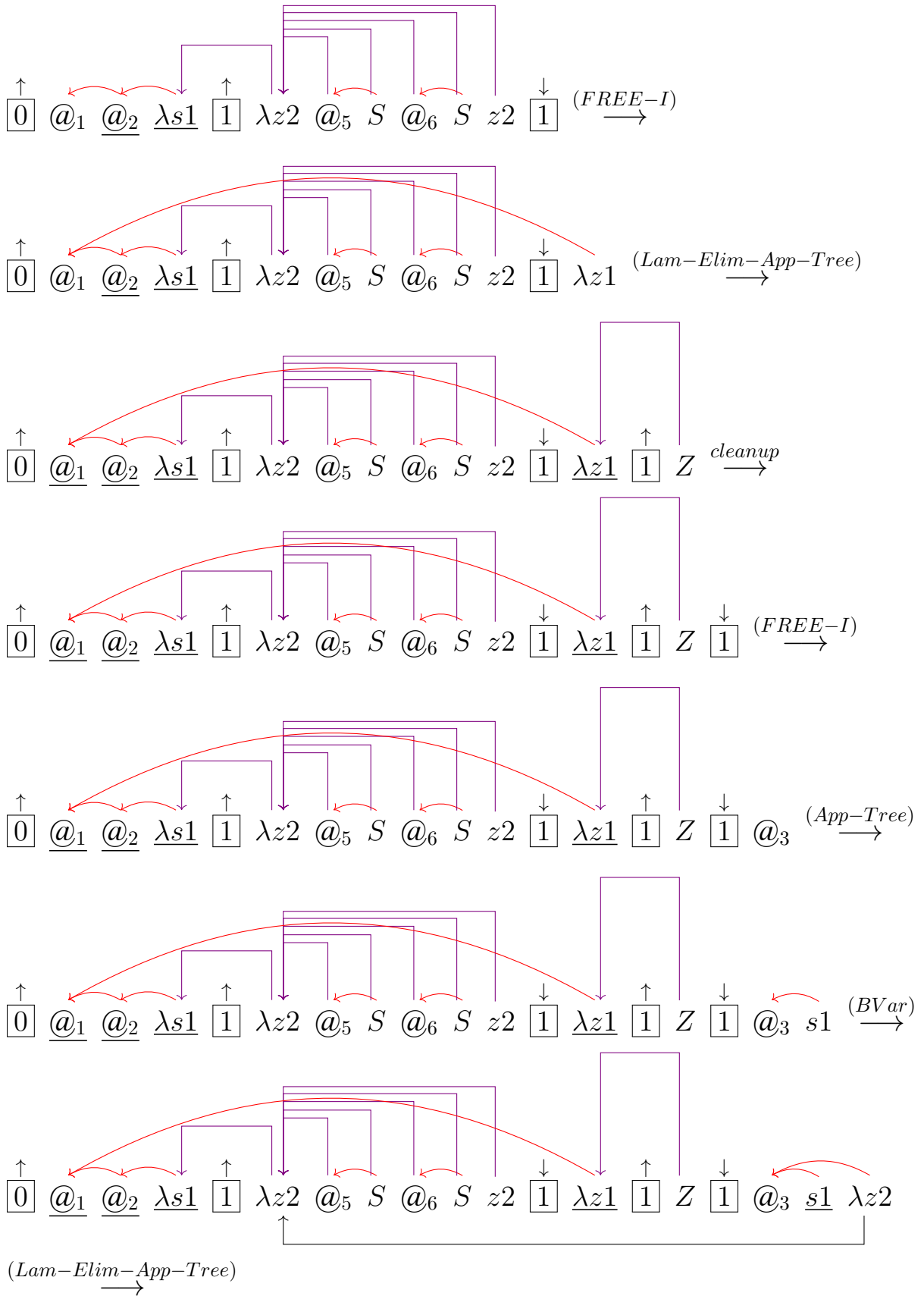
## Пример работы системы переходов для алгоритма трассирующей нормализации, соответствующего аппликативному порядку редукции

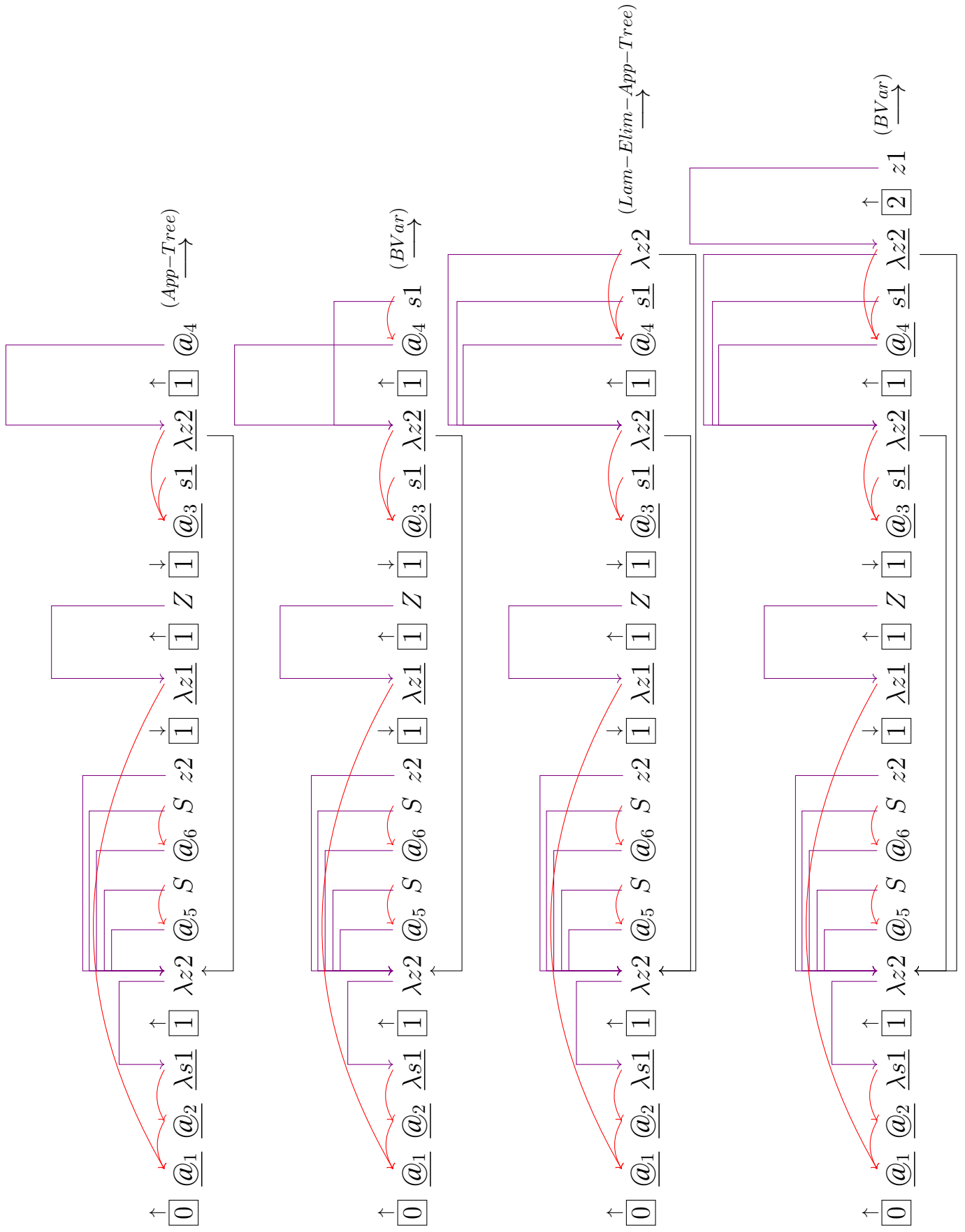
В приложении приведен пример работы системы переходов для алгоритма трассирующей нормализации для нетепизированного лямбда-исчисления, соответствующего аппликативному порядку редукции, представленный в разделе 6.1, на примере терма  $mul \vec{2} \vec{2}$ , АСД которого приведено на рис. 20.



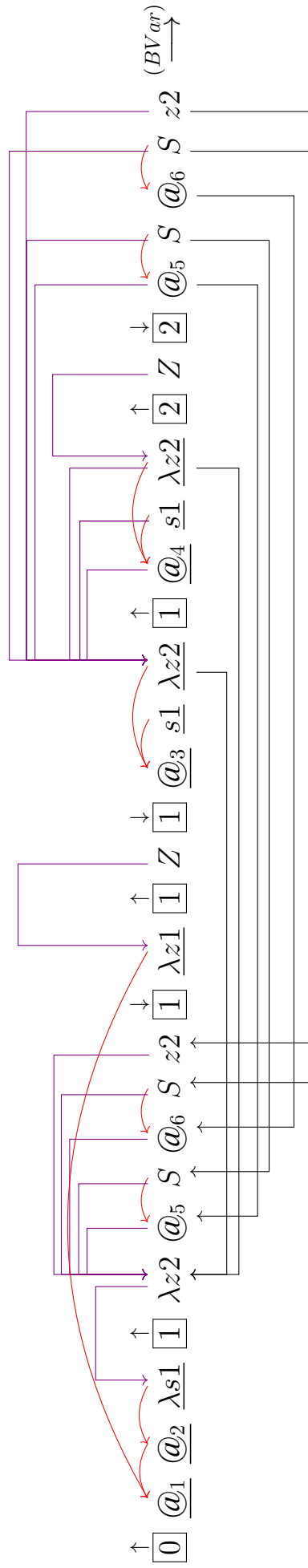
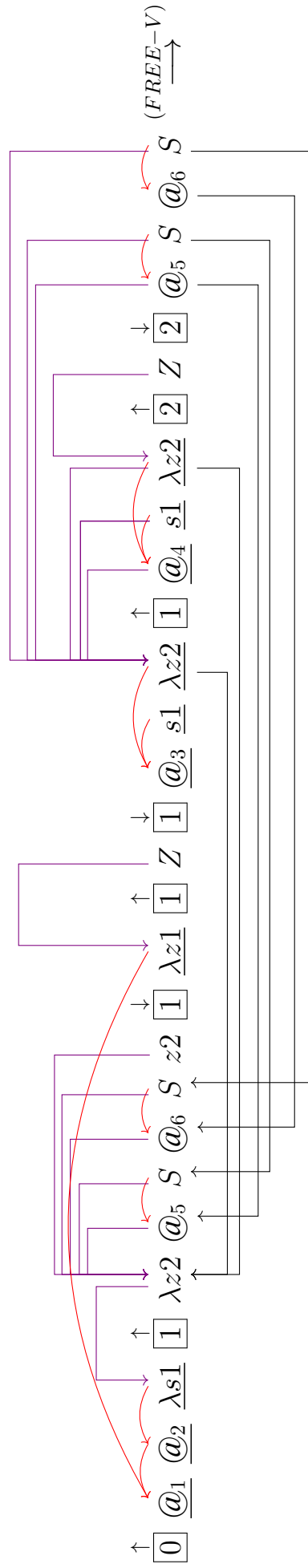
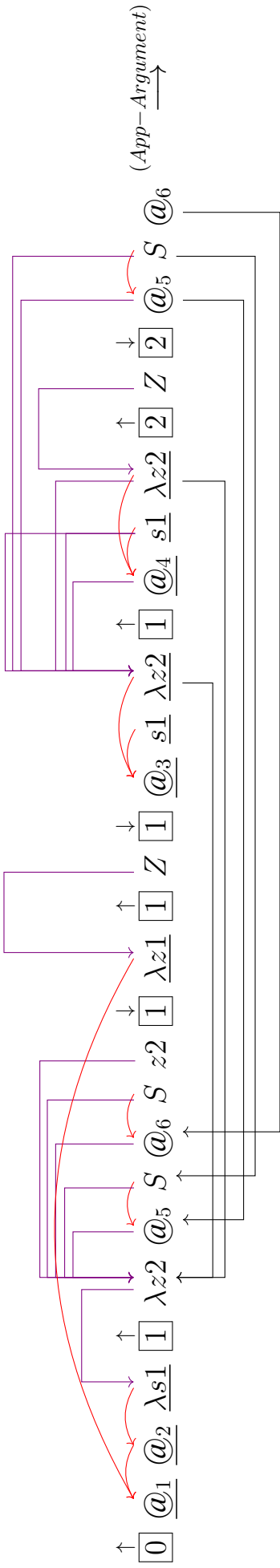




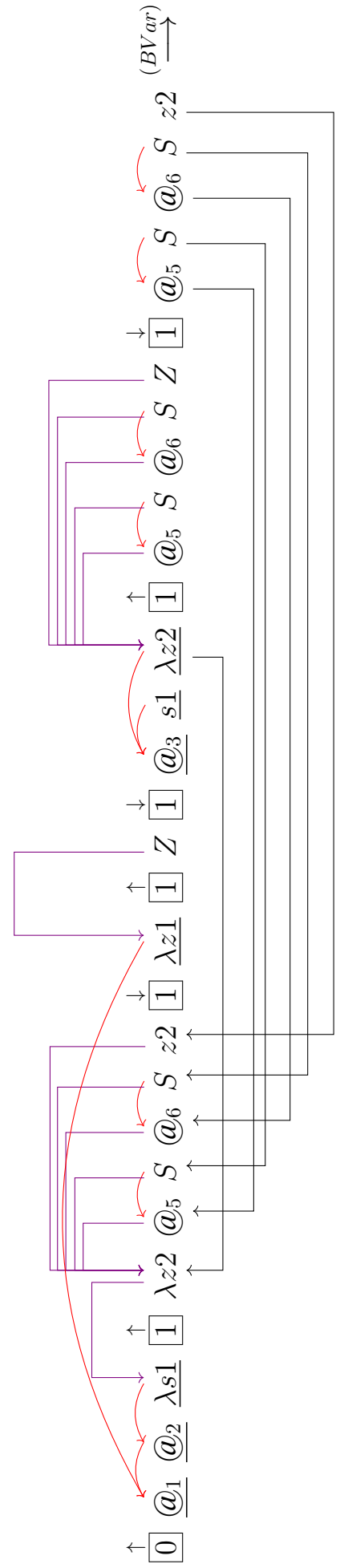
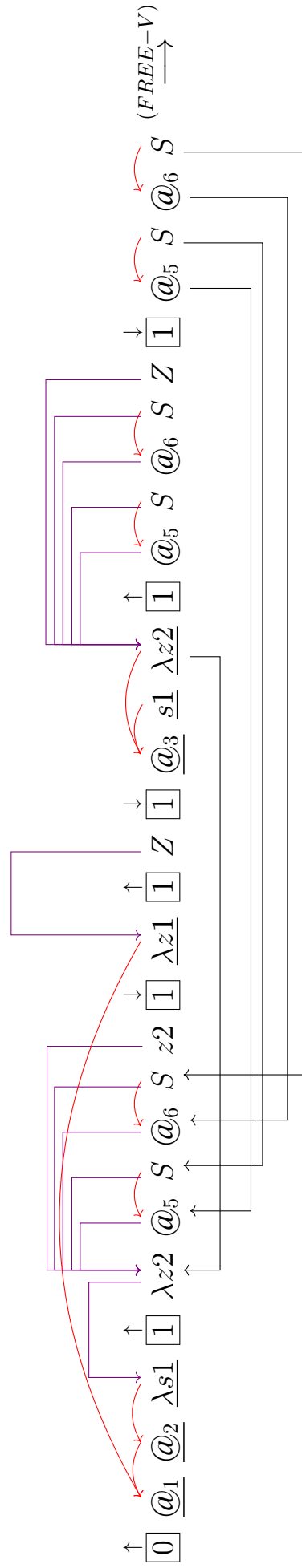
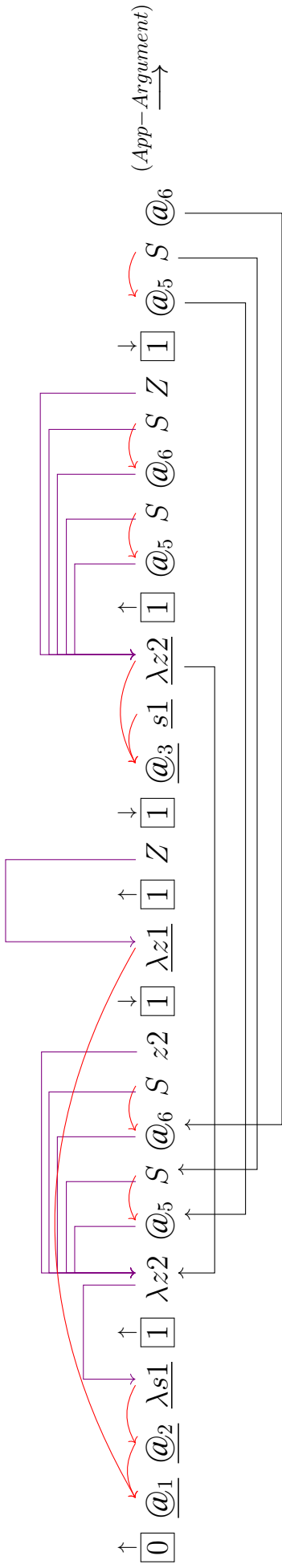


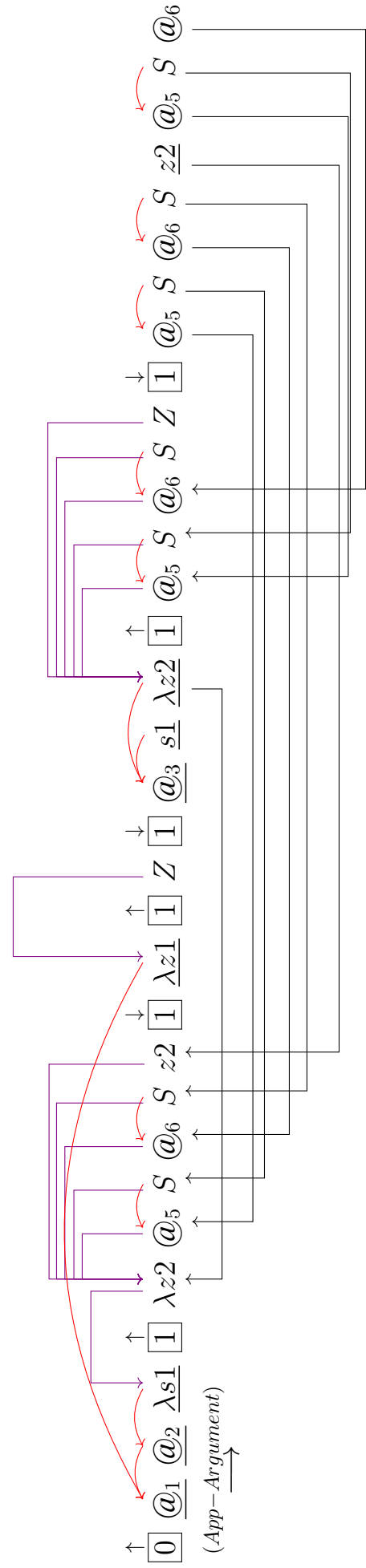
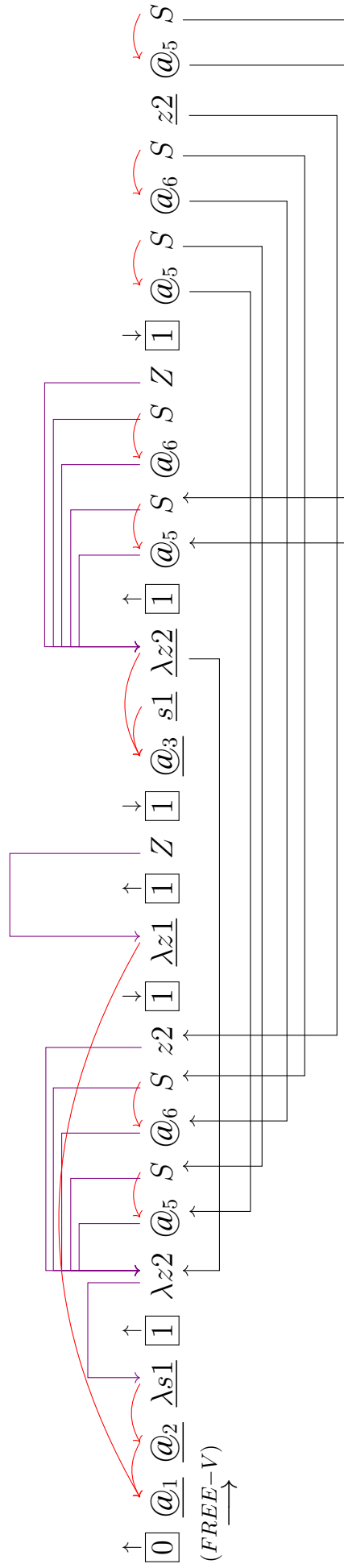
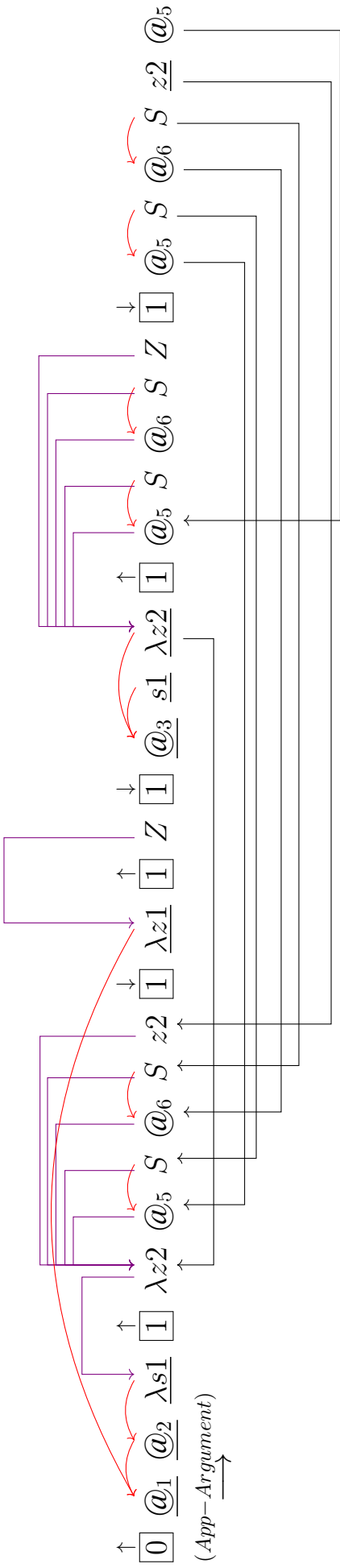




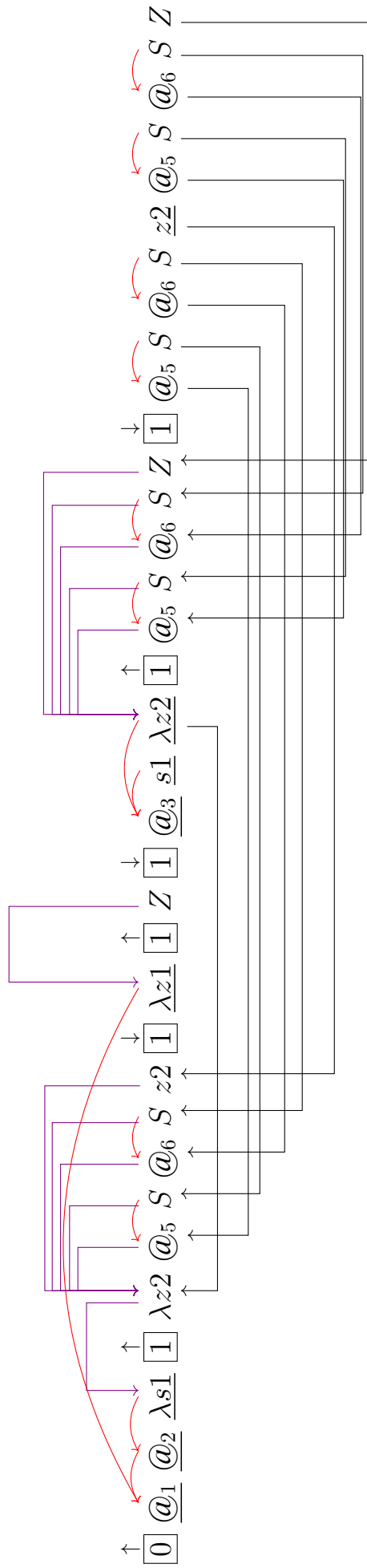
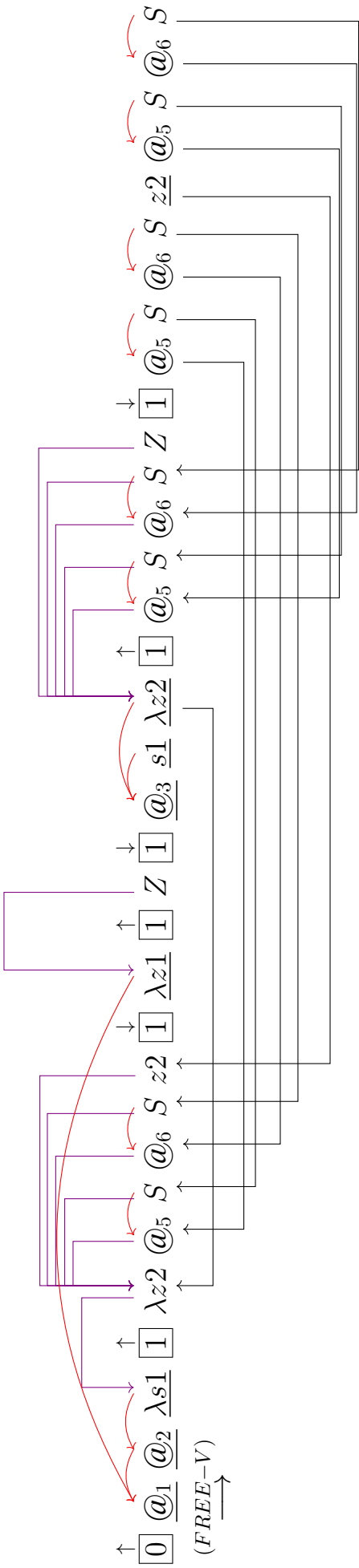












конец; окончательный вызов *cleanup*  $\downarrow$   $\rightarrow$   $\boxed{0}$   $\overset{\curvearrowright}{@_5}$   $S$   $\overset{\curvearrowright}{@_6}$   $S$   $\overset{\curvearrowright}{@_5}$   $S$   $\overset{\curvearrowright}{@_6}$   $S$   $Z$   $\boxed{0}$   $\uparrow$

Результат восстановления нормальной формы терма из итогового обхода приведён на рисунке **Б.1**.

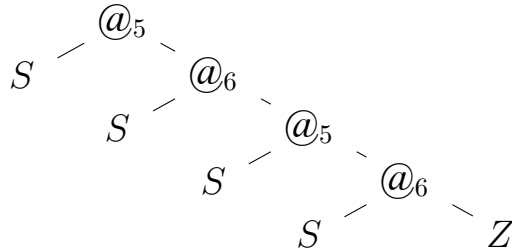


Рисунок Б.1. Результат восстановления нормальной формы терма из итогового обхода