Гречаник С.А.

Доказательство свойств функциональных программ методом насыщения равенствами

Диссертация

Рекомендуемая форма библиографической ссылки: Гречаник С.А. Доказательство свойств функциональных программ методом насыщения равенствами: дис. … канд. физ.-мат. наук: 05.13.11. М., 2017. 215 с. URL: http://library.keldysh.ru/diss.asp?id=2017-grechanik

ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
ИМЕНИ М.В.КЕЛДЫША
РОССИЙСКАЯ АКАДЕМИЯ НАУК

На правах рукописи

Гречаник Сергей Александрович

Доказательство свойств функциональных программ методом насыщения равенствами

05.13.11 — математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени кандидата физико-математических наук

Научный руководитель кандидат физико-математических наук Романенко С.А.

Москва
2017

Оглавление

Введение

Объект исследования и актуальность работы

Одним из декларируемых преимуществ функциональных языков программирования является простота преобразований, а также формулирования и доказательства свойств программ, что достигается главным образом за счёт отсутствия побочных эффектов или более явного контроля за ними. Например, если мы знаем, что $\forall \overline{x_i} f(\overline{x_i}) = g(\overline{x_i})$ для каких-то функций f и g, то мы всегда можем заменить $f(\overline{e_i})$ на $g(\overline{e_i})$ и наоборот в любом месте программы, сохранив при этом её семантику (в императивном языке подобные свойства сформулировать сложнее, т.к. функции могут не только возвращать значения, но и производить побочные эффекты, а также не только использовать свои аргументы, но и читать данные из изменяемой памяти).

Среди функциональных языков программирования наилучшим образом для выявления и доказательства равенств подходят тотальные языки [83], поскольку в них выполняется больше равенств. В тотальных языках все функции тотальны, т.е. определены для всех возможных значений параметров (подходящих по типу), в частности невозможно зацикливание, а значит они не могут быть тьюринг-полными. Такие языки, однако, меньше используются на практике для обычного программирования. Среди оставшихся (нетотальных) функциональных языков можно выделить два класса: строгие языки и нестрогие языки. В строгих языках все определённые пользователем функции являются строгими, то есть значение выражения вызова функции $f(a, b, c)$ не определено (зацикливается), если значение хотя бы одного из аргументов a, b или c не определено. Операционный смысл строгости состоит в том, что значения аргументов вычисляются перед вызовом функции.

Большинство языков являются строгими, однако такая семантика вызовов функций считается менее удобной для анализа и преобразования программ. В нестрогих же языках функции по умолчанию нестрогие. При вызове нестрогой функции даже если среди аргументов есть такой, что его вычисление зацикливается, значение всего вызова $f(a, b, c)$ может быть определено, если этот аргумент не используется (простейший пример — константная функция $f(x) = C$). Обычно это реализуется при помощи вычислений по необходимости (ленивых вычислений). Примером нестрогого языка является Haskell. Нестрогие языки считаются более удобными для анализа и преобразования, поэтому в данной диссертации в качестве входного языка был выбран нестрогий функциональный язык первого порядка.

Среди задач доказательства свойств программ выделим задачу доказательство эквивалентности функций программ, то есть утверждений, имеющих вид $\forall \overline{x_i} f(\overline{x_i}) = g(\overline{x_i})$. Данная задача имеет следующие применения:

- Верификация: многие желательные свойства функций над алгебраическими структурами данных можно сформулировать в таком виде. Широко известным примером являются законы, которые должны выполняться (хотя это и не проверяется в настоящее время компилятором GHC) для реализаций различных классов типов в языке Haskell, таких, как монады, аппликативные функторы и многие другие [84].

- Преобразование программ (в частности, оптимизация): равенства такого вида можно использовать как правила переписывания, например, в GHC, компиляторе языка Haskell, существует возможность использовать правила переписывания, задаваемые пользователем [38].

- Использование в качестве лемм, как для доказательства других свойств подобного типа, так и для доказательства свойств другого вида, например, при верификации.

Во всех случаях желательно удостовериться в том, что равенство действительно выполняется. Доказательство таких эквивалентностей часто требуется проводить по индукции.

Среди методов решения данной задачи отметим такой класс методов, которые применимы не только непосредственно к задаче доказательства эквивалентности, но и к задаче преобразования программ, например, с целью

оптимизации или анализа для выявления и доказательства свойств другого вида. Одним из таких методов является суперкомпиляция [82]. Суперкомпиляция состоит в построении программы, эквивалентной исходной (иногда в некотором слабом смысле, например, на общей области определения) при помощи правил переписывания (прогонка, обобщение) в сочетании со свёрткой (подробнее суперкомпиляция будет рассмотрена в Разделе 1.1.6). Часто суперкомпиляция рассматривается как метод оптимизации, однако суперкомпиляцию можно использовать и для других задач, в частности для доказательства эквивалентности (например, для этого можно сравнить на синтаксическое совпадение результат суперкомпиляции левой и правой частей [45; 49]). Одно из направлений развития суперкомпиляции — многорезультатная суперкомпиляция, позволяющая строить не одну программу, а целое множество различных программ, эквивалентных исходной [44]. Это позволяет использовать вместо эвристик, управляющих процессом переписывания, более точные метода выбора подходящей программы, работающие уже после суперкомпиляции и имеющие в своём распоряжении уже полностью построенные программы (например, в этом случае для выбора наиболее быстрой программы можно просто измерить производительность всех полученных программ).

Ещё один метод, применимый как к задаче доказательства эквивалентности, так и к задаче преобразования программ — насыщение равенствами (equality saturation). Насыщения равенствами — метод преобразования программ, заключающийся в применении преобразований к структуре данных, представляющей не одну программу, а целое множество семантически эквивалентных программ. Сам термин был введён в работе “Equality Saturation: A New Approach to Optimization” Тейта и других [21], где этот метод использовался для преобразования императивных программ, хотя схожие идеи применялись и задолго до этого. В насыщении равенствами каждое преобразование применяется недеструктивно, то есть исходная программа остаётся во множестве, и ко множеству добавляется новая преобразованная программа. При этом потребление памяти уменьшается за счёт совмещения общих частей программ. В итоге строится множество всех возможных программ, полученных из исходной применением преобразований в различном порядке. После этого из построенного множества выделяется одна наилучшая программа, если целью было преобразование программы. Если же целью было доказательство эквивалентности, то построение одной программы не требу-

ется, т.к. информацию об эквивалентности функций можно извлечь гораздо более простым способом.

Таким образом, насыщение равенствами и многорезультатная суперкомпиляцию имеют одинаковую мотивацию, однако использование структуры данных для компактного представления множества программ является характерной чертой насыщения равенствами, но совершенно не обязательно для многорезультатной суперкомпиляции. И хотя в предшествующих работах по многорезультатной суперкомпиляции использовались некоторые приёмы для оптимизации использования памяти [26; 43], они уступали в этом смысле насыщению равенствами. С другой стороны, насыщение равенствами применялось для императивных языков, но не для функциональных в то время, как суперкомпиляция в первую очередь предназначена для функциональных языков. Поэтому представляется актуальным исследование применения комбинации методов насыщения равенствами и многорезультатной суперкомпиляции к задачам выявления и доказательства свойств программ на функциональных языках (главным образом к задачам доказательства эквивалентности).

Цели и задачи работы

Цель работы — исследовать возможность применения комбинации методов насыщения равенствами и многорезультатной суперкомпиляции для выявления и доказательства свойств программ на нестрогом функциональном языке первого порядка.

Были поставлены следующие задачи:

- Разработать компактное представление для множеств функциональных программ.
- На основе методов насыщения равенствами и суперкомпиляции разработать метод преобразования компактного представления множеств функциональных программ.
- На основе метода преобразования разработать метод доказательства эквивалентности функций, включающий в себя в том числе доказательство по индукции и коиндукции.

- Реализовать разработанный метод в экспериментальном средстве доказательства эквивалентности и протестировать его на наборе модельных задач.

Научная новизна работы

В оригинальных работах по насыщению равенствами [21; 73; 78] рассматривается оптимизация императивных программ. При этом, однако, императивные программы преобразуются в функциональное промежуточное представление. Мы же имеем дело изначально с функциональным языком, при этом мы накладываем гораздо меньше ограничений на рекурсию — в императивных языках широко применяются циклы, для представления которых в функциональных языках достаточно, например, хвостовой рекурсии[1], однако при работе с программами на функциональных языках работа с рекурсией более общего вида становится крайне желательной.

В [21] для представления множеств программ используется структура данных называемая E-PEG (PEG расшифровывается как Program Expression Graph), основанная на E-графах (графах, вершины которого разбиты на классы эквивалентности). В данной диссертации для представления множества функциональных программ мы вводим понятие полипрограммы, которое аналогично E-PEG, но концептуально проще: неформально полипрограмма — это программа, в которой разрешены множественные определения одной и той же функции. Главное содержательное отличие состоит в том, что в полипрограмме работа ведётся с функциями со своими собственными входными параметрами, в то время как в E-PEG работа идёт со значениями (точнее, с массивами значений для представления результата работы циклов), а входные параметры являются общими для всего E-PEG.

Кроме того, мы исключаем дублирование в полипрограмме функций, отличающихся только порядком аргументов, что делается впервые в рамках работы со структурами, подобными E-графам (самый близкий из предшествующих результатов — исключение дублирования вершин, обозначающих

[1] В оригинальных работах по насыщению используется другое представление — каждая вершина изображает многомерный массив значений (или другими словами функцию от нескольких целочисленных параметров), каждое измерение соответствует глубине вложенности цикла, при этом на граф накладываются ограничения так, чтобы зависимости значений массивов от элементов с меньшими индексами соответствовали зависимостям между значениями переменных в циклах на императивных языках.

значения, отличающиеся некоторым целочисленным смещением [56]).

Мы также вводим специальное преобразование, которое называем слиянием по бисимуляции, работающее как доказательство равенств по индукции или коиндукции. Его предшественником можно считать специализированную версию слияния по конгруэнтности, способную сливать одинаковые циклы, реализованную в системе насыщения равенствами Peggy, созданной в рамках оригинальных работ по насыщению равенствами, однако она менее мощная и не упоминается в публикациях. Слияние по бисимуляции играет роль аналогичную выявлению, доказательству и применению лемм в многоуровневой суперкомпиляции [46], однако для обеспечения корректности в данной диссертации используются признаки структурной и защищённой рекурсии вместо теории улучшения Сэндса.

Практическая значимость работы

Настоящая работа показывает, что на основе метода насыщения равенствами и преобразований из суперкомпиляции можно построить систему преобразования функциональных программ, применимую для доказательства эквивалентности.

Показано, что при этом удаётся избежать проблемы, связанной с комбинаторным взрывом количества функций, отличающихся только порядком аргументов.

Также было показано, что можно сформулировать принцип индукции как специальное преобразование, работающее в рамках системы насыщения. Полученный метод можно успешно применять для доказательства эквивалентности функций, что продемонстрировано тестированием на наборе примеров и сравнением с аналогичными инструментами.

Апробация работы и публикации

По результатам работы были сделаны доклады на следующих семинарах и конференциях:

1. Международный семинар “Third International Valentin Turchin Workshop on Metacomputation”, Россия, Переславль-Залесский, 5–9 июля 2012.

2. Международная конференция памяти Андрея Ершова “Ershov Informatics Conference”, Россия, Санкт-Петербург, 24–27 июня 2014

3. Международный семинар “Fourth International Valentin Turchin Workshop on Metacomputation”, Россия, Переславль-Залесский, 29 июня – 3 июля 2014..

4. Семинар ИПС им. А.К. Айламазяна под рук. член-корр. С.М. Абрамова, 20 мая 2016.

5. Международный семинар “Fifth International Valentin Turchin Workshop on Metacomputation”, Россия, Переславль-Залесский, 27 июня – 1 июля 2016.

6. Семинар “Теоретические проблемы программирования” под рук. д.ф.-м.н. В.А. Захарова, ВМиК МГУ, 11 ноября 2016.

7. Семинар “Теоретическое и экспериментальное программирование” под рук. к.ф.-м.н. В.А. Непомнящего, ИСИ СО РАН, 22 ноября 2016.

8. Семинар “Системное программирование” под рук. к.ф.-м.н. М.А. Бульонкова и д.ф.-м.н. А.Г. Марчука, ИСИ СО РАН, 24 ноября 2016.

9. Семинар группы “Технологии программирования” под рук. д.ф.-м.н. А.К. Петренко, ИСП РАН, 16 марта 2017.

Также были опубликованы следующие статьи, включая две в научных журналах из списка ВАК:

1. *Grechanik S. A.* Overgraph Representation for Multi-Result Supercompilation // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. Pereslavl-Zalessky, Russia : Pereslavl-Zalessky: Ailamazyan University of Pereslavl, July 2012. Pp. 48–65. ISBN 978-5-901795-28-6. URL: `http://pat.keldysh.ru/~grechanik/doc/overgraph.pdf`

2. *Grechanik S. A.* Supercompilation by Hypergraph Transformation: Preprint / Keldysh Institute of Applied Mathematics. 2013. 24 pp. No. 26.
URL: `http://library.keldysh.ru/preprint.asp?id=2013-26&lg=e`

3. *Grechanik S. A.* Inductive Prover Based on Equality Saturation for a Lazy Functional Language // Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers. Vol. 8974 / ed. by A. Voronkov, I. Virbitskaite. Springer, 2014. Pp. 127–141. (Lecture Notes in Computer Science). ISBN 978-3-662-46822-7. URL: http://dx.doi.org/10.1007/978-3-662-46823-4

4. *Grechanik S.* Inductive Prover Based on Equality Saturation (Extended Version) // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", July 2014. Pp. 26–53. ISBN 978-5-901795-31-6

5. **(ВАК)** *Гречаник С. А.* Доказательство свойств функциональных программ методом насыщения равенствами // Программирование. 2015. № 3. с. 44—61

6. *Grechanik S. A.* Proving properties of functional programs by equality saturation // Programming and Computer Software. 2015. Vol. 41, no. 3. Pp. 149–161. URL: http://dx.doi.org/10.1134/S0361768815030056

7. **(ВАК)** *Гречаник С. А.* Полипрограммы как представление множеств функциональных программ и преобразования над ними // Препринты ИПМ им. М.В.Келдыша. 2017. № 5. DOI: 10.20948/prepr-2017-5. URL: http://library.keldysh.ru/preprint.asp?id=2017-5

Глава 1

Смежные работы

В настоящей работе предлагается использовать комбинацию методов насыщения равенствами и суперкомпиляции для преобразования функциональных программ с целью выявления и доказательства их свойств. В данной главе приведём обзор смежных работ, касающихся насыщения равенствами, суперкомпиляции, а также некоторых других методов.

1.1 Обзор смежных работ

1.1.1 Система преобразования Бёрстолла-Дарлингтона

Многие методы преобразования программ можно рассматривать в качестве частных случаев общей схемы преобразования программ Бёрстолла и Дарлингтона [10] (в том числе насыщение равенствами, суперкомпиляцию и наш метод). Данная схема состоит в том, что программа рассматривается как набор уравнений, после чего при помощи некоторых преобразований данный набор уравнений пополняется новыми уравнениями, и в результате какое-то подмножество этих новых уравнений можно рассматривать как новую программу. Среди преобразований можно отметить следующие три:

- Раскрытие (unfolding) — подстановка тела функции (правой части) вместо вызова функции.
- Свёртка (folding) — обратная операция, подстановка вызова функции вместо тела.

- Переопределение (redefinition) — замена одной функции на другую, если обе функции определяются двумя одинаковыми с точностью до переименования функций подмножествами уравнений. Это преобразование мало изучено в оригинальной работе, однако оно перечислено здесь, потому что соответствует важному преобразованию слияния по бисимуляции из нашего метода.

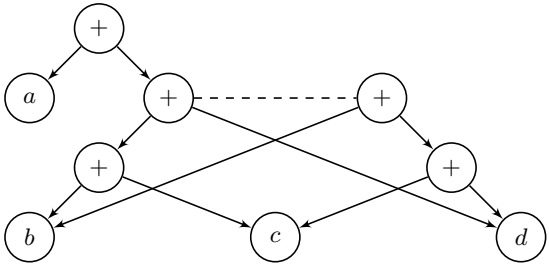
Схема Бёрстолла и Дарлингтона очень общая, она допускает различные стратегии применения правил и оставляет открытым вопрос о корректности преобразований. Как будет видно в Главе 2, наш метод очень близок к данной схеме, однако мы явно вводим отдельное понятие полипрограммы для множества уравнений, и явно вводим семантику для полипрограмм. Это позволяет нам разобраться с корректностью преобразований, особенно с тонким вопросом корректности слияния по бисимуляции. Кроме того, на уровне реализации и формальной работы с полипрограммами мы вводим ограничения на вид определений, что сближает наш метод с насыщением равенствами.

1.1.2 Насыщение равенствами

Насыщение равенствами — термин, введённый в работе “Equality Saturation: A New Approach to Optimization” Тейта и других [21], где этот метод использовался для преобразования императивных программ. Представленный в этой работе оптимизатор называется Peggy. Суть метода насыщения равенствами заключается в использовании структуры данных, компактно представляющей не одну программу, а целое множество программ. В [21] эта структура данных называется E-PEG, PEG расшифровывается как Program Expression Graph. E-PEG основана на понятии E-графа — графа, вершины которого разбиты на классы эквивалентности (буква E от equality). E-графы и подобные им структуры данных эффективно представляют множества равенств над термами (например, как на Рис. 1.1). Они используются в средствах доказательства теорем, в частности для анализа и доказательства свойств программ [15; 20]. Обычно E-графы применяются в сочетании с алгоритмом конгруэнтного замыкания [55], позволяющим поддерживать представляемое E-графом множество равенств замкнутым относительно аксиомы монотонности ($\overline{a_i = b_i} \Rightarrow f(\overline{a_i}) = f(\overline{b_i})$ для всех функциональных символов f) — данный алгоритм собственно является решателем для теории неинтерпрети-

рованных функций с равенством. Отдельно стоит упомянуть работу [56], в которой описывается расширение алгоритма конгруэнтного замыкания, позволяющее сливать вершины, представляющие значения, равные с точностью до целочисленного смещения — схожий подход используется нами для слияния функций, эквивалентных с точностью до перестановки параметров.

Рис. 1.1 E-граф, представляющий множество равенств над термами, а именно $b+(c+d) = (b+c)+d$ и $a+(b+(c+d)) = a+((b+c)+d)$. Первое равенство изображено пунктирной линией, соединяющей две вершины, находящиеся в одном классе эквивалентности. Второе равенство является следствием первого по аксиоме монотонности. На рисунке можно видеть, как обеспечивается компактность представления за счёт обобществления общих подтермов (а именно вершин b, c, и d) и обобществления общего подтерма "с дыркой", а именно $a + \_$. Последнее возможно только для эквивалентных выражений, поэтому является особенностью E-графа (обобществление подтермов без дырок осуществляется и в обычных графах, представляющих термы).



Кроме задачи доказательства эквивалентности E-графы могут быть использованы для других задач, например, для оптимизации программ. Данное применение основано на том, что при помощи E-графа можно компактно представить множество эквивалентных программ (например, можно считать, что E-граф на Рис. 1.1 представляет множество термов $\{a+(b+(c+d)), a+((b+c)+d)\}$). Компактность представления осуществляется за счёт объединения общих подвыражений (sharing). При этом над E-графом можно осуществлять различные преобразования, что даёт эффект одновременного преобразования всего множества программ. Данный подход применяется, например, в супероптимизаторе Denali [39] для поиска оптимальной последовательности инструкций для выполнения небольших программ.

В уже упомянутой работе [21] этот подход под названием насыщение равенствами был использован для оптимизации императивных программ, а кроме того было показано, что тот же метод применим и для доказательства

эквивалентности программ [74]. Общая схема насыщения равенствами в этой работе выглядит следующим образом:

- Исходная программа преобразуется в E-PEG.
- Производится насыщение: к E-PEG применяются преобразования, которые могут добавлять новые вершина и рёбра и сливать классы эквивалентности. Удаление же вершин и рёбер (и разбиение классов) запрещено. Насыщение завершается либо когда больше нельзя применить ни одно преобразование, либо когда достигается некоторый лимит количества преобразований.
- Из E-PEG, представляющего множество программ, выделяется одна наилучшая программа. Для этого формулируется и решается задача целочисленного линейного программирования: каждой вершине E-графа ставится в соответствие переменная, равная 1, если вершина войдёт в программу и 0, если не войдёт, и производится поиск программы наименьшей стоимости при ограничениях на согласованность (например, если вершина входит в программу, то в программу входят какие-то представители классов эквивалентностей дочерних вершин).

Если стоит задача доказательства эквивалентности, то схема ещё проще, т.к. последний пункт выполнять не нужно, и весь процесс можно завершить когда вершины, соответствующие обеим частям целевого утверждения, окажутся в одном классе эквивалентности.

1.1.3 Переписывание графов и джунгли

Переписывание графов — это обобщение переписывания термов на графы. Система переписывания графов состоит из набора правил вида $L \to R$, подразумевающих, что если в графе G есть подграф вида L, то он может быть заменён на подграф вида R. В теории переписывания графов возникают трудности с такими операциями, как удаление вершин, поэтому существует множество подходов к переписыванию графов: подход двойного кодекартова квадрата [19], одинарного кодекартова квадрата [50] и т.д. Переписывание графов также обобщается на гиперграфы (графы, в которых рёбра, называемые гиперрёбрами, могут соединять более двух вершин, гиперграфы часто удобнее графов для некоторых приложений). E-графы можно представлять в

виде ориентированных гиперграфов, при этом классы эквивалентности вершин E-графа отображаются в вершины гиперграфа, а вершины E-графа вместе с исходящими рёбрами отображаются в гиперрёбра. Фактически понятие полипрограммы, используемое в настоящей работе, основано на гиперграфах, и преобразования полипрограмм основаны на преобразовании гиперграфов.

Джунгли [35; 58] — ориентированные гиперграфы без циклов, у которых каждая вершина имеет не более одного исходящего гиперребра. Джунгли удобны для представления термов с учётом обобществления (sharing) одинаковых подвыражений. Джунгли очень похожи на E-графы из насыщения равенствами (представленными гиперграфами) и полипрограммы из настоящей работы, по существу отличаясь от них только ограничением на количество исходящих гиперрёбер и ацикличностью, что не позволяет джунглям представлять множество эквивалентных выражений. На джунглях определяется правило преобразование, называемое свёрткой, которое позволяет сливать подграфы, представляющие равные термы. Это преобразование соответствует конгруэнтному замыканию, позволяющему сливать классы эквивалентности в E-графах.

Систему переписывания термов можно превратить в систему переписывания джунглей, при этом если оригинальная система завершающаяся и конфлюэнтная, то и система для джунглей тоже завершающаяся и конфлюэнтная с точностью до сборки мусора. Благодаря использованию свёртки переписывание на джунглях может быть более эффективным, чем переписывание термов, кроме того свёртка обеспечивает возможность использовать правила с нелинейными левыми частями так же как и при переписывании термов.

1.1.4 Пруверы первого порядка

Одна из задач, родственных доказательству эквивалентности функций на функциональном языке — доказательство истинности формул в логике предикатов первого порядка. Эта задача отличается от доказательства эквивалентности функций на функциональном языке главным образом тем, что её решение не подразумевает использование индукции, поэтому пруверы первого порядка обычно индукцию не поддерживают. Однако, как мы увидим в следующем разделе, существуют инструменты, позволяющие добавить поддержку индукции к пруверам первого порядка (иногда используя эти пруверы как чёрный ящик).

Одним из подходов к решению данной задачи является метод насыщения [22] (в частности, метод резолюций [62]). Идея насыщения (которое не следует путать с насыщением равенствами, хотя идеи похожи) состоит в том, чтобы применять ко множеству формул правила вывода до тех пор, пока не будет достигнуто противоречие (при использовании доказательства от противного). В отличие от насыщения равенствами этот метод не подразумевает использование компактного представления равенств в виде E-графа — обычно в пруверах первого порядка к равенствам используется другой подход, в частности, основанный на парамодуляции [63] и суперпозиции [4]. Данный метод реализован во многих пруверах, в частности, в E [66], Vampire [60], SPASS [71]. Существует обширный сборник задач для пруверов первого порядка TPTP (Thousands of Problems for Theorem Provers) [77], также регулярно проводятся соревнования.

Одной из вариаций задачи доказательства истинности формулы является задача выполнимости в теориях (SMT, satisfiability modulo theories). Отличие в том, что в этой задаче подразумевается, что есть некоторые встроенные теории, аксиомы для которых не нужно задавать в формулировке задачи. Это такие теории, как теории целых и вещественных чисел, списков, массивов и т.п. Для этих теорий должны быть специализированные разрешающие процедуры, которые используются в SMT-решателях (SMT-солверах), что позволяет решать задачу эффективнее, чем общими методами, использующимися в пруверах первого порядка. Заметим, что одна из теорий (наиболее простая) — теория неинтерпретированных функций с конгруэнтным равенством — решается как раз при помощи алгоритма конгруэнтного замыкания на E-графах [55]). Формулы, к которым применяются SMT-солверы часто не содержат кванторов (свободные переменные можно считать связанными квантором существования на верхнем уровне, т.к. решается задача выполнимости) — в этом случае SMT-солверы за конечное время могут выдать ответ, является формула выполнимой или нет. Однако современные SMT-солверы поддерживают кванторы, что позволяет использовать их в качестве общих пруверов для логики первого порядка.

Сейчас большинство SMT-солверов основаны на алгоритме DPLL(T) [16], являющимся модификацией алгоритма решения задачи выполнимости булевых формул DPLL [14]. Суть алгоритма состоит в поиске с возвратом интерпретаций атомарных формул. Упрощённо, решатель теории получает на

вход интерпретацию и проверяет, нет ли в ней противоречий — если нет, то формула выполнима, но для эффективной реализации решатель должен взаимодействовать с алгоритмом поиска, чтобы отсекать тупиковые ветви.

Поддержка кванторов в SMT-решателях обычно осуществляется следующим эвристическим методом. Каждой универсально-квантифицированной формуле сопоставляется некоторый образец, представляющий из себя обычно подтерм этой формулы. Как только в состоянии солвера встречается терм, соответствующий этому образцу, производится инстанцирование формулы. Сопоставление с образцом должно при этом осуществляться с точностью до известных равенств между термами (E-matching, [52; 53]) — эта же проблема возникает и в насыщении равенствами при применении правил.

В качестве примеров SMT-решателей можно назвать Simplify [15], Yices [17], Z3 [54], CVC3 [6], Alt-Ergo [79]. Для SMT-решателей также существует сборник задач [80]

1.1.5 Индуктивные пруверы

Доказательство свойств программ при помощи структурной индукции известно ещё с 50-60-х годов [11]. Впоследствии появились полностью автоматические индуктивные пруверы, одним из первых был прувер Бойера и Мура [8] для языка LISP. Этот прувер был предшественником ACL2 [40], успешным индуктивным прувером, широко применяемым в индустрии. Большинство современных пруверов, как и ACL2 и его предшественники, используют метод явной индукции, суть которого в том, что аксиома индукции применяется явно, точно так же, как и все остальные правила вывода. У этого подхода есть некоторые недостатки, такие как необходимость угадывать подходящую схему индукции и гипотезу индукции до проведения самого рассуждения. Существуют альтернативные подходы, например, “индукция без индукции” [36], основанная на алгоритме Кнута-Бендикса [47]. Есть также подход, заключающийся в том, что допускаются циклические доказательства, для которых потом производится проверка корректности — все циклы должны быть убывающими относительно некоторого фундированного отношения на формулах — при этом схема индукции также применяется неявно [72; 75]. Отметим, что подход с циклическими доказательствами также обычно (хотя и не всегда) применяется в суперкомпиляции, которую мы рассмотрим в следующем разделе, а также он применяется в данной диссертации в форме слияния по

бисимуляции (Глава 5).

Обычно индуктивные пруверы работают следующим образом: к целевому утверждению применяются правила вывода (в обратном порядке) пока не будет построено завершённое дерево доказательства. В случае явного применения индукции индукция применяется как обычное правило вывода. При этом возникает огромное количество точек ветвления, что связано с неоднозначностью выбора правил вывода, применимых к утверждению, в частности очень большое множество вариантов даёт обобщение доказываемого утверждения. Эта проблема решается сочетанием эвристик и перебора. Среди эвристик стоит отметить рипплинг [9; 61]. Рипплинг управляет применением правил переписываний таким образом, чтобы к утверждению можно было применить гипотезу индукции, сдвинув мешающие части выражения наружу или к переменным. Кроме того, отметим индуктивный прувер Zeno [69] для языка Haskell, который использует более свободный подход к применению правил переписывания (внутри Zeno очень похож на суперкомпилятор, хотя и использует явную индукцию).

Существуют также способы использовать пруверы для логики первого порядка и SMT-решатели для доказательства по индукции. Для этого производится инстанцирование схемы индукции для целевого утверждения, после чего база индукции и шаг индукции доказываются отдельно прувером первого порядка [48]. В частности, таким образом работает HipSpec [3], индуктивный прувер для языка Haskell. Отметим, что HipSpec также использует разведку теории (theory exploration): сначала он генерирует набор гипотез при помощи тестирования [13], затем доказывает их по одной, при этом для доказательства гипотез используются уже доказанные гипотезы. Это позволяет находить необходимые обобщения и леммы даже в очень нетривиальных случаях.

1.1.6 Суперкомпиляция

Суперкомпиляция — метод преобразования программ, разработанный Турчиным [82]. Суперкомпиляция похожа на автоматическое доказательство по индукции, однако эти области развивались довольно независимо. Изначально суперкомпиляция была сформулирована для языка Рефал, но потом появились формулировки и для других языков. В частности, в работе Андрея Климова и Роберта Глюка [23] используется язык, похожий на Lisp, работающий

с S-выражениями. В работе Сёренсена [70] используется функциональный язык первого порядка с семантикой вызова по имени и данными, построенными из конструкторов, основанный на языке Miranda. В работе Митчелла [51] используется подмножество языка Haskell. Подмножество Haskell'а используется также в работе Ключникова [87].

В общих чертах суперкомпиляция заключается в применении правил переписывания к исходному выражению для построения так называемого графа конфигураций, который потом может быть превращён в остаточную программу, семантически эквивалентную исходному выражению. В графе конфигураций каждая вершина содержит конфигурацию, обозначающую множество состояний программы и в простейшем случае являющуюся просто выражением входного языка со свободными переменными. Конфигурация любой вершины может быть выражена через конфигурации дочерних вершин. Построение графа происходит примерно следующим образом:

- Если в выражении в вершине можно произвести вычисление, то вычисление производится и к вершине подвешивается дочерняя вершина с вычисленным выражением. В доказательствах по индукции это соответствует применению аксиом, выражающих определения функций.

- Если вычисление произвести нельзя, потому что этому мешает свободная переменная, значение которой неизвестно, то производится разбор случаев — к вершине подвешиваются дочерние вершины, в которых значение упомянутой переменной ограничено значениями определённого вида (например, для переменной типа Nat это может быть $S(x)$ и Z). В доказательствах по индукции это соответствует применению аксиомы индукции (для пруверов, использующих метод явной индукции) или разбору случаев для пруверов, использующих циклические доказательства без явного применения индукции.

- По решению некоторых эвристик возможно применение обобщения — замену некоторых подвыражений на новые свободные переменные. Например, $f(g(h(x)))$ можно обобщить до $f(y)$, где $y = g(h(x))$, после чего $f(y)$ и $g(h(x))$ рассматриваются по отдельности. В доказательствах по индукции это соответствует усилению гипотезы индукции при помощи обобщения или некоторым случаям введения лемм.

- Если в вершине находится конфигурация, которая уже встречалась среди потомков этой вершины (с точностью до переименования переменных), то производится зацикливание (свёртка), т.е. проводится обратная дуга от данной вершины к вершине-потомку. Это соответствует применению гипотезы индукции (фертилизации).

Первые два пункта составляют так называемую прогонку — вычисление с неполными данными. Прогонку саму по себе можно выполнять неограниченно, поэтому необходимы некоторые эвристики, которые запрещают прогонку для некоторой конфигурации, и вместо этого производится обобщение. Такие эвристики называются свистками, и они обеспечивают завершаемость суперкомпилятора.

Одним из усовершенствований суперкомпиляции является возможность применения лемм в процессе суперкомпиляции. Например, в дистилляции, предложенной Гамильтоном [31–34], при зацикливании и обобщении производится работа не с самими конфигурациями, а с их суперкомпилированными версиями, что соответствует применению лемм, доказанных суперкомпилятором. В работах Ключникова и Романенко [42; 46] описана суперкомпиляция высшего уровня, заключающаяся в применении нижнего суперкомпилятора верхним (в частности, для доказательства лемм). Отметим, что в случае применения лемм возникает проблема с корректностью суперкомпиляции — дело в том, что традиционно суперкомпиляторы не совершают проверку того, что внутри циклов построенного графа происходит убывание относительно некоторого фундированного отношения, однако суперкомпиляция устроена так, что в этом нет необходимости. Но при произвольном применении лемм данное свойство суперкомпиляции теряется. В [46] показано, что если применяемые леммы ограничивать улучшающими леммами (в смысле теории Сэндса [64; 65]), то корректность сохраняется. Тот факт, что лемма улучшающая, можно проверить при помощи суперкомпилятора в процессе доказательства леммы.

Ещё одно развитие идеи суперкомпиляции — многорезультатная суперкомпиляция, предложенная Ключниковым и Романенко [44]. Традиционно суперкомпилятор выдаёт на выходе только одну программу, но многорезультатный суперкомпилятор выдаёт целое множество остаточных программ. Это бывает полезно, когда суперкомпиляция используется для анализа программ — при этом суперкомпилятор упрощает остаточную программу, делая её более доступной для дальнейшей обработки анализаторами. Один из примеров

такого использования суперкомпилятора — доказательство эквивалентности программ — при этом входные программы суперкомпилируются, после чего производится синтаксическое сравнение остаточных программ [45; 49]. В случае многорезультатной суперкомпиляции нужно сравнить результирующие множества на пересечение, что позволяет доказывать эквивалентности в большем числе случаев.

Для большинства приложений многорезультатной суперкомпиляции выдавать непосредственно множества остаточных программ не очень эффективно. Эффективнее генерировать некоторое компактное представление множества остаточных программ, которое потом анализируется также быстрее. Например, в работе [26] описано такое представление множеств остаточных программ в виде деревьев, компактность при этом достигается за счёт совмещения общих начальных частей программ, а дальнейшая фильтрация этого множества по некоторому условию может быть произведена эффективнее, чем фильтрация обычного множества программ за счёт того, что в процессе разом могут выбрасываться целые подмножества программ, не соответствующих условию. В работе [28] используется ещё более компактное представление множества программ в виде графа, подобного E-графу, но при этом сохраняется общая схема суперкомпиляции — фактически эта работа является промежуточной между суперкомпиляцией и насыщением равенствами. Настоящая диссертационная работа является следующим шагом по направлению к насыщению равенствами — фактически мы используем преобразования из суперкомпиляции, но в рамках насыщения равенствами.

1.2 Выводы

В данной главе был дан краткий обзор работ по смежным темам: насыщению равенствами, доказательству теорем в логике предикатов, индуктивному доказательству теорем, суперкомпиляции.

Глава 2

Основная идея метода: полипрограммы и преобразования над ними

В этом разделе мы опишем нетипизированный функциональный язык первого порядка, который будем использовать в данной работе и введём понятие полипрограммы — центрального понятия данной диссертации. Мы также введём семантику полипрограмм и опишем общую схему предлагаемого метода, в частности перечислим используемые преобразования.

В данной главе отсутствуют некоторые технические подробности, такие как работа с функциями с точностью до перестановок параметров, порядок применения правил и обоснование корректности слияния по бисимуляции. Этим вопросам будут посвящены оставшиеся главы диссертации, а данную главу можно рассматривать как ознакомительную.

2.1 Используемые обозначения

Во всей оставшейся части диссертации будем использовать символ $\equiv$ для разделения частей определений функций программ и полипрограмм, чтобы отличить его от метатеоретического равенства $=$. Кроме того, часто будем использовать запись $\overline{e_i}$ для обозначения списков вида $e_1, e_2, \ldots, e_n$ для неко-

торого натурального n (определяется из контекста). Значение n будем также обозначать как $\max i$. Также будем вкладывать такие конструкции друг в друга, например, $\overline{f_i(\overline{e_{ij}})}$ обозначает

$$f_1(e_{11}, \ldots, e_{1k_1}), \ldots, f_n(e_{n1}, \ldots, e_{nk_n})$$

для некоторого n и некоторых $k_1, \ldots, k_n$. То, какой индекс к какой верхней черте относится, определяется из контекста. Отметим, что нумерацию мы в таких конструкциях ведём всегда с 1.

Списки будем обозначать квадратными скобками $[a, b, c]$, нумерацию элементов будем вести с 1, т.е. $[a, b, c][1] = a$. Будем также использовать генераторы списков $[x + y \mid x \in xs, y \in ys]$.

Кортежи будем обозначать как в угловых скобках $\langle a, b, c\rangle$, так и в круглых (a, b, c). Первую запись мы предпочитаем использовать для целостных объектов (полипрограмма, граф), а вторую для простого возврата нескольких значений из функции. Доступ к элементам кортежа будем осуществлять при помощи функций π_i, например, $\pi_2(\langle a, b, c\rangle) = b$.

Для функций и морфизмов $\phi : A \to B$ домен будем обозначать как $dom(\phi) = A$, а кодомен как $cod(\phi) = B$. Область значений функции f будем обозначать как $\mathrm{Im}(f)$.

Если $f : A \to B$ — функция, то её сужение на подмножество $X \subseteq A$ будем обозначать как $\left(f\big|_X\right)$. Тождественные функции и морфизмы будем обозначать как id, иногда с указанием объекта или множества id_A. Если $A \subseteq B$, то будем использовать обозначение $id_{A\to B} = \left(id_B\big|_A\right)$.

Для выражения с подвыражениями $e_1, \ldots, e_n$ будем использовать запись $E\langle e_1, \ldots, e_n\rangle$ или $E\langle\overline{e_i}\rangle$. Для подстановки выражения e в выражение E вместо переменной x будем использовать запись $E\{x \mapsto e\}$.

Окончания доказательств будем обозначать символом $\square$, окончания примеров — символом $\triangle$.

2.2 Синтаксис

Синтаксис определений нашего входного языка описан на Рис. 2.1. Будем использовать соглашение, что переменные (x) обозначаются строчными буквами, имена функций (f) — строчными буквами или словами, начинающи-

Рис. 2.1 Синтаксис входного языка

Определение:

$def ::= f(\overline{x_i}) \equiv e$

Выражение:

$e ::= x$	переменная
$\mid f(\overline{e_i})$	вызов функции
$\mid C(\overline{e_i})$	конструктор
$\mid$ **case** e_0 **of** $\{\ \overline{C_i(\overline{x_{ij}}) \ \rightarrow \ e_i;}\ \}$	сопоставление с образцом

мися со строчных букв (например, add), а конструкторы (C) — прописными буквами или словами, начинающимися с прописной буквы. Также будем считать, что все имена функций и конструкторов имеют некоторую арность, обозначенную $arity(f)$ или соответственно $arity(C)$. Будем использовать обозначения $fv(e)$ для множества свободных переменных выражения e и $\overline{fv}(e)$ для списка свободных переменных (через запятую). Также будем опускать пустые скобки у конструкторов и функций нулевой арности (C вместо $C()$) и точки с запятой после определений функций и ветвей case-выражений.

Дополнительно потребуем от определений нашего входного языка, чтобы:

- переменные в левых частях определений и в образцах не повторялись (т.е. в одном списке переменных все переменные разные, но перекрытие переменных во вложенных конструкциях разрешается)
- имена конструкторов в сопоставлениях с образцом не повторялись и ветви были отсортированы по именам конструкторов некоторым образом (главное, чтобы в одной программе использовался один порядок)
- использования имён конструкторов и функций были согласованы с их арностью

Определение 1. *Программой* (на нашем языке) назовём набор определений (как на Рис. 2.1) такое, что каждой функции f, входящей в программу (т.е. упомянутой в каком-либо из этих определений), соответствует ровно одно определение вида $f(\overline{x_i}) \equiv \ldots$ из этого набора, а кроме того все переменные в определениях связаны либо левыми частями, либо образцами.

Т.е. нельзя, чтобы у функции не было определений, нельзя, чтобы было больше одного определения, и нельзя использовать необъявленные переменные ($\text{zero}() \equiv \text{sub}(x, x)$).

Для представления множеств программ, а также множества равенств над выражениями введём понятие полипрограммы.

Определение 2. *Полипрограммой* (на нашем языке) назовём набор определений (как на Рис. 2.1).

(Заметим, что любая программа также является и полипрограммой.)

В отличие от программ, в полипрограмме разрешены множественные определения одной и той же функции (причём определения могут в точности повторяться), а также функции без определений и определения, использующие необъявленные переменные. Если рассматривать определение как уравнение (относительно неизвестных-функций), то полипрограмма является системой уравнений, т.е. семантические функции-решения должны удовлетворять всем определениям (более строго мы введём денотационную семантику в следующем разделе).

Функции без определений нужны для того, чтобы любые фрагменты полипрограмм (т.е. полипрограммы-подмножества некоторой другой полипрограммы) были также полипрограммами. Использование необъявленных переменных позволяет записывать утверждения вида $\forall x \,.\, 0 = x - x$ без потери информации о том, что x может быть любым (при преобразовании в программу такие переменные можно заменять на произвольные значения).

Полипрограммы полезны при анализе и преобразовании программ, поскольку позволяют одновременно работать с несколькими представлениями одной и той же функции.

Множественные определения приводят к некоторым интересным последствиям. Во-первых, некоторые определения могут противоречить друг другу, например, как в этом случае:

$$f() \equiv C()$$
$$f() \equiv D()$$

Т.к. одно и то же значение не может быть равно разным конструкторам (это верно для семантики, которую мы введём в дальнейшем), то данная полипрограмма имеет пустое множество моделей и представляет пустое множе-

ство программ. Такие полипрограммы будем называть *несовместными*. Нас в основном будут интересовать *совместные* полипрограммы, т.е. без таких противоречий.

Во-вторых, в полипрограммах изменяется смысл определений, которые определяют функцию не полностью, например:

$$f(x) \equiv f(f(x))$$

Стандартный подход к семантике программ в таких случаях говорит, что среди всех возможных (семантических) функций, удовлетворяющих данному определению как уравнению, надо брать наименьшую, т.е. $f(x) = \bot$ (другими словами, вычисление f не завершается). Однако если рассматривать данное определение как уравнение, то оно говорит, что функция f идемпотентна, и больше никак её не ограничивает (сам факт идемпотентности может быть в принципе полезен как лемма для доказательства свойств функции f). Действительно, в случае полипрограмм это определение функции f может быть не единственным:

$$f(x) \equiv f(f(x))$$
$$f(x) \equiv C()$$

Такая полипрограмма совместна, т.к. константная функция $f(x) = C$ удовлетворяет обоим определениям.

В данной работе мы отказываемся от стандартного подхода, предписывающего рассматривать наименьшую неподвижную точку, и рассматриваем все возможные неподвижные точки. Это делает семантику композиционной: семантика полипрограммы (множество моделей) является пересечением семантик её определений. В частности, это позволяет заменять фрагменты полипрограммы на эквивалентные им полипрограммы с сохранением семантики всей полипрограммы. Действительно, следующие два определения имеют одинаковые наименьшие неподвижные точки $\bot$ (мы считаем, что в случае ошибки, например, сопоставления с образцом незнакомого конструктора, воз-

вращается $\perp$):

$$f() \equiv f()$$

$$f() \equiv \textbf{case } C \textbf{ of } \{D \to D\}$$

Однако если заменить в следующей совместной полипрограмме одно определение на другое

$$f() \equiv f()$$
$$f() \equiv C()$$

то получится несовместная полипрограмма:

$$f() \equiv \textbf{case } C \textbf{ of } \{D \to D\}$$
$$f() \equiv C()$$

Поэтому совпадение только наименьших неподвижных точек недостаточно для задачи преобразования полипрограмм. Однако если у полипрограмм-фрагментов полностью совпадают множества неподвижных точек, то подобная замена становится корректной.

На практике для задачи доказательства эквивалентности рассмотрение всех неподвижных точек означает, что мы теряем некоторую полноту — для некоторых равенств $f(\overline{x}) \equiv g(\overline{x})$, верных в стандартном подходе, мы не можем установить их истинность, поскольку они неверны в ненаименьших моделях, простейший пример — доказать, что $f() \equiv g()$ для программы:

$$f() \equiv f()$$
$$g() \equiv g()$$

Однако мы не теряем корректность, поэтому если при помощи нашего метода удаётся доказать некоторое свойство программы, то оно будет верно и для стандартного подхода с наименьшей неподвижной точкой.

Теперь разберём, как полипрограммы представляют множества программ. Для удобства введём понятие монопрограммы — оно будет занимать проме-

жуточное положение между полипрограммой и программой.

Определение 3. *Монопрограммой* назовём полипрограмму, в которой каждая функция имеет не более одного определения.

Монопрограмма — почти программа, чтобы достроить её до программы, надо всего лишь добавить определения $f(\overline{x_i}) \equiv \bot$ для функций без определений и заменить несвязанные переменные на $\bot$ (т.е. определения вида $f() \equiv g(x)$ на $f() \equiv g(\bot)$).

Если для каждой функции полипрограммы, имеющей определение, выбрать по одному определению, то получится фрагмент полипрограммы, являющийся монопрограммой. Однако такая монопрограмма может не быть эквивалентна исходной полипрограмме, поэтому добавим ещё ограничение, что монопрограмма должна быть семантически эквивалентна исходной полипрограмме (что значит "семантически эквивалентна" будет сказано в следующем разделе). Таким образом, мы получим множество монопрограмм (а значит и программ), представляемое полипрограммой.

Задача доказательства эквивалентности полипрограммы и выделенной из неё монопрограммы алгоритмически неразрешима, поэтому на практике мы обычно не можем построить всё множество монопрограмм. Однако можно приблизить его снизу, используя алгоритмически разрешимые признаки этой эквивалентности. Задача выделения монопрограмм из полипрограмм не является предметом настоящей диссертации, потому что её решение не требуется для решения задачи доказательства эквивалентности, однако преобразование слияния по бисимуляции очень похоже по своей сути на построение монопрограммы, и также требует использования подобных признаков. Слияние по бисимуляции будет рассмотрено в Разделе 2.5 и более подробно в Главе 5.

Пример 1. Рассмотрим следующую полипрограмму:

$$\mathrm{not}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}$$
$$\mathrm{even}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{odd}(y)\ \}$$
$$\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))$$
$$\mathrm{odd}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ F;\ S(y)\ \to\ \mathrm{even}(y)\ \}$$
$$\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))$$

Мы можем выделить из неё четыре монопрограммы, выбирая по одному определению для каждой функции. Три из них будут эквивалентны исходной полипрограмме:

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{odd}(y)\ \}\\
&\mathrm{odd}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ F;\ S(y)\ \to\ \mathrm{even}(y)\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{odd}(y)\ \}\\
&\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))\\
&\mathrm{odd}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ Z\ \to\ F;\ S(y)\ \to\ \mathrm{even}(y)\ \}
\end{aligned}$$

Но одна не будет эквивалентна исходной полипрограмме (даже наименьшие неподвижные точки у них различны):

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))\\
&\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))
\end{aligned}$$

Поэтому исходная полипрограмма представляет множество монопрограмм, состоящее из трёх вышеуказанных монопрограмм, не включающее четвёртую. △

Можно возразить, что последний пример был построен искусственно, потому что на практике полипрограммы получаются из программ применением преобразований, и можно ввести преобразования полипрограмм таким образом, чтобы такие “плохие” полипрограммы не возникали — тогда можно будет наивно извлекать только эквивалентные программы из полипрограммы. Хотя в принципе такой подход возможен, он требует довольно сильных ограничений на преобразования, поскольку даже очень простые преобразования

могут приводить к “плохим” полипрограммам, как будет показано в следующем примере.

Пример 2. Рассмотрим следующую программу:

$$f(x) \equiv C$$
$$g(x) \equiv f(f(x))$$

Раскроем вызов $f(x)$ в теле функции $g(x)$ (новые определения будем просто дописывать внизу):

$$f(x) \equiv C$$
$$g(x) \equiv f(f(x))$$
$$g(x) \equiv C$$

Теперь произведём свёртку, заменив C, эквивалентное правой части определения f, на f:

$$f(x) \equiv C$$
$$g(x) \equiv f(f(x))$$
$$g(x) \equiv C$$
$$g(x) \equiv f(x)$$

Теперь снова произведём свёртку, заменив f в правой части второго определения на g:

$$f(x) \equiv C$$
$$g(x) \equiv f(f(x))$$
$$g(x) \equiv C$$
$$g(x) \equiv f(x)$$
$$g(x) \equiv g(g(x))$$

Из данной полипрограммы можно извлечь даже несколько разнообразных программ, не эквивалентных исходной. △

Можно заметить, что свёртка в последнем примере является опасным пре-

образованием, приведшим к "плохой" полипрограмме. Поэтому вместо того, чтобы накладывать дополнительные условия при извлечении монопрограмм или при выполнении слияния по бисимуляции, можно было бы ограничить свёртку таким образом, чтобы "плохие" полипрограммы в принципе не возникали. Однако свёртка слишком полезна, чтобы её запрещать или ограничивать, а в насыщении равенствами является вообще фундаментальным преобразованием, выполняющимся всегда, когда появляется такая возможность. Отметим, что аналогичная проблема возникает, например, и в суперкомпиляции высшего уровня, где она решается как раз наложением ограничений на преобразования (например, в суперкомпиляторе Ключникова [46] могут быть использованы только улучшающие леммы в смысле теории Сэндса [64]).

Пример 3. Бывают полипрограммы, из которых нельзя выделить ни одной монопрограммы, эквивалентной исходной. Например, это может быть связано с тем, что нет ни одного самодостаточного определения, как в этой полипрограмме:

$$f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \rightarrow\ A;\ B\ \rightarrow\ f(x)\ \}$$
$$f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \rightarrow\ f(x);\ B\ \rightarrow\ B\ \}$$

Здесь эти два определения вместе однозначно задают функцию такую, что $f(A) = A, f(B) = B$, но по отдельности они задают её только на какой-то своей части возможных входных данных (например, если использовать только первое определение, то мы знаем, что $f(A) = A$, а про $f(B)$ мы знаем только то, что оно равно самому себе — обычно в этом случае полагают, что $f(B) = \bot$). Если полипрограмма состоит только из этих двух определений, то из неё невозможно выделить программу, эквивалентную всей полипрограмме, и являющейся её фрагментом. Отметим, впрочем, что можно "развернуть" это определение, введя дополнительную функцию, однако такая программа не будет фрагментом исходной полипрограммы:

$$f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \rightarrow\ A;\ B\ \rightarrow\ g(x)\ \}$$
$$g(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \rightarrow\ f(x);\ B\ \rightarrow\ B\ \}$$

$\triangle$

2.3 Семантика

Для доказательства корректности преобразований полипрограмм нужно ввести семантику языка, относительно которой мы и будем доказывать корректность. Мы введём семантику в денотационном стиле, однако будем рассматривать не наименьшую неподвижную точку, а все неподвижные точки. Этот подход к денотационной семантике отличается от привычного подхода, однозначно интерпретирующего программу как наименьшую неподвижную точку [67], но он обеспечивает композиционность семантики на уровне определений, позволяя заменять фрагменты полипрограмм на семантически эквивалентные фрагменты.

Аналогично логике предикатов первого порядка мы введём понятие интерпретации и модели, однако интерпретации будут задавать только значения пользовательских функций (функциональных символов) нашей полипрограммы, смысл же конструкций языка будет фиксирован.

Интерпретация будет отображать каждый функциональный символ из полипрограммы в непрерывную функцию типа $[\mathbb{A}^n \to \mathbb{A}]$, где $\mathbb{A}$ — множество данных первого порядка, являющееся доменом Скотта, удовлетворяющим следующему уравнению [76]:

$$\mathbb{A} \cong \bigoplus_{C} \underbrace{\mathbb{A} \times \ldots \times \mathbb{A}}_{arity(C) \text{ раз}}$$

Также $\mathbb{A}$ можно считать *наибольшей* неподвижной точкой следующего определения[1]:

$$\mathbb{A} = \{C(\overline{a_i}) \mid a_i \in \mathbb{A},\ C \text{ — конструктор}\} \cup \{\bot\}$$

Таким образом, $\mathbb{A}$ является множеством конечных и бесконечных деревьев, построенных из конструкторов и $\bot$.

Частичный порядок $\sqsubseteq$ на $\mathbb{A}$ также можно определить как наибольшую неподвижную точку следующего определения:

$$a \sqsubseteq b = (a = \bot) \vee (a = C(\overline{a_i}) \wedge b = C(\overline{b_i}) \wedge \bigwedge_i a_i \sqsubseteq b_i)$$

[1] Для того, чтобы наибольшая неподвижная точка давала то, что мы хотим, то есть в том числе и бесконечные деревья из конструкторов, необходимо, чтобы теория множеств, в которой мы работаем, поддерживала такие циклические конструкции [7].

Отметим, что $\mathbb{A}$ — полурешётка с операцией взятия точной нижней грани $\sqcap$. Но это не решётка, т.к. операция взятия точной верхней грани $\sqcup$ является частичной.

Определение 4. Пусть A — частично упорядоченное множество. Тогда A^n — частично упорядоченное множество кортежей, причём

$$(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n) \Leftrightarrow \forall i \; a_i \sqsubseteq b_i$$

Нам понадобится несколько определений из теории доменов, а именно понятие наименьшей верхней грани и понятие непрерывности по Скотту [68].

Определение 5. Пусть A — частично упорядоченное множество (в частности, $\mathbb{A}$). *Верхней гранью* множества $X \subseteq A$ называется элемент $a \in A$ такой, что $\forall x \in X \Rightarrow x \sqsubseteq a$. *Наименьшей верхней гранью* (супремумом) множества X называется такая верхняя грань a, что для любой другой верхней грани a' выполняется $a \sqsubseteq a'$. Обозначение: $a = \sup X$.

Определение 6. Пусть A, B — частично упорядоченные множества. Функция $f : A \to B$ является *непрерывной*, если для любой монотонной последовательности $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ элементов из A такой, что $\sup\{a_i\} = a$ верно, что $f(a) = \sup\{f(a_i)\}$.

Через $\mathbb{D}$ обозначим множество всех непрерывных функций над $\mathbb{A}$ конечной арности, т.е. $\mathbb{D} = \bigcup_n [\mathbb{A}^n \to \mathbb{A}]$.

Определение 7. Пусть G — полипрограмма, а F — множество функциональных символов данной полипрограммы.

Интерпретацией полипрограммы G будем называть функцию $\eta : F \to \mathbb{D}$ такую, что $arity(\eta(f)) = arity(f)$.

Моделью полипрограммы G будем называть такую интерпретацию μ, что $\mu \models G$, где $\mu \models G \stackrel{\text{def}}{=} \bigwedge_{p \in G} [\![p]\!]_\mu$, т.е. в μ выполняются все определения p полипрограммы G. $[\![p]\!]_\mu$ определено на Рис. 2.2.

Таким образом, если все определения полипрограммы истинны в некоторой интерпретации, то такая интерпретация называется моделью полипрограммы. Нам также понадобится понятие семантической эквивалентности полипрограмм.

Рис. 2.2 Семантика определений

$$(\!|f(\overline{e_j})|\!)_{\mu\sigma} = \mu(f)(\overline{(\!|e_j|\!)_{\mu\sigma}})$$
$$(\!|x|\!)_{\mu\sigma} = \sigma(x)$$
$$(\!|C(\overline{e_j})|\!)_{\mu\sigma} = C(\overline{(\!|e_j|\!)_{\mu\sigma}})$$
$$(\!|\textbf{case } e_0 \textbf{ of } \{\overline{C_j(\overline{x_i}) \to e_j}\}|\!)_{\mu\sigma} =$$
$$= \begin{cases} (\!|e_j|\!)_{\mu\tau} & \text{если } (\!|e_0|\!)_{\mu\sigma} = C_j(\overline{a_i}) \\ & \text{где } \tau(y) = a_i, \text{ если } y = x_i, \text{ и } \tau(y) = \sigma(y) \text{ иначе} \\ \bot & \text{если } (\!|e_0|\!)_{\mu\sigma} = C_l(\ldots) \neq C_j(\ldots) \forall j \\ \bot & \text{если } (\!|e_0|\!)_{\mu\sigma} = \bot \end{cases}$$

$$[\![t_1 \equiv t_2]\!]_\mu \Leftrightarrow \forall \sigma : V \to \mathbb{A} \,.\, (\!|t_1|\!)_{\mu\sigma} = (\!|t_2|\!)_{\mu\sigma}$$
(σ — контекст, отображающий имена переменных из V в значения)

Определение 8. Две полипрограммы G_1 и G_2 со множествами функциональных символов F_1 и F_2 будем называть *семантически эквивалентными* на подмножестве функциональных символов $F \subseteq F_1 \cap F_2$, если для любой модели $\mu_1 \models G_1$ существует модель $\mu_2 \models G_2$ такая, что $\forall f \in F \,.\, \mu_1(f) = \mu_2(f)$, и наоборот, для любой модели $\eta_2 \models G_2$ существует модель $\eta_1 \models G_1$ такая, что $\forall f \in F \,.\, \eta_1(f) = \eta_2(f)$.

Рассмотрим теперь некоторые особенности введённой таким образом семантики полипрограмм.

Во-первых, как уже было сказано, бывает так, что две полипрограммы (даже программы) имеют одинаковые наименьшие неподвижные точки (модели), однако разное множество всех неподвижных точек (моделей), и поэтому мы не считаем их эквивалентными, например, полипрограмма

$$f(x) \equiv f(f(x))$$

обладает такими моделями, что $\mu(f)$ — идемпотентная функция, а для полипрограммы

$$f(x) \equiv f(x)$$

вообще любая интерпретация является моделью.

Отметим, что полипрограмма, в отличие от программы, может не иметь наименьшей модели, даже если полипрограмма совместна. Например, следующая полипрограмма не допускает, чтобы g было равно $\bot$:

$$f \equiv \textbf{case } g \textbf{ of} \{A \to C; B \to C\}$$
$$f \equiv C$$

Все её модели — это $\{f = C, g = A\}$ и $\{f = C, g = B\}$, однако они несравнимы. Тем не менее, наш метод преобразования так устроен, что если изначально полипрограмма имела наименьшую модель (а это так, потому что полипрограммы мы изначально производим из программ), то и преобразованная полипрограмма будет иметь наименьшую модель.

Интересно, что в нашем подходе к семантике можно записать незавершающуюся программу так, чтобы она была определена однозначно (имела единственную модель), например, как в этом случае:

$$\text{inf} \equiv S(\text{inf})$$
$$\text{eat}(x) \equiv \textbf{case } x \textbf{ of } S(x') \ \to \ \text{eat}(x')$$
$$\text{bottom} \equiv \text{eat}(\text{inf})$$

Здесь inf является бесконечным числом $S(S(S(\ldots)))$, а функция eat просто разбирает свой аргумент пока не упрётся в $\bot$ или в незнакомый конструктор. Значение функции bottom будет равно $\bot$ во всех моделях. Действительно, т.к. функция $\mu(\text{eat})$ должна быть непрерывной, а на всех значениях вида $S^n(\bot)$ она равна $\bot$ по определению конструкций языка, то и на бесконечном $S(S(\ldots))$ она также равна $\bot$. Здесь крайне важно, что в нашей семантике в качестве интерпретаций функциональных символов допускаются только непрерывные функции. Альтернативный способ доказательства единственности модели данной программы — использование признаков структурной и защищённой рекурсии, что подробно рассматривается в Главе 5.

Ещё одной особенностью нашего входного языка, усложняющей некоторые примеры, является его нетипизированность. Из-за этого в функцию можно передавать конструкторы, которые она не ожидает, и мы считаем, что в этом случае функция возвращает $\bot$. Это опять же означает, что некоторые

эквивалентности, верные в типизированном языке, в нашем случае не выполняются. Примером может служить следующая пара функций:

$$
\begin{aligned}
&f(x) \equiv x \\
&g(x) \equiv \textbf{case } x \textbf{ of} \\
&\quad Z \to Z \\
&\quad S(x') \to S(g(x'))
\end{aligned}
$$

Пусть также задан тип

$$\textbf{data } \mathrm{Nat} = Z \mid S(\mathrm{Nat})$$

В типизированном языке ясно, что $g(x) = f(x)$ для любого $x : \mathrm{Nat}$ (причём ограничение на тип x в таких случаях обычно выводится автоматически). Однако мы работаем с нетипизированным языком, поэтому данное равенство неверно, т.к. x может принимать значения, отличные от описываемых типом Nat (например, $f(C) = C$, но $g(C) = \bot$). Тем не менее, сформулировать подобные утверждения на нетипизированном языке так, чтобы они были доказуемыми, можно. Для этого нужно обернуть входные переменные в функцию “приведения типов”, заменяющую неизвестные конструкторы на $\bot$. Например, для натуральных чисел эта функция выглядит так:

$$
\begin{aligned}
&\mathrm{toNat}(x) \equiv \textbf{case } x \textbf{ of} \\
&\quad Z \to Z \\
&\quad S(x') \to S(\mathrm{toNat}(x'))
\end{aligned}
$$

Для списков она может быть параметризована типом элементов:

$$
\begin{aligned}
&\mathrm{toList}(f, x) \equiv \textbf{case } x \textbf{ of} \\
&\quad \mathit{Nil} \to \mathit{Nil} \\
&\quad \mathit{Cons}(a, x') \to \mathit{Cons}(f(a), \mathrm{toList}(f, x'))
\end{aligned}
$$

Интересно, что она в точности совпадает со стандартной функцией map. Теперь, например, утверждение $\mathrm{add}(Z, y) \equiv \mathrm{add}(y, Z)$, неверное в нашей семантике, можно переписать как $\mathrm{add}(Z, \mathrm{toNat}(y)) \equiv \mathrm{add}(\mathrm{toNat}(y), Z)$, превратив

его в верное.

Наконец, отметим, что описанный язык является языком первого порядка. Задачу доказательства эквивалентности мы также будем рассматривать именно для случая первого порядка, однако задача доказательства эквивалентности для языка с функциями высших порядков может быть сведена при помощи дефункционализации [59] к задаче доказательства эквивалентности для языка первого порядка (корректность такого сведения следует из [5]). Задача дефункционализации не является темой настоящей диссертации, поэтому далее мы будем работать только с языком первого порядка, однако возможность работы с функциями высших порядков реализована в нашем экспериментальном прувере, и многие примеры, на которых производилось тестирование, используют функции высших порядков.

2.3.1 Формальная постановка задачи

Теперь у нас есть все основные понятия нужные для того, чтобы сформулировать задачу формально. Основная задача преобразования полипрограмм звучит следующим образом:

> Дана полипрограмма P_1. Построить полипрограмму P_2 такую, что P_1 и P_2 семантически эквивалентны на функциональных символах полипрограммы P_1.

Решению этой задачи по большей части и посвящена оставшаяся часть диссертации. Для простоты мы считаем, что нужно сохранить все исходные функции, хотя можно было бы ограничиться только каким-то подмножеством интересных функций.

К этой задаче могут быть сведена задача доказательства эквивалентности. Задача доказательства эквивалентности звучит следующим образом:

> Дана программа P выражение вида $f \equiv g$, где f и g — функции данной программы с одинаковой арностью. Выдать ответ “да” или “не знаю”, причём если дан ответ “да”, то для любой модели μ программы P должно выполняться $\mu(f) = \mu(g)$.

Действительно, чтобы решить задачу эквивалентности, достаточно рассмотреть программу P как полипрограмму, преобразовать её в полипрограмму

P' алгоритмом, решающим задачу преобразования, и проверить, существует ли в полипрограмме P' функция h и два определения $f(\overline{x_i}) \equiv h(\overline{x_i})$ и $g(\overline{x_i}) \equiv h(\overline{x_i})$.

Отметим, что в постановке задач мы не требуем дополнительной генерации доказательства утверждения. С теоретической точки зрения это не имеет особого значения — корректность приведённого метода будет доказана в данной диссертации. С практической точки зрения это достаточно существенный недостаток, потому что как в доказательстве корректности метода, так и в практической реализации этого метода могут быть ошибки. Приведённый метод может быть модифицирован так, чтобы генерировать также и доказательство [57], однако в нашем прувере данная возможность пока не реализована.

2.4 Правила преобразования полипрограмм

Преобразования полипрограмм удобно формулировать в виде правил. Часто правила переписывания формулируются в виде $e_1 \mapsto e_2$, где e_1 и e_2 — выражения, однако в случае полипрограмм необходимо работать не с выражениями, а с определениями, поэтому правила будут иметь вид $G_1 \mapsto G_2$, где G_1 и G_2 — полипрограммы. Правила могут удалять определения и добавлять определения и функции (удалять функции не могут). Записывать их мы будем следующим образом:

$$\left\{\begin{array}{l}\text{удаляемое определение}\\ \text{определение}_1\\ \text{определение}_2\end{array}\right\} \mapsto \left\{\begin{array}{l}\text{новое определение}\\ \text{определение}_1\\ \text{определение}_2\end{array}\right\}$$

Применение такого правила к полипрограмме G заключается в замене фрагмента, соответствующего левой части правила (с точностью до переименования функций и переменных), на фрагмент, соответствующий правой части. В этой главе будем считать, что эта процедура интуитивно понятна, а более формальные определения введём в следующей главе.

Также будем считать очевидным, что если в правиле левая часть семантически эквивалентна правой на функциях из левой части, то исходная полипрограмма также семантически эквивалентна преобразованной на функциях

исходной полипрограммы.

Отметим, что правила, удаляющие определения, могут уменьшать мощность метода, т.к. они приводят к тому, что выводится меньше эквивалентностей. Однако с практической точки зрения они необходимы, иначе будет происходить слишком сильное разрастание полипрограммы. В данной главе для простоты будем считать, что правила не удаляют определения, а в следующих главах уточним некоторые правила так, что они будут удалять определения.

Начнём с технического правила, которое понадобится нам только в этой главе.

Расчленение

$$\left\{ f(\overline{x_i}) \equiv E\langle e' \rangle \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle e' \rangle \\ f(\overline{x_i}) \equiv E\langle g(\overline{fv}(e')) \rangle \\ g(\overline{fv}(e')) \equiv e' \end{array} \right\}$$

Данное правило выносит любое подвыражение в качестве отдельной функции. В сочетании с правилом раскрытия вызова функции оно позволяет стереть грань между подвыражениями и вызовами функций. Оно необходимо из-за особенностей формулировки наших правил. В дальнейших главах этого правила нет, поскольку мы введём понятие полипрограммы в расчленённой форме, в котором все достаточно сложные подвыражения будут вынесены в качестве отдельных функций, причём все правила будут сохранять расчленённость полипрограммы.

Раскрытие вызова функции

$$\left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j}) \rangle \\ g(\overline{y_j}) \equiv H \end{array} \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{y_j}) \rangle \\ g(\overline{y_j}) \equiv H \\ f(\overline{x_i}) \equiv E\langle H\{\overline{y_j \mapsto e_j}\} \rangle \end{array} \right\}$$

Данное правило подставляет тело функции в место вызова (одновременно подставляя выражения-аргументы в тело). Оно достаточно стандартно. Отметим, что в следующих главах оно распадётся на несколько отдельных правил, довольно непохожих на данное правило (чуть более сложных для по-

нимания, но более простых для реализации). Мы также будем считать, что процесс подстановки интуитивно понятен, потому что в последующих главах он нам не понадобится — в нашей реализации подстановка фактически является не атомарной операцией, а разбита на отдельные шаги, представляющие из себя отдельные правила (это похоже на явные подстановки).

Правило раскрытия вызова функции вместе с правилом расчленения позволяет применять любое правило вида

$$\left\{ f(\overline{x_i}) \equiv E \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E \\ f(\overline{x_i}) \equiv H \end{array} \right\}$$

не к отдельным определениям, а к подвыражениям:

$$\left\{ g(\overline{x_i}) \equiv G\langle E \rangle \right\} \mapsto \left\{ \begin{array}{l} g(\overline{x_i}) \equiv G\langle E \rangle \\ g(\overline{x_i}) \equiv G\langle h(\overline{x_i}) \rangle \\ h(\overline{x_i}) \equiv E \end{array} \right\} \mapsto$$

$$\mapsto \left\{ \begin{array}{l} g(\overline{x_i}) \equiv G\langle E \rangle \\ g(\overline{x_i}) \equiv G\langle h(\overline{x_i}) \rangle \\ h(\overline{x_i}) \equiv E \\ h(\overline{x_i}) \equiv H \end{array} \right\} \mapsto \left\{ \begin{array}{l} g(\overline{x_i}) \equiv G\langle E \rangle \\ g(\overline{x_i}) \equiv G\langle h(\overline{x_i}) \rangle \\ h(\overline{x_i}) \equiv E \\ h(\overline{x_i}) \equiv H \\ g(\overline{x_i}) \equiv G\langle H \rangle \end{array} \right\}$$

При этом получается много дополнительных промежуточных определений, но их можно выкинуть. Т.к. вышеуказанным образом записывается большинство правил, мы будем неявно использовать этот приём, чтобы сократить выкладки.

Транзитивность/Свёртка

$$\left\{ \begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \end{array} \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv g(\overline{y_j}) \end{array} \right\}$$

Данное правило говорит, что если функции имеют два определения с одинаковыми правыми частями, то они равны.

Конгруэнтность

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \\ f(\overline{x_i}) \equiv E\langle h(\overline{e_{\theta(j)}})\rangle \end{array}\right\}$$

$$\left\{\begin{array}{l} g(\overline{x_i}) \equiv E \\ g(\overline{x_i}) \equiv h(\overline{x_{\theta(i)}}) \end{array}\right\} \mapsto \left\{\begin{array}{l} g(\overline{x_i}) \equiv E \\ g(\overline{x_i}) \equiv h(\overline{x_{\theta(i)}}) \\ h(\overline{x_{\theta(i)}}) \equiv E \end{array}\right\}$$

Очень важное правило, которое позволяет распространить информацию о равенстве функций путём замены вхождений одного функционального символа на другой. Замену можно производить как слева, так и справа. Т.к. в равенстве двух функций переменные могут идти в разном порядке, правило параметризовано перестановкой θ.

Деструктивную версию данного преобразования, которую мы рассмотрим в Главе 4, удобно применять сразу для всех вхождений, чтобы полностью заменить одну функцию на другую.

Пример 4. Уже введённых правил достаточно для того, чтобы доказывать простейшие эквивалентности. Рассмотрим следующую полипрограмму:

$$\begin{aligned} &f_1(x,y) \equiv C(g(x),h(y)) \\ &f_2(x,y) \equiv C(h(y),g(x)) \\ &g(x) \equiv A(x,g(S(x))) \\ &h(x) \equiv A(x,A(S(x),g(S(S(x))))) \end{aligned}$$

Сначала раскроем вызов функции g в определении $g(x) \equiv A(x,g(S(x)))$ и получим новое определение, добавляемое к полипрограмме:

$$g(x) \equiv A(x,A(S(x),g(S(S(x)))))$$

Это определение имеет ту же правую часть, что и определение функции h, поэтому можно применить транзитивность:

$$g(x) \equiv h(x)$$

Теперь можно применить конгруэнтность и заменить g на h в определениях функций f_1 и f_2:

$$f_1(x, y) \equiv C(h(x), h(y))$$
$$f_2(x, y) \equiv C(h(y), h(x))$$

Можно заметить, что если переименовать переменные в одном из определений, например, во втором $f_2(y, x) \equiv C(h(x), h(y))$, то получится, что у двух определений одинаковые правые части, а значит можно применить транзитивность:

$$f_1(x) \equiv f_2(x)$$

Таким образом, мы доказали эквивалентность функций f_1 и f_2. $\triangle$

Симметричность

$$\left\{ f(\overline{x_i}) \equiv g(\overline{y_j}) \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv g(\overline{y_j}) \\ g(\overline{y_j}) \equiv f(\overline{x_i}) \end{array} \right\}$$

Симметричность равенства может быть нужна главным образом для того, чтобы применить понижение арности в другую сторону для функции справа.

Понижение арности

$$\left\{ \begin{array}{l} f(\overline{x_i}, \overline{y_j}) \equiv E, \\ \qquad \text{где } \{\overline{x_i}\} \subseteq fv(E) \end{array} \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}, \overline{y_j}) \equiv E, \\ f(\overline{x_i}, \overline{y_j}) \equiv g(\overline{x_i}) \end{array} \right\}$$

Это правило позволяет удалять неиспользуемые переменные из функции путём введения новой функции меньшей арности. Потом старая функция f заменяется на g при помощи правила конгруэнтности.

Инъективность конструкторов

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv C(\overline{E_j}) \\ f(\overline{x_i}) \equiv C(\overline{H_j}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv C(\overline{E_j}) \\ f(\overline{x_i}) \equiv C(\overline{H_j}) \\ \overline{g_j(\overline{x_i}) \equiv E_j} \\ \overline{g_j(\overline{x_i}) \equiv H_j} \end{array}\right\}$$

Инъективность сопоставления с образцом

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_j}) \to E_k}\} \\ f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_j}) \to H_k}\} \\ \qquad \text{где } x \notin fv(E_k) \text{ и } x \notin fv(H_k) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\} \\ f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to H_k}\} \\ \overline{g_k(\overline{x_i}, \overline{y_{jk}}) \equiv E_k} \\ \overline{g_k(\overline{x_i}, \overline{y_{jk}}) \equiv H_k} \end{array}\right\}$$

Редукция сопоставления с образцом

$$\left\{ f(\overline{x_i}) \equiv \textbf{case}\ C(\overline{e_j})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to E; \ldots\} \right\} \mapsto$$
$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ C(\overline{e_j})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to E; \ldots\} \\ f(\overline{x_i}) \equiv E\{\overline{y_j \mapsto e_j}\} \end{array}\right\}$$

Распространение позитивной информации

$$\left\{ f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\} \right\} \mapsto$$
$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\} \\ f(\overline{x_i}) \equiv \textbf{case}\ x\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k\{x \mapsto C_k(\overline{y_{jk}})\}}\} \end{array}\right\}$$

Сопоставление с образцом результата сопоставления с образцом

$$\left\{ f(\overline{x_i}) \equiv \textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\})\ \textbf{of}\ \{\ldots\} \right\} \mapsto$$
$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\})\ \textbf{of}\ \{\ldots\} \\ f(\overline{x_i}) \equiv \textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to \textbf{case}\ E_k\ \textbf{of}\ \{\ldots\}}\} \end{array}\right\}$$

Перед применением данного правила переменные переименовываются так, чтобы не было конфликтов имён.

Изменение порядка сопоставлений с образцом

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to \textbf{case}\ e'\ \textbf{of}\ \{\overline{D_l(\overline{z_{jl}}) \to h_{kl}}\}}\} \\ \quad \text{где } fv(e') \subseteq \{\overline{x_i}\} \end{array}\right\} \mapsto$$

$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to \textbf{case}\ e'\ \textbf{of}\ \{\overline{D_l(\overline{z_{jl}}) \to h_{kl}}\}}\} \\ f(\overline{x_i}) \equiv \textbf{case}\ e'\ \textbf{of}\ \{\overline{D_l(\overline{z_{jl}}) \to \textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to h_{kl}}\}}\} \end{array}\right\}$$

Также перед применением данного правила переменные переименовываются так, чтобы не было конфликтов имён.

Пример 5. Рассмотрим классический пример доказательства того, что сложение ассоциативно

(1) $\mathrm{add}(x, y) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to S(\mathrm{add}(x', y))\}$

(2) $f(x, y, z) \equiv \mathrm{add}(\mathrm{add}(x, y), z)$

(3) $g(x, y, z) \equiv \mathrm{add}(x, \mathrm{add}(y, z))$

В этой полипрограмме определены функции f и g, и нужно доказать, что они эквивалентны. Начнём с преобразования функции f. Сначала раскроем внешний вызов add в определении (2):

(4) $f(x, y, z) \equiv$

$\quad \textbf{case}\ \mathrm{add}(x, y)\ \textbf{of}$

$\quad\quad Z \to z$

$\quad\quad S(x') \to S(\mathrm{add}(x', z))$

Теперь раскроем add в разбираемой позиции в определении (4):

(5) $f(x, y, z) \equiv$

$\quad \textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to S(\mathrm{add}(x', y))\})\ \textbf{of}$

$\quad\quad Z \to z$

$\quad\quad S(x') \to S(\mathrm{add}(x', z))$

Теперь применим сопоставление с образцом результата сопоставления с об-

разцом:

$$(6)\ f(x,y,z) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to \textbf{case}\ y\ \textbf{of}\{Z \to z; S(x') \to S(\mathrm{add}(x', z))\}$$
$$S(x') \to \textbf{case}\ S(\mathrm{add}(x', y))\ \textbf{of}\{Z \to z; S(x') \to S(\mathrm{add}(x', z))\}$$

Теперь в определении (6) в ветви Z производим свёртку (применяя расчленение, транзитивность, конгруэнтность и раскрытие вызова), а в ветви S — редукцию сопоставления с образцом:

$$(7)\ f(x,y,z) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to \mathrm{add}(y, z)$$
$$S(x') \to S(\mathrm{add}(\mathrm{add}(x', y), z))$$

Теперь уже в ветви S применяем свёртку

$$(8)\ f(x,y,z) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to \mathrm{add}(y, z)$$
$$S(x') \to S(f(x', y, z))$$

Теперь проделаем примерно то же самое с функцией g. Раскрываем внешний вызов add в определении (3):

$$(9)\ g(x,y,z) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to \mathrm{add}(y, z)$$
$$S(x') \to S(\mathrm{add}(x', \mathrm{add}(y, z)))$$

Теперь производим свёртку в ветви S:

$$
\begin{aligned}
&(10)\, g(x, y, z) \equiv \\
&\qquad \textbf{case } x \textbf{ of} \\
&\qquad\qquad Z \to \mathrm{add}(y, z) \\
&\qquad\qquad S(x') \to S(g(x', y, z))
\end{aligned}
$$

Можно заметить, что определения (8) и (10) совпадают с точностью до переименования функций f на g, однако транзитивность к ним не применима. Чтобы вывести отсюда $f \equiv g$, нужно ещё одно преобразование, которое выводит равенство функций исходя из того, что они имеют одинаковые рекурсивные определения. Это преобразование мы рассмотрим в следующем разделе. △

2.5 Слияние по бисимуляции

При помощи правил транзитивности и конгруэнтности можно добиться слияния двух функций, представленных одинаковыми *ациклическими* фрагментами полипрограммы. Однако если функции представлены фрагментами, имеющими циклы, эта процедура не работает. Простейший пример — два бесконечных числа:

$$
\begin{aligned}
f &\equiv S(f) \\
g &\equiv S(g)
\end{aligned}
$$

Этот факт мотивирует поиск некоторого преобразования полипрограмм, которое бы могло сливать два произвольных изоморфных фрагмента. Оказывается, однако, что требование изоморфизма можно ослабить до отношения, которое мы назовём бисимуляцией полипрограмм (являющееся обобщением понятия бисимуляции для размеченных систем переходов), поэтому мы будем называть это преобразование слиянием по бисимуляции. В данной главе мы введём урезанное понятие бисимуляции, не учитывающее, что в соответствующих друг другу функциях параметры могут идти в разном порядке.

Не все изоморфные фрагменты корректно сливать, например, следующие две функции входят в два изоморфных фрагмента, состоящих из одного опре-

деления каждый:

$$f(x) \equiv f(f(x))$$
$$g(x) \equiv g(g(x))$$

Но эти фрагменты говорят только, что соответствующие функции являются идемпотентными, из чего не следует, что они должны быть равны. Для решения этой проблемы на полипрограммы-фрагменты накладываются некоторые условия, подобные тем, которые применяются в тотальных языках для обеспечения завершаемости функций [1; 2], но в нашем случае обеспечивается не завершаемость, а единственность модели.

Слияние по бисимуляции является преобразованием, которое соответствует частному случаю применения доказательства по индукции или коиндукции для равенства двух функций. В некоторых случаях ему не хватает мощности, однако в большинстве случаев его достаточно для проведения индуктивных доказательств.

Введём теперь понятие бисимуляции (упрощённое).

Определение 9. Будем говорить, что две полипрограммы G_1 и G_2 со множествами функциональных символов F_1 и F_2 находятся в отношении бисимуляции, если существует третья полипрограмма H со множеством функциональных символов F и два отображения $\phi_i : F \to F_i, i = 1, 2$ таких, что если заменить в H все функциональные символы f на $\phi_i(f)$, то получится G_i с точностью до удаления определений-дубликатов. H будем называть полипрограммой бисимуляции.

Оказывается, что если в полипрограмме G существуют полипрограммы-фрагменты G_1 и G_2, находящиеся в отношении бисимуляции, причём H — полипрограмма этой бисимуляции, и если для каждой интерпретации μ_h таких функций h, что $\phi_1(h) = \phi_2(h)$ существует не более одной модели μ полипрограммы H такой, что $\mu(h) = \mu_h$, то в полипрограмму G можно добавить определения вида $\phi_1(f)(\overline{x_i}) \equiv \phi_2(f)(\overline{x_i})$, получив полипрограмму G', эквивалентную исходной полипрограмме G. Один из возможных алгоритмов поиска таких бисимуляций будет описан в Главе 5.

Отметим, что требуется не абсолютная единственность модели, а единственность модели для каждой интерпретации функций, которые отобража-

ются в одну и ту же функцию G. Такие функции соответствуют случаю доказательства по рефлексивности и бывает полезно во многих случаях.

Для того, чтобы убедиться, что H имеет не более одной модели, можно использовать признаки структурной и защищённой рекурсии [1; 2]. Тот факт, что в нашей семантике они действительно обеспечивают единственность модели, будет доказан в Главе 5. Там же будет приведена адаптация алгоритма проверки данных признаков для полипрограмм в бесточечно-расчленённом представлении. Пока приведём несколько иную формулировку данных признаков, которой нам будет достаточно в данной главе.

Нам понадобится следующее отношение порядка на значениях из $\mathbb{A}$.

Определение 10. Пусть $a, b \in \mathbb{A}$. Тогда $a < b \Leftrightarrow (a \leq b \wedge a \neq b)$, а $\leq$ определено как наименьшая неподвижная точка следующих утверждений:

$$a \leq a$$
$$a \leq b \Rightarrow a \leq C_j(\ldots, b, \ldots)$$
$$a \sqsubseteq b \Rightarrow a \leq b$$
$$a \leq b, b \leq c \Rightarrow a \leq c$$

Пусть определение функции f имеет вид $f(\overline{x_i}) \equiv E\langle f(\overline{e_{1i}}), \ldots, f(\overline{e_{ni}})\rangle$, и в данной полипрограмме функция f не имеет других рекурсивных вызовов (в том числе косвенных). Пусть $(\!|E\langle\overline{f(\overline{e_{li}})}\rangle|\!)_{\mu\sigma} = \mathcal{E}\langle\overline{\mu(f)((\!|e_{li}|\!)_{\mu\sigma})}\rangle$ в соответствии с Рис. 2.2. Тогда:

- Если существует j такое, что для любой интерпретации μ и любого контекста σ, назначающего переменным конечные (но, возможно, частичные) значения, верно, что для любого l (т.е. для любого места рекурсивного вызова) $(\!|e_{lj}|\!)_{\mu\sigma} < \sigma(x_j)$, то любые две модели данной полипрограммы, совпадающие на всех функциях, кроме f, совпадают и на f (т.е. $\mu_1(f) = \mu_2(f)$).

- Если для любой интерпретации μ и любого контекста σ верно что значение $(\!|E\langle\overline{f(\overline{e_{li}})}\rangle|\!)_{\mu\sigma}$ имеет вид $C_{\mu\sigma}(\ldots)$, где $C_{\mu\sigma}$ — некоторый конструктор или $\perp$, то любые две модели данной полипрограммы, совпадающие на всех функциях, кроме f, совпадают и на f (т.е. $\mu_1(f) = \mu_2(f)$).

Первый случай соответствует структурной рекурсии, а второй защищён-

ной рекурсии, однако мы не будем формально вводить эти термины в настоящей диссертации, и будем использовать их исключительно неформально.

Рассмотрим теперь примеры применения слияния по бисимуляции. Начнём с бесконечных чисел:

$$f \equiv S(f)$$
$$g \equiv S(g)$$

В качестве полипрограммы бисимуляции возьмём $f \equiv S(f)$, а отображениями функций будут $\{f \mapsto f\}$ и $\{f \mapsto g\}$. Функция f в данной полипрограмме является защищённо-рекурсивной, потому что семантика правой части $(\!|S(f)|\!)_{\mu\sigma}$ имеет вид $S(\mu(f))$. Поэтому можно добавить определение $f \equiv g$ в полипрограмму:

$$f \equiv S(f)$$
$$g \equiv S(g)$$
$$f \equiv g$$

Теперь завершим пример ассоциативности сложения:

$$(8)\ f(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}$$
$$(10)\ g(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(g(x',y,z))\}$$

В этом случае в качестве полипрограммы бисимуляции можно взять просто первое определение:

$$f(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}$$

При этом $\phi_1(f) = f$, $\phi_2(f) = g$, $\phi_i(\mathrm{add}) = \mathrm{add}$. Рассмотрим семантику правой части:

$$(\!|\textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}|\!)_{\mu\sigma} =$$
$$= \begin{cases} \mu(\mathrm{add})(\sigma(y),\sigma(z)) & \text{если } \sigma(x) = Z \\ \mu(f)(a,\sigma(y),\sigma(z)) & \text{если } \sigma(x) = S(a) \\ \bot & \text{иначе} \end{cases}$$

Т.к. $a < \sigma(x)$, если $\sigma(x) = S(a)$ и $S(a)$ конечное, то f структурно рекурсивная. Поэтому для каждой интерпретации add существует не более одной интерпретации f такой, что вместе они образуют модель. Поэтому можно добавить определение

$$(11)\ f(x, y, z) \equiv g(x, y, z)$$

Ещё несколько примеров применения слияния по бисимуляции можно найти в Приложении А, в частности, в Примерах 12 и 13 слияния по бисимуляции применяется не только в самом конце для завершения доказательства, но и в процессе доказательства, что соответствует обнаружению и применению леммы.

2.6 Обобщения

В системе Бёрстолла-Дарлингтона существует правило, называемое абстракцией, в суперкомпиляции аналогичное преобразование называется обобщением. Его суть в том, что составные выражения вида $E\langle\overline{e_i}\rangle$ можно привести к виду **let** $\{\overline{x_i = e_i}\}$ **in** $E\langle\overline{x_i}\rangle$. В нашем случае в языке нет let-выражений, но можно ввести вспомогательную функцию — тогда соответствующее правило выглядело бы как-то так:

$$\left\{ f(\overline{x_i}) \equiv E\langle\overline{e_j}\rangle \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle\overline{e_j}\rangle \\ f(\overline{x_i}) \equiv g(\overline{x_i}, \overline{e_j}) \\ g(\overline{x_i}, \overline{y_j}) \equiv E\langle\overline{y_j}\rangle \end{array} \right\}$$

Проблема с этим правилом в том, что при использовании подхода насыщения равенствами к порядку применения правил (применение правил в ширину) данное правило приводит к комбинаторному взрыву. Поэтому мы обходимся только обобщениями некоторого ограниченного вида, которые реализуются уже введёнными выше правилами.

Во-первых, некоторые выражения уже могут быть записаны в виде $g(\overline{e_j})$ — в таком случае мы может перейти к преобразованию функции g. С точки зрения обобщений в суперкомпиляции такой переход является обобщением, хотя с точки зрения системы Бёрстолла-Дарлингтона он тривиален и даже не требует применения абстракции. Во-вторых, наши правила хоть и крайне

похожи на прогонку из суперкомпиляции, являются немного более выразительными, и позволяют в некоторых случаях добиться эффекта обобщения, как, например, показано в Примере 14 из Приложения А.

2.7 Выводы

В данной главе описан синтаксис входного языка для нашей системы преобразования, и введено понятие полипрограммы. Полипрограммы отличаются от программ главным образом тем, что каждая функция может иметь произвольное число определений (ноль, одно или несколько).

Для полипрограмм была введена денотационная семантика, и некоторые основные понятия, такие как понятие семантической эквивалентности полипрограмм. Также формально были сформулированы задачи, решаемые методом, представленным в диссертации.

Сам метод был описан в общих чертах, без некоторых технических подробностей, которые будут раскрыты в последующих главах. Были введены правила преобразования полипрограмм, используемые в нашем методе, а также введено преобразование, называемое слиянием по бисимуляции, способное выполнять доказательство эквивалентности по индукции. Хотя введённых таким образом понятий и преобразований достаточно для ручного преобразования полипрограмм, для программной реализации они не подходят, потому что наивное применение правил в таком виде будет приводить к очень сильному разрастанию полипрограммы. Этот недостаток и будет устранён в последующих главах.

Глава 3

Расчленённая форма и бесточечное представление полипрограмм

В Главе 2 был описан функциональный нетипизированный язык первого порядка, введено понятие полипрограммы, и введены преобразования полипрограмм. Введённое таким образом понятие полипрограммы и преобразования хорошо подходят для ручного применения нашего метода, однако при реализации возникают некоторые трудности.

Во-первых, не приведено никакого указания, в каком порядке применять правила. Если применять правила в глубину, то метод получится больше похожим на суперкомпиляцию — но при этом нужны также некоторые эвристики, обрезающие бесконечные ветви. Если применять правила в ширину, то метод получится более похожим на насыщение равенствами, в этом случае эвристики не нужны (или, по крайней мере, не обязательны). В данной работе мы применяем правила в ширину, поэтому считаем наш метод вариантом насыщения равенствами.

Однако при применении правил в ширину возникает следующая проблема: слишком быстрое разрастание полипрограммы. Для борьбы с этой проблемой мы используем, во-первых, стандартный приём слияния общих подвыражений (используемый, в частности, в насыщении равенствами), а во-

вторых, применяем некоторые правила деструктивно (этот приём в насыщении равенствами ранее не использовался).

3.1 Расчленённая форма

Рассмотрим следующую полипрограмму:

$$f(\overline{x_i}) \equiv E_1\langle H\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle H\rangle$$

Подвыражение H (которое может быть довольно большим) входит в два определения как подвыражение, поэтому занимает в два раза больше памяти, чем могло бы, и, что хуже, в процессе преобразования вероятно будет преобразовано одним и тем же образом два раза. Поэтому желательно обобществить это подвыражение между определениями, что нетрудно сделать, введя вспомогательную функцию при помощи правила расчленения (точнее, правило расчленения введёт две разные функции, после чего необходимо применить транзитивность, конгруэнтность и удаление ненужных определений, чтобы оставить только одну функцию):

$$f(\overline{x_i}) \equiv E_1\langle h(\overline{fv}(H))\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle h(\overline{fv}(H))\rangle$$
$$h(\overline{fv}(H)) \equiv H$$

Отсюда возникает следующая идея: выносить таким образом не только повторяющиеся подвыражения, но и вообще все подвыражения, кроме подвыражений вида $f(\overline{x_i})$, где x_i — различные переменные (такие подвыражения будем называть элементарными вызовами функции). Будем говорить, что если в полипрограмме вынесены все такие подвыражения, то она находится в расчленённой форме (или просто является расчленённой полипрограммой).

Определение 11. *Полипрограммой в расчленённой форме* назовём полипрограмму, в которой каждое определение имеет вид $f(\overline{x_i}) \equiv s$, где s — простейшее выражение как на Рис. 3.1.

Правая часть любого определения полипрограммы в расчленённой форме состоит из элементарного вызова (обозначено r) или простейшего выражения

Рис. 3.1 Простейшие выражения

Элементарный вызов функции:

$r ::= f(\overline{x_i})$, где x_i — *различные* переменные

Простейшее выражение:

$s ::= r$
$\quad | \; x$
$\quad | \; f(\overline{r_i})$
$\quad | \; C(\overline{r_i})$
$\quad | \; \mathbf{case}\ r_0\ \mathbf{of}\ \{\ \overline{C_i(\overline{x_{ij}})\ \rightarrow\ r_i;}\ \}$

(обозначено s), т.е. выражения, максимальные собственные подвыражения которого являются элементарными вызовами. Например, $f(x, y)$ — элементарный вызов, а $f(x, x)$ — нет, т.к. дублируется переменная, $f(x, C())$ — тоже нет, т.к. второй аргумент — не переменная. Выражение

$$\mathbf{case}\ x\ \mathbf{of}\ \{\ C(y)\ \rightarrow\ f(x, y)\ \}$$

не является простейшим выражением, т.к. его подвыражение x в разбираемой позиции не является элементарным вызовом, а выражение

$$\mathbf{case}\ id(x)\ \mathbf{of}\ \{\ C(y)\ \rightarrow\ f(x, y)\ \}$$

уже является простейшим.

Расчленённая форма не даёт автоматического устранения дублирования — как уже было отмечено, для этого нужно применить транзитивность, конгруэнтность и удаление определений-дубликатов.

Любую полипрограмму можно привести к расчленённой форме за конечное число шагов (примем это без доказательства).

Пример 6. Рассмотрим простой пример преобразования полипрограммы в расчленённую форму. Пусть дана следующая полипрограмма:

$\mathrm{add}(x, y) \equiv$
$\quad \mathbf{case}\ x\ \mathbf{of}$

$$Z \to y$$
$$S(x) \to S(\mathrm{add}(x, y))$$

$$\mathrm{mul}(x, y) \equiv$$
$$\quad \textbf{case } x \textbf{ of}$$
$$\quad\quad Z \to Z$$
$$\quad\quad S(z) \to \mathrm{add}(y, \mathrm{mul}(z, y))$$

Вынесем просто все подвыражения, не являющиеся элементарными вызовами, в отдельные функции:

$$\mathrm{add}(x, y) \equiv \textbf{case } g_1(x) \textbf{ of } \{Z \to g_2(y);\ S(x) \to g_3(x, y)\}$$
$$g_1(x) \equiv x$$
$$g_2(y) \equiv y$$
$$g_3(x, y) \equiv S(\mathrm{add}(x, y))$$

$$\mathrm{mul}(x, y) \equiv \textbf{case } g_4(x) \textbf{ of } \{Z \to g_5();\ S(z) \to g_6(y, z)\}$$
$$g_4(x) \equiv x$$
$$g_5() \equiv Z$$
$$g_6(y, z) \equiv \mathrm{add}(g_7(y), g_8(z, y))$$
$$g_7(y) \equiv y$$
$$g_8(z, y) \equiv \mathrm{mul}(z, y)$$

После такого преобразования в полипрограмме может получиться довольно много дублирования, однако оно всё будет устранено после применения упрощающих преобразований. △

При применении правил преобразования у расчленённой формы есть одна проблема: если левая часть правила находится не в расчленённой форме, как например, у правила редукции сопоставления с образцом

$$\left\{ f(\overline{x_i}) \equiv \textbf{case } C(\overline{e_j}) \textbf{ of } \{\ldots; C(\overline{y_j}) \to E; \ldots\} \right\} \mapsto \ldots$$

то перед применением правила необходимо будет применить правило раскрытия вызова функции. Это разрушит расчленённость программы, что нежелательно, однако после применения правила такое раскрытое определение можно будет удалить. Также если правая часть правила не находится в расчленённой форме, то после его применения расчленённость, опять же, будет нарушена. В этом случае после применения правила необходимо применить расчленение.

Однако есть более изящное решение — переписать все правила так, чтобы обе части находились в расчленённой форме. Для правила редукции сопоставления с образцом, например, левая часть будет выглядеть так (обратите внимание, что неизвестные подвыражения заменены на элементарные вызовы неизвестных функций):

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ g(\overline{z_l})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to h(\overline{y_j}, \overline{x_i}); \ldots\} \\ g(\overline{z_l}) \equiv C(\overline{e_j(\overline{z_{jk}})}) \end{array}\right\} \mapsto \ldots$$

При этом расчленённая форма будет поддерживаться автоматически, и правило расчленения будет не нужно (кроме преобразования из программы в расчленённую полипрограмму в самом начале). Все правила из Главы 2 будут переписаны в таком виде в Главе 4.

Самое интересное изменение после такого переписывания произойдёт с правилом раскрытия вызова функции

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{y_j})\rangle \\ g(\overline{y_j}) \equiv H \\ f(\overline{x_i}) \equiv E\langle H\{\overline{y_j \mapsto e_j}\}\rangle \end{array}\right\}$$

Во-первых, от выражения E здесь можно вообще избавиться, потому что подвыражение $g(\overline{e_j})$ будет из него вынесено отдельно в любом случае (это неэлементарный вызов). Во-вторых, в правой части указана подстановка в выражение H вида $H\{\overline{y_j \mapsto e_j}\}$, однако H имеет простейший вид, а значит можно рассмотреть все конструкции, которые могут составлять H, и для каждой из них ввести своё правило. При этом окажется, что не нужно реализовывать отдельную операцию подстановки в выражение, потому что неэлементарные вызовы будут играть роль явных подстановок, а правила раскрытия вызова функции станут правилами проталкивания этих явных подстановок вниз.

Например, так будет выглядеть правило раскрытия вызова для случая, когда H имеет вид вызова функции:

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k}(\overline{y_j})) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k}(\overline{y_j})) \\ f(\overline{x_i}) \equiv g(\overline{q_k(\overline{x_i})}) \\ \hline q_k(\overline{x_i}) \equiv d_k(\overline{e_j(\overline{x_i})}) \end{array}\right\}$$

Можно заметить одну неприятную вещь: записи правил становятся довольно громоздкими из-за упоминания переменных. Более того, в предыдущей главе было сказано, что правила применяются с точностью до переименования переменных, и что это должно быть интуитивно понятно, однако для реализации этот момент требует прояснения. Поэтому в следующем разделе мы введём нотацию, позволяющую избавиться от упоминания переменных в большинстве случаев.

3.2 Бесточечное представление полипрограмм

Представление расчленённой полипрограммы при помощи синтаксиса с Рисунка 3.1 не очень удобно для реализации, потому что содержит связанные переменные. Поэтому мы введём другое представление расчленённых полипрограмм, очень похожее на исходное, но более удобное для формальной работы и для программной реализации. В этом представлении не будет явного упоминания параметров функций, поэтому будем называть его бесточечным (point-free) представлением полипрограмм в расчленённой форме, или просто бесточечно-расчленённым представлением полипрограмм. Отметим, что бесточечное представление можно ввести и для нерасчленённых полипрограмм, но мы не будем этого делать.

3.2.1 Перестановки параметров

Вместо именованных переменных мы будем работать напрямую с номерами параметров функций, используя для этого перестановки параметров.

Определение 12. *Перестановкой параметров* назовём инъективное отображение $\{1, \ldots, n\} \hookrightarrow \{1, \ldots, m\}$ (для простоты будем писать $n \hookrightarrow m$), где

$m, n \in \mathbb{N}$. Перестановки параметров будем обозначать буквами $\theta, \zeta, \xi, \ldots$. Множество всех перестановок параметров обозначим Θ. Будем также использовать обозначение $dom(\theta) = n$ и $cod(\theta) = m$, если $\theta : n \hookrightarrow m$.

Перестановка параметров $n \hookrightarrow m$ позволяет преобразовать функцию арности n в функцию арности m, попутно переставив некоторые параметры местами (при этом обязательно должно выполняться $m \geq n$, так как перестановки параметров не отождествляют переменные). Происходит это следующим образом. Пусть f — некоторая функция, $arity(f) = n$ и дана перестановка параметров $\theta : n \hookrightarrow m$. Тогда действие перестановки параметров на функцию описывается следующей формулой:

$$(\theta f)(a_1, \ldots a_m) = f(a_{\theta(1)}, \ldots, a_{\theta(n)})$$

Или, с использование черты, $(\theta f)(\overline{a_i}) = f(\overline{a_{\theta(j)}})$. При этом $arity(\theta f) = m$.

Для описания перестановок будем использовать как запись $\{\overline{n_i \to k_i}\}$, так и конструкцию Perm(новые переменные | старые переменные):

$$\mathrm{Perm}(x_1, \ldots, x_n \,|\, y_1, \ldots, y_m) : m \hookrightarrow n$$
$$\mathrm{Perm}(x_1, \ldots, x_n \,|\, y_1, \ldots, y_m)(i) = \min\{j \mid y_i \text{ совпадает с } x_j \}$$

Эта конструкция будет использована при формализации преобразования в бесточечное представление. Минимум нужно брать для работы с повторяющимися переменными, которые в свою очередь полезно допускать при работе с конструкцией сопоставления с образцом, которая может связывать переменные, уже связанные снаружи. Тип получившейся перестановки определяется по полному количеству переменных слева и справа. Например,

$$\mathrm{Perm}(x, y, z, y \,|\, z, y) = \{1 \to 3, 2 \to 2\} : 2 \hookrightarrow 4$$
$$(\mathrm{Perm}(x, y, z, y \,|\, z, y)\, f)(a, b, c, d) = f(c, b)$$

Перестановки параметров часто удобно представлять в виде композиции $\xi = \theta \circ id_{m \to n}$, где θ — биекция $n \leftrightarrow n$, а $id_{m \to n}$ просто расширяет арность, но не переставляет параметры, т.е. $id_{m \to n}(i) = i$. Биекция θ при этом определена

неоднозначно, поэтому канонически будем определять её как $\theta = \mathit{lift}(\xi)$, где

$$\mathit{lift}(\xi)(i) = \begin{cases} \xi(i), & \text{если } i \leq m \\ (i-m)\text{-й элемент } \mathbf{n} \text{ без прообраза при } \xi, & \text{иначе} \end{cases}$$

Мы часто будем опускать тождественные перестановки при записи термов, формул и полипрограмм. Также будем писать просто id_m вместо $id_{m \to m}$.

3.2.2 Полипрограмма в бесточечно-расчленённом представлении

Несмотря на то, что наш язык нетипизированный на уровне значений первого порядка, определяемые пользователем функции могут иметь разную арность, которая играет роль типов. Поэтому каждой функции мы присвоим некоторую арность, кроме того, нам понадобятся простейшие правила вывода арности. Также будем для $\mathit{arity}(f) = n$ использовать альтернативное обозначение $f : n$.

Определение 13. Множеством *функциональных символов* назовём произвольное множество F, на котором определена функция $\mathit{arity} : F \to \mathbb{N}$.

Теперь формально введём понятие полипрограммы в бесточечно-расчленённом представлении. Фактически расчленённые полипрограммы в том виде, в котором они были введены ранее, и полипрограммы в бесточечно-расчленённом представлении, очень близки, и легко преобразуются друг в друга, но в оставшейся части диссертации мы будем формально работать только с полипрограммами в бесточечно-расчленённом представлении (и называть их просто полипрограммами), хотя для записи полипрограмм будем применять и ту и другую форму.

Определение 14. *Полипрограммой в бесточечно-расчленённом представлении* будем называть тройку $\langle H, F, v \rangle$, где H — просто некоторое множество (индексирующее определения), F — множество функциональных символов, а $v : H \to P$, где P — множество формул вида $\theta_0 f_0 \equiv \xi\, L(\overline{\theta_i f_i})$ как на Рис. 3.2, где $f_i \in F$, $\theta_i \in \Theta$, $\xi \in \Theta$, причём обе части формулы должны быть согласованы по арности, т.е. должно существовать такое n, что $\theta_0 f_0 : n$ и $\xi\, L(\overline{\theta_i f_i}) : n$, где правила вывода арности определены на Рис. 3.3.

Рис. 3.2 Виды формул полипрограммы

$$
\begin{array}{rll}
P ::= & \theta_0 f_0 \equiv \xi\, \mathbf{Id}(\theta_1 f_1) & \text{перестановка параметров} \\
\mid & \theta_0 f_0 \equiv \xi\, \mathbf{Var} & \text{переменная} \\
\mid & \theta_0 f_0 \equiv \xi\, \mathbf{Call}(\theta_1 f_1, \ldots, \theta_n f_n) & \text{вызов функции} \\
\mid & \theta_0 f_0 \equiv \xi\, \mathbf{Cons}_C(\theta_1 f_1, \ldots, \theta_n f_n) & \text{конструктор} \\
\mid & \theta_0 f_0 \equiv \xi\, \mathbf{Case}_{\overline{C_i}}(\theta_1 f_1, \ldots, \theta_n f_n) & \text{сопоставление с образцом, } n \geq 1
\end{array}
$$

Рис. 3.3 Правила вывода арности конструкций бесточечного представления

$$\frac{\mathit{arity}(f) = n}{f : n} \qquad \frac{t : n \quad \theta : n \hookrightarrow m}{\theta t : m}$$

$$\frac{t : n}{\mathbf{Id}(t) : n} \qquad \mathbf{Var} : 1$$

$$\frac{t_0 : n \quad n \geq 1 \quad t_1, \ldots, t_n : m}{\mathbf{Call}(t_0, t_1, \ldots, t_n) : m} \qquad \frac{t_0 : 0}{\mathbf{Call}(t_0) : 0}$$

$$\frac{\mathit{arity}(C) = n \quad n \geq 1 \quad t_1, \ldots, t_n : m}{\mathbf{Cons}_C(t_1, \ldots, t_n) : m} \qquad \frac{\mathit{arity}(C) = 0}{\mathbf{Cons}_C() : 0}$$

$$\frac{\mathit{arity}(C_i) = k_i \quad t_0 : n \quad \overline{t_i : k_i + n}}{\mathbf{Case}_{\overline{C_i}}(t_0, \overline{t_i}) : n}$$

Для удобства введём функции доступа к отдельным компонентам полипрограммы:

$$\mathit{defs}(\langle H, F, v\rangle) = H$$
$$\mathit{funs}(\langle H, F, v\rangle) = F$$
$$\mathit{labeling}(\langle H, F, v\rangle) = v$$

Для простоты можно считать, что полипрограмма — мультимножество формул из P. Множество H, которое мы будем называть множеством определений, нужно для индексирования формул определений, чтобы можно было различать даже одинаковые определения, что понадобится для доказательства некоторых теорем. Далее будем называть определениями как сами эле-

менты $h \in H$, так и соответствующие им формулы определений $v(h)$, если из контекста понятно, о каком именно определении h идёт речь.

Данное определение полипрограммы основано на определении понятия гиперграфа. Фактически элементы F можно считать вершинами, а элементы H — гиперрёбрами (отсюда и использование буквы H). Функция v одновременно задаёт как метку гиперребра, так и вершины, инцидентные данному гиперребру, например, если формула $v(h)$ имеет вид

$$\theta f \equiv \xi\, \mathbf{Cons}_C(\phi_1 g_1, \phi_2 g_2),$$

то гиперребро инцидентно вершинам f, g_1 и g_2 и имеет метку

$$\theta \_ \equiv \xi\, \mathbf{Cons}_C(\phi_1 \_, \phi_2 \_)$$

Далее, впрочем, меткой определения будем называть только часть $\mathbf{Cons}_C$.

Полипрограмму можно также рассматривать и как формулу логики предикатов первого порядка в сигнатуре $\{\mathbf{Id}, \mathbf{Var}, \mathbf{Call}, \mathbf{Cons}_C, \mathbf{Case}_s, \equiv\} \cup \Theta \cup F$ (как конъюнкцию формул определений), однако дальнейшее изложение не зависит от теории логики первого порядка.

Формулы определений полипрограммы в бесточечно-расчленённом представлении являются конструкциями одного из пяти видов, соответствующих основным конструкциям языка (неформально):

- $\mathbf{Id}(t)$ соответствует просто t и является вспомогательной.
- **Var** соответствует переменной, причём всегда переменной под номером 1. Остальные переменные получаются применением перестановки параметров (например, $\{1 \to 2\}\,\mathbf{Var}$).
- $\mathbf{Call}(t_0, t_1, \ldots, t_n)$ соответствует вызову функции $t_0(t_1, \ldots, t_n)$.
- $\mathbf{Cons}_C(t_1, \ldots, t_n)$ соответствует конструированию значения $C(t_1, \ldots, t_n)$.
- $\mathbf{Case}_{\overline{C_i}}(t_0, t_1, \ldots, t_n)$ соответствует конструкции сопоставления с образцом **case** t_0 **of** $\{\overline{C_i(\overline{x_{ij}}) \ \to \ t_i}\}$. При этом в ветви t_i переменные образца передаются первыми, а затем идут переменные, полученные из внешней области видимости.

Отметим, что формулы определений в представленном виде содержат избыточное количество перестановок параметров. Это сделано для удобства, в

частности, перестановки ξ нужны для того, чтобы согласовать арность левых и правых частей равенств в определениях вида $f(x) \equiv C()$ — такое определение будет представлено в виде $id_1 f \equiv id_{0\to 1}\mathbf{Cons}_C()$. От ξ на практике можно избавиться, приведя определения к каноническому виду (см. Раздел 3.3) и применив правило понижения арности (arity-reduction).

Рис. 3.4 Преобразование расчленённой полипрограммы в полипрограмму в бесточечном представлении

$$\mathrm{ToIRep}[\![\{d_1; d_2; \ldots d_n\}]\!] = \bigcup_i \{\mathrm{DefToFormula}[\![d_i]\!]\}$$

$$\mathrm{DefToFormula}[\![f(x_1, \ldots, x_n) \equiv s]\!] =$$
$$\quad id_{n\to|fv(s)|}\, f \equiv \mathrm{ExprToTerm}[\![s]\!]_{\overline{x_i},\overline{y_j}} \qquad y_j \in fv(s), \forall i\ y_j \neq x_i$$

$$\mathrm{ExprToTerm}[\![h(\overline{x_j})]\!]_{\overline{x_i}} =$$
$$\quad \mathrm{Perm}(\overline{x_i}|\overline{x_j})\, h$$

$$\mathrm{ExprToTerm}[\![x_k]\!]_{\overline{x_i}} =$$
$$\quad \mathrm{Perm}(\overline{x_i}|x_k)\, \mathbf{Var}$$

$$\mathrm{ExprToTerm}[\![h(\overline{g_j(\overline{x_{jk}})})]\!]_{\overline{x_i}} =$$
$$\quad id_{\ldots\to\max i}\, \mathbf{Call}(h, \overline{\mathrm{Perm}(\overline{x_i}|\overline{x_{jk}})g_j})$$

$$\mathrm{ExprToTerm}[\![C(\overline{g_j(\overline{x_{jk}})})]\!]_{\overline{x_i}} =$$
$$\quad id_{\ldots\to\max i}\, \mathbf{Cons}_C(\overline{\mathrm{Perm}(\overline{x_i}|\overline{x_{jk}})g_j})$$

$$\mathrm{ExprToTerm}[\![\mathbf{case}\ h(\overline{x_{0k}})\ \mathbf{of}\ \{\ \overline{C_j(\overline{y_{jl}})\ \to\ g_j(\overline{x_{jk}})}\ \}]\!]_{\overline{x_i}} =$$
$$\quad id_{\ldots\to\max i}\, \mathbf{Case}_{\overline{C_j}}(\mathrm{Perm}(\overline{x_i}|\overline{x_{0k}})h, \overline{\mathrm{Perm}(\overline{y_{jl}}, \overline{x_i}|\overline{x_{jk}})g_j})$$

Теперь приведём функцию для преобразования расчленённых полипрограмм в бесточечное представление (Рис. 3.4). Данная процедура далее нигде больше не используется, так как далее мы будем работать только с бесточечно-расчленённым представлением, но приведена здесь для полноты описания.

Пример 7. Если преобразовать следующую полипрограмму в бесточечно-расчленённое представление

$$\mathrm{add}(x, y) \equiv \mathbf{case}\ g_1(x)\ \mathbf{of}\ \{Z \to g_2(y);\ S(x) \to g_3(x, y)\}$$
$$g_1(x) \equiv x$$
$$g_2(y) \equiv y$$

$$g_3(x, y) \equiv S(\mathrm{add}(x, y))$$

$$\mathrm{mul}(x, y) \equiv \mathbf{case}\ g_1(x)\ \mathbf{of}\ \{Z \to g_5();\ S(z) \to g_6(y, z)\}$$
$$g_5() \equiv Z$$
$$g_6(y, z) \equiv \mathrm{add}(g_2(y), g_8(z, y))$$
$$g_8(z, y) \equiv \mathrm{mul}(z, y)$$

то получится следующая полипрограмма:

$$id_2\,\mathrm{add} \equiv id_2\,\mathbf{Case}_{Z,S}(\{1 \to 1\}\,g_1, \{1 \to 2\}\,g_2, \{1 \to 1, 2 \to 3\}\,g_3)$$
$$id_1\,g_1 \equiv id_1\,\mathbf{Var}$$
$$id_1\,g_2 \equiv id_1\,\mathbf{Var}$$
$$id_2\,g_3 \equiv id_2\,\mathbf{Cons}_S(id_2\,\mathrm{add})$$
$$id_2\,\mathrm{mul} \equiv id_2\,\mathbf{Case}_{Z,S}(\{1 \to 1\}\,g_4, id_{0\to 2}\,g_5, \{1 \to 3, 2 \to 1\}\,g_6)$$
$$id_0\,g_5 \equiv id_0\,\mathbf{Cons}_Z()$$
$$id_2\,g_6 \equiv id_2\,\mathbf{Call}(id_2\,\mathrm{add}, \{1 \to 1\}\,g_7, \{1 \to 2, 2 \to 1\}\,g_8)$$
$$id_2\,g_8 \equiv id_2\,\mathbf{Id}(id_2\,\mathrm{mul})$$

△

3.2.3 Семантика бесточечно-расчленённого представления

Семантику бесточечно-расчленённого представления полипрограмм можно ввести через семантику обычных полипрограмм, однако для удобства мы перепишем основные определения из Раздела 2.3 в адаптированном для бесточечного представления виде.

Напомним, что множество данных первого порядка $\mathbb{A}$ можно рассматривать как наибольшую неподвижную точку следующего определения:

$$\mathbb{A} = \{C(\overline{a_i}) \mid a_i \in \mathbb{A},\ C \text{ — конструктор}\} \cup \{\bot\}$$

А функциональные символы полипрограммы будут интерпретироваться функциями из $\mathbb{D}$ — множества всех непрерывных функций над $\mathbb{A}$ конечной арно-

Рис. 3.5 Семантика конструкций бесточечного представления

$$(\!|f|\!)_\mu = \mu(f)$$
$$(\!|\theta t|\!)_\mu(\overline{x_i}) = (\!|t|\!)_\mu(\overline{x_{\theta(j)}})$$
$$(\!|\mathbf{Call}(t_0, \overline{t_j})|\!)_\mu(\overline{x_i}) = (\!|t_0|\!)_\mu(\overline{(\!|t_j|\!)_\mu(\overline{x_i})})$$
$$(\!|\mathbf{Var}|\!)_\mu(x) = x$$
$$(\!|\mathbf{Cons}_C(\overline{t_j})|\!)_\mu(\overline{x_i}) = C(\overline{(\!|t_j|\!)_\mu(\overline{x_i})})$$

$$(\!|\mathbf{Case}_{\overline{C_j}}(t_0, \overline{t_j})|\!)_\mu(\overline{x_i}) =$$
$$= \begin{cases} (\!|t_j|\!)_\mu(\overline{a_l}, \overline{x_i}) & \text{если } (\!|t_0|\!)_\mu(\overline{x_i}) = C_j(\overline{a_l}) \\ \bot & \text{если } (\!|t_0|\!)_\mu(\overline{x_i}) = C_l(\ldots) \neq C_j(\ldots) \forall j \\ \bot & \text{если } (\!|t_0|\!)_\mu(\overline{x_i}) = \bot \end{cases}$$

$$[\![t_1 \equiv t_2]\!]_\mu \Leftrightarrow (\!|t_1|\!)_\mu = (\!|t_2|\!)_\mu$$

сти.

Определение 15. *Интерпретацией* полипрограммы G с $\mathit{funs}(G) = F$ будем называть функцию $\eta : F \to \mathbb{D}$ такую, что $\mathit{arity}(\eta(f)) = \mathit{arity}(f)$.

Моделью полипрограммы G будем называть такую интерпретацию μ, что $\mu \models G$, где

$$\mu \models G \stackrel{\text{def}}{=} \bigwedge_{d \in \mathit{defs}(G)} [\![\mathit{labeling}(G)(d)]\!]_\mu$$

$[\![p]\!]_\mu$ определено на Рис. 3.5.

Отметим, что $(\!|t|\!)_\mu$ для любого терма $t : n$ (для которого выводится арность по Рис. 3.3) является функцией с арностью n.

Определение 16. Две полипрограммы G_1 и G_2 будем называть семантически эквивалентными на подмножестве функциональных символов F таком, что $F \subseteq \mathit{funs}(G_1) \cap \mathit{funs}(G_2)$, если для любой модели $\mu_1 \models G_1$ существует модель $\mu_2 \models G_2$ такая, что $\forall f \in F \,.\, \mu_1(f) = \mu_2(f)$, и наоборот, для любой модели $\eta_2 \models G_2$ существует модель $\eta_1 \models G_1$ такая, что $\forall f \in F \,.\, \eta_1(f) = \eta_2(f)$.

3.3 Каноническая форма определений

Переменные несут в себе одну проблему — неоднозначность представления одного и того же определения. Например, следующие два определения эквивалентны с точки зрения семантики полипрограмм:

$$f(x, y) \equiv C(h_1(x), h_2(y))$$
$$f(y, x) \equiv C(h_1(y), h_2(x))$$

Однако из-за различных имён переменных они формально различаются. Перестановки параметров никак эту проблему не решают:

$$\{1 \to 1, 2 \to 2\}f \equiv \mathbf{Cons}_C(\{1 \to 1\}h_1, \{1 \to 2\}h_2)$$
$$\{1 \to 2, 2 \to 1\}f \equiv \mathbf{Cons}_C(\{1 \to 2\}h_1, \{1 \to 1\}h_2)$$

Однако можно ввести некоторое каноническое именование переменных, например, по порядку вхождения, слева направо — тогда оба определения примут следующий вид:

$$f(x_1, x_2) \equiv C(h_1(x_1), h_2(x_2))$$

Этот приём также позволяет избавиться от перестановки при левой части (она всегда будет равна id). Но оно не очень удачное, потому что, например, два следующих определения переименованы именно таким образом:

$$f(x_1, x_2) \equiv C(h_1(x_1), h_2(x_2))$$
$$h(x_1, x_2) \equiv C(h_1(x_2), h_2(x_1))$$

Но если мы переименуем второе определение по-другому, а именно так:

$$h(x_2, x_1) \equiv C(h_1(x_1), h_2(x_2))$$

То отсюда сразу будет видно, что $f(x_1, x_2) \equiv h(x_2, x_1)$ по транзитивности. А транзитивность — очень важное преобразование, поэтому желательно переименовывать переменные так, чтобы применения транзитивности были сразу очевидны. К счастью, это несложно сделать — достаточно именовать переменные в соответствии с порядков их вхождения в *правую часть*, а не во всё

определение. Такое именование мы и будем называть канонической формой определения.

Теперь формализуем данную операцию для определений в бесточечно-расчленённом представлении. Напомним, что формула определения имеет вид $t_1 \equiv t_2$, где t_1 и t_2 — термы вида θf и $\xi\,\mathbf{L}(\overline{\theta_i f_i})$ соответственно (Рис. 3.2), причём согласованные по арности (Рис. 3.3). Будем работать на уровне термов, составляющих формулы определений. Определим операцию применения перестановки параметров $\theta : m \hookrightarrow n$ к терму вида $\xi t : m$ очевидным образом:

$$\theta(\xi t) = (\theta \circ \xi)t$$

При этом $\theta(\xi t) : n$. Теперь определим операцию $push(t)$ как на Рис. 3.6. Эта операция проталкивает внешнюю перестановку при правой части формулы определения внутрь, оставляя снаружи тождественную перестановку или тривиальное вложение $id_{0\to m}$. Пример работы данной операции:

$$\begin{aligned} &push(\{1 \to 2, 2 \to 1\}\,\mathbf{Call}(id_2 f, id_2 g, \{1 \to 2\}h)) = \\ &\quad = id_2\,\mathbf{Call}(id_2 f, \{1 \to 2, 2 \to 1\}id_2 g, \{1 \to 2, 2 \to 1\}\{1 \to 2\}h) = \\ &\quad = id_2\,\mathbf{Call}(id_2 f, \{1 \to 2, 2 \to 1\}g, id_{1\to 2}h) \end{aligned}$$

На том же рисунке определена вспомогательная операция $shift(\theta, k)$, сдвигающая все переменные в θ на k вправо. Она нужна для работы с конструкцией **Case**.

Нетрудно проверить, что операция $push$ сохраняет арность терма. Кроме того, она согласована с нашей семантикой, т.е. выполняется следующее утверждение.

Утверждение 1. Для любого терма t выполняется $(\!|t|\!)_\mu = (\!|push(t)|\!)_\mu$.

Теперь нам надо определить обратную операцию, которая наоборот будет максимально вытаскивать перестановки параметров наружу. Для этого сначала определим операцию взаимной декомпозиции перестановок с общим кодоменом $bisplit(\theta_1, \theta_2)$ как на Рис. 3.7. Легко проверяется основное свойство данной операции — то, что она действительно производит декомпозицию.

Утверждение 2. Пусть $bisplit(\theta_1, \theta_2) = (\delta, \xi_1, \xi_2)$. Тогда $\theta_i = \delta \circ \xi_i$.

Теперь сформулируем и докажем два важных свойства этой операции.

Рис. 3.6 Операция проталкивания перестановки параметров вглубь терма

$$
\begin{aligned}
&push(\theta\,\mathbf{Id}(t)) = id_{cod(\theta)}\,\mathbf{Id}(\theta t)\\
&push(\theta\,\mathbf{Var}) = \theta\,\mathbf{Var}\\
&push(\theta\,\mathbf{Call}(t_0)) = id_{0\to cod(\theta)}\,\mathbf{Call}(t_0)\\
&push(\theta\,\mathbf{Call}(t_0, \overline{t_j})) = id_{cod(\theta)}\,\mathbf{Call}(t_0, \overline{\theta t_j})\\
&push(\theta\,\mathbf{Cons}_C()) = id_{0\to cod(\theta)}\,\mathbf{Cons}_C()\\
&push(\theta\,\mathbf{Cons}_C(\overline{t_j})) = id_{cod(\theta)}\,\mathbf{Cons}_C(\overline{\theta t_j})\\
&push(\theta\,\mathbf{Case}_{\overline{C_j}}(t_0, \overline{t_j})) =\\
&\qquad id\,\mathbf{Case}_{\overline{C_j}}(\theta t_0, \overline{shift(\theta, arity(C_j))t_j})
\end{aligned}
$$

$$
shift(\theta : m \hookrightarrow n, k) : (k+m) \hookrightarrow (k+n)
$$

$$
shift(\theta, k)(i) = \begin{cases} i, \text{ если } i \le k \\ k + \theta(i-k), \text{ иначе} \end{cases}
$$

Во-первых, эта декомпозиция является “наибольшей” — если взять другую декомпозицию $(\delta', \xi_1', \xi_2')$, $\theta_i = \delta' \circ \xi_i'$, то найдётся такая перестановка τ, что $\delta = \delta' \circ \tau$, причём $\tau = bisplit(\xi_1', \xi_2')$. Мы впрочем будем использовать другую, эквивалентную формулировку:

Утверждение 3. Пусть $bisplit(\theta_1, \theta_2) = (\delta, \xi_1, \xi_2)$, тогда для любой перестановки параметров ζ верно $bisplit(\zeta \circ \theta_1, \zeta \circ \theta_2) = (\zeta \circ \delta, \xi_1, \xi_2)$.

Доказательство. Пусть $bisplit(\zeta \circ \theta_1, \zeta \circ \theta_2) = (\delta', \xi_1', \xi_2')$. Тогда:

- $unshared = unshared'$ благодаря тому, что ζ инъективна, а значит $\theta_2(i) = \theta_1(j) \Leftrightarrow (\zeta \circ \theta_2)(i) = (\zeta \circ \theta_1)(j)$.
- Следовательно $m' = m$, $\xi_1' = \xi_1$ и $\xi_2' = \xi_2$.
- Из предыдущих пунктов и определения δ' сразу следует, что $\delta' = \zeta \circ \delta$.

□

Второе свойство (являющееся простым следствием первого) говорит о том, как изменяется декомпозиция, если к исходным перестановкам справа добавить другие перестановки.

Рис. 3.7 Операция взаимной декомпозиции двух перестановок

$$
\begin{aligned}
&\mathit{bisplit}(\theta_1 : m_1 \hookrightarrow n, \theta_2 : m_2 \hookrightarrow n) = (\delta, \xi_1, \xi_2), \text{ где} \\
&\quad \mathit{unshared} = [i \mid i \in [1 \ldots m_2], \nexists j : \theta_2(i) = \theta_1(j)] \\
&\quad m = m_1 + |\mathit{unshared}| \\
&\quad \xi_1 : m_1 \hookrightarrow m \\
&\quad \xi_1(i) = i \\
&\quad \xi_2 : m_2 \hookrightarrow m \\
&\quad \xi_2(i) = \begin{cases} j, & \text{если } \theta_2(i) = \theta_1(j) \\ m_1 + \text{индекс элемента } i \text{ в } \mathit{unshared}, & \text{иначе} \end{cases} \\
&\quad \delta : m \hookrightarrow n \\
&\quad \delta(j) = \begin{cases} \theta_1(j), & \text{если } j \le m_1 \\ \theta_2(\mathit{unshared}(j - m_1)), & \text{иначе} \end{cases}
\end{aligned}
$$

Утверждение 4. Пусть $\mathit{bisplit}(\theta_1, \theta_2) = (\delta, \xi_1, \xi_2)$ и $\mathit{bisplit}(\xi_1 \circ \sigma_1, \xi_2 \circ \sigma_2) = (\delta', \xi_1', \xi_2')$, тогда $\mathit{bisplit}(\theta_1 \circ \sigma_1, \theta_2 \circ \sigma_2) = (\delta \circ \delta', \xi_1', \xi_2')$

Доказательство. Т.к. $\theta_i \circ \sigma_i = \delta \circ \xi_i \circ \sigma_i$, то $\mathit{bisplit}(\theta_1 \circ \sigma_1, \theta_2 \circ \sigma_2) = \mathit{bisplit}(\delta \circ \xi_1 \circ \sigma_1, \delta \circ \xi_2 \circ \sigma_2) = (\delta \circ \delta', \xi_1', \xi_2')$ по предыдущему утверждению. □

Операция *bisplit* легко обобщается до произвольной ненулевой арности:

$$
\begin{aligned}
&\mathit{nsplit}(\theta_1 : m \hookrightarrow n) = (\theta_1, id_{m \to m}) \\
&\mathit{nsplit}(\theta_1, \theta_2) = \mathit{bisplit}(\theta_1, \theta_2) \\
&\mathit{nsplit}(\theta_1, \ldots, \theta_{n-1}, \theta_n) = (\delta', \psi \circ \xi_1, \ldots, \psi \circ \xi_{n-1}, \phi), \text{ где} \\
&\quad (\delta, \xi_1, \ldots, \xi_{n-1}) = \mathit{nsplit}(\theta_1, \ldots, \theta_{n-1}) \\
&\quad (\delta', \psi, \phi) = \mathit{bisplit}(\delta, \theta_n)
\end{aligned}
$$

По индукции легко доказываются следующие утверждения, которые аналогичны утверждениям для *bisplit*.

Утверждение 5. Если $\mathit{nsplit}(\overline{\theta_i}) = (\delta, \overline{\xi_i})$, то $\theta_i = \delta \circ \xi_i$.

Утверждение 6. Если $\mathit{nsplit}(\overline{\theta_i}) = (\delta, \overline{\xi_i})$, то для любой перестановки параметров ζ верно $\mathit{nsplit}(\overline{\zeta \circ \theta_i}) = (\zeta \circ \delta, \overline{\xi_i})$.

Утверждение 7. Если $nsplit(\overline{\theta_i}) = (\delta, \overline{\xi_i})$ и $nsplit(\overline{\xi_i \circ \sigma_i}) = (\delta', \overline{\xi'_i})$, то выполняется $nsplit(\overline{\theta_i \circ \sigma_i}) = (\delta \circ \delta', \overline{\xi'_i})$.

Теперь нам надо будет обобщить данную операцию так, чтобы она могла работать с перестановками, находящимися в ветвях сопоставления с образцом (т.е. с учётом связанных переменных). Для этого введём операцию декомпозиции перестановки параметров на перестановку, отделяющую связанные переменные от несвязанных и перестановку, переставляющую связанные переменные. Оказывается, такую операцию можно определить в терминах уже введённых операций:

$$
\begin{aligned}
&\textit{branch-split}(\theta, k) = (\delta', \xi), \text{ где} \\
&\quad (\delta, id_{k\to(k+m)}, \xi) = \textit{bisplit}(id_{k\to n}, \theta) \\
&\quad \delta' : m \hookrightarrow (n-k) \\
&\quad \delta'(i) = \delta(k+i) - k
\end{aligned}
$$

Заметим, что $shift(\delta', k) = \delta$, поэтому $shift(\delta', k) \circ \xi = \theta$, что и является основным нужным нам свойством. Кроме того, для любой перестановки параметров ζ верно $\textit{branch-split}(shift(\zeta, k) \circ \theta, k) = (\zeta \circ \delta', \xi)$, что тоже легко доказать.

Теперь определим функцию декомпозиции терма, являющегося правой частью формулы определения, как на Рис. 3.8. Данная функция специально спроектирована так, чтобы выполнялись следующие утверждения, которые следуют из аналогичных утверждений для *nsplit* и *branch-split*.

Утверждение 8. Пусть ξt имеет вид правой части формулы определения. Пусть $decompose(\xi t) = \delta t'$. Тогда $push(\xi t) = push(\delta t')$.

Утверждение 9. Пусть ξt имеет вид правой части формулы определения. Пусть $decompose(\xi t) = \delta t'$. Тогда для любой перестановки параметров ζ верно, что $decompose(push((\zeta \circ \xi)t)) = (\zeta \circ \delta)t'$.

Данную операцию уже можно использовать для приведения к каноническому виду, если как-нибудь переносить выделенную перестановку δ в левую часть (для этого можно также воспользоваться операцией *bisplit*). Однако сначала нам надо разобраться с вызовами функций, потому что из первого параметра конструкции **Call** перестановка параметров не выносится, однако её можно применить для перестановки остальных параметров конструкции,

Рис. 3.8 Операция декомпозиции правой части формулы определения

$$
\begin{aligned}
&\mathit{decompose}(\theta t) = \theta\ \mathit{decompose}(t)\\
&\mathit{decompose}(\mathbf{Id}(\theta f)) = \theta\,\mathbf{Id}(\mathit{id}\ f)\\
&\mathit{decompose}(\mathbf{Var}) = \mathit{id}_1\,\mathbf{Var}\\
&\mathit{decompose}(\mathbf{Call}(t_0)) = \mathit{id}_0\,\mathbf{Call}(t_0)\\
&\mathit{decompose}(\mathbf{Call}(t_0, \overline{\theta_i f_i})) = \delta\,\mathbf{Call}(t_0, \overline{\xi_i f_i}),\ \text{где}\\
&\qquad (\delta, \overline{\xi_i}) = \mathit{nsplit}(\overline{\theta_i})\\
&\mathit{decompose}(\mathbf{Cons}_C()) = \mathit{id}_0\,\mathbf{Cons}_C()\\
&\mathit{decompose}(\mathbf{Cons}_C(\overline{\theta_i f_i})) = \delta\,\mathbf{Cons}_C(\overline{\xi_i f_i}),\ \text{где}\\
&\qquad (\delta, \overline{\xi_i}) = \mathit{nsplit}(\overline{\theta_i})\\
&\mathit{decompose}(\mathbf{Case}_{\overline{C_i}}(\theta_0 f_0, \overline{\theta_i f_i})) = \delta\,\mathbf{Case}_{\overline{C_i}}(\xi_0 f_0, \overline{(\mathit{shift}(\phi_i, k_i) \circ \xi_i) f_i}),\ \text{где}\\
&\qquad (\delta, \xi_0, \overline{\phi_i}) = \mathit{nsplit}(\theta_0, \overline{\zeta_i})\\
&\qquad (\zeta_i, \xi_i) = \mathit{branch\text{-}split}(\theta_i, k_i)\\
&\qquad k_i = \mathit{arity}(C_i)
\end{aligned}
$$

например:

$$\mathbf{Call}(\{1 \to 2, 2 \to 1\} f, g, h) \approx \mathbf{Call}(f, h, g)$$

Для этого определим следующую функцию на термах, представляющих правые части формул определений:

$$
\begin{aligned}
&\mathit{canon\text{-}call}(\xi\,\mathbf{Call}(\theta f, \overline{t_i})) = \xi\,\mathbf{Call}(\mathit{id}_m f, \overline{t_{\theta(j)}})\\
&\mathit{canon\text{-}call}(t) = t, \text{ для термов другого вида}
\end{aligned}
$$

Нетрудно проверить, что $(\!|\mathit{canon\text{-}call}(t)|\!)_\mu = (\!|t|\!)_\mu$. Отметим, что данная операция может удалять некоторые из аргументов, если θ не является биекцией. С точки зрения преобразования полипрограмм это будет означать, что после приведения к каноническому виду может быть потерян путь между функциями, если рассматривать полипрограмму как граф, однако это не будет для нас особой проблемой.

Теперь осталось соединить всё вместе.

Определение 17. Пусть $\theta f \equiv t$ — формула определения. Тогда определим

понятие *канонической формы* данной формулы следующим образом:

$$\begin{aligned} &\mathit{canon}(\theta f \equiv t) = (\sigma f \equiv \xi t'),\ \text{где} \\ &\quad \delta t' = \mathit{decompose}(\mathit{canon\text{-}call}(t)) \\ &\quad (\delta', \xi, \sigma) = \mathit{bisplit}(\delta, \theta) \end{aligned}$$

Здесь мы сначала выносим перестановку δ наружу из терма t при помощи *decompose*. Затем мы производим взаимную декомпозицию перестановки при правой части δ и перестановки при левой части θ, получая новые перестановки при правой и левой частях ξ и σ. Тут важно, что мы используем именно такой порядок, сначала δ, потом θ, потому что в результате новая перестановка при правой части ξ будет иметь вид канонического вложения $id_{m\to n}$, что позволит в практической реализации её не хранить. В принципе можно было бы сделать наоборот, но мы предпочитаем, чтобы все нетривиальные перестановки были при функциональных символах, т.к. это упростит дальнейшее изложение.

Утверждение 10. Для любой формулы определения p выполняется $[\![p]\!]_\mu \Leftrightarrow [\![\mathit{canon}(p)]\!]_\mu$

Определение 18. Определим отношение $\approx$ на формулах определений как $p \approx q \Leftrightarrow \mathit{canon}(p) = \mathit{canon}(q)$.

Также будем использовать запись $\mathit{nsplit}(\overline{\theta_i}) \approx \mathit{nsplit}(\overline{\xi_i})$, если выполнено $\pi_j(\mathit{nsplit}(\overline{\theta_i})) = \pi_j(\mathit{nsplit}(\overline{\xi_i}))$ для $j \geq 2$.

Пример 8. Рассмотрим пример приведения определения к каноническому виду. Пусть определение имеет следующий вид:

$$\begin{aligned} id_{3\to4}\, f \equiv id_{3\to4}\, \mathbf{Call}(&\{1 \to 3, 2 \to 2\}_{2\to3}\, g, \\ &id_3\, h_1, \\ &id_{2\to3}\, h_2, \\ &\{1 \to 2, 2 \to 1\}_{2\to3}\, h_3) \end{aligned}$$

Оно приблизительно соответствует такому определению (некоторые переста-

новки параметров изображены лямбда-выражениями):

$$(\lambda(x,y,z,u).f(x,y,z))(x,y,z,u) \equiv$$
$$\quad (\lambda(k,l,m).g(m,l))(h_1(x,y,z), h_2(x,y), h_3(y,x))$$

Сначала применим операцию *canon-call* к правой части:

$$id_{3\to4}\, f \equiv id_{3\to4}\, \mathbf{Call}(id_2\, g, \{1\to2, 2\to1\}_{2\to3}\, h_3, id_{2\to3}\, h_2)$$
$$(\lambda(x,y,z,u).f(x,y,z))(x,y,z,u) \equiv g(h_3(y,x), h_2(x,y))$$

Теперь применим *decompose* к правой части, для чего надо потребуется перестановка параметров $nsplit(\{1\to2, 2\to1\}_{2\to3}, id_{2\to3}) = (\{1\to2, 2\to1\}_{2\to3}, id_2, \{1\to2, 2\to1\}_{2\to2})$.

$$id_{3\to4}\, f \equiv \{1\to2, 2\to1\}_{2\to4}\, \mathbf{Call}(id_2\, g, id_2\, h_3, \{1\to2, 2\to1\}_{2\to2}\, h_2)$$
$$(\lambda(x,y,z,u).f(x,y,z))(y,x,z,u) \equiv g(h_3(x,y), h_2(y,x))$$

Наконец, применим *bisplit* к перестановкам $\{1\to2, 2\to1\}_{2\to4}$ и $id_{3\to4}$ при разных частях формулы.

$$\{1\to2, 2\to1, 3\to3\}_{3\to3}\, f \equiv$$
$$\quad id_{2\to3}\, \mathbf{Call}(id_2\, g, id_2\, h_3, \{1\to2, 2\to1\}_{2\to2}\, h_2)$$
$$f(y,x,z) \equiv g(h_3(x,y), h_2(y,x))$$

Отметим, что в результате оказалось, что f на самом деле не зависит от z, а это значит, что (забегая вперёд) арность f можно будет понизить, что делается при помощи правила (arity-reduction). △

3.4 Категория полипрограмм

В данном разделе мы формально определим, что такое правило и что такое применение правила к полипрограмме (в бесточечно-расчленённом представлении). Для преобразования полипрограмм будем использовать подход двойного кодекартова квадрата (double-pushout approach), используемый для переписывания графов, поэтому нам потребуется описать категорию полипро-

грамм и доказать, что она обладает некоторыми нужными свойствами.

Т.к. обычные преобразования должны выполняться также с точностью до перестановки параметров, удобно включить перестановку параметров в понятие морфизма полипрограмм, т.е. функциональные символы должны отображаться в пары вида (перестановка параметров, функциональный символ). Прежде чем вводить категорию полипрограмм, введём категорию функциональных символов.

Определение 19. *Морфизмом функциональных символов* назовём функцию $\alpha : F_1 \to \Theta \times F_2$, где F_1 и F_2 — множества функциональных символов, а Θ — множество перестановок параметров, причём если $\alpha(f_1) = (\theta, f_2)$, то $\theta : arity(f_2) \hookrightarrow arity(f_1)$. Композиция двух морфизмов функциональных символов определена так:

$$(\alpha \circ \beta)(f) = (\theta \circ \xi, h)$$

где $\beta(f) = (\theta, g)$, а $\alpha(g) = (\xi, h)$. Тождественный морфизм имеет вид $\iota(f) = (id_{arity(f)}, f)$.

Морфизм функциональных символов просто заменяет одни функциональные символы на другие (возможно с отождествлением) с некоторой перестановкой параметров. При этом возможно понижение арности, но не повышение (потому что перестановки параметров инъективные).

Утверждение 11. Множества функциональных символов с морфизмами функциональных символов образуют категорию.

Для категории конечных множеств функциональных символов удаётся доказать конечную кополноту, чего нам будет достаточно, потому что дальше мы будем работать в основном с конечными полипрограммами (множества определений и функциональных символов которых конечны).

Утверждение 12. Категория *конечных* множеств функциональных символов конечно кополна, т.е. в ней есть начальный объект, бинарные копроизведения и коуравнители.

Доказательство этого утверждения приведено в Приложении В на с. 204.

Естественным образом определяется операция применения морфизма функциональных символов к определению полипрограммы. Отметим, что она оставляет формулу определения согласованной по арности.

Определение 20. Пусть $\alpha : F_1 \to F_2$ — морфизм функциональных символов, а $\theta_0 f_0 \equiv \xi\, L(\overline{\theta_i f_i})$ — формула определения, $f_0, f_i \in F_1$. Тогда операция применения морфизма к формуле определения определена как:

$$\alpha(\theta_0 f_0 \equiv \xi\, L(\overline{\theta_i f_i})) = ((\theta_0 \circ \xi_0) g_0 \equiv \xi\, L(\overline{(\theta_i \circ \xi_i) g_i})),\ \text{где}\ \alpha(f_i) = (\xi_i, g_i)$$

Утверждение 13. Морфизмы функциональных символов хорошо взаимодействуют с понятием канонической формы, а именно если $p_1 \approx p_2$, то $\alpha(p_1) \approx \alpha(p_2)$.

Доказательство этого утверждения приведено в Приложении В на с. 206.

Применение морфизма к определению естественным образом распространяется на всю полипрограмму.

Определение 21. Пусть $\alpha : F_1 \to F_2$ — морфизм функциональных символов, $G = \langle H, F_1, v\rangle$ — полипрограмма со множеством определений H, множеством функций F_1 и функцией отображение определений в соответствующие формулы определений v. Тогда определим операцию применения морфизма α к полипрограмме как $\alpha(G) = \langle H, F_2, \alpha \circ v\rangle$, где $(\alpha \circ v)(h) = \alpha(v(h))$.

Определение 22. Введём категорию полипрограмм следующим образом. Объектами этой категории будут полипрограммы, а морфизмами между полипрограммами $G_i = \langle H_i, F_i, v_i\rangle$, $i = 1, 2$ будут пары $\langle\gamma, \alpha\rangle$, где $\alpha : F_1 \to F_2$ — морфизм функциональных символов, а $\gamma : H_1 \to H_2$ — отображение определений, причём $\alpha(v_1(h)) \approx v_2(\gamma(h))$ для всех $h \in H_1$.

Для удобства будем использовать функции доступа к компонентам морфизма $\mathit{defmap}(\langle\gamma, \alpha\rangle) = \gamma$ и $\mathit{funmap}(\langle\gamma, \alpha\rangle) = \alpha$. Кроме того, иногда вместо $\mathit{funmap}(\phi)(f)$ или $\mathit{defmap}(\phi)(h)$ будем писать просто $\phi(f)$ и $\phi(h)$

Утверждение 14. Описанная конструкция является категорией, т.е. выполняется ассоциативность и тождественные морфизмы действительно являются нейтральными элементами.

Отметим, что данную категорию можно определить как категорию запятой (**Set** $\downarrow$ **H**), где **Set** выступает в качестве тождественного функтора над **Set**, а **H** — функтор из категории функциональных символов в **Set**, отображающий множества функциональных символов во множества классов

эквивалентностей формул определений над этими символами относительно $\approx$.

Морфизм между полипрограммами в описанной категории означает вложение одной полипрограммы в другую с точностью до переименования функций с возможным отождествлением функций, модификацией функций при помощи перестановок параметров и отождествлением определений с эквивалентными формулами.

Также отметим, что если α — морфизм функциональных символов такой, что множество формул полипрограммы $\alpha(G_1)$ является подмножеством множества формул полипрограммы G_2, то существует морфизм полипрограмм $\langle \gamma, \alpha \rangle : G_1 \to G_2$ для некоторого γ. Мы иногда будем использовать морфизмы функциональных символов в контексте морфизмов полипрограмм именно в этом смысле (т.е. записывать $\alpha : G_1 \to G_2$).

Также будем использовать запись $\phi(G)$, где $\phi = \langle \gamma, \alpha \rangle : G \to H$, для обозначения полипрограммы $\langle \mathrm{Im}(\gamma), \mathit{funs}(H), \left(\mathit{labeling}(H)\big|_{\mathrm{Im}(\gamma)} \right) \rangle$, т.е. образа G относительно ϕ как фрагмента полипрограммы H. Мы также будем использовать обозначение $\lfloor \phi \rfloor$ для соответствующего морфизма $\lfloor \phi \rfloor : G \to \phi(G)$.

Напомним, что ранее мы ввели обозначения для компонент полипрограммы $G = \langle \mathit{defs}(G), \mathit{funs}(G), \mathit{labeling}(G) \rangle$.

Следующее утверждение говорит о том, что если одна и та же полипрограмма вкладывается в две разные полипрограммы, но с одним и тем же морфизмом функциональных символов, и без отождествления определений, то её образы изоморфны.

Утверждение 15. Если $\phi_i = \langle \gamma_i, \alpha \rangle : G \to H_i$, $i = 1, 2$ — морфизмы полипрограмм и γ_i инъективны, то $\phi_1(G) \cong \phi_2(G)$.

В категории полипрограмм также удобно работать и с семантикой, т.к. семантику можно представить как бесконечную полипрограмму, в которой есть все верные определения, и только они.

Определение 23. *Семантической полипрограммой* назовём (бесконечную) полипрограмму, определённую как $\langle H, \mathbb{D}, id \rangle$, где

$$H = \{ p \mid [\![p]\!]_{id} = 1, p = \mathit{canon}(p) \}$$

Т.е. H — множество всех истинных формул в канонической форме с семантическими функциями из $\mathbb{D}$ в качестве функциональных символов, причём

каждая формула входит в полипрограмму только один раз. Обозначать эту полипрограмму будем $\mathbb{S}$.

Утверждение 16. Полипрограмма G имеет модель тогда и только тогда, когда существует морфизм $\phi : G \to \mathbb{S}$ (будем называть такие морфизмы семантическими).

Отметим, что семантические морфизмы соответствуют моделям полипрограмм, но грубо: из-за перестановок параметров для одной и той же модели можно построить несколько морфизмов [1]. По этой причине нужно аккуратно различать понятия модели и семантического морфизма в тех случаях, когда речь идёт о единственности модели (как в Главе 5).

Далее нам понадобится обобщение понятия семантической эквивалентности полипрограмм, чтобы можно было рассматривать полипрограммы с совсем разными множествами функций.

Определение 24. Две полипрограммы G и H назовём семантически эквивалентными относительно полипрограммы D с парой морфизмов $\psi : D \to G$ и $\phi : D \to H$, если

- Для любого $\mu : G \to \mathbb{S}$ существует $\eta : H \to \mathbb{S}$ такой, что $\mu \circ \psi = \eta \circ \phi$.
- Для любого $\eta : H \to \mathbb{S}$ существует $\mu : G \to \mathbb{S}$ такой, что $\mu \circ \psi = \eta \circ \phi$.

Здесь D вместе с ψ и ϕ просто указывает на соответствующие функции в полипрограммах G и H. Соответствующие функции могут иметь разный порядок аргументов и вообще разную арность, поэтому это определение более общее, чем в предыдущей главе, однако они связаны следующим утверждением.

Утверждение 17. Пусть даны две полипрограммы G и H. Пусть они эквивалентны относительно полипрограммы D с парой морфизмов ψ, ϕ. Тогда существует полипрограмма H' такая, что есть морфизм $\tau : H \to H'$, и что G и H' эквивалентны на подмножестве функциональных символов $\mathrm{Im}(\mathit{funmap}(\psi))$.

[1] В принципе этого можно было бы избежать, используя вместо функций из D в семантической полипрограмме классы эквивалентности функций, отличающихся только порядком аргументов, и изменив определение категории полипрограмм так, чтобы уравнять некоторые морфизмы, работающие с коммутативными функциями.

Доказательство. H' состоит из H, в которой функции переименованы так, чтобы они не пересекались с функциями из G (отсюда сразу получаем τ), и из дополнительных определений вида $\theta g \equiv \mathbf{Id}(\zeta h)$, где g взято из $\mathrm{Im}(\mathit{funmap}(\psi))$, а h взято из полипрограммы H', причём существует f из D такая, что $\psi(f) = (\theta, g)$ и $\tau(\phi(f)) = (\zeta, h)$. □

Теперь опишем основные конструкции, существующие в категории полипрограмм, которые нам понадобятся в дальнейшем.

Утверждение 18. Категория *конечных* полипрограмм является конечно кополной, или, эквивалентно, в ней существуют:

- Начальный объект — пустая полипрограмма
- Копроизведение — размеченное объединение двух полипрограмм
- Коуравнители

Кроме того, если морфизмы полипрограмм образуют некоторую конструкцию в категории полипрограмм, то соответствующие морфизмы функциональных символов будут образовывать ту же конструкцию в категории функциональных символов.

Доказательство. Данное утверждение следует из свойств категорий запятой.

□

Самая важная конструкция, которая нам понадобится — кодекартов квадрат, который выражается при помощи копроизведений и коуравнителей. Кодекартовы квадраты используются при применении правил преобразований к полипрограмме.

Определение 25. *Правилом* преобразования полипрограмм будем называть конструкцию вида $L \hookleftarrow K \to R$ в категории полипрограмм (т.е. два морфизма с общим доменом, один из которых является мономорфизмом).

Полипрограмма L — это левая часть, R — правая часть, K — полипрограмма, состоящая из функций и определений, которые сохраняются при применении правила.

Определение 26. Пусть $r : L \hookleftarrow K \to R$ — правило, G — полипрограмма, $\varepsilon : L \to G$. Тогда *применением* данного правила будем называть конструкцию из двух кодекартовых квадратов следующего вида:

$$\begin{array}{ccccc} L & \hookleftarrow & K & \to & R \\ \downarrow{\scriptstyle \varepsilon} & & \downarrow{\scriptstyle \omega} & & \downarrow \\ G & \hookleftarrow & D & \to & H \end{array}$$

Применение правила будем записывать как $G \Rightarrow_{r,\varepsilon} H$.

Отметим, что левый кодекартов квадрат строится не по тем морфизмам, на которых обычно строятся кодекартовы квадраты. Такое построение не всегда существует, а кроме того оно не единственно, поэтому мы потом введём дополнительное ограничение на правила и определим стандартный способ построения промежуточной полипрограммы D. Но сначала покажем, что конструкция из кодекартовых квадратов хорошо взаимодействует с семантикой.

Определение 27. Правило $r : L \overset{\psi}{\hookleftarrow} K \overset{\phi}{\to} R$ семантически *корректно* , если L и R эквивалентны относительно полипрограммы K с морфизмами ψ и ϕ.

Утверждение 19. Пусть $r : L \overset{\psi_1}{\hookleftarrow} K \overset{\phi_1}{\to} R$ корректно. Пусть $G \Rightarrow_{r,\varepsilon} H$:

$$\begin{array}{ccccc} L & \overset{\psi_1}{\hookleftarrow} & K & \overset{\phi_1}{\to} & R \\ \downarrow{\scriptstyle \varepsilon} & (1) & \downarrow{\scriptstyle \omega} & (2) & \downarrow{\scriptstyle \gamma} \\ G & \overset{\psi_2}{\hookleftarrow} & D & \overset{\phi_2}{\to} & H \end{array}$$

Тогда G и H эквивалентны относительно полипрограммы D с морфизмами ψ_2 и ϕ_2.

Доказательство. Пусть $\mu_1 = \mu \circ \varepsilon$. Тогда по определению корректности правила существует $\eta_1 : R \to \mathbb{S}$, причём $\eta_1 \circ \phi_1 = \mu_1 \circ \psi_1 = \mu \circ \varepsilon \circ \psi_1 = \mu \circ \psi_2 \circ \omega$. Получаем, что $\eta_1 \circ \phi_1 = (\mu \circ \psi_2) \circ \omega$, а значит по определению кодекартова квадрата существует $\eta : H \to S$ такой, что $\eta \circ \phi_2 = \mu \circ \psi_2$. Второй пункт доказывается точно так же. □

Отметим, что корректность правила ещё не гарантирует того, что оно полностью сохраняет семантическую информацию, потому что корректность

гарантируется только на сохранённых функциях (т.е. входящих в K) — если какие-то функции будут удалены, то информация о них пропадёт. *По этой причине будем ограничиваться только такими правилами, которые не удаляют функции.* Это очень сильно упрощают теорию преобразования полипрограмм. При этом ограничении гарантируется существование левого кодекартова квадрата, хотя не гарантируется его единственность (даже с точностью до изоморфизма).

Также наложим ещё одно (несущественное) ограничение на правила: будем считать, что $K \subseteq L$ в следующем смысле.

Определение 28. Пусть $G_i = \langle H_i, F, v_i \rangle$, $i = 1, 2$. Тогда $G_1 \subseteq G_2 \Leftrightarrow H_1 \subseteq H_2 \wedge v_1 = \left(v_2\big|_{H_1}\right)$

Заметим, что в данном определении мы требуем, чтобы множества функциональных символов у полипрограмм были одинаковые.

Определение 29. На множестве всех полипрограмм-фрагментов некоторой полипрограммы (в смысле $\subseteq$) определены привычные теоретико-множественные операции $(\cup, \cap, \setminus)$, работающие только со множеством определений, т.е.:

$$\langle H_1, F, \left(v\big|_{H_1}\right)\rangle \star \langle H_2, F, \left(v\big|_{H_2}\right)\rangle = \langle H_1 \star H_2, F, \left(v\big|_{H_1 \star H_2}\right)\rangle$$

где $\star \in \{\cup, \cap, \setminus\}$.

Можно заметить, что если $G_1 \subseteq G_2$, то существует морфизм $\langle id_{H_1 \to H_2}, id \rangle : G_1 \to G_2$. Такие морфизмы будем обозначать как $id_{G_1 \to G_2} : G_1 \subseteq G_2$. Кроме того, любой морфизм полипрограмм $\psi : G_1 \to G_2$, можно представить в виде композиции $id_{G_1' \to G_2} \circ \lfloor \psi \rfloor$, где $\lfloor \psi \rfloor : G_1 \to G_1'$, а $G_1' = \psi(G_1)$.

Далее будем иногда опускать скобки и $\circ$, т.е. $\phi\psi G = (\phi \circ \psi)(G)$. Также иногда будем опускать морфизмы вида $id_{A \to B}$ при применении морфизмов к полипрограмме, например, будем писать εK вместо $\varepsilon id_{K \to L} K$

Утверждение 20. $\phi(G \cup H) = \phi G \cup \phi H$

Утверждение 21. Если $defmap(\phi)$ инъективна, то $\phi(G \setminus H) = \phi G \setminus \phi H$.

Определение 30. Пусть $r : L \supseteq K \to R$ — правило, $\varepsilon : L \to G$. Тогда *стандартным применением правила* назовём такое, при котором промежуточная полипрограмма $D = (G \setminus \varepsilon L) \cup \varepsilon' K \subseteq G$, где $\varepsilon' = \varepsilon \circ id_{K \to L}$, а $\omega : K \to D$

равно $id_{\varepsilon' K \to D} \circ \lfloor \varepsilon' \rfloor$. В дальнейшем символом $\Rightarrow_{r,\varepsilon}$ будем обозначать именно стандартное применение правила.

В данном определении промежуточная полипрограмма D строится при помощи обычных теоретико-множественных операций, а именно сначала удаляются определения, входящие в L (пропущенный через ε), а потом добавляются определения, входящие в K. При таком определении левый кодекартов квадрат становится точно определён при данном ε, а правый только с точностью до изоморфизма, чего нам достаточно. Стандартное применение правил построено таким образом, чтобы удалять все определения, которые можно удалить, старое же неоднозначное определение позволяло применять удаляющие правила так, чтобы они ничего не удаляли, что не является тем, что нам нужно.

Также все правила, которые мы будем рассматривать, будут иметь вид $L \supseteq K \to R$, где $K \to R$ инъективен по функциональным символам (неинъективность по определениям возможна, см. правило (remove-dups)). Склеивание функций при помощи морфизма $K \to R$ удобно для того, чтобы моделировать слияние функций в полипрограмме, однако оно плохо взаимодействуют с семантикой. Например, рассмотрим правило $\{P(f); P(g)\} \xrightarrow{\phi} \{P(h)\}$, в котором $L = K$ (т.е. оно не производит явного удаления) и $\phi(f) = \phi(g) = (id, h)$. Пусть, например, $P(f)$ означает, что $f(x, y) = x + y$, и в $\mathbb{S}$ существует функция add такая, что $\{P(add), P(\{1 \to 2, 2 \to 1\}add)\} \subseteq \mathbb{S}$. Тогда рассмотрим семантический морфизм левой полипрограммы $\mu : L \to \mathbb{S}$ такой, что $\mu(f) = (id, add)$ и $\mu(g) = (\{1 \to 2, 2 \to 1\}, add)$. Тогда, если правило корректное, то мы можем построить $\eta : \{P(h)\} \to S$ так, что $\mu = \eta \circ \phi$. Но это значит, что $\mu(f) = \mu(g)$, потому что $\phi(f) = \phi(g)$, что по построению μ неверно. Это означает, что такое правило не может быть корректным.

Очевидный выход из сложившейся ситуации — ограничиваться правилами с морфизмами, инъективными по функциям. Слияние функций при этом приходится моделировать деструктивными преобразованиями. Существует также альтернативное решение. Можно заметить, что проблема возникает оттого, что в семантике есть коммутативные функции, и из-за этого морфизмы $\mu(f) = (id, add), \mu(g) = (\{1 \to 2, 2 \to 1\}, add)$ и $\mu'(f) = \mu'(g) = (id, add)$ оказываются различными, хотя из-за коммутативности функции add их можно было бы считать одинаковыми — это также главная причина, по которой семантические морфизмы не находятся во взаимно-однозначном соответствии

с моделями. Поэтому можно было бы ввести равенство морфизмов, учитывающее коммутативность. Для этого можно было бы расширить категорию перестановок параметров Θ, введя отдельные типы для коммутативных функций. Данный подход выглядит привлекательно, но его исследование выходит за рамки данной работы.

Отметим, что можно было бы ввести семантику иначе, так, чтобы в ней в принципе не было коммутативных функций — это значительно упростило бы многие вещи, однако такая семантика недостаточно общепринята (хотя на практике коммутативные функции по всей видимости встречаются редко), поэтому данный подход выходит за рамки данной работы.

Определение 31. Пусть $r : L \supseteq K \to R$ — правило, применение которого к G даёт морфизмы $G \supseteq D \overset{\langle \dots, \alpha \rangle}{\to} H$. Тогда морфизм функциональных символов α будем называть *функцией отображения функциональных символов* и использовать запись $G \overset{\alpha}{\Rightarrow}_{r,\varepsilon} H$.

Если правило не отождествляет функциональные символы и не изменяет их арность (в нашей системе правил этим условиям не удовлетворяет только правило понижения арности), то всегда можно выбрать (среди изоморфных вариантов) H и α так, чтобы $funs(G) \subseteq funs(H)$ и $\alpha(f) = (id, f)$ для любого f. Это бывает удобно для проведения рассуждений о правилах.

Определение 32. Пусть дано некоторое множество правил $\mathcal{R}$. Тогда $G \overset{\alpha}{\Rightarrow}{}^*_{\mathcal{R}} H$, если выполнено хотя бы одно из:

- $G \overset{\alpha}{\cong} H$ (полипрограммы изоморфны с изоморфизмом $\langle \dots, \alpha \rangle$)
- $G \overset{\alpha}{\Rightarrow}_r H$ для некоторого $r \in \mathcal{R}$
- $G \overset{\beta}{\Rightarrow}{}^*_{\mathcal{R}} J$, $J \overset{\gamma}{\Rightarrow}{}^*_{\mathcal{R}} H$ и $\alpha = \gamma \circ \beta$

Также будем обозначать $G \overset{\alpha}{\Rightarrow}{}^?_{\mathcal{R}} H$, если выполнено $G \overset{\alpha}{\cong} H$ или $G \overset{\alpha}{\Rightarrow}_{\mathcal{R}} H$.

Наконец, приведём несколько утверждений, которые нам понадобятся в дальнейшем. Начнём с утверждения, позволяющее преобразовывать любое корректное (или хотя бы корректное в одну сторону) деструктивное правило в корректное недеструктивное правило (не удаляющее определения).

Утверждение 22. Пусть $r : L \overset{\psi}{\supseteq} K \overset{\phi}{\to} R$ корректно или хотя бы "полукорректно", т.е. для любого $\mu : L \to \mathbb{S}$ существует $\eta : R \to \mathbb{S}$ такой, что $\mu \circ \psi = \eta \circ \phi$.

Пусть $\phi' : L \to R'$ и $\psi' : R \to R'$ таковы, что $\phi' \circ \psi = \psi' \circ \phi$ — кодекартов квадрат. Тогда $r' : L \supseteq L \xrightarrow{\phi'} R'$ также корректно.

Доказательство. Пусть $\mu : L \to S$. Тогда существует $\eta : R \to \mathbb{S}$ и $\eta \circ \phi = \mu \circ \psi$. Тогда по свойству кодекартовых квадратов существует $\eta' : R' \to \mathbb{S}$ такой, что $\eta' \circ \phi' = \mu = \mu \circ id$.

Теперь в другую сторону, пусть $\eta : R' \to \mathbb{S}$. Тогда сразу можно взять $\mu \circ id = \mu = \eta \circ \phi'$, корректность исходного правила при этом не требуется. □

Также нам понадобится утверждение, позволяющее применять некоторые правила как бы параллельно, используя уже введённую механику последовательного применения правил.

Утверждение 23. Пусть $r_i : L_i \supseteq K_i \to R_i$, $i = 1, 2$ — два правила. Пусть $G \Rightarrow_{r_i, \varepsilon_i} H_i$ и верно, что $\varepsilon_1 L_1 \cap \varepsilon_2 L_2 = \varepsilon_1 K_1 \cap \varepsilon_2 K_2$. Тогда существует полипрограмма M такая, что $H_1 \Rightarrow_{r_2} M$ и $H_2 \Rightarrow_{r_1} M$. Такую пару применений правил будем называть *параллельно независимой*.

Доказательство. См. [18]. □

3.5 Неоднозначность применения правил

Результат применения правила может существенно зависеть от того, с каким морфизмом, отображающим левую часть правила в полипрограмму, оно применяется, даже если отличие только в перестановках параметров.

Пример 9. Рассмотрим следующее правило, являющееся частным случаем правила (trans):

$$\left\{ \begin{array}{l} id_N f \equiv id_{0 \to N} \mathbf{Cons}_C() \\ id_N g \equiv id_{0 \to N} \mathbf{Cons}_C() \end{array} \right\} \mapsto \left\{ \begin{array}{l} id_N f \equiv id_{0 \to N} \mathbf{Cons}_C() \\ id_N g \equiv id_{0 \to N} \mathbf{Cons}_C() \\ id_N f \equiv id_N \mathbf{Id}(id_N g) \end{array} \right\}$$

N — какое-то большое число, выбранное так, чтобы быть больше арности любой функции, встречающейся на практике. Применим правило к следующей

полипрограмме:

$$f(x) \equiv C()$$
$$g(x) \equiv C()$$

Эта полипрограмма в бесточечно-расчленённом представлении выглядит так:

$$id_1 f \equiv id_{0\to 1}\mathbf{Cons}_C()$$
$$id_1 g \equiv id_{0\to 1}\mathbf{Cons}_C()$$

Применим правило сначала с морфизмом ϕ из левой части правила в полипрограмму, заданным на функциях следующим образом:

$$\phi(f) = (id_{1\to N}, f)$$
$$\phi(g) = (id_{1\to N}, g)$$

В результате получится (после приведения к каноническому виду) следующая полипрограмма (либо изоморфная ей):

$$id_1 f \equiv id_{0\to 1}\mathbf{Cons}_C()$$
$$id_1 g \equiv id_{0\to 1}\mathbf{Cons}_C()$$
$$id_1 f \equiv id_1\mathbf{Id}(id_1 g)$$

Новое определение в человеко-читаемом виде записывается как $f(x) \equiv g(x)$.

Теперь рассмотрим другой морфизм, ϕ', заданный на функциях иначе:

$$\phi(f) = (id_{1\to N}, f)$$
$$\phi(g) = (\{1 \to 2\}_{1\to N}, g)$$

Действительно, он является морфизмом указанных полипрограмм, потому что определение

$$\{1 \to 2\}_{1\to N} g \equiv id_{0\to N}\mathbf{Cons}_C()$$

после приведения к каноническому виду превращается в определение

$$id_1 g \equiv id_{0\to 1}\mathbf{Cons}_C()$$

После применения правила получится следующая полипрограмма (новое определение не в каноническом виде для наглядности):

$$id_1 f \equiv id_{0\to 1}\mathbf{Cons}_C()$$
$$id_1 g \equiv id_{0\to 1}\mathbf{Cons}_C()$$
$$id_{1\to N} f \equiv \{1 \to 2\}_{1\to N}\mathbf{Id}(id_1 g)$$

После приведения к канонической форме новое определение будет выглядеть как

$$\{1 \to 2\}_{1\to 2} f \equiv id_{1\to 2}\mathbf{Id}(id_1 g)$$

Или, иначе, $f(y) \equiv g(x)$. Это определение означает, что f и g являются одинаковыми константами. По смыслу это совершенно не то же самое, что $f(x) \equiv g(x)$. △

Теоретически, в этом нет ничего страшного, однако таких отличающихся лишь перестановкой параметров морфизмов может быть очень много, на практике рассматривать их все невозможно. Поэтому крайне желательно, чтобы правила обладали таким свойством, чтобы результат их применения не зависел от выбора перестановок параметров в морфизме. Однако из примера видно, что это свойство не выполняется даже для самого главного правила. Тем не менее, если наложить ограничение на полипрограмму, оно начнёт выполняться.

Определение 33. Будем говорить, что определение находится в *полностью канонической форме*, если оно находится в канонической форме и имеет вид $\theta f \equiv id_m\mathbf{L}(\ldots)$ (важно, что при правой части тождественная перестановка id_m, а не каноническое вложение $id_{m\to n}$). Будем говорить, что полипрограмма находится в полностью канонической форме, если все её определения находятся в полностью канонической форме.

Полипрограмму можно всегда привести к полностью канонической форме, применив правило понижения арности (arity-reduction). Например, програм-

ма из примера станет выглядеть следующим образом (в человеко-читаемой форме):

$$f() \equiv C()$$
$$g() \equiv C()$$

Как видно, из f и g были удалены фиктивные переменные, что позволило уравнять арности обеих частей.

Как будет показано в следующей главе, для правила (trans) верно, что если полипрограмма находится в полностью канонической форме, то при применении правила с морфизмами, отличающимися только перестановками параметров, получится одинаковый результат.

К сожалению, общего способа решить проблему сразу для всех используемых нами правил найдено не было, поэтому данный вопрос мы решим отдельно только для некоторых правил, а для остальных вопрос остаётся открытым.

3.6 Выводы

В данной главе было введено понятие полипрограммы в расчленённой форме. Все определения расчленённой полипрограммы являются простейшими определениями, что достигается за счёт введения промежуточных функций. Расчленённая форма полезна для увеличения обобществления общих подвыражений. Кроме того, она удобна для программной реализации, т.к. полипрограмма фактически превращается в гиперграф.

Также было введено понятие полипрограммы в бесточечно-расчленённом представления, с которым ещё легче работать формально и программно. При этом была формально описана функция преобразования из расчленённой формы в бесточечно-расчленённое представление. Основной особенностью бесточечно-расчленённого представления является особый подход к работе с переменными, основанный на явных и повсеместных переименованиях переменных, формально описываемых инъективными функциями над подмножествами натуральных чисел.

Также были приведены основные понятия и утверждения теории преобразования полипрограмм в бесточечно-расчленённом представлении, явля-

ющейся специализацией теории преобразования гиперграфов. Основное отличие от простой теории гиперграфов (кроме смены терминологии) заключается в наличии перестановок параметров — из-за них многие понятия и утверждения претерпевают изменения, т.к. большинство действий требуется производить с точностью до обратимых изменений перестановок параметров. Отметим, что встраивание перестановок параметров непосредственно в теорию не является однозначным решением — можно было бы работать с ними, как с обычными метками определений, но тогда потребовалось бы усложнение в других частях теории (в частности, приведение определений к каноническому виду стало бы деструктивным преобразованием).

Отметим некоторые важные результаты данной главы:

- Было показано, что перестановки параметров можно перемещать внутри формулы определения, сохраняя при этом её семантику. Было введено понятие канонической формы формулы определения, позволяющей алгоритмически определять, можно ли одну формулу превратить в другую такими перемещениями. Кроме того, для формул в канонической форме можно быстро определить, являются ли определяемые ими функции эквивалентными с точностью до перестановки параметров по транзитивности.

- Было показано, что полипрограммы образуют категорию.

- Было введено понятие правила преобразования и применения правила.

Глава 4

Локальные правила преобразования

В этой главе мы опишем локальные правила преобразования, которые мы применяем к полипрограммам. Мы называем их локальными, потому что они работают только с некоторым фрагментом, и для их применения не нужно анализировать всю полипрограмму, в отличие, например, от слияния по бисимуляции. Некоторые правила будут на самом деле семействами правил, параметризованные конкретными видами формул и иногда перестановками параметров и арностями функциональных символов. Самое главное свойство преобразований — корректность, поэтому мы явно докажем корректность нескольких преобразований, для остальных корректность доказывается аналогично. Кроме того, важным свойством является завершаемость. В целом наша система правил завершаемостью не обладает, поэтому проблему завершаемости мы решаем путём прерывания процесса в какой-то момент времени. При этом возникает проблема детерминированности, которую мы частично решаем путём установления порядка применения преобразований. Однако мы изгоняем таким образом не весь недетерминизм из нашего метода. Вопрос о том, является ли наша система преобразований детерминированной зависит от вопроса о конфлюэнтности группы правил, которые мы называем упрощающими. Разрешение данного вопроса не входит в настоящую диссертацию и оставлено на будущее.

При записи правил $L \supseteq K \to R$ мы не будем явно выписывать полипро-

грамму K, а будем считать, что она состоит из всех функций полипрограммы L и определений полипрограммы L, входящих также и в R.

Мы будем, если не сказано иначе, считать что все функции, упомянутые в правилах, имеют арности заведомо большие арности любой из функций преобразуемой полипрограммы. Это не влечёт проблем, т.к. наша система устроена так, что арности новых функций не могут неограниченно расти. Отметим, что вместо этого можно было бы ввести бесконечную арность, однако это усложнило бы запись некоторых операций из предыдущего раздела, поэтому мы так не делаем.

В большинстве правил мы не будем явно писать перестановки параметров в левой и правой частях, т.к. будем считать их по умолчанию каноническими вложениями $id_{m\to n}$, где m и n выводятся из контекста. Использование канонических вложений в правилах позволяет легче доказывать некоторые свойства правил. Иногда мы будем требовать, чтобы перестановки параметров были не просто каноническими вложениями, а тождественными перестановками (конечно, не все правила могут удовлетворить этому требованию).

Мы делим правила на две группы. Первая группа — упрощающие правила, применение которых организовано так, чтобы выполнялась завершаемость. Вторая группа — насыщающие правила, многие из них потенциально приводят к неограниченному разрастанию полипрограммы, поэтому их применение выполняется послойно и чередуется с применением упрощающих правил, а кроме того, все они выполняются недеструктивно.

Для большинства упрощающих правил мы укажем способ нахождения вложения левой части в преобразуемую полипрограмму (точнее, дополнения соответствия между определениями и функциями до морфизмов перестановками параметров) и докажем, что достаточно рассмотреть только конечное количество таких морфизмов для правила или семейства правил.

Кроме собственно правил преобразования нам понадобится преобразование, которое мы будем называть конгруэнтностью (оно соответствую конгруэнтности из Главы 2). Оно полностью аналогично "плохому" правилу $\{f \equiv \mathbf{Id}(g)\} \to \{f \equiv \mathbf{Id}(f)\}$, отождествляющему две функции, однако мы не будем считать это преобразование правилом, чтобы избежать проблем с его корректностью.

4.1 Упрощающие правила

Список упрощающих правил приведён в Таблице 4.1. Хотя все они сформулированы в виде правил, в нашей практической реализации они встроены в ядро движка преобразования полипрограмм, а не являются отдельными правилами. Это связано с тем, что они довольно сложно взаимодействуют друг с другом. К примеру, правило (remove-dups) обеспечивает завершаемость всех остальных правил, т.к. если не рассматривать применимость правил с точностью до удаления дубликатов, то любое недеструктивное правило можно применять бесконечное число раз. Некоторые правила, а именно (clone), (eq-symm), (eq-refl) вообще фактически не применяются в практической реализации и нужны только для того, чтобы выполнялись некоторые теоретические утверждения о свойствах системы переписывания.

Далее мы подробнее рассмотрим все упрощающие правила по одному.

4.1.1 Транзитивность

Правило 1 (trans)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{L}(\overline{h_i}) \\ g \equiv \mathbf{L}(\overline{h_i}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{L}(\overline{h_i}) \\ g \equiv \mathbf{L}(\overline{h_i}) \\ f \equiv \mathbf{Id}(g) \end{array}\right\}$$

Параметры: L.

Отметим, что мы опускаем перестановки параметров при функциях, т.к. они все являются каноническими вложениями. Далее левую часть правила будем называть L, а правую R.

Данное правило крайне важно для насыщения равенствами, т.к. позволяет (вместе с конгруэнтностью) поддерживать равенство функций в полипрограмме в состоянии конгруэнтной замкнутости. Оно также аналогично свёртке в суперкомпиляции и переписывании джунглей. Именно ради эффективности применения данного правила мы поддерживаем формулы определений в канонической форме. Действительно, если существует $\phi : L \to G$, то канонические формы формул образов обоих определений в G будут отличаться только перестановкой параметров при функций слева от равенства.

Таблица 4.1 Упрощающие правила

Обозначение	Смысл
(trans)	Транзитивность равенства $f \equiv L(h), g \equiv L(h) \Rightarrow f \equiv g$
(remove-dups)	Удаление дубликатов $f \equiv L(h), f \equiv L(h) \Leftrightarrow f \equiv L(h)$
(clone)	Дублирование определения $f \equiv L(h) \Rightarrow f \equiv L(h), f \equiv L(h)$
(eq-symm)	Симметричность равенства $f \equiv g \Rightarrow g \equiv f$
(eq-refl)	Рефлексивность равенства $f \equiv f$
(arity-reduction)	Редукция арности $f \equiv \theta L(h) \overset{f \to \theta g}{\Leftrightarrow} g \equiv L(h)$
(call-to-permutation)	Преобразование **Call** в **Id** $f \equiv \mathbf{Call}(h, v_1, v_2) \Leftrightarrow$ $f \equiv h$
(commutativity)	Коммутативность $h \equiv L(f), f \equiv \theta f \Rightarrow$ $h \equiv L(\theta f)$
(inj-cons)	Инъективность **Cons** $f \equiv \mathbf{Cons}_C(g), f \equiv \mathbf{Cons}_C(h) \Rightarrow$ $g \equiv h$
(inj-case)	Инъективность **Case** $f \equiv \mathbf{Case}_C(v, g), f \equiv \mathbf{Case}_C(v, h) \Rightarrow$ $g \equiv h$
(red-call-var)	Редукция **Call-Var** $f \equiv \mathbf{Call}(v, g) \Leftrightarrow$ $f \equiv g$
(red-case-cons)	Редукция **Case-Cons** $f \equiv \mathbf{Case}_C(c, h), c \equiv \mathbf{Cons}_C(g) \Rightarrow$ $f \equiv \mathbf{Call}(h, c)$
(red-case-cons-bot)	Незнакомый конструктор $f \equiv \mathbf{Case}_C(c, h), c \equiv \mathbf{Cons}_D(g) \Rightarrow$ $f \equiv \mathbf{Cons}_\perp$

Из-за некоторых особенностей нашей теории преобразования полипрограмм данное правило не производит само слияние функций f и g, а только добавляет определение, обозначающее равенство между ними. Слияние функций реализуется конгруэнтностью, которое не может быть сформулировано в виде обычного правила.

Корректность Корректность данного правила требуется доказать только в одну сторону благодаря его недеструктивности. Здесь и далее будем использовать запись $\alpha(f)$ внутри формул вместо более строгой записи "θg, где $(\theta, g) = \alpha(f)$", а также $[\![p]\!]$ вместо $[\![p]\!]_{id}$.

Пусть $\mu : L \to \mathbb{S}$ — семантический морфизм, тогда $\eta : R \to \mathbb{S}$ строится так, что $funmap(\eta) = funmap(\mu)$, а $defmap(\eta)$ совпадает с $defmap(\mu)$ на всех исходных определениях, а на новом $defmap(\eta)(f \equiv \mathbf{Id}(g)) = (\eta(f) \equiv \mathbf{Id}(\eta(g)))$.

Остаётся доказать, что $(\eta(f) \equiv \mathbf{Id}(\eta(g))) \in defs(\mathbb{S})$. Т.к. выполняется $[\![\mu(f) \equiv \mathbf{L}(\overline{\mu(h_i)})]\!]$ и $[\![\mu(g) \equiv \mathbf{L}(\overline{\mu(h_i)})]\!]$, то по Рис. 3.5 выполнено $(\!|\mu(f)|\!) = (\!|\mathbf{L}(\overline{\mu(h_i)})|\!) = (\!|\mu(g)|\!)$, а значит верно $[\![\mu(f) = \mathbf{Id}(\mu(g))]\!]$, что то же самое, что $[\![\eta(f) = \mathbf{Id}(\eta(g))]\!]$, а значит $(\eta(f) \equiv \mathbf{Id}(\eta(g))) \in defs(\mathbb{S})$ по определению семантической полипрограммы $\mathbb{S}$.

Сопоставление с левой частью

Утверждение 24. Пусть в некоторой полипрограмме H был найден фрагмент G, являющийся кандидатом на соответствие левой части правила L, причём G имеет следующий вид (оба определения записаны в канонической форме, также для простоты мы используем те же самые имена функций, что и в образце правила, но на самом деле они отличаются, и у них может быть другая арность):

$$\xi f \equiv id_{m \to M}\mathbf{L}(\overline{\theta_i h_i})$$
$$\zeta g \equiv id_{m' \to M'}\mathbf{L}(\overline{\theta'_i h_i})$$

Причём оба определения могут быть одним и тем же определением и некоторые функциональные символы также могут быть одними и теми же. Тогда морфизм $\phi : L \to G$, отображающий соответствующие определения в соответствующие определения и соответствующие функциональные символы в соответствующие функциональные символы, существует тогда и только то-

гда, когда правые части канонических форм определений полипрограммы G без внешней перестановки $id_{...}$ (т.е. $\mathbf{L}(\overline{\theta_i h_i})$ и $\mathbf{L}(\overline{\theta'_i h_i})$) совпадают. Более того, если G находится в полностью канонической форме, то результат применения правила не зависит от того, каким именно образом ϕ дополнен перестановками параметров (а значит достаточно найти только одно такое дополнение).

Доказательство. Если морфизм ϕ существует, то правые части без внешней перестановки канонических форм $\phi(f \equiv \mathbf{L}(\overline{h_i}))$ и $\phi(g \equiv \mathbf{L}(\overline{h_i}))$ совпадают по определению канонической формы, т.к. правые части исходных определений совпадают (а новая правая часть без внешней перестановки зависит только от исходной правой части).

В обратную сторону, пусть $\theta'_i = \theta_i$. Тогда доопределим ϕ так:

$$\phi(f) = id_{M \to ...} \xi f$$
$$\phi(g) = id_{M' \to ...} \zeta g$$
$$\phi(h_i) = id_{m \to ...} \theta_i h_i$$

При применении ϕ к L и приведении к каноническому виду канонические вложения $id_{...}$ уйдут и останется в точности G. (Здесь важную роль играет то, что канонические вложения остаются каноническими вложениями после применения операции *shift*.)

Теперь осталось доказать, что результат применения данного правила не зависит от того, каким образом морфизм ϕ дополнен конкретными перестановками параметров. Здесь будет важно, что полипрограмма находится в полностью канонической форме (с полностью редуцированными арностями функций). Тогда $m = m' = M = M'$. Пусть существует другой морфизм ϕ', который отличается от ϕ только перестановками параметров. Введём следующие обозначения:

$$\phi'(f) = \delta_f \xi f$$
$$\phi'(g) = \delta_g \zeta g$$
$$decompose(\phi'(\mathbf{L}(\overline{h_i}))) = \tau t$$
$$bisplit(\tau, \delta_f \xi) = (\delta_f, id_m, \xi)$$
$$bisplit(\tau, \delta_g \zeta) = (\delta_g, id_m, \zeta)$$

Вид $\phi'(f)$ и $\phi'(g)$ следует из последних двух строчек, которые в свою очередь следуют из того, что определения из L после преобразования морфизмом ϕ' переводятся в определения, эквивалентные определениям из G. Также выполнено равенство $\tau = \delta_f id_m = \delta_g id_m$, откуда следует, что $\delta_f = \delta_g$. Добавляемое определение будет иметь вид $\delta_f \xi f \equiv \mathbf{Id}(\delta_g \zeta g)$. Требуется доказать, что оно эквивалентно $\xi f \equiv \mathbf{Id}(\zeta g)$, что следует из того, что $\delta_f = \delta_g$ и оба $\delta_{\dots}$ сразу уничтожаются по свойствам *bisplit*.

□

В Примере 15 из Приложения А демонстрируется применение правила (trans).

4.1.2 Удаление дубликатов

Правило 2 (remove-dups)**.**

$$\left\{ \begin{array}{l} f_0 \equiv \mathbf{L}(\overline{f_i}) \\ f_0 \equiv \mathbf{L}(\overline{f_i}) \end{array} \right\} \mapsto \left\{ f_0 \equiv \mathbf{L}(\overline{f_i}) \right\}$$

Данное правило удобно считать не удаляющим определение, а отождествляющим два одинаковых определения — это позволяет рассматривать его как недеструктивное.

Параметры: $\mathbf{L}$.

Данное правило позволяет очищать полипрограмму от дублирования. С его корректностью не возникает сложностей, несмотря на то, что оно склеивает определения, потому что в семантических полипрограммах отсутствует дублирование определений, т.е. семантический морфизм $\mu : L \to \mathbb{S}$ обязан отображать оба определения в определения с одной и той же формулой, а значит эти определения-образы одинаковые.

4.1.3 Дублирование определения

Правило 3 (clone)**.**

$$\left\{ f_0 \equiv \mathbf{L}(\overline{f_i}) \right\} \mapsto \left\{ \begin{array}{l} f_0 \equiv \mathbf{L}(\overline{f_i}) \\ f_0 \equiv \mathbf{L}(\overline{f_i}) \end{array} \right\}$$

Параметры: **L**.

Это правило обратное предыдущему. Отметим тот факт, что при помощи совместного использования правил (commutativity) и (eq-refl) можно добиться того же эффекта.

4.1.4 Симметричность равенства

Правило 4 (eq-symm)**.**

$$\left\{ f \equiv \mathbf{Id}(g) \right\} \mapsto \left\{ \begin{array}{l} g \equiv \mathbf{Id}(f) \\ f \equiv \mathbf{Id}(g) \end{array} \right\}$$

Это правило добавляет определение, означающее то же равенство, записанное наоборот. Оно нужно главным образом для гарантированного понижения арности правой части.

4.1.5 Рефлексивность равенства

Правило 5 (eq-refl)**.**

$$\left\{ f \right\} \mapsto \left\{ f \equiv \mathbf{Id}(f) \right\}$$

(Левая полипрограмма содержит только одну функцию и ни одного определения).

В большинстве случаев тривиальное равенство $f \equiv \mathbf{Id}(f)$ может быть добавлено другими правилами, например, транзитивностью, но это правило нужно для случая, когда функция не имеет определений.

4.1.6 Понижение арности

Правило 6 (arity-reduction)**.**

$$\left\{ id_N f \equiv id_{m \to N} \mathbf{L}(\overline{h_i}) \right\} \mapsto \left\{ id_m g \equiv id_m \mathbf{L}(\overline{h_i}) \right\}$$

где $f : N$, и f отображается в $id_{m \to N} g$, поэтому правило неразрушающее.
Параметры: **L**, m.

Понижение арности позволяет удалять из функций фиктивные переменные.

Корректность Пусть μ — семантический морфизм и выполнено

$$[\![\mu(f) \equiv id_{m\to N}\mathbf{L}(\overline{\mu(h_i)})]\!]$$

Тогда существует функция $g' \in \mathbb{S}$ такая, что $[\![g' \equiv id_m\mathbf{L}(\overline{\mu(h_i)})]\!]$, а значит также $\mu(f) = id_{m\to N}g'$. Построим семантический морфизм $\eta : R \to \mathbb{S}$ так, что $\eta(g) = g'$, а $\eta(h_i) = \mu(h_i)$. Определение R отобразится в определение $g' \equiv id_m\mathbf{L}(\overline{\mu(h_i)})$, присутствующее в $\mathbb{S}$ по построению g'. Пусть $\psi : L \to R$ — морфизм правила, тогда нетрудно проверить, что $\eta\psi = \mu$. Действительно, $\eta(\psi(f)) = \eta(id_{m\to N}g) = id_{m\to N}\eta(g) = id_{m\to N}g' = \mu(f)$. Совпадение остальных компонентов морфизмов тривиально.

В обратную сторону аналогично.

Сопоставление с левой частью Пусть в полипрограмме G кандидат на соответствие L выглядит так (в канонической форме):

$$\xi f \equiv id_{m\to n}\mathbf{L}(\overline{\zeta_i h_i})$$

Тогда всегда найдётся морфизм $L \to G$ для правила с параметром m — действительно, надо просто взять $\{f \to id_{n\to N}\xi f, h_i \to \zeta_i h_i\}$). Таким образом, хотя бы одно правило из семейства применимо всегда. Более того, после применения правила указанным способом f отобразится в $\sigma f'$, причём будет выполнено $\xi\sigma = id_{m\to n}\sigma'$ для некоторого σ' (т.к. σ получается построением кодекартова квадрата в категории морфизмов полипрограмм, или, соответственно построением декартового квадрата в категории перестановок параметров). Это значит, что определение будет иметь следующий вид:

$$id_{m\to n}\sigma' f' \equiv id_{m\to n}\mathbf{L}(\overline{\zeta'_i h'_i})$$

Здесь $\zeta'_i h'_i$ либо останутся $\zeta_i h_i$, либо будут превращены в $\zeta_i\sigma f'$, если $h_i = f$ для некоторого i. Если среди h_i нет f, то после приведения к каноническому виду мы получим:

$$\sigma' f' \equiv id_m\mathbf{L}(\overline{\zeta_i h_i})$$

Это значит, что определение будет полностью каноническим. В случае, если среди h_i есть f, это может быть не так (см. пример ниже). Тем не менее, если $m < n$, то арности обеих частей будут понижены, а значит всегда можно применить правило понижения арности так, чтобы привести определение к полностью каноническому виду за конечное число шагов.

Пусть теперь определение находится в полностью каноническом виде:

$$\xi f \equiv id_n \mathbf{L}(\overline{\zeta_i h_i})$$

Пусть выбран какой-то параметр m и какой-то морфизм $\phi : L \to G$, причём $\phi(f) = \chi f$, и выполняется следующее:

$$\begin{aligned}(\xi f \equiv id_n \mathbf{L}(\overline{\zeta_i h_i})) \approx \\ \approx (\chi f \equiv id_{m \to N} \mathbf{L}(\overline{\phi(h_i)}))\end{aligned}$$

Т.е. $\chi = \delta\xi$ для некоторого δ, и $\mathit{decompose}(\mathbf{L}(\overline{\phi(h_i)})) = (\tau, \mathbf{L}(\overline{\zeta_i h_i}))$, $\tau : n \to m$, причём $\delta = id_{m \to N}\tau$. Значит $\chi = id_{m \to N}\tau\xi$. Тогда после преобразования f отобразится в $\sigma f'$, причём $\chi\sigma = id_{m \to N}\sigma'$ — декартов квадрат (здесь важно, что это будет декартовым квадратом, даже если некоторые h_i отображаются в f морфизмом ϕ, что следует из вида нашего правила). Это эквивалентно тому, что $(id_{m \to N}\tau\xi)\sigma = id_{m \to N}\sigma'$ — декартов квадрат, откуда следует, что $(\tau\xi)\sigma = id\sigma'$ — декартов квадрат, а значит $\sigma = id_{\mathit{arity}(f)}$ или любая другая биекция над $\mathit{arity}(f)$ — в любом случае арность f понижена не будет, а значит результирующая полипрограмма будет изоморфна исходной. Таким образом, применение данного правила к определениям в полностью канонической форме бесполезно. В сочетании с тем, что правило можно применять так, чтобы привести определение к полностью каноническому виду, и с тем, что данное правило недеструктивно, получаем, что его достаточно применять только указанным выше способом.

В Примере 16 из Приложения А демонстрируется применение правила (arity-reduction), причём приведение к полностью каноническому виду происходит не за одно, а за два применения.

4.1.7 Преобразование вызова в перестановку параметров

Правило 7 (call-to-permutation)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{\{1 \mapsto i\}v}) \\ v \equiv \mathbf{Var} \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Id}(g) \\ v \equiv \mathbf{Var} \end{array}\right\}$$

Причём $f, g : N$, $v : 1$.

Данное правило преобразует вызов, в котором в качестве аргументов используются разные переменные, в перестановку параметров, что сильно упрощает полипрограмму. Корректность правила следует из определений семантики вызова и перестановки параметров.

Взаимодействие правила с самим собой при морфизмах, отличающихся перестановками параметров мы рассматривать не будем.

Отметим, что v в левой части имеет арность 1, что не уменьшает общность правила, т.к. если в полипрограмме содержится определение вида $\xi u \equiv id_{1\to n}\mathbf{Var}$, то к нему можно применить понижение арности и уменьшить арность u до 1.

Применение данного правила продемонстрировано в Примере 17.

4.1.8 Насыщение по коммутативности

Правило 8 (commutativity)**.**

$$\left\{\begin{array}{l} P(\overline{h_i}, f, \overline{h'_i}) \\ \theta f \equiv \mathbf{Id}(f) \end{array}\right\} \mapsto \left\{\begin{array}{l} P(\overline{h_i}, f, \overline{h'_i}) \\ P(\overline{h_i}, \theta f, \overline{h'_i}) \\ \theta f \equiv \mathbf{Id}(f) \end{array}\right\}$$

Параметры: $P, \theta : N \hookrightarrow N$. $P(\overline{h_i}, f, \overline{h'_i})$ образует определение, в котором f может быть как слева, так и справа от $\equiv$ и входит в него ровно один раз.

Это недеструктивная версия конгруэнтности, которая работает только для функций, эквивалентных самим себе с точностью до нетривиальной перестановки параметров (т.е. для коммутативных функций). Данное правило просто добавляет альтернативные версии инцидентных определений. С прак-

тической точки зрения это может привести к сильному разрастанию полипрограммы, однако в нашем случае коммутативные функции достаточно редки для этого.

Корректность следует из монотонности (конгруэнтности) семантического равенства.

Сопоставление с левой частью Пусть фрагмент-кандидат имеет следующий вид (в полностью канонической форме):

$$P'(\ldots, f', \ldots)$$
$$\xi f' \equiv id_n \mathbf{Id}(id_n f')$$

Пусть последнее определение должно соответствовать определению $\theta f \equiv \mathbf{Id}(f)$ из L. Пусть $\phi(f) = \zeta f'$. Тогда

$$(\theta\zeta f' \equiv \zeta \mathbf{Id}(id_n f')) \approx (\xi f' \equiv id_n \mathbf{Id}(id_n f'))$$

Это выполняется тогда и только тогда, когда $\zeta\xi = \theta\zeta$. Значит хотя бы один морфизм всегда можно найти, положив $\zeta = id_{n\to N}$ — тогда $\zeta\xi = \mathit{lift}(\zeta\xi) \circ id_{n\to N} = \theta \circ id_{n\to N}$, и поэтому можно взять $\theta = \mathit{lift}(id_{n\to N}\xi)$.

Теперь рассмотрим произвольные подходящие ζ и θ. Добавляемое определение будет иметь вид

$$P(\ldots, \theta\zeta f', \ldots) = P(\ldots, \zeta\xi f', \ldots)$$

Т.к. $P'(\ldots, f', \ldots) \approx P(\ldots, \zeta f', \ldots)$, то новое определение будет иметь вид $P(\ldots, \zeta(\xi f'), \ldots) \approx P'(\ldots, \xi f, \ldots)$, а значит результат не зависит ни от θ, ни от ζ.

Пример Рассмотрим следующую полипрограмму:

$$\xi f \equiv id_n \mathbf{Id}(id_n f)$$

Рассмотрим случай, когда $\phi : L \to G$ отображает оба определения в одно и то же определение. Т.к. f входит в это определение дважды, то можно рассмотреть как $P(f, h) = (\xi f \equiv id_n \mathbf{Id}(id_n h))$, так и $P(h, f)$. В первом случае

в результате применения правила будет добавлено следующее определение:

$$\xi^2 f \equiv id_n \mathbf{Id}(id_n f)$$

Во втором случае получается следующая тавтология:

$$id_n f \equiv id_n \mathbf{Id}(id_n f)$$

При этом оба применения правила открывает возможность для новых применений правила, поэтому в результате применения его до нормальной формы (с точностью до удаления дубликатов), получится полипрограмма следующего вида:

$$\begin{aligned}
&\dots \\
&\xi^{-2} f \equiv id_n \mathbf{Id}(id_n f) \\
&\xi^{-1} f \equiv id_n \mathbf{Id}(id_n f) \\
&id_n f \equiv id_n \mathbf{Id}(id_n f) \\
&\xi f \equiv id_n \mathbf{Id}(id_n f) \\
&\xi^2 f \equiv id_n \mathbf{Id}(id_n f) \\
&\dots
\end{aligned}$$

Полипрограмма будет конечная, т.к. перестановка ξ конечная, и всего существует конечное число таких определений в канонической форме.

Т.к. правило приводит к разрастанию полипрограммы, лучшем решением было бы вместо этого правила использовать сопоставление с точностью до перестановке параметров коммутативных функций. Это возможно, но оставлено в качестве возможного направления будущей работы. Отметим ещё раз, что существенно коммутативные функции встречаются на практике не очень часто.

4.1.9 Инъективность конструкторов

Правило 9 (inj-cons)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Cons}_C(\overline{h_i}) \\ f \equiv \mathbf{Cons}_C(\overline{g_i}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Cons}_C(\overline{h_i}) \\ f \equiv \mathbf{Cons}_C(\overline{g_i}) \\ \overline{h_i \equiv \mathbf{Id}(g_i)} \end{array}\right\}$$

Перестановки параметров в L тождественны кроме случая $arity(C) = 0$.
Параметры: C.

Данное преобразование аналогично транзитивности, но в обратную сторону. Его корректность следует из инъективности конструкторов.

Сопоставление с левой частью Пусть дан фрагмент-кандидат, и нужно найти перестановки параметров соответствующего морфизма ϕ:

$$\xi_1 f \equiv \mathbf{Cons}_C(\overline{\theta'_i h_i})$$
$$\xi_2 f \equiv \mathbf{Cons}_C(\overline{\zeta'_i g_i})$$

Перенесём все перестановки параметров вправо (это можно сделать при помощи *bisplit* и *push*) и получим следующую эквивалентную полипрограмму:

$$id_{n\to N} f \equiv \mathbf{Cons}_C(\overline{\theta_i h_i})$$
$$id_{n\to N} f \equiv \mathbf{Cons}_C(\overline{\zeta_i g_i})$$

Отсюда сразу видно, что морфизм можно дополнить перестановками параметров в любом случае (единственное условие — арности функций L больше арностей соответствующих функций полипрограммы-кандидата — выполняется по договорённости из начала главы).

Что касается независимости от выбора перестановок параметров, здесь возникает та же проблема, что и для правила (trans), однако она не решается применением правила (arity-reduction), потому что уравниваемые функции стоят справа. Поэтому сначала рассмотрим случай, когда ξ_1 и ξ_2 являются биекциями.

Пусть ϕ таков, что

$$\phi(f \equiv \mathbf{Cons}_C(\overline{h_i})) \approx (\xi_1 f \equiv \mathbf{Cons}_C(\overline{\theta'_i h_i}))$$
$$\phi(f \equiv \mathbf{Cons}_C(\overline{g_i})) \approx (\xi_2 f \equiv \mathbf{Cons}_C(\overline{\zeta'_i g_i}))$$

Тогда существуют некоторые перестановки параметров δ_1 и δ_2, что:

$$\phi(f) = \delta_1 \xi_1 f = \delta_2 \xi_2 f$$
$$\phi(h_i) = \delta_1 \theta'_i h_i$$
$$\phi(g_i) = \delta_2 \zeta'_i g_i$$

Т.к. в левой части правила два определения, имеющие общую функцию, то δ_1 и δ_2 связаны соотношением $\delta_1 \xi_1 = \delta_2 \xi_2$. Добавляемые определения будут иметь вид

$$\delta_1 \theta'_i h_i \equiv \mathbf{Id}(\delta_2 \zeta'_i g_i)$$

Т.к. мы считаем, что ξ_1 и ξ_2 — биекции, это определение равно такому:

$$\delta_1 \xi_1 \xi_1^{-1} \theta'_i h_i \equiv \mathbf{Id}(\delta_2 \xi_2 \xi_2^{-1} \zeta'_i g_i)$$

А т.к. $\delta_1 \xi_1 = \delta_2 \xi_2$, то эта часть уничтожается и определение становится эквивалентно такому, не зависящему от параметров δ_1 и δ_2:

$$\xi_1^{-1} \theta'_i h_i \equiv \mathbf{Id}(\xi_2^{-1} \zeta'_i g_i)$$

Остаётся доказать, что определения вида $\xi_1 f \equiv id_{m \to n} \mathbf{Cons}_C(\overline{\theta'_i h_i})$ всегда можно преобразовать так, чтобы ξ_1 стала биекцией. Приведём определение к виду

$$id_{n \to N} f \equiv id_N \mathbf{Cons}_C(\overline{\theta_i h_i})$$

Теперь возьмём две такие перестановки параметров $\tau, \tau' : N \to N$, чтобы для всех $x \leq n$ было выполнено $\tau(x) = \tau'(x) = x$, а для всех $x > n$ было выполнено $N \geq \tau'(x) > M_1 \geq \tau(x) > n$ (будем считать, что $N > 3n$). Тогда можно построить морфизм $\phi : L \to G$, отображающий оба определения левой части нашего правила L в одно рассматриваемое определение, и действующий

на функции так:

$$\phi(f) = \tau id_{n\to N} f = \tau' id_{n\to N} f = id_{n\to N} f$$
$$\phi(h_i) = \tau' \theta_i h_i$$
$$\phi(g_i) = \tau \theta_i h_i$$

Действительно, при этих условиях ϕ будет морфизмом, т.к. τ и τ' сокращаются и получаются в точности последнее определение. Новые определения будут иметь следующий вид:

$$\tau' \theta_i h_i \equiv \mathbf{Id}(\tau \theta_i h_i)$$

Все такие определения следует привести к полностью канонической форме по правилу понижения арности. Это приведёт к тому, что арность h_i будет понижена по крайней мере до n (это арность f). Затем следует применить понижение арности к исходному определению и повторять всю процедуру до тех пор, пока арность правой части не станет равна арности f (это рано или поздно произойдёт, т.к. арность нельзя уменьшать бесконечно). Тогда после приведения определения к канонической форме перестановка параметров при функции в левой части станет биекцией.

4.1.10 Инъективность сопоставления с образцом

Правило 10 (inj-case)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(v, \ldots, h_{i-1}, \sigma h_i, h_{i+1}, \ldots) \\ f \equiv \mathbf{Case}_{\overline{C_i}}(v, \ldots, g_{i-1}, \sigma g_i, g_{i+1}, \ldots) \\ v \equiv id_1 \mathbf{Var} \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(v, \ldots) \\ f \equiv \mathbf{Case}_{\overline{C_i}}(v, \ldots) \\ v \equiv id_1 \mathbf{Var} \\ h_i \equiv \mathbf{Id}(g_i) \end{array}\right\}$$

Где $v : 1\ f : N$, $h_i, g_i : k_i + N - 1$, $\sigma : k_i + N - 1 \to k_i + N$, $\sigma(j) = j$, если $j \leq k_i$ и $\sigma(j) = j + 1$, если $j > k_i$, где $k_i = arity(C_i)$. Таким образом h_i и g_i не смогут использовать разбираемую переменную (под номером 1).
Параметры: $\overline{C_i}, i$.

Это правило аналогично предыдущему, но только действует на сопоставления с образцом. Корректность этого правила следует из определения се-

мантики сопоставлений с образцом. Т.к. данное правило нетривиально и при этом обладает теми же проблемами, что и предыдущее, мы не будем доказывать, что применение его с различными морфизмами ведёт к одному и тому же результату.

Пример Рассмотрим следующую полипрограмму:

$$f(x,y) \equiv \textbf{case}\ v(x)\ \textbf{of}\{A(z) \to g_1(z,y); B(z) \to h_1(z,x,y)\}$$
$$f(x,y) \equiv \textbf{case}\ v(x)\ \textbf{of}\{A(z) \to g_2(z,y); B(z) \to h_2(z,x,y)\}$$
$$v(x) \equiv x$$

Применяя правило для ветви A, получим следующее новое определение:

$$g_1(z,y) \equiv g_2(z,y)$$

(Действительно, они оба равны $f(A(z),y)$). Однако к ветви B данное правило неприменимо, потому что h использует переменную x и нельзя сказать, что $h_1(z,x,y) = h_2(z,x,y)$, потому что возможно, что они равны только в случае, когда $x = B(z)$. Однако можно применить правило распространения позитивной информации (propagation-case-var) (не являющееся упрощающим), получив такие новые определения:

$$f(x,y) \equiv \textbf{case}\ v(x)\ \textbf{of}\{A(z) \to g_1(z,y); B(z) \to h'_1(z,y)\}$$
$$f(x,y) \equiv \textbf{case}\ v(x)\ \textbf{of}\{A(z) \to g_2(z,y); B(z) \to h'_2(z,y)\}$$
$$h'_1(z,y) = h_1(z,B(z),y)$$
$$h'_2(z,y) = h_2(z,B(z),y)$$

И тогда h'_1 и h'_2 уже будут равны.

4.1.11 Редукция вызова тождественной функции

Правило 11 (red-call-var)**.**

$$\left\{\begin{array}{l} f \equiv \textbf{Call}(v,h) \\ v \equiv \textbf{Var} \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \textbf{Id}(h) \\ v \equiv \textbf{Var} \end{array}\right\}$$

Данное правило фактически означает, что $id(x) = x$. Его полезно применять деструктивно для небольшого упрощения полипрограммы.

4.1.12 Редукция сопоставления с образцом

Правило 12 (red-case-cons)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(c, \overline{h_i}) \\ c \equiv \mathbf{Cons}_{C_k}(\overline{g_j}) \\ v \equiv \mathbf{Var} \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Call}(h_k, \overline{g_j}, \overline{\{1 \mapsto l\}v}) \\ c \equiv \mathbf{Cons}_{C_k}(\overline{g_j}) \\ v \equiv \mathbf{Var} \end{array}\right\}$$

При этом l пробегает значения $1 \ldots N$, а значит $arity(h_k) = arity(C_k) + N$.
Параметры: $\overline{C_i}$, C_k.

Отметим, что данное правило требует, чтобы в полипрограмме уже содержалась функция, соответствующая **Var**. В большинстве случаев это так, в оставшихся надо добавить такую функцию в полипрограмму сразу после преобразования из исходной программы. Мы не можем просто так добавить новую функцию в правой части, потому что это усложнит доказательство завершаемости.

4.1.13 Сопоставление с образцом незнакомого конструктора

Правило 13 (red-case-cons-bot)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(c, \overline{h_i}) \\ c \equiv \mathbf{Cons}_{C}(\overline{g_j}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv id_{0 \to N}\mathbf{Cons}_{\perp}() \\ c \equiv \mathbf{Cons}_{C}(\overline{g_j}) \end{array}\right\}$$

при этом $\forall i\ C \neq C_i$.
Параметры: $\overline{C_i}$, C.

Данное правило аналогично предыдущему, но работает в случае, когда разбираемый конструктор отсутствую в списке образцов.

4.2 Конгруэнтность

Ни одно из упрощающих правил не отождествляет функциональные символы, они лишь могут добавлять определения вида $f \equiv \mathbf{Id}(g)$, означающие, что функции эквивалентны и их в принципе можно отождествить. Связано это, как было отмечено в предыдущей главе, с тем, что в присутствии коммутативных функций правила, отождествляющие функции, не могут быть корректны. Поэтому мы вводим специальное преобразование, не являющееся правилом — конгруэнтность. Конгруэнтность заменяет все вхождения одного функционального символа в полипрограмме на другой функциональный символ, а старый функциональный символ удаляет. Удаление исходного функционального символа мы производим для удобства, это не влечёт проблем, потому что конгруэнтность не является обычным правилом в нашей системе, зато становится проще доказать завершаемость, потому что функциональный символ выводится из дальнейшего преобразования.

Далее будем записывать как $\{f \mapsto \theta g\}$ морфизм функциональных символов, отображающий f в (θ, g), а все остальные функциональные символы в себя с тождественной перестановкой.

Определение 34. Пусть G — полипрограмма, $\theta g \equiv \mathbf{Id}(f)$ — одно из её определений, $\theta : \mathit{arity}(g) \to \mathit{arity}(f)$. Тогда *конгруэнтностью* назовём преобразование этой полипрограммы в полипрограмму $\alpha(G)$, где $\alpha = \{f \mapsto \theta g\}$. Применение конгруэнтности будем обозначать $G \Rightarrow_{f=\theta g} G'$.

Мы применяем конгруэнтность вдоль определений вида $\theta g \equiv id_n\mathbf{Id}(f)$, однако можно её применять и для определений вида $\theta g \equiv id_{n\to m}\mathbf{Id}(f)$ (а к такому виду приводятся любые определения с меткой **Id**). Для этого надо сначала применить понижение арности, чтобы привести его к *полностью* канонической форме.

Конгруэнтность обладает важным свойством — завершаемость с точностью до изоморфизма. Действительно, применение конгруэнтности вдоль $\theta g \equiv \mathbf{Id}(f)$, $f \neq g$ приводит к уменьшению числа функций. Если же $f = g$, то конгруэнтность вдоль $\theta f \equiv \mathbf{Id}(f)$ превратит исходную полипрограмму в изоморфную, т.к. θ обязана быть биекцией, учитывая что определение находится в полностью канонической форме.

К сожалению, конгруэнтность не обладает конфлюэнтностью, т.е. результат нескольких последовательных применений конгруэнтности существенно

зависит от того, в каком порядке их производить. По всей видимости, если добавить правила (commutativity), (eq-symm) и (remove-dups), то полученная система становится конфлюэнтной, однако в настоящей диссертации мы не рассматриваем конфлюэнтность, поэтому данное заявление мы оставляем только как гипотезу.

Пример 10. Рассмотрим следующую полипрограмму $(f, g, h : 2, \theta = \{1 \to 2, 2 \to 1\}_{2\to 2})$:

$$h \equiv \mathbf{Cons}_C(f, g)$$
$$id_2 f \equiv \mathbf{Id}(g)$$
$$\theta g \equiv \mathbf{Id}(f)$$

Если применить конгруэнтность вдоль $id_2 f \equiv \mathbf{Id}(g)$, то получится

$$h \equiv \mathbf{Cons}_C(f, f)$$
$$id_2 f \equiv \mathbf{Id}(f)$$
$$\theta f \equiv \mathbf{Id}(f)$$

Если же сначала применить конгруэнтность вдоль $\theta f \equiv \mathbf{Id}(g)$, то получится

$$h \equiv \mathbf{Cons}_C(\theta g, g)$$
$$\theta g \equiv \mathbf{Id}(g)$$
$$\theta g \equiv \mathbf{Id}(\theta g) \;\approx\; g \equiv \mathbf{Id}(g)$$

Эти полипрограммы не являются изоморфными, что сразу видно из первых определений, если записать их в человеко-читаемом виде:

$$h(x, y) \equiv C(f(x, y), f(x, y))$$
$$h(x, y) \equiv C(g(y, x), g(x, y))$$

Привести их друг к другу дальнейшими применениями конгруэнтности (вдоль петель) также нельзя. Всё дело в том, что изоморфизм (а значит и конгруэнтность вдоль петли) меняет все вхождения функционального символа, а в данных полипрограммах различие только в одном вхождении. Решить проблему можно при помощи правила (commutativity), способного изменять вхожде-

ния по одному. Применив это правило к данным полипрограммам, можно получить следующие две полипрограммы:

$$h \equiv \mathbf{Cons}_C(f, f)$$
$$h \equiv \mathbf{Cons}_C(\theta f, f)$$
$$id_2 f \equiv \mathbf{Id}(f)$$
$$\theta f \equiv \mathbf{Id}(f)$$

И

$$h \equiv \mathbf{Cons}_C(\theta g, g)$$
$$h \equiv \mathbf{Cons}_C(\theta^2 g, g) \;\approx\; h \equiv \mathbf{Cons}_C(g, g)$$
$$\theta g \equiv \mathbf{Id}(g)$$
$$\theta g \equiv \mathbf{Id}(\theta g) \;\approx\; g \equiv \mathbf{Id}(g)$$

В результате получаем две изоморфные полипрограммы. △

К сожалению, правило (commutativity) несовместимо с завершаемостью. Действительно, пусть правило r удаляет определение $h(\theta f)$, тогда возможен такой цикл преобразований:

$$\{h(f); \theta f \equiv \mathbf{Id}(f)\} \Rightarrow_{(\text{commutativity})}$$
$$\{h(f); h(\theta f); \theta f \equiv \mathbf{Id}(f)\} \Rightarrow_r$$
$$\{h(f); \theta f \equiv \mathbf{Id}(f)\}$$

Здесь коммутативность успевает восстановить удалённое определение. Чтобы такого не происходило, надо либо запретить деструктивные преобразования, либо применять их сразу ко всем определениям, эквивалентным по коммутативности, чтобы после их удаления коммутативность не могла их восстановить. Фактически, последнее и является нашим решением, для реализации которого мы будем отделять преобразования конгруэнтности и коммутативности от остальных преобразований.

4.3 Завершаемость упрощающих правил

Здесь мы будет говорить исключительно о конечных полипрограммах, т.к. для бесконечных полипрограмм завершаемость выполняться в общем случае не может. Также мы предполагаем совместность полипрограммы (существование хотя бы одной модели), поскольку несовместные полипрограммы нам неинтересны, а некоторые свойства работают только в предположении совместности.

Определение 35. Отношение $\Rightarrow$ завершается, если для любой последовательности $A_1 \Rightarrow \ldots \Rightarrow A_i \Rightarrow \ldots$ существует N такое, что для любого $n > N$ выполняется $A_n = A_N$.

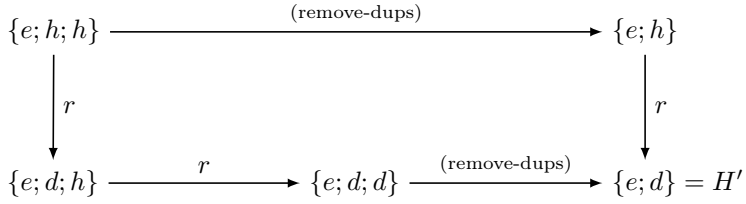
Говоря о завершаемости в применении к полипрограммам можно рассматривать классы эквивалентности полипрограмм, и соответственно распространять отношение $\Rightarrow$ с полипрограмм на классы так, что $A \Rightarrow B$ если существуют $G \in A$ и $H \in B$ такие, что $G \Rightarrow H$. Нам понадобится эквивалентность, основанная на удалении определений-дубликатов, потому что без удаления дубликатов правила, только добавляющие определения в полипрограмм, не могут быть завершающимися в принципе.

Определение 36. $G_1 \sim G_2$, если существует H такой, что $G_i \Rightarrow^*_{(\text{remove-dups})} H$ при $i = 1, 2$.

Отметим, что если G_1 и G_2 изоморфны, то по определению $G_1 \sim G_2$, т.к. применение правил происходит с точностью до изоморфизма. Данное отношение является отношением эквивалентности. Завершаемость остальных преобразований будем рассматривать с точностью до $\sim$.

Утверждение 25. Пусть правило r либо недеструктивно, либо удаляет не более одного определения и при этом не добавляет новые функции. Пусть $G \Rightarrow^*_{(\text{remove-dups})} H$ и $G \Rightarrow_r G'$. Тогда существует H' такой, что $H \Rightarrow_r H'$ и $G' \Rightarrow^*_{(\text{remove-dups}),r} H'$.

Доказательство. Для одного применения $\Rightarrow_{(\text{remove-dups})}$ единственный нетривиальный случай — когда определение h удаляется правилом r и правилом удаления дубликатов одновременно. Пусть e — некоторый контекст (мультимножество определений), а определения d добавляется правилом r, тогда схема преобразования G в H' представлена следующей диаграммой:



Для $\Rightarrow^*_{\text{(remove-dups)}}$ утверждение следует по индукции. □

Так же хорошо удаление дубликатов взаимодействует с конгруэнтностью.

Утверждение 26. Пусть $G \Rightarrow^*_{\text{(remove-dups)}} H$ и $G \Rightarrow_{f=\theta g} G'$. Тогда существует H' такой, что $H \Rightarrow_{f=\theta g} H'$ и $G' \Rightarrow^*_{\text{(remove-dups)}} H'$.

Теперь докажем, что некоторое подмножество упрощающих преобразований является завершающимся с точностью до удаления дубликатов. К сожалению, сюда не входит (commutativity), поэтому следующее утверждение имеет ограниченную применимость.

Утверждение 27. Любое объединение отношений из $\{\Rightarrow_{f=\theta g}\} \cup \{\Rightarrow_r \mid r \in \mathcal{R}\}$, где $\mathcal{R}$ — упрощающие правила кроме (commutativity), является завершающимся с точностью до удаления дубликатов.

Доказательство. Сначала отметим, что (remove-dups) и (clone) не изменяют полипрограмму с точностью до удаления дубликатов (т.е. они в этом смысле никогда не применимы).

Пусть существует бесконечная цепочка преобразований из условия. Т.к. мы считаем, что в полипрограммах конечное число функций, и ни одно преобразование не добавляет новые, то начиная с некоторого момента применения конгруэнтности перестанут удалять функции, а значит будут просто переводить полипрограммы в изоморфные им же, поэтому их можно выкинуть, а начальную часть цепочки преобразований, в которых применения конгруэнтности были существенными, можно обрезать. Таким образом мы получаем бесконечную цепочку преобразований без применений конгруэнтности.

Аналогичным образом выкинем из цепочки (trans), (eq-symm), (eq-refl), (inj-cons), (inj-case), которые добавляют только равенства вида $\xi f \equiv \theta \mathbf{Id}(g)$, т.к. таких равенств конечное количество для полипрограммы с конечным числом функций и неповышаемой арностью (ни одно упрощающее преобразование не повышает арность), и ни одно преобразование такие равенства не уда-

ляет. Также выкинем (arity-reduction), т.к. оно понижает арность, и ни одно преобразование арность не может повысить.

Остаются только правила (call-to-permutation), (red-call-var), (red-case-cons) и (red-case-cons-bot). Сопоставим полипрограмме кортеж (n, m), где n — количество различных определений типа **Case**, а m — количество различных определений типа **Call**. Тогда правила (call-to-permutation) и (red-call-var) уменьшают m и не изменяют n, правило (red-case-cons-bot) уменьшает n, а правило (red-case-cons) увеличивает m, но уменьшает n. Таким образом, все эти правила уменьшают (n, m) относительно лексикографического порядка, а значит не могут применяться до бесконечности, что доказывает завершаемость.

□

Мы будем разносить применения конгруэнтности с коммутативностью и остальные правила, применяя их поочерёдно до насыщения.

Утверждение 28. Пусть $\mathcal{R}$ — некоторое подмножество упрощающих правил, для которых выполнено последнее утверждение. Пусть $A \Downarrow_{\mathcal{R}} B$ означает, что B — нормальная форма A относительно $\Rightarrow_{\mathcal{R}}$ с точностью до удаления дубликатов. Пусть $A \downarrow B$ означает, что B — нормальная форма A относительно $(\Rightarrow_{f=\theta g} \cup \Rightarrow_{(\text{commutativity})})$, также с точностью до удаления дубликатов. Пусть $A \rightsquigarrow C$ означает, что существует B такая, что $A \Downarrow_{\mathcal{R}} B$ и $B \downarrow C$. Тогда отношение $\rightsquigarrow$ является завершающимся.

Доказательство. Аналогично предыдущему доказательству из предположительно бесконечной цепочки можно удалить все преобразования кроме правила (commutativity) и деструктивных правил (call-to-permutation), (red-call-var), (red-case-cons) и (red-case-cons-bot) (но включая конгруэнтность). Правило (commutativity) нельзя удалить, потому что деструктивные преобразования могут удалять определения, вводимые коммутативностью.

Пусть теперь в оставшейся цепочке $P \Downarrow_{\mathcal{R}} G \downarrow H \Rightarrow_r K$, где r — одно из перечисленных деструктивных преобразований, причём $G \Rightarrow^*_{(\text{commutativity})} H$. Мы докажем, что r в таком случае можно перенести обратно к G, причём r будет существенно применим к G, что противоречит тому, что G — нормальная форма относительно $\Rightarrow_{\mathcal{R}}$.

Пусть $G \Rightarrow_{(\text{commutativity})} H$, более общий случай будет следовать по индукции. Пусть по коммутативности происходит клонирование определения $P(f)$

вдоль $\theta f \equiv \mathbf{Id}(f)$. Если образ левой части r в H не содержит $P(\theta f)$, то r и коммутативность никак не взаимодействуют и их можно поменять местами, что означает, что утверждение выполнено. Пусть тогда левая часть r содержит прообраз $P(\theta f)$. Если левая часть r линейна по f, то это значит, что вместо θf можно использовать f (для этого требуется биективность θ, которая присутствует), а значит r можно перенести к G, в котором есть $P(f)$. Это доказывает утверждение для (call-to-permutation) и (red-call-var) и для входящих линейно функций двух правил (red-case-cons) или (red-case-cons-bot). Для нелинейно входящей функции c правил (red-case-cons) или (red-case-cons-bot) рассуждения аналогичны, поскольку при изменении только одного вхождения c в левые части этих правил правила всё равно остаются применимыми (только с другими морфизмами).

Таким образом, в оставшейся цепочке не может вышеуказанных комбинаций, и она имеет вид $P \Downarrow_{\mathcal{R}} G \downarrow H \Downarrow_{\mathcal{R}} H \downarrow H \ldots$ □

Данное утверждение говорит о том, как надо применять упрощающие правила: сначала все упрощающие правила кроме коммутативности и конгруэнтности до достижения нормальной формы, затем конгруэнтность и коммутативность до нормальной формы, потом повторить заново, и так далее пока применение правил возможно. Данный процесс гарантировано завершится.

4.4 Насыщающие правила

В этом разделе мы опишем остальные локальные правила преобразования, которые приводят к раздуванию полипрограммы, незавершаемости или просто недостаточно фундаментальны, чтобы быть включёнными в список упрощающих правил. Все правила из этого раздела применяются недеструктивно.

Все правила из этого раздела соответствуют прогонке из суперкомпиляции, однако мы не называем насыщающие правила правилами прогонки, потому что некоторые правила прогонки попали в набор упрощающих правил. Кроме того, в принципе насыщающие правила можно расширить и другими преобразованиями (например, соответствующими обобщению), потому что к ним не предъявляются никакие специальные требования.

В этом разделе мы будем использовать сокращение v_i для $\{1 \mapsto i\}v$.

4.4.1 Редукция вызова функции

В условиях преобразования полипрограмм удобнее работать с вызовами функций как с явными подстановками с соответствующими правилами редукции. Правила данного подраздела фактически говорят, как производить подстановку в тело функции, если оно начинается с различных конструкций. Все эти правила могут приводить к незавершаемости, поэтому не входят в список упрощающих. Среди упрощающих правил есть только простейшее из правил такого рода — (red-call-var).

4.4.1.1 Редукция вызова функции, состоящей из вызова функции

Правило 14 (red-call-call)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ g \equiv \mathbf{Call}(s, \overline{t_j}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ f \equiv \mathbf{Call}(s, \overline{k_j}) \\ \hline k_j \equiv \mathbf{Call}(t_j, \overline{h_i}) \\ g \equiv \mathbf{Call}(s, \overline{t_j}) \end{array}\right\}$$

4.4.1.2 Редукция вызова функции, состоящей из сопоставления с образцом

Правило 15 (red-call-case)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ g \equiv \mathbf{Case}_{\overline{C_j}}(s, \overline{t_j}) \\ v \equiv \mathbf{Var} \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ f \equiv \mathbf{Case}_{\overline{C_j}}(s', \overline{k_j}) \\ s' \equiv \mathbf{Call}(s, \overline{h_i}) \\ \hline k_j \equiv \mathbf{Call}(t_j, v_1 \ldots v_{arity(C_j)}, \overline{h_i}) \\ g \equiv \mathbf{Case}_{\overline{C_j}}(s, \overline{t_j}) \\ v \equiv \mathbf{Var} \end{array}\right\}$$

4.4.1.3 Редукция вызова функции, состоящей из конструктора

Правило 16 (red-call-cons)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ g \equiv \mathbf{Cons}_C(\overline{t_j}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f \equiv \mathbf{Call}(g, \overline{h_i}) \\ f \equiv \mathbf{Cons}_C(\overline{k_j}) \\ \hline k_j \equiv \mathbf{Call}(t_j, \overline{h_i}) \\ g \equiv \mathbf{Cons}_C(\overline{t_j}) \end{array}\right\}$$

4.4.2 Преобразования сопоставлений с образцом

Редукция сопоставления с образцом (red-case-cons) находится в упрощающих правилах. Здесь собраны более сложные правила, касающиеся сопоставления с образцом.

4.4.2.1 Распространение позитивной информации

Правило 17 (propagation-case-var)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(v, \overline{g_i}) \\ v \equiv \mathbf{Var} \end{array}\right\} \mapsto$$

$$\mapsto \left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(v, \overline{g_i}) \\ v \equiv \mathbf{Var} \\ f \equiv \mathbf{Case}_{\overline{C_i}}(v, \overline{h_i}) \\ \hline h_i \equiv \mathbf{Call}(g_i, v_1 \ldots v_{arity(C_i)}, c_i, v_{arity(C_i)+2}, \ldots) \\ \hline c_i \equiv \mathbf{Cons}_{C_i}(v_1 \ldots v_{arity(C_i)}) \end{array}\right\}$$

Это преобразование является очень важным в таких методах, как суперкомпиляция. Оно заменяет разбираемую переменную в ветвях сопоставления с образцом на её реконструкцию в терминах переменных образца, что позволяет в теле ветви использовать информацию о том, какой случай выполняется.

4.4.2.2 Сопоставление с образцом результата сопоставления с образцом

Правило 18 (distribute-case-case)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(g, \overline{h_i}) \\ g \equiv \mathbf{Case}_{\overline{D_j}}(s, \overline{t_j}) \\ v \equiv \mathbf{Var} \end{array}\right\} \mapsto$$

$$\mapsto \left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(g, \overline{h_i}) \\ g \equiv \mathbf{Case}_{\overline{D_j}}(s, \overline{t_j}) \\ v \equiv \mathbf{Var} \\ f \equiv \mathbf{Case}_{\overline{D_j}}(s, \overline{k_j}) \\ \overline{k_j \equiv \mathbf{Case}_{\overline{C_i}}(t_j, \overline{h'_{ji}})} \\ \overline{h'_{ji} \equiv \mathbf{Call}(h_i, v_1 \ldots v_{arity(C_i)}, v_{arity(C_i)+arity(D_j)+1} \ldots)} \end{array}\right\}$$

Это правило проталкивает внешнее сопоставление с образцом в ветви внутреннего.

4.4.2.3 Изменение порядка сопоставлений с образцом

Правило 19 (swap-case-case)**.**

$$\left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(g, \overline{h_i}) \\ \overline{h_i \equiv \mathbf{Case}_{\overline{D_j}}(\theta_i s, \overline{t_{ij}})} \end{array}\right\} \mapsto$$

$$\mapsto \left\{\begin{array}{l} f \equiv \mathbf{Case}_{\overline{C_i}}(g, \overline{h_i}) \\ \overline{h_i \equiv \mathbf{Case}_{\overline{D_j}}(\theta_i s, \overline{t_i j})} \\ f \equiv \mathbf{Case}_{\overline{D_j}}(s, \overline{k_j}) \\ \overline{k_j \equiv \mathbf{Case}_{\overline{C_i}}(\xi_j g, \overline{\sigma_{ij} t_{ij}})} \end{array}\right\}$$

где $\theta_i, \xi_j : N \hookrightarrow N$, $\theta_i(n) = arity(C_i) + n$, $\xi_j(n) = arity(D_j) + n$, таким образом s не может использовать переменные образца. В нашей реализации мы используем более ограниченный случай, когда $s = v = \mathbf{Var}$.

σ_{ij} определена следующим образом:

$$\sigma_{ij}(n) \equiv \begin{cases} arity(C_j) + n & \text{если } n \leq arity(D_j) \\ n - arity(D_j) & \text{если } arity(D_j) < n \leq arity(D_j) + arity(C_i) \\ n & \text{если } n > arity(D_j) + arity(C_i) \end{cases}$$

Это правило позволяет менять порядок двух последовательных сопоставлений с образцом (если между ними нет зависимости по данным). Отметим, что с его помощью можно доказывать коммутативность функций.

4.4.3 Применение насыщающих правил

В нашей реализации мы применяем насыщающие правила *недеструктивно* и *параллельно* (т.е. "в ширину"). Если у нас есть полипрограмма G, то мы находим все применения правил к этой полипрограмме $G \Rightarrow_{r_i,\varepsilon_i} G'_i$, после чего мы секвенциализируем эти применения по Утверждению 23, и получаем полипрограмму H такую, что $G \Rightarrow_{r_1,\varepsilon_1} G_1 \Rightarrow_{r_2,\varepsilon_2} \ldots \Rightarrow_{r_n,\varepsilon_n} H$. После этого мы применяем к H упрощающие правила до тех пор, пока это возможно, и получаем полипрограмму H', к которой опять применяем насыщающие преобразования, потом опять упрощающие и т.д.

4.5 Выводы

В данной главе были приведены правила, которые используются при преобразовании (насыщении) полипрограмм. Локальными они были названы потому, что для определения применимости правила используется лишь фрагмент преобразуемой полипрограммы — несколько соседних определений. Все эти преобразования могут быть сформулированы либо как обычные правила в форме, введённой в предыдущей главе, либо как семейства правил, немного отличающихся метками и количеством вовлечённых функций. Мы предпочитали формулировку именно в виде правил, а не семейств, для более удобной работы с ними. Это возможно в большинстве случаев благодаря встраиванию понятия перестановки параметров в ядро теории преобразования полипрограмм — без этого все правила должны были бы формулироваться как семейства правил, отличающихся перестановками параметров, и работа с пере-

становками параметров должна была бы происходить явно и более сложным образом. Отметим, что в нашей практической реализации по историческим причинам был использован именно второй подход, что привело к некоторому усложнению кода из-за необходимости явно учитывать перестановки в формулировке всех правил.

Правила были разделены нами на две группы — упрощающие правила и насыщающие правила. Упрощающие правила устроены так, что обладают завершаемостью при небольших ограничениях на порядок их применения. Вопрос о конфлюэнтности упрощающих правил оставлен в качестве работы на будущее, однако есть экспериментальные основания полагать, что она выполняется. Насыщающие правила такими свойствами не обладают, поэтому насыщающие правила применяются всегда недеструктивно (чтобы гарантировать детерминизм) и только на один слой (чтобы избежать незавершаемости).

Глава 5

Слияние по бисимуляции

При помощи правила (trans) с последующим применением конгруэнтности и удаления дубликатов можно добиться слияния двух функций, заданных одинаковыми *ациклическими* фрагментами полипрограммы (как в Примере 18 из Приложения А).

Однако если функции представлены фрагментами полипрограммы, имеющими циклы, эта процедура не работает. Простейший пример — два бесконечных числа:

$$f \equiv S(f)$$
$$g \equiv S(g)$$

Этот факт мотивирует поиск некоторого преобразования полипрограмм, которое бы могло сливать два произвольных изоморфных фрагмента полипрограммы. Оказывается, однако, что требование изоморфизма можно ослабить до отношения, которое мы назовём бисимуляцией полипрограмм (являющееся обобщением понятия бисимуляции для размеченных систем переходов). Поэтому будем называть это преобразование слиянием по бисимуляции. Отметим, что бисимуляция здесь понимается в синтаксическом смысле, как синтаксическая эквивалентность фрагментов полипрограммы с точностью до переименования функций и перестановки параметров, и не имеет прямого отношения к семантике (ни денотационной, ни операционной).

Однако не все изоморфные фрагменты полипрограммы корректно сли-

вать, например, следующие две функции входят в два изоморфных фрагмента (каждый состоит из одного определения):

$$f(x) \equiv f(f(x))$$
$$g(x) \equiv g(g(x))$$

Но эти фрагменты говорят только, что соответствующие функции являются идемпотентными, из чего не следует, что они должны быть равны. Для решения этой проблемы на фрагменты полипрограммы накладываются некоторые условия, подобные тем, которые применяются в тотальных языках для обеспечения завершаемости функций [1; 2], но в нашем случае обеспечивается не завершаемость, а единственность модели.

Слияние по бисимуляции является преобразованием, которое соответствует частному случаю применения доказательства по индукции или коиндукции для равенства двух функций. В некоторых случаях ему не хватает мощности по сравнению с полноценной индукцией, однако в большинстве случаев его достаточно для проведения индуктивных доказательств.

5.1 Понятие бисимуляции полипрограмм

Будем использовать следующее определение понятия бисимуляции полипрограмм:

Определение 37. Две полипрограммы G_1 и G_2 находятся в отношении *бисимуляции*, если существует полипрограмма H и два сюръективных (по функциональным символам и определениям) морфизма $\phi_i : H \to G_i$, $i = 1, 2$. Полипрограмму H вместе с морфизмами ϕ_i будем называть *бисимуляцией*. H будем называть полипрограммой бисимуляции.

Отметим, что если существует H и $\phi_i : H \to G_i$, то фрагменты $\phi_i(H) \subseteq G_i$ находятся в отношении бисимуляции, т.к. сужения ϕ_i на $\phi_i(H)$ являются сюръективными автоматически. H вместе с исходными морфизмами ϕ_i также будем называть бисимуляцией.

Замечание. Введённое таким образом понятие бисимуляции полипрограмм является обобщением классического понятия бисимуляции, и отличается тем,

что учитывает возможность множественности определений. Для случая монопрограмм данное понятие соответствует коалгебраическому определению бисимуляции.

Следующее утверждение говорит, что если в полипрограмме есть два фрагмента, находящихся в отношении бисимуляции и имеющих не более одной модели, то их соответствующие функции можно пометить как равные определениями вида **Id** (фактическое слияние будет проведено при помощи конгруэнтности и удаления дубликатов). Более того, если эти фрагменты имеют некоторые общие функции, то единственность модели требуется только для каждой интерпретации этих функций (т.е. для каждой интерпретации данных функций существует не более одной модели, совпадающей на данных функциях с этой интерпретацией) — это нужно для того, чтобы можно было использовать доказательство равенства по рефлексивности (соответствующий пример будет рассмотрен сразу после доказательства утверждения). Отметим, что здесь идёт речь именно о моделях, а не о семантических морфизмах.

Утверждение 29. Пусть H, G — полипрограммы, а $W \subseteq H$ — некоторый фрагмент полипрограммы H, состоящий только из функциональных символов (без определений). Пусть $\phi_1, \phi_2 : H \to G$, причём $\phi_1(w) = \phi_2(w)$ для любого функционального символа w из W. Пусть для любой интерпретации $\eta : \mathit{funs}(W) \to \mathit{funs}(\mathbb{S})$ существует не более одной модели $\mu : \mathit{funs}(H) \to \mathit{funs}(\mathbb{S})$ такой, что μ совпадает с η на функциональных символах из W (напомним, что $\mathbb{S}$ — семантическая полипрограмма, и $\mathit{funs}(\mathbb{S}) = \mathbb{D}$ в обозначениях Главы 2). Пусть полипрограмма G' состоит из полипрограммы G, к которой добавлены определения $\theta_1 f_1 \equiv \mathbf{Id}(\theta_2 f_2)$, где $(\theta_i, f_i) = \phi_i(f)$ для всех функциональных символов f из H, не входящих в W. Тогда G и G' являются семантически эквивалентными, т.е. для любого морфизма $\rho : G \to \mathbb{S}$ существует морфизм $\rho' : G' \to \mathbb{S}$ такой, что $\rho = \rho' \circ \subseteq_{G \to G'}$ (обратное выполняется автоматически, т.к. G — фрагмент G').

Доказательство. Введём операцию $[\_]$, отображающую семантические морфизмы в соответствующие модели, т.е. $[\phi](f) = \pi_1(\phi(f))\pi_2(\phi(f))$ (в этой записи перестановка параметров применяется к функции в семантическом смысле, а не к функциональному символу).

Пусть $\rho : G \to \mathbb{S}$. Тогда $\mu_1 = [\rho \circ \phi_1]$ и $\mu_2 = [\rho \circ \phi_2]$ — модели H. μ_1 и

μ_2 будут совпадать на функциях из W, т.к. $\phi_1(w) = \phi_2(w)$ для $w \in W$. А значит $\mu_1 = \mu_2 = \mu$ по свойству единственности модели для H при данной интерпретации W.

Пусть f — произвольная функция из H и $\phi_i(f) = (\theta_i, f_i)$. Пусть $\rho(f_i) = (\sigma_i, m_i)$. Тогда из того, что $\mu_1 = \mu_2$ следует, что $\theta_1\sigma_1 m_1 = \theta_2\sigma_2 m_2$.

Теперь естественным образом дополним ρ до ρ' — при этом определения $\theta_1 f_1 \equiv \mathbf{Id}(\theta_2 f_2)$ должны будут отображаться в $\theta_1\sigma_1 m_1 \equiv \mathbf{Id}(\theta_2\sigma_2 m_2)$, которые обязаны содержаться в $\mathbb{S}$, т.к. выполнено $\theta_1\sigma_1 m_1 = \theta_2\sigma_2 m_2$. Значит ρ' действительно будет семантическим морфизмом. При этом по построению будет выполнено $\rho = \rho' \circ \subseteq_{G \to G'}$. □

Пример 11. Рассмотрим следующую полипрограмму G:

$$f_1(x) \equiv C(g(x), h_1(x))$$
$$h_1(x) \equiv C(g(x), f_1(x))$$
$$f_2(x) \equiv C(g(x), f_2(x))$$

Обе функции f_1 и f_2 генерируют бесконечные списки вида $C(g(x), C(g(x), \ldots))$. Рассмотрим полипрограмму H:

$$f(x) \equiv C(g(x), h(x))$$
$$h(x) \equiv C(g(x), f(x))$$

Эта полипрограмма имеет много моделей, потому что на функцию g не накладываются никакие ограничения, однако для каждой интерпретации функции g она имеет единственную модель (для доказательства этого проще всего воспользоваться признаком единственности модели, который мы рассмотрим позже). Зададим морфизмы $\phi_{1,2} : H \to G$ как $\phi_i = \{f \to f_i, h \to h_i, g \to g\}$

$$\phi_1 = \{f \to f_1, h \to h_1, g \to g\}$$
$$\phi_2 = \{f \to f_2, h \to f_2, g \to g\}$$

При этом на определениях морфизмы заданы естественным образом так, чтобы быть морфизмами. Фактически H этими двумя морфизмами задают бисимуляцию между фрагментами полипрограммы G. Важно отметить, что h отображается в одну и ту же функцию для ϕ_2, но в разные для ϕ_1. Заметим

также, что g отображается в одну и ту же функцию (тоже названную g). Теперь по последнему утверждению можно добавить в G равенства соответствующих функций, после чего полипрограмма примет следующий вид:

$$f_1(x) \equiv C(g(x), h_1(x))$$
$$h_1(x) \equiv C(g(x), f_1(x))$$
$$f_2(x) \equiv C(g(x), f_2(x))$$
$$f_1(x) \equiv f_2(x)$$
$$h_1(x) \equiv f_2(x)$$

После применения конгруэнтности и удаления дубликатов останется следующая полипрограмма:

$$f_2(x) \equiv C(g(x), f_2(x))$$
$$f_2(x) \equiv f_2(x)$$

△

В Примере 19 из Приложения А показано, как слияние по бисимуляции приводит в конечном итоге к понижению арности.

5.2 Описание алгоритма слияния по бисимуляции

Утверждение 29 позволяет на основе бисимуляции полипрограмм вида $\phi_i : H \to G$, $i = 1, 2$ и факта единственности модели полипрограммы H вывести равенства соответствующих функций в G. Отсюда выходит следующий алгоритм слияния по бисимуляции:

1. Найти набор бисимуляций фрагментов полипрограммы.
2. Среди найденного набора бисимуляций найти бисимуляции, удовлетворяющие некоторому условию, обеспечивающему единственность модели соответствующей полипрограммы бисимуляции.
3. Для каждой найденной бисимуляции, удовлетворяющей условию, добавить определения, обозначающие эквивалентность соответствующих

функций (по Утверждению 29) и применить конгруэнтность.

В данном разделе мы рассмотрим первый пункт — поиск бисимуляций. Алгоритм, приведённый здесь, не является полным (находит не все бисимуляции, а только бисимуляции некоторого определённого вида), однако он достаточно хорошо работает на практике. Вопрос фильтрации бисимуляций для обеспечения единственности модели будет рассмотрен в следующем разделе.

Задача поиска бисимуляций осложняется наличием перестановок параметров: у соответствующих функций соответствующие параметры могут идти в разном порядке. Поэтому построение бисимуляций производится в два этапа: сначала производится построение кандидатов в бисимуляции (будем называть их пребисимуляциями), потом для каждой пребисимуляции производится попытка дополнить её перестановками параметров до бисимуляции — эта процедура не всегда завершается успешно, поэтому не все пребисимуляции становятся бисимуляциями.

Отметим, что хотя дополнение перестановками параметров происходит после построения пребисимуляции, т.к. для этого требуется полностью построенная пребисимуляция, для эффективности желательно заранее отсекать пребисимуляции, для которых дополнение перестановками параметров невозможно. То же самое касается и проверки условий единственности — их можно точно проверить только после построения бисимуляции, однако уже при построении желательно отсекать бисимуляции, для которых условия заведомо не могут выполняться. Данные приёмы реализованы в нашей экспериментальной системе доказательства эквивалентности, однако для ясности изложения мы не будем здесь их описывать, считая, что эти проверки чётко разнесены по разным этапам.

Теперь приступим к описанию алгоритма поиска пребисимуляций (дополнением до бисимуляций займёмся в следующем подразделе). Будем искать пребисимуляции в форме корневых деревьев с обратными дугами, каждая вершина соответствует паре функций и паре определений этих функций. Описываются они следующей структурой данных:

PreBisimulation ::=	$\mathbf{Cong}(f_1, f_2, q_1, q_2, \overline{b_i})$	конгруэнтность
	$\mathbf{Refl}(f)$	рефлексивность
	$\mathbf{Fold}(f_1, f_2)$	свёртка (обратная дуга)

Рис. 5.1 Алгоритм поиска пребисимуляций в виде деревьев

$$
\begin{aligned}
&\mathit{prebisimulations}(G, f, f, \mathit{history}) \\
&\qquad = [\mathbf{Refl}(f)] \\
&\mathit{prebisimulations}(G, f_1, f_2, \mathit{history}) \\
&\qquad \mid \text{если } (f_1, f_2) \in \mathit{history} \\
&\qquad = [\mathbf{Fold}(f_1, f_2)] \\
&\mathit{prebisimulations}(G, f_1, f_2, \mathit{history}) \\
&\qquad \mid \text{если } f_1 \text{ и } f_2 \text{ вместе несовместимы} \\
&\qquad = [] \\
&\mathit{prebisimulations}(G, f_1, f_2, \mathit{history}) \\
&\qquad \mid \text{если выполняется некоторая эвристика остановки} \\
&\qquad = [] \\
&\mathit{prebisimulations}(G, f_1, f_2, \mathit{history}) \\
&= [\mathbf{Cong}(f_1, f_2, q_1, q_2, \overline{b_i}) \mid \\
&\qquad q_1 = (\xi_1 f_1 \equiv L(\sigma_1 g_1, \ldots, \sigma_n g_n)) \in G, \\
&\qquad q_2 = (\xi_2 f_2 \equiv L(\zeta_1 h_1, \ldots, \zeta_n h_n)) \in G, \\
&\qquad \overline{bs_i = \mathit{prebisimulations}(G, g_i, h_i, (f_1, f_2) :: \mathit{history})}, \\
&\qquad \overline{b_i \in bs_i}]
\end{aligned}
$$

Буквами f обозначены функции исходной полипрограммы, буквами b — объекты типа **PreBisimulation**, q — определения исходной полипрограммы. Случай конгруэнтности означает, что две функции имеют два эквивалентных определения, случай рефлексивности означает, что мы пришли к одной и той же функции, случай свёртки означает, что мы столкнулись с парой функций, которая уже встречалась (в смысле дерева доказательства этот случай соответствует использованию гипотезы индукции).

На Рис. 5.1 показан алгоритм, строящий список таких деревьев. Он принимает на вход полипрограмму, два функциональных символа $f_1 : n_1$ и $f_2 : n_2$ этой полипрограммы и список пар вида (f, g) уже просмотренных функций (изначально пустой).

Данный алгоритм следует запускать для всех пар функций, для кото-

рых есть подозрение в их равенстве (или просто для всех пар функций). В результате для каждой пары он вернёт список пребисимуляций, представленных в виде дерева. Основная идея алгоритма состоит в том, чтобы обойти полипрограмму одновременно из двух функций, проходя по эквивалентным определениям. При этом если мы встречаем пару одинаковых функций, то мы прекращаем обход, возвращая узел **Refl**, т.к. функция равна себе (она попадёт в W). Если мы встречаем пару функций, которую мы уже встречали, то мы также прекращаем обход, возвращая узел **Fold**, — отметим, что теоретически можно было бы обход продолжить, что было бы аналогично развороту циклов, при этом обеспечилась бы "большая полнота", поскольку могут встречаться полипрограммы, для которых условие единственности модели не выполняется, но если в них развернуть некоторые циклы, то условие единственности выполняться будет.

Дальше если пара функций несовместима, то мы также прекращаем обход, но при этом возвращаем пустой список. Несовместимость функций означает, что они не могут быть равны ни в одной модели, по сути этот пункт алгоритма является поиском контрпримеров. В нашей практической реализации мы считаем две функции несовместимыми, если у них есть несовместимые определения, а именно:

- Различные конструкторы $f_1 \equiv \mathbf{Cons}_C(\ldots)$ и $f_2 \equiv \mathbf{Cons}_D(\ldots)$.
- Одинаковые конструкторы $f_{1,2} \equiv \mathbf{Cons}_C(\ldots, g_{1,2}, \ldots)$ и где g_1 и g_2 стоят в одной позиции и являются несовместимыми функциями.
- Конструктор и сопоставление переменной с образцом.
- Два сопоставления с образцом одной и той же переменной и полностью распространённой информацией (как для правила (inj-case)), причём существует пара соответствующих ветвей с несовместимыми функциями.
- Переменная и конструктор.
- Переменная и сопоставление переменной (любой) с образцом.

Мы также завершаем обход, если выполняется некоторая эвристика завершения, это нужно для сокращения перебора. В нашей реализации используется следующая эвристика: мы завершаем обход, если хоть одна функция

(левая или правая) уже встречалась ранее (для свёртки нужно, чтобы пара функций встречалась ранее, поэтому это более слабое условие). Таким образом мы опять же теряем "полноту", но повышаем эффективность на практике.

Наконец, если ни один из случаев завершения обхода не выполняется, алгоритм спускается к дочерним функциям по определениям с одинаковыми метками и количествами инцидентных функциональных символов. При этом для всех определений кроме **Call** достаточно рассмотреть определения в канонической форме, а для определений вида **Call** надо рассмотреть все возможные перестановки аргументов.

Как уже было сказано, после построения пребисимуляций необходимо отфильтровать такие пребисимуляции, для которых нельзя найти согласованный набор перестановок параметров. Это можно сделать при помощи процедуры, основанной на анализе потока данных, которую мы опишем в оставшейся части раздела.

5.2.1 Частичные перестановки параметров

Далее в алгоритме дополнения пребисимуляции до бисимуляции нам понадобится обобщение понятия перестановки параметров — частичные перестановки параметров (или, по-другому, отношения, инъективные в обе стороны). Будем обозначать их $\theta : m \hookrightarrow n$. Для обращения таких перестановок будем использовать запись θ^{op}, более того, будем распространять её и на обычные перестановки параметров (в результате будет получаться частичная перестановка). За $|\theta|$ будем обозначать количество элементов, на которых θ определена.

Частичную перестановку $\theta : m \hookrightarrow n$ можно представить как следующую пару перестановок: $\theta^R : m \hookrightarrow (n+m-|\theta|)$ и $\theta^L : n \hookrightarrow (n+m-|\theta|)$. Определим θ^R и θ^L как на Рис. 5.2. Таким образом будет выполнено $\theta^L(j) = \theta^R(i)$ тогда и только тогда, когда $j = \theta(i)$. Данное определение устроено специально таким образом, чтобы выполнялось следующее утверждение, позволяющее применять операцию *bisplit* для определения θ^R и θ^L.

Утверждение 30. Пусть $\xi : m \hookrightarrow N$ и $\zeta : n \hookrightarrow N$ — частичные перестановки. Пусть $\theta = \zeta^{op} \circ \xi$, $\theta : m \hookrightarrow n$. Тогда

$$(\_, \theta^L, \theta^R) = bisplit(\zeta, \xi)$$

Рис. 5.2 Определение θ^R и θ^L

$\theta : m \hookrightarrow n$
$unshared = [i \mid i \in [1 \dots m], \theta(i)$ не определено$]$
$|unshared| = m - |\theta|$
$\theta^L : n \hookrightarrow n + m - |\theta|$
$\theta^L = id_{n \to n+m-|\theta|}$
$\theta^R : m \hookrightarrow n + m - |\theta|$

$$\theta^R(i) = \begin{cases} \theta(i), & \text{если } \theta(i) \text{ определено} \\ n + \text{индекс элемента } i \text{ в } unshared, & \text{иначе} \end{cases}$$

Нам также понадобится следующее утверждение.

Утверждение 31. Пусть $\theta : m \hookrightarrow n$ — частичная перестановка, а $\xi : m' \to m$ и $\zeta : n' \to n$ — обычные перестановки параметров. Пусть $\tau = \zeta^{op} \circ \theta \circ \xi$. Тогда

$$(\_, \tau^L, \tau^R) = bisplit(\theta^L \circ \zeta, \theta^R \circ \xi)$$

Доказательство. Т.к. $\theta = (\theta^L)^{op} \circ \theta^R$, то $\tau = (\theta^L \circ \zeta)^{op} \circ (\theta^R \circ \xi)$. Тогда требуемое выполняется по предыдущему утверждению. □

Введём также на частичных перестановках параметров операцию $\oplus$ такую, что $\theta \oplus \xi$ определено только если θ и ξ имеют одинаковые типы и для любого i верно, что или $\theta(i) = \xi(i)$, или $\theta(i)$ не определено, или $\xi(i)$ не определено:

$$(\theta \oplus \xi)(i) = \begin{cases} \theta(i) & \text{если } \theta(i) \text{ определено} \\ \xi(i) & \text{если } \xi(i) \text{ определено} \end{cases}$$

Если же есть такое i, что $\theta(i) \neq \xi(i)$ (оба значения определены), то будем считать, что $\theta \oplus \xi = \top$, где $\top$ — специальный символ, обозначающий противоречие в перестановках. Отметим, что частичные перестановки вместе с $\top$ образуют таким образом полурешётку. Мы не будем считать $\top$ частичной перестановкой, когда нам надо будет подчеркнуть, что переменная θ обозначает частичную перестановку или $\top$, мы так и будем говорить "θ — частичная

перестановка или $\top$".

Нам также понадобится операция *shift*, распространённая на частичные перестановки вместе с $\top$ естественным образом:

$$\begin{aligned}
&\mathit{shift}(\theta : m \hookrightarrow n, k) : (k+m) \hookrightarrow (k+n)\\
&\mathit{shift}(\theta, k)(i) = \begin{cases} i, \text{ если } i \le k \\ k + \theta(i-k), \text{ иначе, если } \theta(i-k) \text{ определено} \end{cases}\\
&\mathit{shift}(\top, k) = \top
\end{aligned}$$

Данная операция согласуется с аналогичной операцией для обычных перестановок следующим образом.

Утверждение 32. Пусть $\theta : m \hookrightarrow n$ — частичная перестановка, пусть $\tau = \mathit{shift}(\theta, k)$. Тогда

$$(\_, \tau^L, \tau^R) = \mathit{bisplit}(\mathit{shift}(\theta^L, k), \mathit{shift}(\theta^R, k))$$

Также нам понадобится утверждение о взаимодействии *shift* и $\oplus$.

Утверждение 33. Пусть $\theta, \xi : m \hookrightarrow n$ — частичные перестановки или $\top$. Тогда $\mathit{shift}(\theta \oplus \xi, k) = \mathit{shift}(\theta, k) \oplus \mathit{shift}(\xi, k)$.

Доказательство очевидно.

Кроме того, нам будет нужна обратная операция *unshift*. Для удобства мы определим её так, чтобы она работала только для перестановок, которые первые k переменных отображают в их самих или хотя бы не определены на них:

$$\begin{aligned}
&\mathit{unshift}(\theta : (k+m) \hookrightarrow (k+n), k) : m \hookrightarrow n\\
&\mathit{unshift}(\theta, k) = \top, \text{ если } \exists j \le k : \theta(j) \text{ определено и } \theta(j) \ne j\\
&\mathit{unshift}(\theta, k)(i) = \theta(k+i) - k\\
&\mathit{unshift}(\top, k) = \top
\end{aligned}$$

Нетрудно проверить, что $\mathit{unshift}(\mathit{shift}(\theta, k), k) = \theta$. Кроме того, если переставить функции в обратном порядке, то $\theta \sqsubseteq \mathit{shift}(\mathit{unshift}(\theta, k), k)$.

5.2.2 Дополнение перестановками параметров и обоснование факта бисимуляции

Теперь опишем, как структуру, представленную деревом **PreBisimulation**, преобразовать в обычную бисимуляцию. Для этого нужно построить по дереву полипрограмму и предоставить два морфизма, отображающих эту полипрограмму в исходную. Основная проблема — построение необходимых перестановок параметров. Т.к. у соответствующих функций параметры могут идти в разном порядке, сопоставим каждому узлу b дерева пребисимуляции частичную перестановку параметров (включая $\top$) $perm(b)$, описывающую связь между параметрами соответствующих функций исходной полипрограммы. Тип $perm(b)$ зависит от узла и имеет следующий вид:

$$perm(\mathbf{Cong}(f_L, f_R, q_L, q_R, \overline{b_i})) : arity(f_R) \nrightarrow arity(f_L)$$
$$perm(\mathbf{Refl}(f)) : arity(f) \nrightarrow arity(f)$$
$$perm(\mathbf{Fold}(f_L, f_R)) : arity(f_R) \nrightarrow arity(f_L)$$

Кроме того, каждому узлу вида **Cong** поставим в соответствие частичную перестановку (включая $\top$) $iperm(b)$. Эта перестановка описывает связь между параметрами "внутри" определения и содержит больше информации, чем $perm(b)$. Она будет играть важную роль при построении бисимуляции. Её тип определяется следующим образом:

$$iperm(\mathbf{Cong}(f_L, f_R, (\xi_L f_L \equiv rhs_L), (\xi_R f_R \equiv rhs_R), \overline{b_i})) :$$
$$cod(\xi_R) \nrightarrow cod(\xi_L)$$

Для построения бисимуляции необходимо, чтобы все эти перестановки были согласованы друг с другом, для этого необходимо решить задачу анализа потока данных (например, простым итеративным алгоритмом). Как уже было сказано, частичные перестановки, дополненные элементом $\top$, обозначающим отсутствие перестановки, вместе с операцией $\oplus$ образуют полурешётку. После решения задачи анализа потока данных либо окажется, что какие-то из частичных перестановок превратились в $\top$ и такую пребисимуляцию нужно выкинуть из дальнейшего рассмотрения, либо частичные перестановки стабилизируются так, что по пребисимуляции можно будет построить бисимуляцию.

Рис. 5.3 Вспомогательные определения

$$
\begin{aligned}
&\mathit{lift\text{-}perms}(L(\overline{\zeta_i g_i}), L(\overline{\sigma_i h_i}), \overline{\theta_i}) = \\
&\qquad = \mathit{lift\text{-}perms}'(L, \overline{\zeta_i \circ \theta_i \circ \sigma_i^{op}}) \\
&\mathit{lift\text{-}perms}'(\mathbf{Cons}_C, \theta_1, \ldots, \theta_n) = \theta_1 \oplus \ldots \oplus \theta_n \\
&\mathit{lift\text{-}perms}'(\mathbf{Call}, id, \theta_1, \ldots, \theta_n) = \theta_1 \oplus \ldots \oplus \theta_n \\
&\mathit{lift\text{-}perms}'(\mathbf{Case}_{\overline{C_i}}, \xi, \theta_1, \ldots, \theta_n) = \\
&\qquad = \xi \oplus \mathit{unshift}(\theta_1, k_1) \oplus \ldots \oplus \mathit{unshift}(\theta_n, k_n) \\
&\qquad \text{где } k_i = \mathit{arity}(C_i) \\
&\mathit{lift\text{-}perms}'(\mathbf{Var}) = id_1 \\
&\mathit{lift\text{-}perms}'(\mathbf{Id}, \theta) = \theta
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{push\text{-}perm}(\theta, L(\overline{\zeta_i g_i}), L(\overline{\sigma_i h_i})) = \\
&\qquad = [\overline{\zeta_i^{op} \circ \theta_i \circ \sigma_i}] \\
&\qquad \text{где } \theta_i = \mathit{push\text{-}perm}'(\theta, L, n)[i] \\
&\qquad n = \max i \\
&\mathit{push\text{-}perm}'(\theta, \mathbf{Cons}_C, n) = [\theta, \ldots, \theta] \text{ (длины } n) \\
&\mathit{push\text{-}perm}'(\theta, \mathbf{Call}, n) = [id_{n-1}, \theta, \ldots, \theta] \text{ (длины } n) \\
&\mathit{push\text{-}perm}'(\theta, \mathbf{Case}_{\overline{C_i}}, n) = [\theta, \overline{\mathit{shift}(\theta, k_i)}] \\
&\qquad \text{где } k_i = \mathit{arity}(C_i) \\
&\mathit{push\text{-}perm}'(\theta, \mathbf{Var}, 0) = [] \\
&\mathit{push\text{-}perm}'(\theta, \mathbf{Id}, 1) = [\theta]
\end{aligned}
$$

Инициализируем перестановки, соответствующие узлам **Cong** и **Fold**, пустыми частичными перестановками, а перестановки, соответствующие **Refl**, тождественными перестановками:

$$
\begin{aligned}
&\mathit{iperm}(\mathbf{Cong}(\ldots)) := \{\} \\
&\mathit{perm}(\mathbf{Cong}(\ldots)) := \{\} \\
&\mathit{perm}(\mathbf{Refl}(f)) := id_{\mathit{arity}(f)} \\
&\mathit{perm}(\mathbf{Fold}(f_L, f_R)) := \{\}
\end{aligned}
$$

Далее, для каждого узла b вида

$$b = \mathbf{Cong}(f_L, f_R, (\xi_L f_L \equiv rhs_L), (\xi_R f_R \equiv rhs_R), \overline{b_i})$$

будем выполнять следующие присваивания (определения вспомогательных функций приведены на Рис. 5.3):

$$\begin{aligned}
&perm(b) := perm(b) \oplus (\xi_L^{op} \circ iperm(b) \circ \xi_R) \\
&iperm(b) := iperm(b) \oplus \\
&\qquad \oplus (\xi_L \circ perm(b) \circ \xi_R^{op}) \oplus \mathit{lift\text{-}perms}(rhs_L, rhs_R, \overline{perm(b_i)}) \\
&perm(b_i) := perm(b_i) \oplus \mathit{push\text{-}perm}(iperm(b), rhs_L, rhs_R)[i]
\end{aligned}$$

А для каждого узла b вида **Fold**, соответствующего узлу b', будем выполнять такие присваивания:

$$\begin{aligned}
&perm(b) := perm(b') \oplus perm(b) \\
&perm(b') := perm(b') \oplus perm(b)
\end{aligned}$$

Благодаря монотонности обновлений и конечности перестановок данный процесс рано или поздно стабилизируется. При этом будут выполняться следующие соотношения (равенства $\alpha = \alpha \oplus \beta$ переписаны в виде $\alpha \sqsupseteq \beta$):

для $b = \mathbf{Cong}(f_L, f_R, (\xi_L f_L \equiv rhs_L), (\xi_R f_R \equiv rhs_R), \overline{b_i})$:

$$\begin{aligned}
&\quad perm(b) \sqsupseteq (\xi_L^{op} \circ iperm(b) \circ \xi_R) \\
&\quad iperm(b) \sqsupseteq (\xi_L \circ perm(b) \circ \xi_R^{op}) \oplus \mathit{lift\text{-}perms}(rhs_L, rhs_R, \overline{perm(b_i)}) \\
&\quad perm(b_i) \sqsupseteq \mathit{push\text{-}perm}(iperm(b), rhs_L, rhs_R)[i] \\
&perm(\mathbf{Refl}(f)) \sqsupseteq id_{arity(f)} \\
&perm(\mathbf{Fold}(\ldots)) = perm(b'),
\end{aligned}$$

где b' — соответствующий данной свёртке узел

Докажем следующее утверждение, которое позволит обосновать существование бисимуляции:

Утверждение 34. Если для узла b пребисимуляции вида

$$b = \mathbf{Cong}(f_L, f_R, (\xi_L f_L \equiv rhs_L), (\xi_R f_R \equiv rhs_R), \overline{b_i})$$

выполняются неравенства вида

$$perm(b) \sqsupseteq (\xi_L^{op} \circ iperm(b) \circ \xi_R)$$
$$iperm(b) \sqsupseteq (\xi_L \circ perm(b) \circ \xi_R^{op}) \oplus \mathit{lift\text{-}perms}(rhs_L, rhs_R, \overline{perm(b_i)})$$
$$perm(b_i) \sqsupseteq \mathit{push\text{-}perm}(iperm(b), rhs_L, rhs_R)[i]$$

то выполняется равенства

$$perm(b) = \xi_L^{op} \circ iperm(b) \circ \xi_R$$
$$perm(b_i) = \mathit{push\text{-}perm}(iperm(b), rhs_L, rhs_R)[i]$$

Доказательство этого утверждения приведено в Приложении В на с. 207.

Пусть теперь дана пребисимуляция, представленная деревом, причём в результате анализа потока данных были построены перестановки параметров *perm* и *iperm*, удовлетворяющие указанным выше соотношениям, причём ни одна перестановка в результате не оказалась равна $\top$. Пусть функция $node(b)$ отображает узел b этого дерева вида **Cong** и **Refl** в функции, однозначно соответствующие b, а узлы вида $\mathbf{Fold}(\theta, f_L, f_R)$ в функции, соответствующие ближайшим предкам этого поддерева вида $\mathbf{Cong}(\theta, f_L, f_R, \ldots)$. При этом арность функций будем считать так:

$$arity(node(b)) = m + n - |\theta|$$
$$\text{где } \theta = perm(b) : m \hookrightarrow n$$

Эти функции будут функциями полипрограммы бисимуляции. Множество функций W будет состоять из функций, соответствующих узлам вида **Refl**. С функциями исходной полипрограммы новые функции будут связаны при

помощи следующих морфизмов функциональных символов:

$$\alpha_{L,R}(node(\mathbf{Refl}(f))) = (id, f)$$
$$\alpha_L(node(\mathbf{Cong}(\theta, f_L, f_R, \ldots))) = (\theta^L, f_L)$$
$$\alpha_R(node(\mathbf{Cong}(\theta, f_L, f_R, \ldots))) = (\theta^R, f_R)$$

Таким образом, если мы обнаружили, что функции $f_L : n$ и $f_R : m$ эквивалентны с точностью до частичной перестановки параметров $\theta : m \hookrightarrow n$, то соответствующая им функция полипрограммы бисимуляции будет иметь арность $(m + n - |\theta|)$ и отображаться в исходные функции с перестановками θ^L и θ^R.

Теперь нам нужно описать, как строить определения полипрограммы бисимуляции по определениям исходной полипрограммы. Каждое такое определение будет получаться из узла вида **Cong**. Пусть в дереве пребисимуляции есть узел следующего вида:

$$b = \mathbf{Cong}(\theta, f_L, f_R, q_L, q_R, \overline{b_i}), \text{ где}$$
$$q_L = (\xi_L f_L \equiv L(\overline{\zeta_{L,i} g_i}))$$
$$q_R = (\xi_R f_R \equiv L(\overline{\zeta_{R,i} h_i}))$$

Обозначим $\theta = perm(b)$, $\theta_i = perm(b_i)$ и $\tau = iperm(b)$. Нужно построить такое определение $q' = (\xi' node(b) \equiv L(\overline{\zeta'_i node(b_i)}))$, чтобы $\alpha_{L,R}(q') \approx q_{L,R}$. При этом

$$\alpha_L(q') = (\xi' \theta^R f_L \equiv L(\overline{\zeta'_i \theta^R_i g_i}))$$
$$\alpha_R(q') = (\xi' \theta^L f_R \equiv L(\overline{\zeta'_i \theta^L_i h_i}))$$

Построим ξ' следующим образом. По Утверждению 34 выполняется $\theta = \xi_L^{op} \circ \tau \circ \xi_R$. Значит по Утверждению 31 выполняется следующее:

$$(\xi', \theta^L, \theta^R) = bisplit(\tau^L \circ \xi_L, \tau^R \circ \xi_R)$$

Тогда будет также выполняться $\xi' \theta^R = \tau^R \xi_R$ и $\xi' \theta^L = \tau^L \xi_L$.

Теперь необходимо найти перестановки ζ'_i из правой части. При этом надо учитывать конкретный вид определения L, поэтому для определённости рассмотрим случай сопоставления с образцом **Case** (остальные случаи рас-

сматриваются аналогично).

По Утверждению 34 выполняется

$$\theta_i = \mathit{push\text{-}perm}(\tau, L(\overline{\zeta_{L,i} g_i}), L(\overline{\zeta_{R,i} h_i}))[i]$$

Пусть количество связанных переменных в соответствующей ветви равно k_i. Перепишем данное равенство:

$$\theta_i = \zeta_{L,i}^{op} \circ \mathit{shift}(\tau, k_i) \circ \zeta_{R,i}$$

Теперь, используя Утверждения 31 и 32 и свойства *bisplit*, находим ζ'_i:

$$(\zeta'_i, \theta_i^L, \theta_i^R) = \mathit{bisplit}(\mathit{shift}(\tau^L, k_i) \circ \zeta_{L,i}, \mathit{shift}(\tau^R, k_i) \circ \zeta_{R,i})$$

Проверим, что найденные ξ' и ζ'_i действительно нам подходят:

$$\begin{aligned}
\alpha_L(q') &= (\xi' \theta^L f_L \equiv L(\overline{\zeta'_i \theta_i^L g_i})) = \\
&= (\tau^L \xi_L \equiv L(\overline{\mathit{shift}(\tau^L, k_i) \circ \zeta_{L,i} g_i})) \approx \\
&\approx (\xi_L \equiv L(\overline{\zeta_{L,i} g_i})) = \\
&= q_L
\end{aligned}$$

Для случая остальных L доказательство аналогично.

Таким образом, мы построили полипрограмму H такую, что $\alpha_{L,R}$ можно дополнить до морфизмов полипрограмм $\phi_{L,R} : H \to G$. Теперь если H имеет не более одной модели для заданных интерпретаций функциональных символов из W (**Refl**), то по Утверждению 29 можно будет в полипрограмму G добавить равенства между функциями $f_{L,R}$. Алгоритм поиска бисимуляций автоматически не обеспечивает единственность модели, поэтому её надо проверять отдельно, о чём пойдёт речь в следующем разделе.

5.3 Обеспечение единственности модели

При слиянии по бисимуляции возникает вопрос корректности этой операции. По сути мы хотим сделать вывод о равенстве двух функций на основании того, что они удовлетворяют одной и той же системе уравнений. Конечно, делать такой вывод можно только если система уравнений имеет не более

одного решения, что в нашем случае автоматически не выполняется (например, уравнению $f(x) \equiv f(f(x))$ удовлетворяет любая идемпотентная функция). Поэтому нам нужно какое-то достаточное условие того, что у системы не может быть более одного решения (причём алгоритмически проверяемое, так как нам надо будет убеждаться перед слиянием по бисимуляции, что оно выполняется).

Эта задача похожа на другую задачу — проверку завершаемости. В самом деле, если система уравнений удовлетворяет достаточному условию завершаемости, то её наименьшая неподвижная точка состоит из функций, которые на каждом входе возвращает некоторое значение, являющееся максимальным элементом соответствующей полурешётки — а это значит, что больше неподвижной точки, чем наименьшая, не существует. Таким образом из завершаемости следует единственность. Ещё одна схожая задача – проверка продуктивности. Точно также, если система уравнений удовлетворяет достаточному условию продуктивности, то она тоже имеет единственное решение. Таким образом, задача проверки единственности является обобщением этих двух задач.

Отметим ещё, что обратное неверно: если решение единственно, это ещё не значит, что оно продуктивно или завершается:

$$\begin{aligned}
&\mathrm{inf} \equiv S(\mathrm{inf}) \\
&\mathrm{eat}(x) \equiv \textbf{case}\ x\ \textbf{of}\ S(x')\ \rightarrow\ \mathrm{eat}(x') \\
&\mathrm{bottom} \equiv \mathrm{eat}(\mathrm{inf})
\end{aligned}$$

В данном случае система уравнений имеет единственное решение, в котором $\mathrm{bottom} = \bot$, то есть bottom не завершается и не продуктивна. Однако inf продуктивна, а eat завершается на любом конечном входе, поэтому они имеют единственное решение, а значит и bottom имеет единственное решение как их композиция.

В качестве признаков единственности будем использовать признаки продуктивности и завершаемости (а точнее признаки защищённой рекурсии, из которой следует продуктивность, и структурной рекурсии, из которой следует завершаемость) подобно тому, как это сделано для языка Agda [1; 2].

Синтаксические условия защищенности и структурности проверяются при помощи алгоритма, похожего на анализ потока данных. В результате реше-

ния задачи анализа мы будем знать, что все циклы в полипрограмме удовлетворяют некоторому синтаксическому условию. Если мы затем рассмотрим бесконечную развёртку полипрограммы в виде дерева (бесконечного), то в нём пути между узлами, соответствующими одной и той же функции в оригинальной полипрограмме, будут удовлетворять тому же условию. Бесконечная развёртка хороша тем, что простой разметкой узлов значениями параметров она превращается в дерево процесса, и из синтаксического свойства путей мы можем вывести соотношения между значениями параметров в узлах.

5.3.1 Выражение защищённой рекурсии через структурную

В нашей семантике мы допускаем использование бесконечных данных, а значит нам нужна поддержка защищённой рекурсии для конструирования таких данных. Чтобы упростить задачу, мы сведём бесконечные данные к конечным при помощи приближения монотонными последовательностями частичных, но конечных данных [37], а защищённую рекурсию сведём к структурной. Для этого мы преобразуем полипрограмму, добавив к каждой функции дополнительный параметр, указывающий на какую глубину следует вычислять значение этой функции. Фактически это означает преобразование функции $f(\overline{x_i})$ в функцию $approx(n, f(\overline{x_i}))$, где $approx$ определяется следующим образом:

$$
\begin{array}{l}
approx(n, x) = \\
\quad \textbf{case } n \textbf{ of} \\
\qquad S(m) \ \rightarrow \ \textbf{case } x \textbf{ of } \{ \ \overline{C_i(\overline{y_j}) \ \rightarrow \ C_i(\overline{approx(m, y_j)})} \ \}
\end{array}
$$

То есть $approx(\bot, x) = \bot$, а $approx(S^n(\bot), x)$ вернёт x обрезанный после глубины n. Заметим, что верно следующее:

$$a = approx(\infty, a) = \lim_{n \to \infty} approx(n, a) = \sup\{approx(S^m(\bot), a) \mid m \in \mathbb{N}\}$$

где $\infty = S(S(S(\ldots)))$. Также нам понадобится тот факт, что если $a \sqsubseteq b$, то $approx(n, a) \sqsubseteq approx(n, b)$. Также будем использовать обозначение

$$approx_n(a) \stackrel{\text{def}}{=} approx(S^n(\bot), a)$$

Рис. 5.4 Преобразование функций в аппроксимации

$$
\begin{array}{l}
\mathcal{A}[\![\{\overline{d_i}\}]\!] = \{\overline{A[\![d_i]\!]}\} \cup \\
\quad \cup \left\{ \begin{array}{l} \mathrm{approx}(n, x) \equiv \\ \qquad \textbf{case}\ n\ \textbf{of}\{S(m) \to \\ \qquad\qquad \textbf{case}\ x\ \textbf{of}\{\overline{C_i(\overline{y_j}) \to C_i(\overline{\mathrm{approx}(m, y_j)})}\}\} \end{array} \right\} \cup \\
\quad \cup \{\mathrm{inf} \equiv S(\mathrm{inf})\} \\
A[\![f(\overline{x_i}) \equiv e]\!] = (f(n, \overline{x_i}) \equiv A_n[\![e]\!]) \\
A_n[\![C(\overline{e_i})]\!] = \textbf{case}\ n\ \textbf{of}\{\ S(m)\ \to\ C(\overline{A_m[\![e_i]\!]})\ \} \\
A_n[\![f(\overline{x_i})]\!] = f(n, \overline{x_i}) \\
A_n[\![f(\overline{e_i})]\!] = f(n, \overline{A_{\mathrm{inf}}[\![e_i]\!]}) \\
A_n[\![\textbf{case}\ e\ \textbf{of}\ \{\ \overline{C_i(\overline{x_j}) \to e_i}\ \}]\!] = \textbf{case}\ A_{\mathrm{inf}}[\![e]\!]\ \textbf{of}\ \{\ \overline{C_i(\overline{x_j}) \to A_n[\![e_i]\!]}\ \} \\
A_n[\![x]\!] = \mathrm{approx}(n, x)
\end{array}
$$

На Рис. 5.4 представлено преобразование, сводящее защищённую рекурсию к структурной, сформулированное для человеко-читаемых полипрограмм. Аналогично можно было бы сформулировать его для бесточечного представления, но т.к. принципиальной разницы нет, мы это делать не будем. Будем считать, что inf и approx — свежие имена, которые не встречаются в исходной полипрограмме. Данное преобразование добавляет каждому функциональному символу исходной полипрограммы дополнительный параметр, и некоторым образом преобразует каждое определение. При этом все конструкторы становятся защищены сопоставлениями с образцом, разбирающими первую переменную (обозначающую глубину). В вызовы функций, как и в ветви сопоставлений с образцом, глубина передаётся в неизменном виде. А в аргументы функций и в разбираемые выражения вместо глубины передаётся бесконечность, потому что эти значения могут понадобиться до произвольной заведомо неизвестной глубины. Отметим, что условие защищённой рекурсии на исходной программе переходит в условие структурной рекурсии по первой переменной (глубине) в преобразованной программе, поэтому дальше будем работать только со структурной рекурсией. Единственный нюанс — новая функция inf (константа ∞) не является структурно-рекурсивной, поэтому её надо исключить из проверки структурной рекурсии.

Основное свойство преобразования $\mathcal{A}$ звучит следующим образом:

Утверждение 35. Пусть $\mu \models G$, тогда существует $\mu' \models \mathcal{A}[\![G]\!]$ такая, что на общих функциях $\mu'(f)(n, \overline{x_i}) = approx(n, \mu(f)(\overline{x_i})))$. И наоборот, по $\mu' \models \mathcal{A}[\![G]\!]$ можно построить $\mu \models G$, причём на общих функциях $\mu(f)(\overline{x_i}) = \mu'(f)(\infty, \overline{x_i})$. Более того, модели находятся во взаимно-однозначном соответствии (даже с учётом добавленных функций).

5.3.2 Достаточное условие структурной рекурсии

Пусть функция $f(\overline{x_i})$ арности n вызывает некоторую функцию $g(\overline{y_j})$ арности m. Тогда связь между их аргументами можно приблизить при помощи матрицы $M : R_{m \times n}$, где R — некоторое подмножество отношений между возможными значениями аргументов, $R \subseteq \mathrm{Rel}(\mathbb{A}, \mathbb{A})$ (конкретный R мы определим ниже). Т.е. каждая ячейка матрицы — это отношение, при этом (неформально) должно выполнятся $y_j \ M(j,i) \ x_i$, если в результате вызова функции f с аргументами x_i произошёл вызов функции g с аргументами y_j.

Напомним, что множество возможных значений аргументов в нашей семантике можно рассматривать как наибольшую неподвижную точку следующего определения:

$$\mathbb{A} = \{C(\overline{a_i}) \mid a_i \in \mathbb{A},\ C \text{ — конструктор}\} \cup \{\bot\}$$

Дальнейшее изложение полностью соответствует [2]. Единственная разница — мы считаем, что $a \leq b$ не только когда a является подвыражением b, но и когда $a \sqsubseteq b$, однако это сыграет роль только в следующем разделе. Отметим, что при этом сохраняется главное свойство отношения $<$ — фундированность для конечных элементов из $\mathbb{A}$.

Нам понадобятся следующие отношения на $\mathbb{A}$: $<$, $\leq$ и $?$, где $a \ ? \ b$ выполняется всегда, $a < b \Leftrightarrow (a \leq b \wedge a \neq b)$, а $\leq$ определено как наименьшая неподвижная точка следующих утверждений:

$$a \leq a$$
$$a \leq b \Rightarrow a \leq C_j(\ldots, b, \ldots)$$
$$a \sqsubseteq b \Rightarrow a \leq b$$
$$a \leq b, b \leq c \Rightarrow a \leq c$$

На $\{<, \leq, ?\}$ определим операции умножения (композиции) $\cdot$ и сложения (конъюнкции) $+$ следующим образом:

$+$	$<$	$\leq$	$?$
$<$	$<$	$<$	$<$
$\leq$	$<$	$\leq$	$\leq$
$?$	$<$	$\leq$	$?$

$\cdot$	$<$	$\leq$	$?$
$<$	$<$	$<$	$?$
$\leq$	$<$	$\leq$	$?$
$?$	$?$	$?$	$?$

Операция $+$ устроена так, что если $a\ R\ b$ и $a\ Q\ b$, то $a\ (R+Q)\ b$ и наоборот. Операция $\cdot$ работает почти как композиция: если $a\ R\ b$ и $b\ Q\ c$, то $a\ (R \cdot Q)\ b$. Однако обратное неверно, например, $Z\ (< \cdot <)\ S(Z)$, но не существует такого b, что $Z < b < S(Z)$.

Далее будем рассматривать матрицы отношений из $R_{m\times n}$, где $R = \{<, \leq, ?\}$.

Отметим, что матрицы отношений можно воспринимать как отношения между кортежами:

$$(\overline{a_j})\ M\ (\overline{b_i}) \Leftrightarrow \forall i, j(a_j\ M(j,i)\ b_i)$$

На матрицах отношений можно определить операцию умножения $\cdot$ через операции $\cdot$ и $+$ на отношения точно так же как для обычных матриц, т.е.

$$(M \cdot N)(j,i) = \sum_k M(j,k) \cdot N(k,i)$$

При этом будет выполняться $\overline{a}\ M\ \overline{b}\ N\ \overline{c} \Rightarrow \overline{a}\ (M \cdot N)\ \overline{c}$.

Теперь каждому определению вида $\xi f \equiv L(\overline{\theta_l g_l})$ нужно сопоставить набор матриц вызова $\mathcal{M}_l[\![q]\!]$, каждая матрица описывает как соотносятся аргументы, передаваемые функции f и аргументы, передаваемые функции g_l. Для начала определим операцию воздействия перестановки на матрицу. Пусть $M : R_{m\times n}$, $\xi : n' \hookrightarrow n$ и $\theta : m' \hookrightarrow m$, тогда:

$$M\xi : R_{m\times n'}$$
$$(M\xi)(j,i) = M(j, \xi(i))$$
$$\theta M : R_{m'\times n}$$
$$(\theta M)(j,i) = M(\theta(j), i)$$

Теперь определим $\mathcal{M}_k[\![\xi f \equiv L(\overline{\theta_l g_l})]\!]$ как на Рис. 5.5. Отметим, что матрица определения зависит не только от самой формулы определения, но и от

Рис. 5.5 Определение матриц вызова для определений

$$\mathcal{M}_k[\![\xi f \equiv L(\overline{\theta_l g_l})]\!] : R_{arity(g_k)\times arity(f)}$$

$$\mathcal{M}_k[\![\xi f \equiv L(\overline{\theta_l g_l})]\!] = \theta \mathcal{M}_k[\![f' \equiv L(\ldots, \theta_{k-1} g_{k-1}, g_k, \theta_{k+1} g_{k+1}, \ldots)]\!]\xi$$

$$\mathcal{M}_k[\![f \equiv \mathbf{Id}(g_k)]\!](j,i) =$$

$$= \begin{cases} \leq & \text{если } i = j \\ ? & \text{иначе} \end{cases}$$

$$\mathcal{M}_k[\![f \equiv \mathbf{Cons}_C(\ldots, g_k, \ldots)]\!](j,i) =$$

$$= \begin{cases} \leq & \text{если } i = j \\ ? & \text{иначе} \end{cases}$$

$$\mathcal{M}_1[\![f \equiv \mathbf{Call}(g_1, \overline{\theta_{l+1} g_{l+1}})]\!](j,i) =$$

$$= \begin{cases} \leq & \text{если в полипрограмме есть } g_{1+j} \equiv \mathbf{Var} \text{ и } \theta_{1+j}(1) = i \\ ? & \text{иначе} \end{cases}$$

$$\mathcal{M}_k[\![f \equiv \mathbf{Call}(\ldots, g_k, \ldots)]\!](j,i) =$$

$$= \begin{cases} \leq & \text{если } i = j \\ ? & \text{иначе} \end{cases}$$

$$\mathcal{M}_1[\![f \equiv \mathbf{Case}_{\overline{C_l}}(g_1, \overline{\theta_l g_l})]\!](j,i) =$$

$$= \begin{cases} \leq & \text{если } i = j \\ ? & \text{иначе} \end{cases}$$

$$\mathcal{M}_k[\![f \equiv \mathbf{Case}_{\overline{C_l}}(\theta_1 g_1, \ldots, g_k, \ldots)]\!](j,i) =$$

$$= \begin{cases} < & \text{если в полипрограмме есть } g_1 \equiv \mathbf{Var} \\ & \qquad \text{и верно, что } \theta_1(1) = i \text{ и } j \leq arity(C_{k-1}) \\ \leq & \text{если } i = j - arity(C_{k-1}) \\ ? & \text{иначе} \end{cases}$$

присутствия некоторых других определений в полипрограмме.

Понятие матрицы вызова можно обобщить на пути в полипрограмме. Для этого сначала введём понятие пути.

Определение 38. Пусть G — полипрограмма. *Путём* в полипрограмме назовём конечную последовательность определений с выделенными функциональными символами вида

$$\xi_1 f_1 \equiv p_1(\ldots, \theta_2 f_2, \ldots); \xi_2 f_2 \equiv p_2(\ldots, \theta_3 f_3, \ldots); \ldots$$
$$\ldots \xi_n f_n \equiv p_n(\ldots, \theta_{n+1} f_{n+1}, \ldots)$$

Матрица пути в полипрограмме определяется как композиция матриц отдельных определений:

$$\mathcal{M}[\![\overline{\xi_i f_i \equiv p_i(\ldots, \theta_{i+1} f_{i+1}, \ldots)}]\!] =$$
$$= \mathcal{M}_{k_n}[\![\xi_n f_n \equiv p_n(\ldots, \theta_{n+1} f_{n+1}, \ldots)]\!] \cdot \ldots$$
$$\ldots \cdot \mathcal{M}_{k_1}[\![\xi_1 f_1 \equiv p_1(\ldots, \theta_2 f_2, \ldots)]\!]$$

Матрица пустого пути является тождественной матрицей, с $\leq$ на диагонали и ? в остальных местах.

Для того, чтобы обеспечить структурную рекурсию, мы потребуем, чтобы все циклы (т.е. пути, где $f_{n+1} = f_1$) обладали матрицей с определёнными свойствами. Для проверки этого надо найти все возможные матрицы замкнутых путей, что проще всего сделать построив семейство множеств матриц путей между всеми парами функций, т.е. следующее семейство множеств:

$$M(f, g) = \{\mathcal{M}[\![path]\!] \mid path \text{ — путь от } f \text{ до } g\}$$

Его можно построить найдя наименьшую неподвижную точку следующей системы уравнений для множеств матриц:

$$M(f, g) = \{\mathcal{M}[\![p]\!] \mid p \text{ — путь длины 1 между } f \text{ и } g\ \} \cup$$
$$\cup \{M_{hg} \cdot M_{fh} \mid M_{hg} \in M(h, g), M_{fh} \in M(f, h),\ h \text{ — функция}\}$$

Данное семейство множеств строится за конечное время благодаря конечному числу различных матриц путей (несмотря на бесконечное число самих путей).

Остаётся только проверить, что для любой функции полипрограммы f арности n существует некоторая биекция $\theta : n \hookrightarrow n$ такая, что для любой матрицы $N \in M(f, f)$ (а значит и для любой матрицы цикла полипрограммы, проходящего через f) верно, что если $(\overline{a_i})\ N\ (\overline{b_i})$, то $(\overline{a_{\theta(i)}}) \prec (\overline{b_{\theta(i)}})$, где $\prec$ — лексикографическое обобщение $<$ на кортежи, т.е.

$$(\overline{a_i}) \prec (\overline{b_i}) \Leftrightarrow a_1 < b_1 \vee (a_1 \leq b_1 \wedge (a_2, \ldots, a_n) \prec (b_2, \ldots, b_n))$$

Такой θ найти несложно (или проверить его отсутствие), см. [2]. Данное условие будем называть достаточным условием структурной рекурсии, как мы увидим в следующем разделе, оно обеспечивает единственность модели полипрограммы.

Чтобы разобраться также и с защищённой рекурсией, надо ослабить требование для функции inf с определением $\mathrm{inf} \equiv S(\mathrm{inf})$, которую вводит преобразование $\mathcal{A}$. Эта функция не является структурно-рекурсивной, однако она является простейшей защищённо-рекурсивной функцией, и её можно рассматривать как особый случай. Проще всего её просто выкинуть

Определение 39. Будем говорить, что полипрограмма G удовлетворяет *достаточному условию структурно-защищённой рекурсии*, если $H = \mathcal{A}[\![G]\!]$ удовлетворяет условию структурной рекурсии, не считая функцию inf, т.е. для любой функции $f \neq \mathrm{inf}$ из H существует некоторая биекция θ такая, что для любой матрицы $N \in M(f, f)$ верно, что если $(\overline{a_i})\ N\ (\overline{b_i})$, то $(\overline{a_{\theta(i)}}) \prec (\overline{b_{\theta(i)}})$.

5.3.3 Доказательство единственности модели

Для случая тотальной семантики единственность модели для полипрограммы, в которой выполняется условие на матрицы путей из предыдущего раздела, следует из завершаемости. Для случая нашей, нетотальной, семантики рассуждения должны быть несколько модифицированы, потому что в этом случае завершаемость не гарантируется из-за наличия бесконечных данных. В этом разделе мы проведём рассуждения, необходимые для доказательства единственности модели в нашей семантике.

Напомним определение функции, непрерывной по Скотту:

Определение 40. Пусть A, B — частично упорядоченные множества. Функ-

ция $f : A \to B$ является *непрерывной*, если для любой монотонной последовательности $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ элементов из A такой, что $\sup\{a_i\} = a$ верно, что $f(a) = \sup\{f(a_i)\}$.

Далее нам понадобится более общее понятие непрерывности, потому что мы будем строить непрерывные функции, и при построении нам будет известно, что они непрерывны только вплоть до какого-то значения.

Определение 41. Пусть A, B — частично упорядоченные множества. Функция $f : A \to B$ является *непрерывной вплоть до* a, если для любой монотонной последовательности $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ элементов из A такой, что $\sup\{a_i\} \sqsubseteq a$ верно, что $f(\sup\{a_i\}) = \sup\{f(a_i)\}$.

Также будем использовать обозначение $\lim_{x \to a} f(x) = b$, если для любой монотонной последовательности $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ такой, что $\sup\{a_i\} = a$ верно, что $\sup\{f(a_i)\} = b$. Соответственно, функция непрерывна, если для любого a верно $\lim_{x \to a} f(x) = f(a)$.

Отметим, что если функция непрерывна вплоть до какого-то значения a, то она также монотонна вплоть до a, т.е. $a_1 \sqsubseteq a_2 \sqsubseteq a \Rightarrow f(a_1) \sqsubseteq f(a_2)$.

Также нам понадобится следующее свойство сопоставлений с образцом,.

Утверждение 36. Пусть даны функции $h, \overline{g_k} \in \mathbb{D}$ Пусть h является непрерывной вплоть до $\overline{a_i}$, и если $h(\overline{a_i}) = C_k(\overline{b_j})$, то g_k является непрерывной вплоть до $(\overline{b_j}, \overline{a_i})$. Тогда функция

$$f(\overline{x_i}) = \mathbf{case}\ h(\overline{x_i})\ \mathbf{of}\{\overline{C_k(\overline{y_j}) \to g_k(\overline{y_j}, \overline{x_i})}\}$$

где

$$\mathbf{case}\ a\ \mathbf{of}\{\overline{C_k(\overline{y_j}) \to e_k\langle\overline{y_j}\rangle}\} = \begin{cases} e_k\langle\overline{b_j}\rangle & \text{если } a = C_k(\overline{b_j}) \\ \bot & \text{иначе} \end{cases}$$

также является непрерывной вплоть до $\overline{a_i}$.

Определение 42. Пусть дана некоторая полипрограмма G. *Состоянием полипрограммы* назовём выражение вида $f(\overline{a_i})$, где f — функция G, а $a_i \in \mathbb{A}$.

Мы введём отношение на состояниях полипрограммы, означающее, что аргументы этих состояний связаны соответствующей матрицей вызова или просто уменьшаются.

Определение 43. Будем говорить, что $f(\overline{a_i}) \rightsquigarrow g(\overline{b_j})$, если выполняется хотя бы одно из двух:

- В G есть определение $\xi f \equiv L(\ldots, \theta g, \ldots)$ и для соответствующей матрицы $M = \mathcal{M}[\![\xi f \equiv L(\ldots, \theta g, \ldots)]\!]$ этого определения верно, что $(\overline{b_j})\ M\ (\overline{a_i})$.
- $f = g$ и $(\overline{b_i}) \sqsubset (\overline{a_i})$ (т.е. $b_i \sqsubseteq a_i$ и для некоторого k выполняется $b_k \sqsubset a_k$).

Неформально, суть данного отношения в том, что если s_2 понадобится в процессе вычисления состояния с *конечными* аргументами s_1, то выполняется $s_1 \rightsquigarrow^+ s_2$. Обратное, конечно, неверно — $s_1 \rightsquigarrow s_2$ не означает, что состояние s_2 обязательно понадобится в процессе вычисления s_1.

Для состояний с бесконечными аргументами возникает проблема, состоящая в том, что они не всегда подчиняются матрице определения. Например, для следующего определения q

$$f(x) = \textbf{case}\ x\ \textbf{of}\ \{S(y) \to g(y)\}$$

матрица $\mathcal{M}_2[\![q]\!]$ (описывающая соотношение между аргументами f и g) имеет вид $\begin{bmatrix} < \end{bmatrix}$. Если вычислять значение $f(\inf)$ по этому определению ($\inf = S(S(\ldots))$), то должен будет произойти вызов $g(\inf)$, однако $f(\inf) \not\rightsquigarrow g(\inf)$, потому что $\inf \not< \inf$. Для того, чтобы обойти эту проблему, мы будем вычислять значения для бесконечных состояний через предел $f(\overline{a_i}) = \lim_{(\overline{x_i}) \to (\overline{a_i})} f(\overline{x_i})$, и именно поэтому нам в определении $\rightsquigarrow$ понадобился случай простого уменьшения аргументов без матрицы.

Определение $\rightsquigarrow$ обобщается и для его транзитивного замыкания $\rightsquigarrow^+$.

Утверждение 37. Если $f(\overline{a_i}) \rightsquigarrow^+ g(\overline{b_j})$, то $f = g$ и $(\overline{b_i}) \sqsubset (\overline{a_i})$ или для некоторого пути p из f в g с матрицей $M = \mathcal{M}[\![p]\!]$ верно, что $(\overline{b_j})\ M\ (\overline{a_i})$.

Доказательство. Докажем по индукции. Для случая $f(\overline{a_i}) \rightsquigarrow g(\overline{b_j})$ утверждение выполняется по определению.

Пусть теперь $f(\overline{a_i}) \rightsquigarrow^+ g(\overline{b_j}) \rightsquigarrow h(\overline{c_l})$ Рассмотрим четыре случая:

- $f = g$ и $(\overline{b_i}) \sqsubset (\overline{a_i})$, и $g = h$ и $(\overline{c_i}) \sqsubset (\overline{b_i})$. Тогда по транзитивности $f = h$ и $(\overline{c_i}) \sqsubset (\overline{a_i})$.

- $(\overline{b_j})\ M_1\ (\overline{a_i})$, где M_1 — матрица пути p_1 из f в g, и $(\overline{c_l})\ M_2\ (\overline{b_j})$, где M_2 — матрица пути p_2 из g в h. Тогда $(\overline{c_l})\ M\ (\overline{a_i})$, где $M = M_2 \cdot M_1$ — матрица пути $p1; p2$.

- $(\overline{b_j})\ M\ (\overline{a_i})$, где M — матрица пути p из f в g, и $g = h$ и $(\overline{c_j}) \sqsubset (\overline{b_j})$. Тогда т.к. $b_j\ M(j,i)\ a_i$ и $M(j,i) \in \{<, \leq, ?\}$, то из того, что $c_j \sqsubseteq b_j$ следует, что $c_j\ M(j,i)\ a_i$, а значит выполняется $(\overline{c_j})\ M\ (\overline{a_i})$. Здесь как раз важен тот факт, что определение $\leq$ включает в себя случай $\sqsubseteq$.

- $(\overline{c_l})\ M\ (\overline{b_j})$, где M — матрица пути p из g в h, и $f = g$ и $(\overline{b_j}) \sqsubset (\overline{a_j})$. Этот случай аналогичен предыдущему.

□

Отметим, что если $(\overline{b_i}) \sqsubset (\overline{a_i})$, то $(\overline{b_i}) \prec (\overline{a_i})$ и $\theta(\overline{b_i}) \prec \theta(\overline{a_i})$ для любой биективной перестановки θ.

Утверждение 38. Если для конечной полипрограммы выполнено достаточное условие структурной рекурсии, то множество состояний программы с *конечными аргументами* (но, возможно, частичными) является фундированным относительно отношения $\rightsquigarrow^+$.

Доказательство. Пусть $s_1 \rightsquigarrow^+ s_2 \rightsquigarrow^+ \ldots$ — бесконечная последовательность состояний. Так как полипрограмма конечна, в ней содержится конечное количество функций, поэтому хотя бы одна (пусть f) будет встречаться бесконечное число раз, а значит можно выделить бесконечную подпоследовательность $f(\overline{a_i^1}) \rightsquigarrow^+ f(\overline{a_i^2}) \rightsquigarrow^+ \ldots$. Используя предыдущее утверждение, получаем, что соседние значения $(\overline{a_i^j}), (\overline{a_i^{j+1}})$ либо связаны соотношениями, определяющимися матрицами путей из f в f, а значит по достаточному условию структурной рекурсии существует θ (общее для всей последовательности) такое, что $\theta(\overline{a_i^{j+1}}) \prec \theta(\overline{a_i^j})$, либо выполняется $(\overline{a_i^{j+1}}) \sqsubset (\overline{a_i^j})$, а значит также $\theta(\overline{a_i^{j+1}}) \prec \theta(\overline{a_i^j})$. Так как множество кортежей *конечных* элементов $\mathbb{A}$ является фундированным относительно $\succ$, то последовательность $(\overline{a_i^1}), (\overline{a_i^2}), \ldots$ является конечной, а значит и $f(\overline{a_i^1}) \rightsquigarrow^+ f(\overline{a_i^2}) \rightsquigarrow^+ \ldots$ также является конечной. Получаем противоречие, значит бесконечных убывающих последовательностей состояний нет и множество состояний с конечными аргументами фундировано. □

Теперь для доказательства единственности модели мы можем воспользоваться трансфинитной индукцией.

Теорема 1. Если для конечной полипрограммы выполнено достаточное условие структурной рекурсии и каждая её функция имеет не более одного определения, то для каждой интерпретации $\eta : W \to \mathbb{D}$ её функций без определений она имеет ровно одну модель $\mu(\eta)$.

Случай структурно-защищённой рекурсии и случай полноценных полипрограмм (с произвольным количеством определений) будет простым следствием данной теоремы.

Доказательство. При помощи трансфинитной рекурсии/индукции мы для каждого состояния программы $f(\overline{a_i})$ (в т.ч. бесконечного) построим четыре вещи:

1. Некоторое значение результата $\mu(f)(\overline{a_i})$.
2. Доказательство, что для единственного определения q функции f выполняется $[\![q]\!]_\mu$, или, если $f \in W$, то $\mu(f)(\overline{a_i}) = \eta(f)(\overline{a_i})$.
3. Доказательство, что если μ' — модель, согласованная с η, то $\mu'(f)(\overline{a_i}) = \mu(f)(\overline{a_i})$.
4. Доказательство, что $\mu(f)$ непрерывна вплоть до $\overline{a_i}$.

Обозначим все эти четыре вещи $P(s)$, где s — состояние. Т.к. множество *конечных* состояний s фундировано относительно $\rightsquigarrow^+$, то чтобы построить $P(s)$ для *конечных* s по трансфинитной индукции, необходимо построить $P(s)$ для каждого конечного s при условии, что для всех конечных s' таких, что $s \rightsquigarrow^+ s'$, построено $P(s')$, т.е. доказать, что:

$$(\forall s' \,.\, (s \rightsquigarrow^+ s' \wedge s' \text{ конечное}) \to P(s')) \to P(s)$$

Однако этого недостаточно для того, чтобы построить $P(S)$ для бесконечных состояний S. Хуже того, для построения $P(s)$ для конечных состояний нам будет необходимо, чтобы уже были построены $P(S')$ для бесконечных состояний S' таких, что $s \rightsquigarrow^+ S'$, а не только для конечных.

Чтобы справиться с этими проблемами, мы также построим $P(S)$ для бесконечных S при условии, что у нас есть $P(s')$ для конечных s' таких, что $S \rightsquigarrow^+ s'$, т.е. докажем, что:

$$(\forall s' \,.\, (S \rightsquigarrow^+ s' \wedge s' \text{ конечное}) \to P(s')) \to P(S), \text{ где } S \text{ — бесконечное}$$

Это утверждение во-первых сразу даст нам $P(S)$ для бесконечных состояний, если построены $P(s)$ для конечных состояний, а во вторых оно будет использовано в качестве леммы при доказательстве шага индукции для конечного случая. Поэтому мы начнём именно с бесконечного случая, т.е. с доказательства этой леммы.

Бесконечный случай Пусть $f(\overline{a_i})$ — состояние, причём среди a_i есть бесконечные. Пусть для всех конечных s' таких, что $f(\overline{a_i}) \rightsquigarrow^+ s'$ построены вышеуказанные вещи, т.е. $P(s')$. Построим тогда $P(f(\overline{a_i}))$.

Т.к. нам доступны значения $\mu(f)$ только для некоторых *конечных* состояний, то мы не можем напрямую применять семантику определения для определения $\mu(f)(\overline{a_i})$ через подвыражения (это свело бы наше состояние к *бесконечным* состояниям). Поэтому мы воспользуемся следующей формулой, чтобы свести бесконечное состояние к счётному множеству конечных состояний:

$$\mu(f)(\overline{a_i}) = \sup\{\mu(f)(\overline{approx_n(a_i)}) \mid n \in \mathbb{N}\}$$

Это определение даёт нам первый пункт. При этом $f(\overline{a_i}) \rightsquigarrow f(\overline{approx_n(a_i)})$, т.к. $\overline{approx_n(a_i)} \sqsubset \overline{a_i}$ (что в свою очередь следует из бесконечности $\overline{a_i}$, для конечных значений строгого уменьшения не было бы). Также все $approx_n(a_i)$ конечные, а значит соответствующие значения нам доступны.

Легко доказывается третий пункт (единственность). Действительно, пусть μ' — модель полипрограммы, согласованная с η. Тогда $\mu'(f)$ непрерывна в $\overline{a_i}$ (т.к. в нашей семантике модели обязательно непрерывны), а значит верно,

что

$$\begin{aligned}
&\mu'(f)(\overline{a_i}) = \\
&\qquad \text{(по непрерывности)} \\
&= \sup\{\mu'(f)(\overline{approx_n(a_i)}) \mid n \in \mathbb{N}\} = \\
&\qquad \text{(по условию, т.к. значения аргументов конечны)} \\
&= \sup\{\mu(f)(\overline{approx_n(a_i)}) \mid n \in \mathbb{N}\} = \\
&\qquad \text{(по определению)} \\
&= \mu(f)(\overline{a_i})
\end{aligned}$$

Здесь мы воспользовались тем, что третий пункт уже доказан для конечных состояний s таких, что $f(\overline{a_i}) \rightsquigarrow^+ s$.

Докажем, что выполняется четвёртый пункт, т.е. $\mu(f)$ непрерывна вплоть до $\overline{a_i}$. Для простоты записи будем считать, что функция принимает один аргумент, т.е. вместо $\overline{a_i}$ будем писать просто a. Пусть $a' \sqsubseteq a$, причём a' является бесконечным (в противном случае утверждение следует по условию). Пусть монотонная последовательность $b_1 \sqsubseteq b_2 \sqsubseteq \ldots$ сходится к a' (при этом в последовательности могут быть как конечные, так и бесконечные элементы). Без потери общности будем считать, что $b_l \neq a$, а значит $b_l \sqsubset a$. Отметим, что в этом случае верно:

$$\begin{aligned}
&\mu(f)(b_l) = \sup\{\mu(f)(approx_n(b_l)) \mid n \in \mathbb{N}\} \\
&\mu(f)(a') = \sup\{\mu(f)(approx_n(a')) \mid n \in \mathbb{N}\}
\end{aligned}$$

Для конечных b_l это следует по условию из четвёртого пункта, т.к. $b_l \sqsubset a$, а значит $f(a) \rightsquigarrow^+ f(b_l)$. Для бесконечных b_l это верно по определению $\mu(f)(b_l)$ — этим определением можно пользоваться, т.к. $f(a) \rightsquigarrow^+ f(approx_n(b_l))$, и значит соответствующие значения $\mu(f)(approx_n(b_l))$ определены по условию.

Т.к. для всех l верно $approx_n(b_l) \sqsubseteq approx_n(b_{l+1})$, а по условию $\mu(f)$ монотонна в точках $approx_n(b_l)$ (четвёртый пункт), то верно, что

$$\mu(f)(approx_n(b_l)) \sqsubseteq \mu(f)(approx_n(b_{l+1}))$$

а значит последовательность супремумов $\mu(f)(b_1) \sqsubseteq \mu(f)(b_2) \sqsubseteq \ldots$ также монотонная.

Докажем, что $\mu(f)(a')$ является верхней гранью этой последовательности ($\mu(f)(b_1) \sqsubseteq \mu(f)(b_2) \sqsubseteq \ldots$). Действительно, для всех l верно $b_l \sqsubseteq a'$, а значит $approx_n(b_l) \sqsubseteq approx_n(a')$, поэтому по условию (четвёртый пункт, о непрерывности) $\mu(f)(approx_n(b_l)) \sqsubseteq \mu(f)(approx_n(a'))$, а значит супремумы соответствующих последовательностей находятся в том же отношении, т.е. $\mu(f)(b_l) \sqsubseteq \mu(f)(a')$.

Теперь докажем, что верхняя грань $\mu(f)(a')$ — наименьшая. Для этого сначала построим последовательность $approx_{m(n)}(b_{m(n)})$ такую, что $m(n)$ не убывают (а значит и вся последовательность неубывающая), а кроме того $approx_n(a') \sqsubseteq approx_{m(n)}(b_{m(n)})$. Для этого для каждого n рассмотрим последовательность $approx_{n+1}(b_l)$ (от l) — эта последовательность по непрерывности $approx$ сходится к $approx_{n+1}(a')$, а т.к. этот предел конечен, то начиная с некоторого l они все будут равны пределу: $approx_{n+1}(b_l) = approx_{n+1}(a') \sqsupseteq approx_n(a')$ для $l \geq L$. За $m(n)$ возьмём $\max\{L, m(n-1)+1, n+1\}$. Тогда будет выполнено

$$approx_n(a') \sqsubseteq approx_{m(n)}(b_{m(n)}) \sqsubseteq a'$$
$$approx_{m(n-1)}(b_{m(n-1)}) \sqsubseteq approx_{m(n)}(b_{m(n)})$$

Т.к. все элементы построенной последовательности $approx_{m(n)}(b_{m(n)})$ ограничены снизу приближениями a', а сверху самим a', то последовательность сходится к a'. Т.к. все элементы построенной последовательности конечны, получаем по монотонности $\mu(f)$, что $\mu(f)(approx_n(a')) \sqsubseteq \mu(f)(approx_{m(n)}(b_{m(n)}))$. Кроме того $\mu(f)(approx_{m(n)}(b_{m(n)})) \sqsubseteq \mu(f)(b_{m(n)})$ по определению $\mu(f)(b_{m(n)})$, а значит по транзитивности верно

$$\mu(f)(approx_n(a')) \sqsubseteq \mu(f)(b_{m(n)})$$

А так как $\mu(f)(b_{m(n)})$ как подпоследовательность сходится к $\sup\{\mu(f)(b_l)\}$, а $\mu(f)(approx_n(a'))$ сходится к $\mu(f)(a')$, то

$$\mu(f)(a') \sqsubseteq \sup\{\mu(f)(b_l)\}$$

А значит $\mu(f)(a')$ не больше, чем супремум последовательности $\mu(f)(b_l)$, а значит является её супремумом. Таким образом, мы доказали, что $\mu(f)$ непрерывна вплоть до $\overline{a_i}$.

Наконец докажем, что выполняется второй пункт, т.е. согласованность с определением или с η. Для этого рассмотрим различные случаи. В случае, когда $\xi f \equiv L(\overline{\theta_i g_i})$ — единственное определение f, мы рассмотрим ξ и θ_i отличные от тождественных только для случая $\xi f \equiv \mathbf{Id}(\theta g)$, в остальных случаях для простоты будем считать, что перестановки параметров тождественны (формально более общий случай можно свести к этому случаю введением промежуточных функций).

- Если $f \in W$, то $\mu(f)(\overline{a_i}) = \eta(f)(\overline{a_i})$ по непрерывности η и определению $\mu(f)$.
- Случай $\xi f \equiv \mathbf{Id}(\theta g)$. По Рис. 3.5 должно быть выполнено $\mu(f)(\overline{x_{\xi(i)}}) = \mu(g)(\overline{x_{\theta(j)}})$ и значит нужно доказать, что $\mu(f)(\overline{a_i}) = \mu(g)(\overline{h_j(\overline{a_i})})$, где

$$h_j(\overline{x_i}) = \begin{cases} x_i & \text{если } \theta(j) = \xi(i) \\ \bot & \text{иначе} \end{cases}$$

  (вместо $\bot$ можно поставить что угодно, эти значения не используются). Отметим, что h_j — монотонная и непрерывная функция, а значит если $\mu(g)$ непрерывна вплоть до $\overline{b_j} = \overline{h_j(\overline{a_i})}$, то их композиция $\mu(g)(\overline{h_j(\overline{x_i})})$ будет непрерывна вплоть до $\overline{a_i}$.

  Непрерывность $\mu(g)$ вплоть до $\overline{b_j}$ следует из четвёртого пункта, однако для его применения требуется, чтобы все $\mu(g)(\overline{y_j})$, где $\overline{y_j}$ конечны и $\overline{y_j} \sqsubseteq \overline{b_j}$, уже были определены по индукции, для чего надо доказать, что $f(\overline{a_i}) \rightsquigarrow g(\overline{b_j})$. Рассмотрим матрицу определения. Она будет равна $M = \theta E \xi$, где E — тождественная матрица (с $\leq$) на диагонали. Значит будет выполнено:

$$\begin{aligned} & b_j \ (\theta E \xi)(j,i) \ a_i = \\ & \quad = b_j \ E(\theta(j), \xi(i)) \ a_i = \\ & \quad = \begin{cases} (a_i \leq a_i) = 1 & \text{если } \theta(j) = \xi(i) \\ (\bot \ ? \ a_i) = 1 & \text{иначе} \end{cases} \end{aligned}$$

  А значит по определению $f(\overline{a_i}) \rightsquigarrow g(\overline{b_j})$, а значит $f(\overline{a_i}) \rightsquigarrow^+ g(\overline{y_j})$ для конечных y_j не превышающих b_j, и мы можем воспользоваться непре-

рывностью $\mu(g)$ вплоть до $\overline{b_j}$:

$$\begin{aligned}
&\mu(f)(\overline{a_i}) = \\
&\qquad \text{(по определению)} \\
&= \sup\{\mu(f)(\overline{approx_n(a_i)})\} = \\
&\qquad \text{(по условию (второй пункт))} \\
&= \sup\{\mu(g)(h_j(\overline{approx_n(a_i)}))\} = \\
&\qquad \text{(по непрерывности композиции } \mu(g) \text{ и } h_j) \\
&= \mu(g)(h_j(\overline{a_i}))
\end{aligned}$$

- Случай $f \equiv \mathbf{Call}(h, \overline{g_j})$. Некоторые g_j могут иметь вид $\theta_j v$, если есть определение $v \equiv \mathbf{Var}$. Матрицы для всех дочерних функциональных символов, кроме первого, являются тождественными, поэтому $f(\overline{a_i}) \rightsquigarrow g_j(\overline{a_i})$, и значит значения $\mu(g_j)(\overline{approx_n(a_i)})$ определены по условию, а значит определены и $b_j = \mu(g_j)(\overline{a_i})$, причём для $g_j = \theta_j v$ соответствующие значения равны $a_{\theta_j(1)}$. Матрица же для первой функции имеет такой вид, что для любых конечных y_j таких, что если $g_j = \theta_j v$, то $y_j \sqsubseteq a_{\theta_j(1)}$, выполняется $f(\overline{a_i}) \rightsquigarrow h(\overline{y_j})$, а значит $\mu(h)$ определена и непрерывна в точке $\overline{b_j}$, и поэтому композиция $\mu(h)(\mu(g_j)(\overline{x_i}))$ также непрерывна. Получаем:

$$\begin{aligned}
&\mu(f)(\overline{a_i}) = \\
&\qquad \text{(по определению)} \\
&= \sup\{\mu(f)(\overline{approx_n(a_i)})\} = \\
&\qquad \text{(по условию)} \\
&= \sup\{h(\overline{\mu(g_j)(\overline{approx_n(a_i)})})\} = \\
&\qquad \text{(по непрерывности композиции)} \\
&= h(\overline{\mu(g_j)(\overline{a_i})})
\end{aligned}$$

- Случай $f \equiv \mathbf{Case}_{\overline{C_j}}(h, \overline{g_j})$, где h либо просто функция, либо $h = \theta v$ и есть определение $v \equiv \mathbf{Var}$. Для простоты будем считать, что θ во втором

случае равно $id_{1\to n}$. По Рис. 3.5 нужно доказать следующее:

$$\mu(f)(\overline{a_i}) = \\ \quad \textbf{case}\ \mu(h)(\overline{a_i})\ \textbf{of} \\ \quad\quad \overline{C_j(\overline{y_l}) \to \mu(g_j)(\overline{y_l}, \overline{a_i})}$$

(Определение конструкции **case of** см. в Утверждении 36). По Рис. 5.5 матрицы определения имеют следующий вид:

$$M_1 = E$$

$$M_{1+k}(j,i) = \begin{cases} < & \text{если } h = \theta v,\ i = \theta(1) = 1 \text{ и } j \le \mathit{arity}(C_k) \\ \le & \text{если } i = j - \mathit{arity}(C_k) \\ ? & \text{иначе} \end{cases}$$

Если $h = \theta v$, то будет выполнено $f(\overline{a_i}) \rightsquigarrow v(a_1)$, а значит выполнено $\mu(h)(\overline{a_i}) = a_1$. Случаи, когда соответствующей ветви нет, тривиальны, поэтому рассмотрим случай $a_1 = C_k(\overline{b_l})$, тогда нужно доказать, что $\mu(f)(\overline{a_i}) = \mu(g_k)(\overline{b_l}, \overline{a_i})$. Матрица M_{1+k} такая, что $(\overline{b_l}, \overline{a_i})\ M_{1+k}\ (\overline{a_i})$, поэтому $f(\overline{a_i}) \rightsquigarrow g_k(\overline{b_l}, \overline{a_i})$, а значит $\mu(g_k)(\overline{b_l}, \overline{a_i})$ определено и непрерывно через приближения. Поэтому

$$\begin{aligned} &\mu(f)(\overline{a_i}) = \\ &\qquad \text{(по определению)} \\ &= \sup\{\mu(f)(\overline{\mathit{approx}_n(a_i)})\} = \\ &\qquad \text{(по условию)} \\ &= \sup\{\textbf{case}\ \mathit{approx}_n(a_1)\ \textbf{of}\{C_j(\overline{y_l}) \to \mu(g_j)(\overline{y_l}, \overline{\mathit{approx}_n(a_i)})\}\} = \\ &\qquad \text{(по виду } a_1,\ n \ge 1) \\ &= \sup\{\mu(g_j)(\overline{\mathit{approx}_{n-1}(b_l)}, \overline{\mathit{approx}_n(a_i)})\} = \\ &\qquad \text{(по непрерывности)} \\ &= \mu(g_j)(\overline{b_l}, \overline{a_i}) \end{aligned}$$

Случай $n = 0$ ведёт к значению $\perp$ и не интересен.

Пусть теперь h не является переменной. Тогда $f(\overline{a_i}) \rightsquigarrow h(\overline{a_i})$, а кроме

того $f(\overline{a_i}) \rightsquigarrow g_k(\overline{y_l}, \overline{a_i})$ для любых y_l, а значит определены их приближения и выполняется непрерывность.

$$
\begin{aligned}
&\mu(f)(\overline{a_i}) = \\
&\qquad \text{(по определению)} \\
&= \sup\{\mu(f)(\overline{approx_n(a_i)})\} = \\
&\qquad \text{(по условию)} \\
&= \sup\{\textbf{case } \mu(h)(\overline{approx_n(a_i)}) \textbf{ of}\{ \\
&\qquad\qquad C_j(\overline{y_l}) \to \mu(g_j)(\overline{y_l}, \overline{approx_n(a_i)})\}\} = \\
&\qquad \text{(по непрерывности } g_j\text{, } h \text{ и Утверждению 36)} \\
&= \textbf{case } \mu(h)(\overline{a_i}) \textbf{ of}\{C_j(\overline{y_l}) \to \mu(g_j)(\overline{y_l}, \overline{a_i})\}\} =
\end{aligned}
$$

- Случаи **Var** и **Cons** аналогичны.

Таким образом мы доказали второй пункт для бесконечного случая, и значит все пункты для бесконечного случая доказаны.

Конечный случай Пусть все a_i конечные и для всех конечных $g(\overline{b_i})$ таких, что $f(\overline{a_i}) \rightsquigarrow^+ g(\overline{b_i})$, построены вышеуказанные объекты (это гипотеза индукции). Используя бесконечный случай как лемму, получаем, что вышеуказанные объекты также построены и для бесконечных состояний S таких, что $f(\overline{a_i}) \rightsquigarrow^+ S$, потому что они построены для всех конечных состояний s таких, что $S \rightsquigarrow^+ s$. Также будем ссылаться на вышеуказанные объекты для бесконечных состояний со словами “по гипотезе индукции”, хотя формально они не входят в гипотезу индукции.

Если для бесконечного случая мы определяли значение $\mu(f)$ через супремум, а затем доказывали непрерывность и согласованность с определениями, то для конечных состояний мы сделаем наоборот: определим их результаты в соответствии с семантикой определений, а затем докажем непрерывность. Третий пункт (единственность) будет следовать из второго (согласованность с определением) и из единственности значений подвыражений по гипотезе индукции.

Отметим, что в конечном случае для доказательства непрерывности надо только доказать монотонность в точке $\overline{a_i}$ (т.е. что для любых $\overline{a'_i} \sqsubseteq \overline{a_i}$ верно $\mu(f)(\overline{a'_i}) \sqsubseteq \mu(f)(\overline{a_i})$). Непрерывность для меньших значений будет следовать

по гипотезе индукции, а непрерывность в точке $\overline{a_i}$ автоматически следует из конечности и монотонности.

Рассмотрим те же самые случаи, что и при доказательстве согласованности для бесконечных состояний. Т.к. все случаи аналогичны, мы рассмотрим только несколько

- Если $f \in W$, то $\mu(f) = \eta(f)$, а значит $\mu(f)$ непрерывна по непрерывности $\eta(f)$.
- Случай $\xi f = \mathbf{Id}(\theta g)$. Тогда $\mu(f)(\overline{a_i}) = \mu(g)(\overline{h_j(\overline{a_i})})$, где

$$h_j(\overline{x_i}) = \begin{cases} x_i & \text{если } \theta(j) = \xi(i) \\ \bot & \text{иначе} \end{cases}$$

  Тогда для любых $\overline{a'_i} \sqsubseteq \overline{a_i}$ будет выполнено

$$\begin{aligned} &\mu(f)(\overline{a'_i}) = \\ &\qquad \text{(по гипотезе индукции)} \\ &= \mu(g)(\overline{h_j(\overline{a'_i})}) \sqsubseteq \\ &\qquad \text{(по гипотезе индукции)} \\ &\sqsubseteq \mu(g)(\overline{h_j(\overline{a_i})}) = \\ &= \mu(f)(\overline{a_i}) \end{aligned}$$

- Случай $f \equiv \mathbf{Call}(h, \overline{g_j})$. Некоторые g_j могут иметь вид $\theta_j v$, если есть определение $v \equiv \mathbf{Var}$. По гипотезе индукции $\mu(g_j)(\overline{a'_i}) \sqsubseteq \mu(g_j)(\overline{a_i})$, а значит и $h(\overline{\mu(g_j)(\overline{a'_i})}) \sqsubseteq h(\overline{\mu(g_j)(\overline{a_i})})$, т.к. по гипотезе индукции h монотонна вплоть до точки $\overline{\mu(g_j)(\overline{a_i})}$. Поэтому:

$$\begin{aligned} &\mu(f)(\overline{a'_i}) = \\ &\qquad \text{(по гипотезе индукции)} \\ &= h(\overline{\mu(g_j)(\overline{a'_i})}) \sqsubseteq \\ &\qquad \text{(по гипотезе индукции и монотонности композиции)} \\ &= h(\overline{\mu(g_j)(\overline{a_i})}) = \\ &= \mu(f)(\overline{a_i}) \end{aligned}$$

- Все остальные случаи аналогичны.

Окончание Таким образом мы построили значения $\mu(f)(\overline{a_i})$ для всех f и всех $\overline{a_i}$. При этом все эти значения согласованы с определениями и с η, а значит мы построили модель. Кроме того, любая другая модель μ' будет равна модели μ, т.е. модель единственная.

□

Случай структурно-защищённой рекурсии является простым следствием доказанной теоремы.

Следствие 1. Если для конечной полипрограммы выполнено достаточное условие *структурной-защищённой* рекурсии и каждая её функция имеет не более одного определения, то для каждой интерпретации $\eta : W \to \mathbb{D}$ её функций без определений она имеет ровно одну модель $\mu(\eta)$.

Доказательство. Пусть G — исходная полипрограмма и $H = \mathcal{A}[\![G]\!]$, W_H — множество функций H без определений, отличающееся от W только арностью функциональных символов (арность больше на 1).

Выкинем из H все определения функции inf, получим полипрограмму H', удовлетворяющую достаточному условию структурной рекурсии. Значит по доказанной теореме она имеет ровно одну модель $\mu'(\eta')$ для каждой $\eta' : W_H \cup \{\text{inf}\} \to \mathbb{D}$. Т.к. все определения inf в H имеют вид $\text{inf} \equiv S(\text{inf})$, то существует ровно одно возможное семантическое значение этой функции, удовлетворяющее этим определениям, а значит по композиционности семантики для каждой $\eta_H : W_H \to \mathbb{D}$ существует ровно одна модель $\mu_H(\eta_H)$ полипрограммы H.

Теперь возьмём $\eta : W \to \mathbb{D}$ и построим $\eta_H : W_H \to \mathbb{D}$ так, что $\eta_H(f)(n, \overline{x_i}) = \mathit{approx}(n, \eta(f)(\overline{x_i}))$. Тогда для H существует ровно одна модель μ_H, согласованная с η_H, и по Утверждению 35 её соответствует модель $\mu \models G$, причём для $f \in W$ выполнено

$$\begin{aligned}\mu(f)(\overline{x_i}) &= \mu_H(f)(\infty, \overline{x_i}) = \eta_H(f)(\infty, \overline{x_i}) = \\ &= \mathit{approx}(\infty, \eta(f)(\overline{x_i})) = \eta(f)(\overline{x_i})\end{aligned}$$

Т.е. μ согласована с η.

Пусть теперь существует ещё одна модель $\mu_1 \models G$, согласованная с η. Тогда по ней можно построить модель $\mu_{H1} \models H$, причём $\mu_{H1} \neq \mu_H$ по по Утверждению 35. Однако по тому же утверждению для $f \in W$ выполнено $\mu_{H1}(f)(n, \overline{x_i}) = approx(n, \mu_1(f)(\overline{x_i})) = \eta_H(f)(n, \overline{x_i})$, а значит μ_{H1} также согласована с η_H, поэтому $\mu_{H1} = \mu_H$. Получаем противоречие, значит модель μ единственная. □

Если ослабить требование единственности определений, то существование модели нельзя гарантировать, однако их будет всё равно не более одной.

Следствие 2. Если для конечной полипрограммы выполнено достаточное условие структурной или структурной-защищённой рекурсии, то для каждой интерпретации $\eta : W \to \mathbb{D}$ её функций без определений она имеет *не более одной модели* $\mu(\eta)$.

5.4 Выводы

В этой главе было описано преобразование слияние по бисимуляции, фактически выполняющее доказательство эквивалентности функций по индукции. В отличие от локальных правил преобразования, для выполнения которых требуется найти несколько соседних определений некоторого довольно фиксированного вида, для выполнения слияния по бисимуляции надо найти в полипрограмме два фрагмента, находящихся в отношении бисимуляции. В главе был описан алгоритм поиска бисимуляций, и было доказано, что в результате он действительно находит два фрагмента, находящихся в отношении бисимуляции (отметим, что дополнительная сложность при этом исходит из необходимости искать бисимуляции с точностью до перестановок параметров). Кроме того, было доказано, что если в полипрограмме была найдена бисимуляция, и она удовлетворяет некоторым дополнительным условиям (которые можно проверить алгоритмически), то соответствующие функции полипрограммы можно слить как эквивалентные (особо стоит отметить, что это верно для семантики с бесконечными и частичными данными, поскольку обычно этот метод применяется только для тотальных языков с конечными данными или с конечными и бесконечными, но строго разделёнными при помощи типизации). Условия, которым должна удовлетворять найденная бисимуляция, позволяет проводить как доказательство по индукции, так и по

коиндукции.

Глава 6

Реализация и экспериментальные результаты

Насыщение равенствами может применяться для разных задач, две наиболее известные — оптимизация и доказательство эквивалентности. Доказательство эквивалентности является очень естественной задачей для насыщения равенствами, поскольку суть насыщения равенствами как раз состоит в выводе новых эквивалентностей из уже существующих. Поэтому в данной работе мы и рассматриваем построение индуктивного прувера для доказательства эквивалентности на основе насыщения равенствами.

Более конкретно задача формулируется следующим образом: даны две функции на входном языке (язык из Главы 2), определить, являются ли они эквивалентными. Отметим, что нас будет интересовать только положительный ответ на поставленный вопрос: наш набор тестов состоит только из верных эквивалентностей, на которых прувер должен выдать “да”, если может их доказать и ничего не выдать или зациклиться/выйти по таймеру, если не может. Задача опровержения эквивалентности является более простой и в любом случае решается внутри прувера (например, при отсечении заведомо нерезультативных ветвей при поиске бисимуляции), однако она нас в данной работе не интересует сама по себе.

В этой главе мы приведём результаты тестирования нашего прувера на наборе примеров и сравним результаты его работы с некоторыми другими пруверами.

6.1 Практическая реализация прувера

Для практической проверки методов, описанных в предыдущих главах, был разработан экспериментальный прувер, решающий приведённую задачу. Прувер написан на языке Scala и имеет открытый исходный код [24]. Работает он следующим образом:

- Сначала над программой производятся некоторые упрощающие преобразования, самое главное из которых — дефункционализация, позволяющая избавиться от функций высшего порядка. Это позволяет решать задачу эквивалентности для языка высшего порядка не смотря на то, что сам прувер внутри работает с языком первого порядка.
- Программа преобразуется в бесточечно-расчленённое представление, как описано в Главе 3. Утверждение, которое требуется доказать преобразуется в форму $\theta f \equiv \xi g$, где f и g являются функциями полипрограммы.
- Производится преобразование полипрограммы (насыщение) до тех пор, пока не будет доказано целевое утверждение (подробнее описано ниже).

6.1.1 Представление полипрограммы

Наш прувер написан в императивном стиле, и преобразования производятся при помощи модификации полипрограммы. Полипрограмма представлена мутабельным множеством функций и определений. Каждая функция является объектом, причём функция может находиться в одном из двух состояний: либо она является обычной функцией, содержащей мутабельное множество инцидентных определений (в которых она упоминается слева или справа), либо она является косвенной функцией, содержащей ссылку на функцию с перестановкой, т.е. пару (θ, f), где θ — некоторая перестановка параметров, а f — какая-то другая функция. Функция может перейти из состояния обычной функции в состояние косвенной функции, но не наоборот. Это сделано для того, чтобы можно было выполнять применение конгруэнтности, и при

этом информация о том, что одна функция была слита с другой распространилась во все места, где она используется. Также каждая функция содержит информацию о её арности (арность может уменьшаться).

Каждое определение представляет из себя тройку (l, s, d), где l — метка, означающая вид определения (**Var**, $\mathbf{Cons}_C$, $\mathbf{Case}_{\overline{C_i}}$, **Call** или **Id**), $s = (\theta, f)$ — функция с перестановкой параметров, стоящая слева от $\equiv$, а d — список функций с перестановками параметров, которые упомянуты справа от $\equiv$. Отметим, что в нашей реализации перестановка параметров при правой части определения всегда равна id, поэтому мы её не храним. Это связано с тем, что мы всегда производим редукцию арности при добавлении определения. На определениях определена операция замены косвенных функций на обычные, т.е. замена пар вида (θ, f) на $(\theta \circ \xi, g)$, где f — косвенная функция, ссылающаяся на (ξ, g) — будем называть эту операцию устранением косвенности.

Над полипрограммой определены следующие основные операции:

- Операция добавления определения, принимающая на вход определение и добавляющее его в полипрограмму. Для этого в определении сначала производится устранение косвенности, потом производится приведение определения к каноническому виду (Раздел 3.3) и упрощение при помощи правил (call-to-permutation), (red-call-var), (red-case-cons) и (red-case-cons-bot) — так как эти правила модифицируют только одно определение, то их можно применять как бы до фактического добавления этого определения в полипрограмму. Если в результате оказывается, что определение имеет вид $\theta f \equiv \mathbf{Id}(\xi g)$, где f и g различаются, то вместо добавления определения мы осуществляем слияние функций (т.е. конгруэнтность). В противном случае производится применение правил (inj-cons) и (inj-case) (если они применимы). Отметим, что если данные правила применимы, то фактическое добавление нового определения не требуется, поскольку в полипрограмме уже есть эквивалентное ему определение (здесь неявно применяется правило (remove-dups)). Если же эти правила неприменимы, то производится фактическое добавление определения, если такого же определения ещё нет (здесь также неявно применяется (remove-dups)). Сразу после добавления определения происходит редукция арности и упрощение инцидентных определений по правилам (call-to-permutation), (red-call-var), (red-case-cons) и

(red-case-cons-bot) — это необходимо сделать, если добавленное определение имеет метку **Var** или **Cons**, поскольку такое определение может открыть возможность применения этих правил.

- Служебная операция слияния функций (конгруэнтность), принимающая на вход две функции вместе с перестановками параметров (θf и ξg) и переклеивающая определения, инцидентные функции f, к функции g (или наоборот). При этом функция f становится косвенной функцией, ссылающейся на g. Отметим, что если f и g совпадают (или являются косвенными функциями, ссылающимися на одну и ту же функцию), операция ничего не делает — это значит, что добавление равенств вида $\theta f \equiv \mathbf{Id}(\xi g)$ следует производить при помощи добавления соответствующего определения. Для выполнения слияния функций сначала производится редукция арности как если бы в полипрограмме были определения вида $\theta f \equiv \mathbf{Id}(\xi g)$ и $\xi g \equiv \mathbf{Id}(\theta f)$ (на самом деле они фактически временно добавляются в полипрограмму), затем, если функции не были слиты в результате редукции арности и сопутствующих упрощений, происходит фактическое переклеивание определений. После этого производится упрощение определений, инцидентных слитой функции, а затем происходит слияние функций-предков по правилу (trans) и функций-потомков по правилам (inj-cons) и (inj-case).

Завершаемость данных операций следует из Утверждения 27.

6.1.2 Процесс насыщения полипрограммы

Для того, чтобы применять правила только к тем частям полипрограммы, которые могут дать новые определения, организовано множество новых определений, в которые попадают все определения, которые добавляются в полипрограмму. Таким образом, после преобразования исходной программы в полипрограмму во множестве новых определений содержатся все определения полипрограммы.

Процесс насыщения полипрограммы состоит в повторении следующих действий:

- Из множества новых определений вынимаются все определения и для каждого из них проводятся следующие действия:

  - Во-первых, в определении производится устранение косвенности, и оно приводится к канонической форме и упрощается по правилам (call-to-permutation), (red-call-var), (red-case-cons), (red-case-cons-bot). Это необходимо сделать, т.к. между добавлением определения во множество новых определений и извлечением его оттуда в полипрограмму могли быть добавлены новые определения, что могло привести к изменению исходного определения в полипрограмме.
  - Затем происходит проверка, что определение (уже в упрощённом виде) всё ещё присутствует в полипрограмме. Если не присутствует, то дальше с ним ничего делать не нужно.
  - Наконец, для каждого фрагмента полипрограммы, содержащего данное определение, и каждого насыщающего правила, применимого к этому фрагменту, строится множество определений, которые следует добавить в полипрограмму. Однако это множество определений добавляется в полипрограмму не сразу, а после того, как аналогичные множества будут построены для всех новых определений, иначе они могут влиять друг на друга недетерминированным образом. Поэтому эти определения помещаются во временный буфер.

- Определения, собранные во временном буфере в процессе применения насыщающих правил на предыдущем шаге, добавляются в полипрограмму при помощи метода добавления определения. При этом добавление новых определений перемежается с их упрощением упрощающими правилами, однако это никак не влияет на детерминированность, поскольку на этом этапе добавление определений происходит безусловно (т.е. даже если правило, по которому определение оказалось во множестве добавляемых определений становится больше неприменимым, определение всё равно будет добавлено). Фактически добавленные определения (после всех упрощений) помещаются также и во множество новых определений.

- Производится насыщение по коммутативности. Для этого берутся все определения из множества новых определений (при этом они не удаляются оттуда), и аналогично двум предыдущим пунктам ко всем фрагментам полипрограммы, содержащим эти определения, применяется пра-

вило (commutativity). Новые определения добавляются в полипрограмму (также через временный буфер), а также во множество новых определений. Данный пункт повторяется до тех пор, пока не перестанут порождаться новые определения. Отметим, что данный пункт является необязательным на практике, т.к. коммутативность в решаемых нами задачах встречается редко, поэтому мы производим его лишь для лучшего соответствия теории из Главы 4.

- Далее производится слияние по бисимуляции, при этом происходит следующее:
  - Берутся все пары функций полипрограммы, из них исключаются те пары функций, слияние которых не приведёт к слиянию других функций по транзитивности или инъективности, и которые при этом не входят в целевое утверждение (т.е. доказательство их эквивалентности в рамках нашего метода бесполезно). Отметим, что рассмотрение всех пар позволяет найти некоторые леммы, необходимые для доказательства целевого утверждения, однако большинство наших примеров берутся и без лемм, поэтому в нашей реализации есть флаг, ограничивающий доказательство по индукции только целевым утверждением.
  - Пары сортируются в соответствии с эвристической метрикой похожести. Упрощённо похожесть считается исходя из глубины совпадающих поддеревьев, растущих из двух функций. Сортировка требуется скорее для детерминированности, чем для реального повышения скорости работы, поскольку всё равно поиск бисимуляций производится для всех пар.
  - Для всех пар последовательно производится слияние по бисимуляции, т.е. происходит поиск фрагментов полипрограммы, находящихся в отношении бисимуляции и удовлетворяющих признаку структурно-защищённой рекурсии, и в случае успешного обнаружения таковых производится слияние функций. После каждого слияния функций выполняется насыщение по коммутативности. Отметим, что в нашей реализации результаты поиска фрагментов полипрограммы кэшируются, причём кэш является общим для всех пар функций, однако при обнаружении эквивалентных функ-

ций кэш очищается, поскольку при слиянии функций он может устареть.

Данные действия повторяются до тех пор пока не будет доказано целевое утверждение.

6.1.3 Процесс слияния по бисимуляции

Процесс слияния по бисимуляции в целом соответствует описанию из Главы 5 и выглядит следующим образом:

- Сначала происходит поиск пребисимуляции при помощи обхода полипрограммы в глубину одновременно из двух функций. Здесь есть несколько деталей реализации:
  - Во-первых поиск продолжается только до того, как будет найдена первая пребисимуляция, полное множество пребисимуляций не строится. Из-за этого становится важным порядок рассмотрения определений, который фиксируется также в соответствии с метрикой похожести. Кроме того, из-за того, что не любая пребисимуляция влечёт эквивалентность функций, это решение может снизить мощность прувера, однако на практике такие случаи происходят редко.
  - Условия структурно-защищённой рекурсии частично проверяются уже в процессе поиска пребисимуляции.
  - В процессе поиска пребисимуляции результаты поиска кэшируются. Это даёт небольшое ускорение на некоторых примерах, если включено рассмотрение всех пар функций.
- Затем производится вычисление правильных переименований в найденной пребисимуляции, этот этап может завершиться неудачей, но редко. В результате получается бисимуляция фрагментов полипрограммы.
- Проверяются условия структурно-защищённой рекурсии. Эта проверка также может вернуть отрицательный результат, но такие случаи встречаются редко, т.к. частичная проверка этих условий в процессе построения пребисимуляции отфильтровывает большинство некорректных бисимуляций.

- Наконец, производится слияние соответствующих функций бисимуляции, если все проверки завершились с положительным результатом.

6.2 Результаты сравнения с другими пруверами

6.2.1 Набор примеров

Для экспериментальной оценки нашего прувера и сравнения его с похожими инструментами мы использовали два набора простых эквивалентностей, порядка 50 примеров каждый. Первый набор — наш собственный. Основная причина его создания заключается в том, что мы работаем с нетотальным языком, с частичными и бесконечными данными, обычно же индуктивные пруверы создаются для тотального случая с конечными данными, поэтому большинство эквивалентностей из других наборов просто неверны в интересующей нас семантике. Для второго набора мы использовали проект TIP (Tons of Inductive Problems) [81], включающий как собственные примеры, так и некоторые более старые наборы. Из входящих туда примеров мы отфильтровали (тестированием при помощи QuickCheck [12]) 66 примеров, вероятно верных в нашей нетотальной семантике, из них 16 оказались на самом деле неверны, т.к. для них были найдены контрпримеры при помощи прувера Zeno [69] (на этапе предварительного тестирования ни один из тестируемых пруверов не смог “доказать” эти 16 неверных эквивалентностей). Таким образом осталось 50 примеров из TIP.

Т.к. большинство пруверов работают с тотальной семантикой, а нас интересует частичная, то для них мы использовали сведение задачи доказательства эквивалентности в частичной семантике к задаче доказательства эквивалентности в тотальной семантике путём добавления дополнительного конструктора, означающего $\bot$, ко всем типам данных (и соответственно $\bot$ отображается в $\bot$ во всех сопоставлениях с образцом).

6.2.2 Сравниваемые инструменты

Мы использовали следующие пруверы на наборах тестовых примеров:

- **graphsc**. Graphsc (название расшифровывается как Graphical Super-Compiler, т.к. изначально наш инструмент задумывался как суперкомпилятор) — наш экспериментальный прувер. Мы тестировали его в двух

режимах: с поиском лемм, когда производится попытка доказать по индукции равенство всех пар функций из полипрограммы, и без поиска лемм, когда доказательство по индукции проводится только для целевого утверждения (в таблицах с результатами соответствующий столбец помечен как “без лемм”).

- **hipspec**. HipSpec [3] — индуктивный прувер для языка Haskell, который генерирует набор гипотез при помощи тестирования на случайных данных, доказывает их при помощи внешнего прувера первого порядка, а затем использует их как леммы для доказательства других гипотез и основной цели. HipSpec предполагает тотальность, поэтому мы использовали упомянутое выше преобразование исходных примеров для моделирования частичности. HipSpec позволяет использовать разные внешние пруверы, а именно Z3, CVC4, Alt Ergo, Vampire, E и SPASS. Из доступных нам пруверов наилучший результат для нашей задачи HipSpec показывает при использовании прувера SPASS, поэтому мы его и используем (однако Z3 работает чуть лучше на втором наборе примеров, взятых из TIP, хотя на двух наборах проигрывает). Отметим также, что результаты HipSpec’а чувствительны к тому, как реализованы экземпляры класса `Arbitrary` для типов данных, участвующих в программе. Мы генерируем данные экземпляры автоматически и при запуске используем флаг `--quick-check-size=10` для максимизации количества взятых HipSpec’ом примеров. Также мы используем максимальную глубину индукции 2 (флаг `-d2`), чтобы HipSpec брал некоторые примеры, требующие сильную индукцию. Кроме того, мы используем флаг `-U`, ставящий целевое утверждение на первое место — это позволяет успешнее брать примеры из наших наборов. HipSpec также можно запускать в режиме работы без лемм (для этого можно передать флаг `-size=0`, чтобы запретить генерацию термов для построения гипотез) — в этом случае результаты HipSpec фактически отражают чистый результат прувера SPASS, поскольку сам HipSpec в этом случае только лишь инстанцирует схему индукции. Соответствующий столбец также помечен как “без лемм”. В этом режиме для максимизации количества взятых примером мы увеличили тайм-аут одного запуска прувера до 5 секунд (`-t5`).

- **hosc**. HOSC [41] — суперкомпилятор, разработанный для анализа программ, в частности для доказательства эквивалентностей. Для доказательства эквивалентности он использует следующий метод: сначала производится суперкомпиляция обеих частей равенства по отдельности, потом две результирующие программы сравниваются друг с другом синтаксически [45; 49]. HOSC имеет возможность использовать метод двухуровневой суперкомпиляции [42; 46], способной находить и доказывать леммы, нужные для доказательства основной цели. Мы считали, что HOSC берёт пример, если он берёт его в одноуровневом или в двухуровневом режиме, в отличие от предыдущих примеров мы эти два случая не разделяем.

- **zeno**. Zeno [69] — индуктивный прувер для языка Haskell. Внутренне он очень похож на суперкомпилятор. Zeno предполагает, что язык тотальный, поэтому мы использовали описанный выше приём, сводящий задачу эквивалентности для нетотального языка к задаче эквивалентности для тотального языка.

6.2.3 Результаты

В таблицах приведено медианное (за три запуска) реальное время в секундах, которое тестируемые инструменты потратили на доказательство эквивалентностей. Мы ограничивали время работы пятью минутами, запуски, превышающие данное время, считались неуспешными. Тестирование проводилось на машине с процессором Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz и 6 Гб оперативной памяти (своп был отключён).

Первый (наш) набор примеров мы разделили на две группы: основная группа сравнительно простых эквивалентностей (Таблица 6.1) и группа более сложных примеров, требующих специальных возможностей (Таблица 6.2) — а именно нетривиальных обобщений, сильную индукцию и коиндукцию. В приведённых таблицах для некоторых примеров даны приблизительные формулировки целевых утверждений во втором столбце — они приблизительны в том смысле, что не учитывают, что мы работаем с нетотальным (а также нетипизированным) языком, реальные целевые утверждения обычно несколько сложнее (например, утверждение $f(x) = x$, где $f(x) = \mathbf{case}\ x\ \mathbf{of}\{\ S(x) \to S(f(x));\ Z \to Z\ \}$, в нашем языке некорректно, т.к. он нетипизированный и

x может принимать, например, значение C, а $f(C) = \bot$). Результаты тестирования на наборе примеров взятых из TIP приведены в Таблице 6.3. Общее количество взятых каждым из пруверов примеров приведено в Таблице 6.4.

Далее мы приведём анализ результатов тестирования пруверов на нашем наборе примеров.

Zeno и HOSC очень быстры благодаря тому, что работают “в глубину”. Zeno также очень мощный инструмент, способный соревноваться с более медленным HipSpec’ом. HOSC работает хуже потому, что не специализирован под данную задачу. Например, эквивалентность под названием `idle-simple` гораздо проще доказать, преобразуя обе части целевого утверждения одновременно, а не по отдельности. Также HOSC не может взять примеры `bool-eq` и `sadd-comm`, потому что они требуют преобразования, подобного (swap-case-case), которое обычно отсутствует в суперкомпиляторах.

Наш инструмент, Graphsc, находится где-то посередине: он медленнее, чем HOSC и Zeno, т.к. работает “в ширину”, но редко требует более 10 секунд. Интересно проанализировать примеры, не взятые нашим прувером. Из основной части нашего набора он не взял три примера, два из них, `ho/church-pred` и `ho/church-add` требуют более глубокой прогонки (мы можем взять их используя экспериментальный режим суперкомпиляции, в котором используется другой порядок применения преобразований, более близкий к применяемому в суперкомпиляторах — к сожалению, данный режим пока очень медленный и ведёт к провалам на других примерах). Тест `ho/filter-idemp` требует распространения большего количества информации, а именно того факта, что `p x` равно `True` в соответствующей ветви. Т.к. это выражение не является переменной, то мы эту информацию не распространяем (HOSC тоже её по умолчанию не распространяет, но имеет экспериментальный режим, в котором он это делает и берёт данный пример). В режиме работы без лемм наш прувер не берёт ещё четыре последних примера. Интересно, что другим пруверам на этих примерах леммы не требуются, по всей видимости, это связано с тем, что индукция, выраженная слиянием по бисимуляции, является более ограниченной в плане использования гипотезы индукции.

Теперь рассмотрим вторую часть нашего набора примеров (Таблица 6.2). Начнём с примеров, требующих нетривиальных обобщений. Будем называть обобщение тривиальным, если оно является просто отбрасыванием внешнего вызова функции, например, $f(g(a), h(b, c))$ тривиально обобщается до $f(x, y)$,

Таблица 6.1 Сравнение различных инструментов на основной части первого набора примеров

Название	Описание	graphsc	graphsc без лемм	hipspec	hipspec без лемм	zeno	hosc
add-assoc	x + (y + z) = (x + y) + z	1.6	1.5	1.3	1.1	0.4	0.5
append-assoc	x ++ (y ++ z) = (x ++ y) ++ z	1.8	1.7	1.6	1.2	0.4	0.5
double-add	double (x+y) = double x + double y	2.5	1.8	1.1	1.0	0.5	2.0
even-double	even (double x) = true	1.7	1.6	6.2	0.9	0.4	0.6
ho/concat-concat	concat (concat xs) = concat (map concat xs)	3.3	2.6	16.4	fail	0.5	0.6
ho/filter-append	filter p (xs ++ ys) = filter p xs ++ filter p ys	3.3	2.8	2.2	1.3	0.4	0.6
ho/map-append	map f (xs ++ ys) = map f xs ++ map f ys	2.1	2.1	2.6	1.1	0.4	0.6
ho/map-comp	map (f . g) xs = (map f . map g) xs	4.3	3.4	4.2	1.0	0.4	0.6
ho/map-concat	map f (concat x) = concat (map (map f) x)	2.9	2.3	19.4	fail	0.4	0.6
ho/map-filter	filter p (map f xs) = map f (filter (p . f) xs)	3.6	2.8	4.4	1.1	0.4	0.7
idnat-idemp	idNat (idNat x) = idNat x	1.5	1.5	0.9	0.8	0.4	0.5
take-drop	drop n (take n x) = []	2.4	2.0	1.7	1.2	0.4	0.6
take-length	take (length x) x = x	2.3	1.8	1.2	1.0	0.4	0.6
length-concat	length (concat x) = sum (map length x)	2.8	2.3	fail	fail	0.5	0.7
append-take-drop	take n x ++ drop n x = x	3.7	3.0	3.1	1.3	0.5	1.1
deepseq-idemp	deepseq x (deepseq x y) = deepseq x y	1.8	1.8	1.1	1.0	0.4	0.6
deepseq-s	deepseq x (S y) = deepseq x (S (deepseq x y))	2.2	2.0	1.0	0.9	5.3	0.7
mul-assoc	(x * y) * z = x * (y * z)	12.8	8.7	fail	fail	0.4	0.7
mul-distrib	(x*y) + (z*y) = (x + z)*y	4.0	3.4	21.9	fail	0.4	0.8
mul-double	x * double y = double (x*y)	5.7	4.0	21.5	fail	0.4	0.7
ho/fold-append	foldr f (foldr f a ys) xs = foldr f a (xs ++ ys)	2.2	1.9	20.4	1.3	fail	0.6
ho/church-id	unchurch (church x) = x	7.4	3.9	fail	0.9	2.7	0.6
ho/church-pred		fail	fail	fail	1.1	fail	0.6
ho/church-add		fail	fail	fail	1.6	0.4	2.3
idle-simple	idle x = idle (idle x)	1.4	1.3	0.8	0.8	0.4	fail
bool-eq		1.3	1.3	0.9	0.8	0.4	fail
sadd-comm		2.1	1.9	1.0	1.0	0.4	fail
ho/filter-idemp	filter p (filter p xs) = filter p xs	fail	fail	1.1	1.0	0.4	fail
even-slow-fast	even x = evenSlow x	1.8	1.7	1.1	fail	fail	fail
or-even-odd	even x || odd x = true	4.4	2.7	1.3	0.9	fail	fail
dummy		1.6	fail	1.0	1.0	0.4	fail
idle	idle x = deepseq x 0	1.5	fail	0.9	0.8	1.1	fail
quad-idle		1.9	fail	0.9	0.9	0.4	fail
exp-idle		4.6	fail	0.9	0.9	0.5	fail

Таблица 6.2 Сравнение инструментов на второй части первого набора примеров

Название	Описание	graphsc	graphsc без лемм	hipspec	hipspec без лемм	zeno	hosc
Примеры, требующие нетривиальных обобщений							
even-dbl-acc-lemma	even (doubleAcc x (S y)) = odd (doubleAcc x y)	fail	fail	1.2	1.1	0.4	0.6
nrev-idemp-nat		fail	fail	2.5	fail	0.4	fail
deepseq-add-comm		fail	fail	7.9	fail	fail	fail
even-double-acc	even (doubleAcc x 0) = true	fail	fail	fail	fail	fail	0.8
nrev-list	naiveReverse = reverse	fail	fail	18.6	fail	fail	fail
nrev-nat		fail	fail	51.1	fail	fail	fail
sadd-assoc		fail	fail	fail	fail	fail	fail
Примеры, требующие сильную индукцию							
add-assoc-bot		2.1	1.8	fail	fail	fail	0.6
double-half	double (half x) + mod2 x = x	4.4	2.9	1.1	0.9	fail	1.4
length-intersperse	length (intersperse x xs) = length (intersperse y xs)	fail	fail	1.7	1.6	fail	0.6
kmp-eq		fail	fail	fail	1.4	fail	1.2
Примеры, требующие коиндукцию							
inf	fix S = fix S	1.2	1.2	fail	fail	fail	0.4
shuffled-let		1.4	1.4	fail	fail	fail	0.5
shifted-cycle	cycle [A,B] = A : cycle [B,A]	4.0	3.4	fail	fail	fail	fail
ho/map-iterate	map f (iterate f a) = iterate f (f a)	fail	fail	fail	fail	fail	0.5

где $x = g(a)$ и $y = h(b, c)$. Наш инструмент поддерживает только такие тривиальные обобщения (потому что в полипрограмме все выражения уже представлены в таком виде, а кроме того разрешено преобразование любых дочерних функций), и их достаточно для того, чтобы взять довольно большое количество примеров. Однако в некоторых случаях нужны более сложные обобщения, например, для того, чтобы взять `even-dbl-acc-lemma`, нужно обобщить выражение odd (doubleAcc x (S (S y))) до odd (doubleAcc x z), где z = S (S y). Т.к. выражение odd (doubleAcc x z) является композицией двух функций, это обобщение нашему пруверу не доступно без применения дополнительных преобразований (у нас есть режим, включающий произвольные обобщения путём применения некоторых правил в обратную сторону, но он приводит к сильному разрастанию полипрограммы, хотя и помогает в некоторых случаях). В суперкомпиляторах, таких как HOSC, обычно используются наиболее тесные обобщения, которые часто помогают, однако для доказательства подобных эквивалентностей наилучшем инструментом из протестированных нами является HipSpec, в котором обобщения фактически берутся из найденных лемм.

Таблица 6.3 Сравнение инструментов на наборе примеров, взятых из TIP

Название	graphsc	graphsc без лемм	hipspec	hipspec без лемм	zeno	hosc
isaplanner/prop_02	10.3	4.2	3.8	1.8	0.5	0.8
isaplanner/prop_09	3.0	2.5	1.8	1.3	0.4	fail
isaplanner/prop_11	1.3	1.2	1.2	1.0	0.4	0.5
isaplanner/prop_13	1.4	1.3	1.7	1.5	0.4	0.5
isaplanner/prop_17	2.0	1.5	1.0	0.9	0.4	0.5
isaplanner/prop_22	219.3	48.2	1.3	1.2	0.4	fail
isaplanner/prop_23	fail	fail	1.0	1.0	0.4	fail
isaplanner/prop_31	2.1	1.8	1.3	1.2	0.4	0.5
isaplanner/prop_33	3.3	2.2	1.6	1.2	0.5	0.6
isaplanner/prop_39	4.6	3.0	3.0	1.9	0.4	fail
isaplanner/prop_40	1.3	1.3	1.2	1.0	0.4	1.9
isaplanner/prop_42	1.4	1.4	1.7	1.6	0.4	0.6
isaplanner/prop_44	1.3	1.4	125.8	1.4	0.4	0.6
isaplanner/prop_45	1.4	1.4	3.4	2.6	0.4	0.6
isaplanner/prop_46	1.3	1.3	1.8	1.0	0.4	0.5
isaplanner/prop_47	fail	fail	66.8	fail	2.8	fail
isaplanner/prop_50	6.2	5.3	1.5	1.1	0.4	fail
isaplanner/prop_55	fail	fail	2.8	1.6	0.4	fail
isaplanner/prop_67	2.7	2.3	1.2	1.0	0.5	fail
isaplanner/prop_80	6.5	5.7	2.8	1.6	0.5	1.2
isaplanner/prop_82	3.5	3.1	29.6	1.6	0.4	0.7
prod/prop_12	fail	fail	21.4	fail	0.5	fail
prod/prop_22	fail	fail	fail	fail	fail	fail
prod/prop_24	fail	fail	1.3	1.1	fail	fail
prod/prop_25	fail	fail	6.3	fail	fail	fail
prod/prop_27	fail	fail	20.9	fail	fail	fail
prod/prop_28	fail	fail	152.7	fail	fail	fail
tip2015/bin_plus_assoc	fail	fail	fail	fail	fail	fail
tip2015/bin_plus_comm	2.8	2.4	1.3	1.1	0.6	fail
tip2015/heap_minimum	fail	3.7	224.9	1.3	0.4	0.7
tip2015/int_add_ident_left	fail	fail	1.5	1.0	0.5	fail
tip2015/int_add_ident_right	fail	fail	1.5	1.0	0.4	fail
tip2015/int_mul_ident_left	fail	fail	fail	fail	0.4	fail
tip2015/int_mul_ident_right	fail	fail	124.9	fail	0.5	fail
tip2015/list_Interleave	fail	fail	1.5	0.9	0.4	fail
tip2015/list_weird_is_normal	1.7	1.7	3.3	0.9	0.5	0.6
tip2015/nat_alt_mul_assoc	fail	fail	fail	fail	fail	fail
tip2015/rotate_self	fail	fail	13.7	fail	fail	fail
tip2015/rotate_snoc_self	fail	fail	16.1	fail	fail	fail
tip2015/tree_Flatten1	fail	fail	fail	fail	fail	fail
tip2015/tree_Flatten2	fail	fail	145.7	fail	fail	fail
tip2015/tree_Flatten3	fail	fail	fail	fail	0.9	fail
tip2015/weird_nat_add3_assoc1	2.5	2.0	126.6	2.4	0.4	0.6
tip2015/weird_nat_add3_assoc2	2.7	2.2	fail	fail	2.8	0.6
tip2015/weird_nat_add3_assoc3	2.7	2.3	fail	fail	0.5	0.7
tip2015/weird_nat_add3acc_assoc2	fail	fail	fail	fail	fail	fail
tip2015/weird_nat_add3acc_comm12	fail	fail	fail	fail	fail	fail
tip2015/weird_nat_add3acc_rot	fail	fail	fail	fail	fail	fail
tip2015/weird_nat_mul2_assoc	fail	fail	fail	fail	fail	fail
tip2015/weird_nat_op_assoc	fail	fail	fail	fail	fail	fail

Таблица 6.4 Суммарное количество взятых примеров

Набор	graphsc	graphsc без лемм	hipspec	hipspec без лемм	zeno	hosc	всего
Наш набор	36	32	36	31	32	33	49
TIP	23	24	37	28	34	18	50
Всего	59	56	73	59	66	51	99

Теперь рассмотрим примеры, требующие сильную индукцию, а именно такую формулировку принципа индукции, которая может отбрасывать более одного конструктора за один шаг. Для Graphsc'а и HOSC'а это не проблема, т.к. эти инструменты не инстанцируют явно схемы индукции, а вместо этого проверяют условия корректности для графов доказательства. Однако Zeno и HipSpec инстанцируют схемы индукции, поэтому такие примеры являются для них проблемой. В случае HipSpec'а максимальная глубина индукции может быть увеличена, и мы использовали глубину равную 2, что позволило HipSpec'у взять два примера из этого набора, но за счёт некоторого увеличения времени работы на других примерах.

Наш инструмент не может пройти KMP-тест, поскольку он требует глубокой прогонки (и опять же, наш экспериментальный режим суперкомпиляции помогает и в этом случае). В случае примера `length-intersperse` наш инструмент не выявляет целевое утверждение как нечто, что стоит доказывать, поскольку обе его части эквивалентны с точностью до переименования, а значит изображаются в полипрограмме одним классом эквивалентности. Данная проблема выглядит чисто технической, однако пока непонятно, как её можно решить.

Последний набор примеров, который осталось разобрать — примеры, требующие коиндукцию. Коиндукция в настоящее время не поддерживается пруверами Zeno и HipSpec, хотя особых препятствий перед реализацией её поддержки в будущем нет. (Отметим, что коиндукцию можно закодировать через индукцию подобно тому, как мы это делаем в Главе 5, однако коиндуктивных примеров у нас довольно мало, поэтому мы не стали этого делать). Пример `ho/map-iterate` не берётся нашим инструментом, потому что кроме коиндукции он требует сложных обобщений.

В целом наш прувер работает хуже современных инструментов Zeno и HipSpec, главным образом за счёт того, что не умеет производить нетриви-

альные обобщения, однако при отключении лемм наш прувер проигрывает три примера HipSpec'у без лемм (т.е. фактически SPASS'у), что является неплохим результатом.

6.2.4 Эксперименты с Graphsc

В Приложении C приведены результаты тестирования Graphsc с модифицированными наборами правил преобразования полипрограмм, в частности, проверялось, насколько важна деструктивность правил. Главный вывод — без деструктивных правил прувер был бы гораздо менее эффективным. Также проверялась возможность отключить или ограничить некоторые другие правила, в частности, правило распространения позитивной информации не стоит делать деструктивным, потому что это приводит к тому, что становится невозможно взять некоторые примеры.

6.2.5 Эффективность полипрограмм по сравнению с прямым представлением

Т.к. полипрограмму можно рассматривать как представление множества программ более эффективное чем прямое представление этого множества, возникает вопрос, насколько оно более эффективно на практике. Для ответа на этот вопрос было проведено измерение количества программ, которые можно извлечь из полипрограммы на момент непосредственно перед слиянием по бисимуляции для выражений, составляющих левую и правую части целевых утверждений некоторых примеров. Результаты приведены в таблице 6.5, в колонках которой указаны следующие величины:

- **Количество определений в исходной программе** — количество определений в исходной программе после расчленения.
- **Количество определений в преобразованной полипрограмме** — количество определений в полипрограмме после применения преобразований (но до слияния по бисимуляции).
- **Количество программ** — сумма количества программ для левой части и количества программ для правой части целевого утверждения, которые можно извлечь из данной полипрограммы.

- **Суммарное количество определений в программах** — количество определений, нужных для представления этих программ в виде множества напрямую.

Из таблицы видно, что представление в виде полипрограммы значительно эффективнее, чем наивное представление, причём этот эффект тем сильнее, чем сложнее пример.

Таблица 6.5 Сравнение прямого представления множеств программ и в виде полипрограмм.

	Количество определений в исходной программе	Количество определений в преобразованной полипрограмме	Количество программ	Количество определений в программах
or-even-odd	9	82	> 100000	> 6961566
double-half	17	175	87128	5389154
append-take-drop	17	104	1050	59579
ho/concat-concat	9	57	466	19580
take-drop	9	36	45	1479
length-concat	16	59	62	1432
double-add	7	29	25	523
take-length	10	28	26	499
deepseq-s	2	22	21	313
add-assoc-bot	6	24	12	293
even-double	9	21	14	206
sadd-comm	11	20	8	112
append-assoc	3	13	7	96
add-assoc	3	13	7	78
deepseq-idemp	2	13	7	60
idnat-idemp	4	8	5	39
shuffled-let	6	6	4	36
even-slow-fast	8	12	5	32
inf	3	2	2	2

Построение программ проводилось следующим образом: сначала производилось преобразование полипрограммы как обычно вплоть до обнаружения бисимуляции, сливающей функции, представляющие левую и правую части целевого утверждения. Затем (в момент *перед* фактическим слиянием) для этих функций строился набор программ, определяющих данные функции, и удовлетворяющих условиям единственности. Построение таких программ проводилось аналогично поиску бисимуляций — фактически алгоритм построения программ для данной функции f повторяет алгоритм поиска бисимуляций для пары (f, f), но с запретом использования рефлексивности. Среди программ оставлялись только те, которые удовлетворяли достаточным условиям единственности решения.

В качестве примеров были выбраны примеры, которые берутся нашим

прувером без использования лемм, не использующие существенно функции высших порядков, и для которых вышеописанная процедура работает за приемлемое время (например, примеры, описывающие свойства умножения, производят слишком много программ, не удовлетворяющих условиям единственности, поэтому не были включены в таблицу).

6.3 Выводы

В данной главе была описана практическая реализация нашего прувера. Фактически она очень близко следует теоретическому описанию метода из предыдущих глав.

Было также приведено сравнение нашего прувера с некоторыми другими на наборе примеров. Набор примеров был частично составлен нами, частично взят из TIP. Главным недостатком нашего инструмента по результатам тестов является невозможность производить сложные обобщения, в остальном на выбранной задаче он работает примерно на уровне других инструментов.

Заключение

В настоящей работе были получены следующие результаты:

- На основе насыщения равенствами и суперкомпиляции был разработан метод преобразования множеств программ на нестрогом функциональном языке первого порядка.
  - Было предложено понятие полипрограммы, позволяющее представлять множества функциональных программ, и была определена семантика полипрограмм.
  - Был разработан набор локальных правил эквивалентных преобразований полипрограмм.
  - Было предложено глобальное преобразование слияние по бисимуляции, соответствующий доказательству эквивалентности функций по индукции и коиндукции.
- Разработанный метод был реализован в системе доказательства эквивалентности функций на нестрогом функциональном языке.
- Разработанная система доказательства эквивалентности была протестирована на наборе модельных задач.

Список литературы

1. *Abel A.* foetus – Termination Checker for Simple Functional Programs: Programming Lab Report / LMU München. — July 1998.
2. *Abel A.*, *Altenkrich T.* A Predicative Analysis of Structural Recursion // Journal of Functional Programming. — 2002. — Vol. 12, no. 1. — Pp. 1–41. — URL: `http://www.cs.cmu.edu/~abel/foetuswf.ps.gz`.
3. Automating Inductive Proofs Using Theory Exploration / K. Claessen, M. Johansson, D. Rosén, N. Smallbone // Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Vol. 7898 / ed. by M. P. Bonacina. — Springer, 2013. — Pp. 392–406. — (Lecture Notes in Computer Science). — ISBN 978-3-642-38573-5. — URL: `http://dx.doi.org/10.1007/978-3-642-38574-2`.
4. *Bachmair L.*, *Ganzinger H.* Equational Reasoning in Saturation-Based Theorem Proving // Automated Deduction: A Basis for Applications. Volume I, Foundations: Calculi and Methods / ed. by W. Bibel, P. H. Schmidt. — Dordrecht : Kluwer Academic Publishers, 1998.
5. *Banerjee*, *Heintze*, *Riecke* Design and Correctness of Program Transformations Based on Control-Flow Analysis // TACS: 4th International Conference on Theoretical Aspects of Computer Software. — 2001.
6. *Barrett C.*, *Tinelli C.* CVC3 // Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Vol. 4590 / ed. by W. Damm, H. Hermanns. — Springer-Verlag, July 2007. — Pp. 298–302. — (Lecture Notes in Computer Science). — Berlin, Germany.

7. *Barwise J.*, *Moss L. S.* Vicious Circles: On the Mathematics of Non-wellfounded Phenomena. — CSLI, 1996.

8. *Boyer R. S.*, *Moore J. S.* Proving theorems about LISP functions // Journal of the ACM (JACM). — 1975. — Vol. 22, no. 1. — Pp. 129–144.

9. *Bundy A.* The use of explicit plans to guide inductive proofs // Conf. on Automated Deduction (CADE 9). — 1987.

10. *Burstall R. M.*, *Darlington J.* A transformational system for developing recursive programs // Journal of the ACM. — 1977. — Jan. — Vol. 24, no. 1. — Pp. 44–67.

11. *Burstall R. M.* Proving properties of programs by structural induction // The Computer Journal. — 1969. — Vol. 12, no. 1. — Pp. 41–48.

12. *Claessen K.*, *Hughes J.* QuickCheck: a lightweight tool for random testing of Haskell programs // ACM SIGPLAN Notices. — 2011. — Apr. — Vol. 46, no. 4. — Pp. 53–64. — ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). — DOI: `http://dx.doi.org/10.1145/1988042.1988046`.

13. *Claessen K.*, *Smallbone N.*, *Hughes J.* QuickSpec: Guessing Formal Specifications Using Testing // Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings. Vol. 6143 / ed. by G. Fraser, A. Gargantini. — Springer, 2010. — Pp. 6–21. — (Lecture Notes in Computer Science). — ISBN 978-3-642-13976-5. — URL: `http://dx.doi.org/10.1007/978-3-642-13977-2`.

14. *Davis M.*, *Logemann G.*, *Loveland D. W.* A machine program for theorem-proving // Commun. ACM. — 1962. — Vol. 5, no. 7. — Pp. 394–397. — URL: `http://doi.acm.org/10.1145/368273.368557`.

15. *Detlefs*, *Nelson*, *Saxe* Simplify: A Theorem Prover for Program Checking // JACM: Journal of the ACM. — 2005. — Vol. 52.

16. DPLL(T): Fast Decision Procedures / H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli // Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts). Vol. 3114 / ed. by R. Alur, D. Peled. — Springer, 2004. — Pp. 175–188. — (Lecture Notes in Computer Science).

17. *Dutertre B.*, *De Moura L.* The yices smt solver // Tool paper at http://yices. csl. sri. com/tool-paper. pdf. — 2006. — Vol. 2, no. 2.

18. *Ehrig H.*, *Kreowski H.-J.* Parallelism of Manipulations in Multidimensional Information Structures // Mathematical Foundations of Computer Science. Vol. 45 / ed. by A. Mazurkiewicz. — 1976. — Pp. 284–293. — (Lecture Notes in Computer Science).

19. *Ehrig*, *Pfender*, *Schneider* Graph Grammars: An Algebraic Approach // FOCS: IEEE Symposium on Foundations of Computer Science (FOCS). — 1973.

20. *Emelianov P. G.* Analysis of the Equality Relations for the Program Terms // Lecture Notes in Computer Science. — 1996. — Vol. 1145. — Pp. 174–188. — ISSN 0302-9743.

21. Equality saturation: a new approach to optimization / R. Tate, M. Stepp, Z. Tatlock, S. Lerner // SIGPLAN Not. — New York, NY, USA, 2009. — Jan. — Vol. 44, issue 1. — Pp. 264–276. — ISSN 0362-1340. — URL: `http://doi.acm.org/10.1145/1594834.1480915`.

22. *Ganzinger H.* Saturation-Based Theorem Proving // Lecture Notes in Computer Science. — 1996. — Vol. 1099. — ISSN 0302-9743.

23. *Glück R.*, *Klimov A. V.* Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree // Static Analysis, Third International Workshop, WSA'93, Padova, Italy, September 22-24, 1993, Proceedings. Vol. 724 / ed. by P. Cousot, M. Falaschi, G. Filé, A. Rauzy. — Springer, 1993. — Pp. 112–123. — (Lecture Notes in Computer Science). — ISBN 3-540-57264-3. — URL: `http://dx.doi.org/10.1007/3-540-57264-3_34`.

24. Graphsc source code and the test suite. — `https://github.com/sergei-grechanik/supercompilation-hypergraph`.

25. *Grechanik S.* Inductive Prover Based on Equality Saturation (Extended Version) // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", July 2014. — Pp. 26–53. — ISBN 978-5-901795-31-6.

26. *Grechanik S.*, *Klyuchnikov I.*, *Romanenko S.* Staged Multi-Result Supercompilation: Filtering by Transformation // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl Zalessky:

Publishing House "University of Pereslavl", July 2014. — Pp. 54–78. — ISBN 978-5-901795-31-6.

27. *Grechanik S. A.* Inductive Prover Based on Equality Saturation for a Lazy Functional Language // Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers. Vol. 8974 / ed. by A. Voronkov, I. Virbitskaite. — Springer, 2014. — Pp. 127–141. — (Lecture Notes in Computer Science). — ISBN 978-3-662-46822-7. — URL: http://dx.doi.org/10.1007/978-3-662-46823-4.

28. *Grechanik S. A.* Overgraph Representation for Multi-Result Supercompilation // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl-Zalessky: Ailamazyan University of Pereslavl, July 2012. — Pp. 48–65. — ISBN 978-5-901795-28-6. — URL: http://pat.keldysh.ru/~grechanik/doc/overgraph.pdf.

29. *Grechanik S. A.* Proving properties of functional programs by equality saturation // Programming and Computer Software. — 2015. — Vol. 41, no. 3. — Pp. 149–161. — URL: http://dx.doi.org/10.1134/S0361768815030056.

30. *Grechanik S. A.* Supercompilation by Hypergraph Transformation: Preprint / Keldysh Institute of Applied Mathematics. — 2013. — 24 pp. — No. 26. — URL: http://library.keldysh.ru/preprint.asp?id=2013-26&lg=e.

31. *Hamilton G. W.* A Graph-Based Definition of Distillation // Second International Workshop on Metacomputation in Russia. — 2010.

32. *Hamilton G. W.* Distillation: extracting the essence of programs // Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. — ACM Press New York, NY, USA. 2007. — Pp. 61–70.

33. *Hamilton G. W.* A Hierarchy of Program Transformers // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl-Zalessky: Ailamazyan University of Pereslavl, July 2012. — Pp. 66–86. — ISBN 978-5-901795-28-6.

34. *Hamilton G. W.*, *Jones N. D.* Distillation with labelled transition systems // Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012 / ed. by O. Kiselyov, S. Thompson. — ACM, 2012. — Pp. 15–24. — ISBN 978-1-4503-1118-2. — URL: `http://dl.acm.org/citation.cfm?id=2103746`.

35. *Hoffmann B.*, *Plump D.* Implementing term rewriting by jungle evaluation // ITA. — 1991. — Vol. 25. — Pp. 445–472.

36. *Huet G.*, *Hullot J.-M.* Proofs by Induction of Equational Theories with Constructors // Journal of Computer and System Sciences 25. — 1982. — Pp. 239–266.

37. *Hutton G.*, *Gibbons J.* The Generic Approximation Lemma // Information Processing Letters. — 2001. — Vol. 79, no. 4. — Pp. 197–201.

38. *Jones S. P.*, *Tolmach A.*, *Hoare T.* Playing by the rules: rewriting as a practical optimisation technique in GHC // Haskell workshop. Vol. 1. — 2001. — Pp. 203–233.

39. *Joshi R.*, *Nelson G.*, *Randall K.* Denali: a goal-directed superoptimizer // ACM SIGPLAN Notices. — 2002. — May. — Vol. 37, no. 5. — Pp. 304–314. — ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic).

40. *Kaufmann M.*, *Moore J. S.* An Industrial Strength Theorem Prover for a Logic Based on Common Lisp // IEEE Transactions on Software Engineering. — 1997. — Apr. — Vol. 23, no. 4. — Pp. 203–213.

41. *Klyuchnikov I.* Supercompiler HOSC 1.0: under the hood: Preprint / Keldysh Institute of Applied Mathematics. — Moscow, 2009. — No. 63.

42. *Klyuchnikov I.* Towards effective two-level supercompilation: Preprint / Keldysh Institute of Applied Mathematics. — 2010. — 28 pp. — No. 81. — URL: `http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e`.

43. *Klyuchnikov I. G.*, *Romanenko S. A.* MRSC: a toolkit for building multi-result supercompilers: Preprint / Keldysh Institute of Applied Mathematics. — 2011. — No. 77. — URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77`.

44. *Klyuchnikov I. G.*, *Romanenko S. A.* Multi-Result Supercompilation as Branching Growth of the Penultimate Level in Metasystem Transitions // Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011. Vol. 7162 / ed. by E. Clarke, I. Virbitskaite, A. Voronkov. — Springer, 2012. — Pp. 210–226. — (Lecture Notes in Computer Science). — ISBN 978-3-642-11485-4.

45. *Klyuchnikov I.*, *Romanenko S.* Proving the Equivalence of Higher-Order Terms by Means of Supercompilation // Perspectives of Systems Informatics. Vol. 5947. — 2010. — Pp. 193–205. — (LNCS).

46. *Klyuchnikov I.*, *Romanenko S.* Towards Higher-Level Supercompilation // Second International Workshop on Metacomputation in Russia. — 2010.

47. *Knuth D. E.*, *Bendix P.* Simple Word Problems in Universal Algebras // Computational Problems in Abstract Algebra / ed. by J. Leech. — Pergamon Press, 1970. — Pp. 263–297.

48. *Leino K. R. M.* Automating Induction with an SMT Solver // Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings. Vol. 7148 / ed. by V. Kuncak, A. Rybalchenko. — Springer, 2012. — Pp. 315–331. — (Lecture Notes in Computer Science). — ISBN 978-3-642-27939-3. — URL: `http://dx.doi.org/10.1007/978-3-642-27940-9`.

49. *Lisitsa A.*, *Webster M.* Supercompilation for Equivalence Testing in Metamorphic Computer Viruses Detection // Proceedings of the First International Workshop on Metacomputation in Russia. — 2008.

50. *Lowe*, *Ehrig* Algebraic Approach to Graph Transformation Based on Single Pushout Derivations // WG: Graph-Theoretic Concepts in Computer Science, International Workshop WG. — 1990.

51. *Mitchell N.*, *Runciman C.* A Supercompiler for Core Haskell // Implementation and Application of Functional Languages. Vol. 5083. — Berlin, Heidelberg : Springer-Verlag, 2008. — Pp. 147–164. — (Lecture Notes In Computer Science). — ISBN 978-3-540-85372-5. — DOI: `http://dx.doi.org/10.1007/978-3-540-85373-2_9`.

52. *Moskal M.*, *Lopuszanski J.*, *Kiniry J. R.* E-matching for Fun and Profit // Electr. Notes Theor. Comput. Sci. — 2008. — Vol. 198, no. 2. — Pp. 19–35. — URL: http://dx.doi.org/10.1016/j.entcs.2008.04.078.

53. *Moura L. M. de*, *Bjørner N.* Efficient E-Matching for SMT Solvers // Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. Vol. 4603 / ed. by F. Pfenning. — Springer, 2007. — Pp. 183–198. — (Lecture Notes in Computer Science). — ISBN 978-3-540-73594-6. — URL: http://dx.doi.org/10.1007/978-3-540-73595-3_13.

54. *Moura L. de*, *Bjørner N.* Z3: An Efficient SMT Solver // Tools and Algorithms for the Construction and Analysis (TACAS). Vol. 4963. — Berlin : Springer-Verlag, 2008. — Pp. 337–340. — (Lecture Notes in Computer Science). — URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24.

55. *Nelson*, *Oppen* Fast Decision Procedures Based on Congruence Closure // JACM: Journal of the ACM. — 1980. — Vol. 27, no. 2. — Pp. 356–364.

56. *Nieuwenhuis R.*, *Oliveras A.* Congruence Closure with Integer Offsets // Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings. Vol. 2850 / ed. by M. Y. Vardi, A. Voronkov. — Springer, 2003. — Pp. 78–90. — (Lecture Notes in Computer Science). — ISBN 3-540-20101-7. — URL: http://dx.doi.org/10.1007/978-3-540-39813-4_5.

57. *Nieuwenhuis*, *Oliveras* Proof-Producing Congruence Closure // RTA: Rewriting Techniques and Applications, 1987, LNCS 256. — 2005.

58. *Plump D.* Evaluation of Functional Expressions by Hypergraph Rewriting: PhD thesis / Plump D. — Univ. Bremen, 1993.

59. *Reynolds J.* Definitional Interpreters for Higher Order Programming Languages // ACM Conference Proceedings. — ACM, 1972. — Pp. 717–740.

60. *Riazanov A.*, *Voronkov A.* Vampire // Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings. Vol. 1632 / ed. by H. Ganzinger. — Springer, 1999. — Pp. 292–296. — (Lecture Notes in Computer Science). — ISBN 3-540-66222-7. — URL: http://dx.doi.org/10.1007/3-540-48660-7_26.

61. Rippling: A Heuristic for Guiding Inductive Proofs / A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, A. Smaill // Artificial Intelligence. — 1993. — Aug. — Vol. 62, no. 2. — Pp. 185–253.

62. *Robinson J. A.* A Machine Oriented Logic Based on the Resolution Principle // JACM. — 1965. — Vol. 12, no. 1. — Pp. 23–41.

63. *Robinson G.*, *Wos L.* Paramodulation and theorem-proving in first-order theories with equality // Automation of Reasoning. — Springer, 1983. — Pp. 298–313.

64. *Sands D.* Total correctness by local improvement in the transformation of functional programs // ACM Trans. Program. Lang. Syst. — 1996. — Vol. 18, no. 2. — Pp. 175–234.

65. *Sands D.* Proving the correctness of recursion-based automatic program transformations // Theor. Comput. Sci. — Amsterdam-London-New York-Oxford-Paris-Shannon-Tokyo, 1996. — Vol. 167, no. 1–2. — Pp. 193–233.

66. *Schulz S.* E - a brainiac theorem prover // AI Commun. — 2002. — Vol. 15, no. 2–3. — Pp. 111–126. — URL: `http://content.iospress.com/articles/ai-communications/aic260`.

67. *Scott D.*, *Strachey C.* Towards a Mathematical Semantics for Computer Languages: Technical Report / Oxford University Computer Laboratory. — 1971. — PRG–6.

68. *Scott D. S.* Domains for Denotational Semantics // Automata, Languages, and Programming: 9th Colloquium, Aarhus, Denmark / ed. by M. Nielson, E. M. Schmidt. — Springer-Verlag, 1982. — Pp. 577–613. — (Lecture Notes in Computer Science ; 140).

69. *Sonnex W.*, *Drossopoulou S.*, *Eisenbach S.* Zeno: An automated prover for properties of recursive data structures // TACAS. — Mar. 2012. — (Lecture Notes in Computer Science). — URL: `http://pubs.doc.ic.ac.uk/zenoTwo/`.

70. *Sørensen M. H.* Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation: MA thesis / Sørensen M. H. — Københavns Universitet, Datalogisk Institut, 1994.

71. SPASS Version 3.5 / C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. S. 0001, P. Wischnewski // Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings. Vol. 5663 / ed. by R. A. Schmidt. — Springer, 2009. — Pp. 140–145. — (Lecture Notes in Computer Science). — ISBN 978-3-642-02958-5. — URL: http://dx.doi.org/10.1007/978-3-642-02959-2.

72. *Sprenger C.*, *Dam M.* On global induction mechanisms in a μ-calculus with explicit approximations // ITA. — 2003. — Vol. 37, no. 4. — Pp. 365–391. — URL: http://dx.doi.org/10.1051/ita:2003024.

73. *Stepp M. B.* Equality saturation : engineering challenges and applications. — Jan. 2011.

74. *Stepp M.*, *Tate R.*, *Lerner S.* Equality-Based Translation Validator for LLVM // Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Vol. 6806 / ed. by G. Gopalakrishnan, S. Qadeer. — Springer, 2011. — Pp. 737–742. — (Lecture Notes in Computer Science). — ISBN 978-3-642-22109-5. — URL: http://dx.doi.org/10.1007/978-3-642-22110-1.

75. *Stratulat S.* A Unified View of Induction Reasoning for First-Order Logic // Turing-100 - The Alan Turing Centenary, Manchester, UK, June 22-25, 2012. Vol. 10 / ed. by A. Voronkov. — EasyChair, 2012. — Pp. 326–352. — (EPiC Series). — URL: http://www.easychair.org/publications/?page=1900403647.

76. *Streicher T.* Domain-theoretic foundations of functional programming. — World Scientific, 2006. — ISBN 978-981-270-142-8.

77. *Sutcliffe G.* The TPTP Problem Library and Associated Infrastructure // J. Autom. Reasoning. — 2009. — Vol. 43, no. 4. — Pp. 337–362. — URL: http://dx.doi.org/10.1007/s10817-009-9143-8.

78. *Tate R.* Equality saturation: using equational reasoning to optimize imperative functions. — Jan. 2012.

79. The Alt-Ergo Theorem Prover. — http://alt-ergo.lri.fr/.

80. The Satisfiability Modulo Theories Library. — http://smtlib.cs.uiowa.edu/.

81. TIP: Tons of Inductive Problems / K. Claessen, M. Johansson, D. Rosén, N. Smallbone.

82. *Turchin V.* The concept of a supercompiler // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1986. — Vol. 8, no. 3. — Pp. 292–325.

83. *Turner D.* Total Functional Programming // Journal of Universal Computer Science. — 2004. — Vol. 10, no. 7. — Pp. 751–768.

84. *Yorgey B.* The typeclassopedia // The Monad. Reader Issue 13. — 2009. — P. 17.

85. *Гречаник С. А.* Доказательство свойств функциональных программ методом насыщения равенствами // Программирование. — 2015. — № 3. — с. 44—61.

86. *Гречаник С. А.* Полипрограммы как представление множеств функциональных программ и преобразования над ними // Препринты ИПМ им. М.В.Келдыша. — 2017. — № 5. — DOI: 10.20948/prepr-2017-5. — URL: http://library.keldysh.ru/preprint.asp?id=2017-5.

87. *Ключников И. Г.* Выявление и доказательство свойств функциональных программ методами суперкомпиляции / Ключников Илья Григорьевич. — Институт прикладной математики им М.В. Келдыша РАН, 2010.

88. *Ключников И. Г.* Суперкомпиляция: идеи и методы // Практика функционального программирования. — 2011. — т. 7.

Приложение A
Дополнительные примеры

В данном приложении будут рассмотрены дополнительные примеры.

Пример 12. Слияние по бисимуляции обычно применяется для завершения доказательства, однако иногда его имеет смысл применять и в середине процесса преобразования полипрограммы, что соответствует обнаружению и применению леммы. Рассмотрим следующую полипрограмму:

$$(1)\, f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to f(f(y))\}$$
$$(2)\, g(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to g(x)\}$$

Требуется доказать, что f и g эквивалентны. Для удобства вынесем подвыражение $f(f(y))$ в отдельную функцию при помощи правила расчленения:

$$(3)\, f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}$$
$$(4)\, h(x) \equiv f(f(x))$$

Раскроем внешний вызов f в определении (4), получится следующее определение:

$$(5)\, h(x) \equiv \textbf{case}\ f(x)\ \textbf{of}\{Z \to Z; S(y) \to h(y)\}$$

Теперь раскроем $f(x)$ в разбираемой позиции:

$$
\begin{array}{l}
(6)\, h(x) \equiv \\
\quad \textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\})\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to h(y)
\end{array}
$$

Теперь применим правило сопоставления с образцом результата сопоставления с образцом:

$$
\begin{array}{l}
(7)\, h(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to \textbf{case}\ Z\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\} \\
\qquad S(y) \to \textbf{case}\ h(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

В ветви Z произведём редукцию сопоставления с образцом:

$$
\begin{array}{l}
(8)\, h(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case}\ h(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

Теперь возьмём определение (3) и раскроем в нём $h(x)$ по определению (5):

$$
\begin{array}{l}
(9)\, f(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case}\ f(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

Определения (8) и (9) очень похожи, и действительно, к ним можно применить слияние по бисимуляции. Полипрограмма бисимуляции выглядит сле-

дующим образом:

$$
\begin{array}{l}
f(x) \equiv \\
\quad \textbf{case } x \textbf{ of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case } f(y) \textbf{ of } \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

При этом $\phi_1 = \{f \mapsto f, h \mapsto h\}$ и $\phi_2 = \{f \mapsto h, h \mapsto h\}$. Это как раз пример, когда фрагменты не изоморфны. Нетрудно видеть, что рекурсия в полипрограмме бисимуляции структурная, поэтому можно добавить следующее определение к нашей полипрограмме:

(10) $f(x) \equiv h(x)$

Теперь применим конгруэнтность, заменив h на f в определении (3):

(11) $f(x) \equiv \textbf{case } x \textbf{ of } \{Z \to Z; S(y) \to f(y)\}$

Теперь можно применить ещё одно слияние по бисимуляции для определений (11) и (2), что даст нам требуемое $f(x) \equiv g(x)$ $\triangle$

Пример 13. Рассмотрим пример доказательства того, что эффективная реализация функции чётности эквивалентна неэффективной реализации:

(1) $\text{even}(x) \equiv \textbf{case } x \textbf{ of } \{Z \to T; S(y) \to \text{odd}(y)\}$
(2) $\text{odd}(x) \equiv \textbf{case } x \textbf{ of } \{Z \to F; S(y) \to \text{even}(y)\}$

(3) $\text{not}(t) \equiv \textbf{case } t \textbf{ of } \{F \to T; T \to F\}$
(4) $\text{evenSlow}(x) \equiv \textbf{case } x \textbf{ of } \{Z \to T; S(y) \to \text{oddSlow}(y)\}$
(5) $\text{oddSlow}(x) \equiv \text{not}(\text{evenSlow}(x))$

Надо доказать, что $\text{evenSlow}(x) \equiv \text{even}(x)$. Раскроем вызов функции not в определении (5):

(6) $\text{oddSlow}(x) \equiv \textbf{case } \text{evenSlow}(x) \textbf{ of } \{F \to T; T \to F\}$

Теперь раскроем вызов evenSlow:

$$
\begin{array}{l}
(7)\,\mathrm{oddSlow}(x) \equiv \\
\quad \textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to T; S(y) \to \mathrm{oddSlow}(y)\})\ \textbf{of} \\
\quad\quad F \to T \\
\quad\quad T \to F
\end{array}
$$

Применим правило сопоставления с образцом результата сопоставления с образцом:

$$
\begin{array}{l}
(8)\,\mathrm{oddSlow}(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \textbf{case}\ T\ \textbf{of}\ \{F \to T; T \to F\} \\
\quad\quad S(y) \to \textbf{case}\ \mathrm{oddSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}
\end{array}
$$

Теперь произведём редукцию в ветви Z:

$$
\begin{array}{l}
(9)\,\mathrm{oddSlow}(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to F \\
\quad\quad S(y) \to \textbf{case}\ \mathrm{oddSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}
\end{array}
$$

Ветвь S вынесем в качестве отдельной функции:

$$
\begin{array}{l}
(10)\,\mathrm{oddSlow}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to F; S(y) \to f(y)\} \\
(11)\,f(x) \equiv \textbf{case}\ \mathrm{oddSlow}(x)\ \textbf{of}\ \{F \to T; T \to F\}
\end{array}
$$

Раскроем в (11) вызов oddSlow с использованием определения (10), применим сопоставление с образцом результата сопоставлении с образцом и произведём

редукцию в ветви Z (сделаем это в одно действие для краткости):

$$(12)\ f(x) \equiv$$
$$\quad \textbf{case}\ x\ \textbf{of}$$
$$\quad\quad Z \to T$$
$$\quad\quad S(y) \to \textbf{case}\ f(y)\ \textbf{of}\ \{F \to T; T \to F\}$$

Теперь в определении (4) раскроем oddSlow по определению (6):

$$(13)\ \mathrm{evenSlow}(x) \equiv$$
$$\quad \textbf{case}\ x\ \textbf{of}$$
$$\quad\quad Z \to T$$
$$\quad\quad S(y) \to \textbf{case}\ \mathrm{evenSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}$$

Теперь можно применить слияние по бисимуляции к определениям (13) и (12):

$$(14)\ f(x) \equiv \mathrm{evenSlow}(x)$$

Теперь по конгруэнтности заменим f на evenSlow в (10):

$$(15)\ \mathrm{oddSlow}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to F; S(y) \to \mathrm{evenSlow}(y)\}$$

Определения (1) и (2) совпадают с определениями (4) и (15) с точностью до переименования функций, поэтому применяем слияние по бисимуляции и получаем

$$(16)\ \mathrm{oddSlow}(x) \equiv \mathrm{odd}(x)$$
$$(17)\ \mathrm{evenSlow}(x) \equiv \mathrm{even}(x)$$

△

Пример 14. В Примере 13 слияние по бисимуляции применялось два раза. Однако можно обойтись и одним слиянием по бисимуляции. Этот пример интересен тем, что раскрывает одну вещь о нашей системе преобразований: в терминах суперкомпиляции она реализует не только прогонку, но и неко-

торые виды обобщений. Начало преобразования будет таким же как и в том примере:

$$(1)\ \mathrm{even}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to T; S(y) \to \mathrm{odd}(y)\}$$
$$(2)\ \mathrm{odd}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to F; S(y) \to \mathrm{even}(y)\}$$

$$(3)\ \mathrm{not}(t) \equiv \textbf{case}\ t\ \textbf{of}\ \{F \to T; T \to F\}$$
$$(4)\ \mathrm{evenSlow}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to T; S(y) \to \mathrm{oddSlow}(y)\}$$
$$(5)\ \mathrm{oddSlow}(x) \equiv \mathrm{not}(\mathrm{evenSlow}(x))$$

$$(6)\ \mathrm{oddSlow}(x) \equiv \textbf{case}\ \mathrm{evenSlow}(x)\ \textbf{of}\ \{F \to T; T \to F\}$$
$$(7)\ \mathrm{oddSlow}(x) \equiv$$
$$\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to T; S(y) \to \mathrm{oddSlow}(y)\})\ \textbf{of}$$
$$F \to T$$
$$T \to F$$
$$(8)\ \mathrm{oddSlow}(x) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to \textbf{case}\ T\ \textbf{of}\ \{F \to T; T \to F\}$$
$$S(y) \to \textbf{case}\ \mathrm{oddSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}$$
$$(9)\ \mathrm{oddSlow}(x) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to F$$
$$S(y) \to \textbf{case}\ \mathrm{oddSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}$$
$$(10)\ \mathrm{oddSlow}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to F; S(y) \to f(y)\}$$
$$(11)\ f(x) \equiv \textbf{case}\ \mathrm{oddSlow}(x)\ \textbf{of}\ \{F \to T; T \to F\}$$
$$(12)\ f(x) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to T$$
$$S(y) \to \textbf{case}\ f(y)\ \textbf{of}\ \{F \to T; T \to F\}$$

Теперь в ветви S определения (12) раскроем вызов функции f по определению (11), и сразу после этого раскроем вызов oddSlow по определению (6):

$$(13)\ f(x) \equiv \textbf{case}\ x\ \textbf{of}$$
$$Z \to T$$
$$S(y) \to \textbf{case} \begin{pmatrix} \textbf{case}\ (\textbf{case}\ \text{evenSlow}(x)\ \textbf{of}\ \{F \to T; T \to F\})\ \textbf{of} \\ F \to T \\ T \to F \end{pmatrix}$$
$$\textbf{of}\{F \to T; T \to F\}$$

Фактически в ветви S записано тройное отрицание выражения evenSlow(x), которое эквивалентно одинарному отрицанию этого выражения. Применим в ветви S правило сопоставления с образцом результата сопоставления с образцом два раза (возможность такого применения ключевая в этом примере), а в ветвях получившегося выражения произведём редукцию сопоставления с образцом, и получим в точности одинарное отрицание:

$$(14)\ f(x) \equiv$$
$$\textbf{case}\ x\ \textbf{of}$$
$$Z \to T$$
$$S(y) \to \textbf{case}\ \text{evenSlow}(y)\ \textbf{of}\ \{F \to T; T \to F\}$$

Теперь заметим, что правые части определений (9) и (14) совпадают, а значит по транзитивности:

$$(15)\ f(x) \equiv \text{evenSlow}(x)$$

Дальнейшее происходит как в прошлом примере. По конгруэнтности заменяем f на evenSlow в (10):

$$(16)\ \text{oddSlow}(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to F; S(y) \to \text{evenSlow}(y)\}$$

Определения (1) и (2) совпадают с определениями (4) и (16) с точностью до переименования функций, поэтому применяем слияние по бисимуляции и

получаем

$$(17)\, \mathrm{oddSlow}(x) \equiv \mathrm{odd}(x)$$
$$(18)\, \mathrm{evenSlow}(x) \equiv \mathrm{even}(x)$$

Отметим, что ключевой момент состоял в применении правила сопоставления с образцом результата сопоставления с образцом к выражению вида **case case** e **of**$\{\ldots\}$ **of**$\{\ldots\}$, где e — произвольное выражение. Интересный нюанс заключается в том, что для реализации прогонки (как в суперкомпиляции) нам достаточно того, чтобы это правило было применимо только в случае, когда $e = x$, т.е. некоторая переменная. Но когда e может быть произвольным, это правило становится более мощным и способным на эффекты, подобные нетривиальным обобщениям. △

Пример 15. Рассмотрим пример применения правила (trans) к следующей полипрограмме:

$$f_1(x, y) \equiv g(h(x, y), h(y, x))$$
$$f_2(y) \equiv g(h(x, y), h(y, x))$$

В бесточечно-расчленённом представлении она записывается следующим образом:

$$id_2 f_1 \equiv \mathbf{Call}(id_2 g, id_2 h, \{1 \to 2, 2 \to 1\}h)$$
$$\{1 \to 2\}_{1 \to 2} f_2 \equiv \mathbf{Call}(id_2 g, id_2 h, \{1 \to 2, 2 \to 1\}h)$$

Существует четыре способа отобразить определения образца L в полипрограмму G. Первые два способа отображают оба определения L в какое-то одно определение G, и в результате применения правила с такими отображениями получатся тривиальные определения $f_1 \equiv \mathbf{Id}(f_1)$ и $f_2 \equiv \mathbf{Id}(f_2)$, что эквивалентно применению (eq-refl). Рассмотрим более интересные случаи, когда определения L отображаются в различные определения G, например, первое в первое и второе во второе. Соответствующее отображение функциональных символов выглядит как $\{f \to f_1, g \to f_2, h_1 \to g, h_2 \to h, h_3 \to h, h_i \to$ произвольная функция$\}$. Т.к. в образце L есть лишние функции $h_i, i > 3$, то их тоже надо куда-то отобразить, но совершенно не важно, куда, поскольку

это ни на что не повлияет. Теперь дополним данные отображения до морфизма $\phi : L \to G$. На функциональных символах одно из возможных дополнений выглядит следующим образом (здесь $N_f, N_g, \ldots$ обозначают арности функций в образце L):

$$\phi(f) = id_{2 \to N_f} f_1$$
$$\phi(g) = \{1 \to 2\}_{1 \to N_g} f_2$$
$$\phi(h_1) = id_{2 \to N_{h_1}} g$$
$$\phi(h_2) = id_{2 \to N_{h_2}} h$$
$$\phi(h_3) = \{1 \to 2, 2 \to 1\}_{2 \to N_{h_3}} h$$

Получаем, что $G \Rightarrow_{(\text{trans}),\phi} H$, где H имеет следующий вид:

$$id_2 f_1 \equiv \mathbf{Call}(id_2 g, id_2 h, \{1 \to 2, 2 \to 1\} h)$$
$$\{1 \to 2\}_{1 \to 2} f_2 \equiv \mathbf{Call}(id_2 g, id_2 h, \{1 \to 2, 2 \to 1\} h)$$
$$id_{2 \to N_f} f_1 \equiv id_{N_g \to N_f} \mathbf{Id}(\{1 \to 2\}_{1 \to N_g} f_2)$$

Новое (последнее) определение записано не в канонической форме, в канонической форме оно выглядит так:

$$\{1 \to 2, 2 \to 1\}_{2 \to 2} f_1 \equiv id_{1 \to 2} \mathbf{Id}(id_1 f_2)$$

Или, в более удобном для чтения виде:

$$f_1(y, x) \equiv f_2(x)$$

Проделывая то же самое для случая, когда первое определение отображается во второе, а второе в первое, получим симметричное определение (с тем же смыслом):

$$f_2(y) \equiv f_1(x, y)$$

Отметим, что в данном примере арности левых функций различаются, однако это ничему не мешает (арность f_1 может быть понижена при помощи (arity-reduction)).

Если рассмотреть другое дополнение отображений определений и функ-

ций перестановками параметров до морфизма, то получится тот же самый результат. Например, возьмём следующий морфизм:

$$\phi(f) = \{1 \to 2, 2 \to 1\}_{2 \to N_f} f_1$$
$$\phi(g) = id_{1 \to N_g} f_2$$
$$\phi(h_1) = \{1 \to 2, 2 \to 1\}_{2 \to N_{h_1}} g$$
$$\phi(h_2) = id_{2 \to N_{h_2}} h$$
$$\phi(h_3) = \{1 \to 2, 2 \to 1\}_{2 \to N_{h_3}} h$$

Здесь отличаются перестановки при вызываемой функции g, что возможно только благодаря тому, что оба аргумента представлены одинаковыми функциями h. Формулы определений L отобразятся в следующие определения (до приведения к каноническому виду):

$$\{1 \to 2, 2 \to 1\}_{2 \to N_f} f_1 \equiv id_{N_{h_2} \to N_f} \mathbf{Call}(\{1 \to 2, 2 \to 1\}_{2 \to N_{h_1}} g, id_{2 \to N_{h_2}} h, \{1 \to 2, 2 \to 1\}_{2 \to N_{h_3}} h)$$
$$id_{1 \to N_g} f_2 \equiv id_{N_{h_2} \to N_f} \mathbf{Call}(\{1 \to 2, 2 \to 1\}_{2 \to N_{h_1}} g, id_{2 \to N_{h_2}} h, \{1 \to 2, 2 \to 1\}_{2 \to N_{h_3}} h)$$

Или, в более читаемой форме:

$$f_1(y, x) \equiv g(h(y, x), h(x, y))$$
$$f_2(x) \equiv g(h(y, x), h(x, y))$$

После приведения к каноническому виду получится в точности G.

При применении правила добавляемое определение будет иметь также немного другой вид до приведения к каноническому виду:

$$\{1 \to 2, 2 \to 1\}_{2 \to N_f} f_1 \equiv id_{N_g \to N_f} \mathbf{Id}(id_{1 \to N_g} f_2)$$

Однако после приведения к каноническому виду оно будет в точности совпа-

дать с определением, добавленным при помощи предыдущего морфизма:

$$\{1 \to 2, 2 \to 1\}_{2\to 2} f_1 \equiv id_{1\to 2}\mathbf{Id}(id_1 f_2)$$

△

Пример 16. Рассмотрим пример применения правила (arity-reduction). Пусть полипрограмма имеет следующий вид (важный нюанс — f и слева, и справа):

$$f(x, y) \equiv f(g_1(), g_2(x))$$

Или, в бесточечно-расчленённом представлении:

$$id_2 f \equiv id_{1\to 2}\mathbf{Call}(id_2 f, id_{0\to 1} g_1, id_1 g_2)$$

Применим правило понижения арности для $m = 1$. При этом морфизм $\phi : L \to G$ на функциях действует следующим образом:

$$\phi(f) = id_{2\to N} f$$
$$\phi(h_1) = id_{2\to 2} f$$
$$\phi(h_2) = id_{0\to 1} g_1$$
$$\phi(h_3) = id_{1\to 1} g_2$$

В результате функция f полипрограммы G отобразится в $id_{1\to 2} f'$ и получится следующая полипрограмма:

$$id_{1\to 2} f' \equiv id_{1\to 2}\mathbf{Call}(id_{1\to 2} f', id_{0\to 1} g_1, id_1 g_2)$$

Что после приведения к каноническому виду превратится в

$$id_1 f' \equiv id_{0\to 1}\mathbf{Call}(id_1 f', id_0 g_1)$$

Или, иначе

$$f'(x) \equiv f'(g_1())$$

Получившаяся полипрограмма не находится в полностью канонической форме, и так произошло именно потому, что f находится слева и справа. Ещё

одно применение правила приведёт данную полипрограмму к полностью каноническому виду:

$$f''() \equiv f''()$$

Это определение выражено при помощи конструкции **Call**:

$$id_0 f'' \equiv id_0 \mathbf{Call}(id_0 f'')$$

Оно может быть превращено в конструкцию **Id** при помощи применения правила (call-to-permutation). △

Пример 17. Рассмотрим пример применения правила (call-to-permutation) к следующей полипрограмме:

$$f(x, y, z, u) \equiv g(v(y), v(x), v(u))$$
$$v(x) \equiv x$$

Она же в бесточечно-расчленённом представлении:

$$id_4 f \equiv id_4 \mathbf{Call}(id_3 g, \{1 \to 2\}v, \{1 \to 1\}v, \{1 \to 4\}v)$$
$$id_1 v \equiv id_1 \mathbf{Var}$$

Её первое определение эквивалентно следующему (неиспользуемый аргумент $\{1 \to 3\}v$ добавлен для наглядности):

$$id_4 f \equiv id_4 \mathbf{Call}(\{1 \to 2, 2 \to 1, 3 \to 4\}g,$$
$$\{1 \to 1\}v, \{1 \to 2\}v, \{1 \to 3\}v, \{1 \to 4\}v)$$

Отсюда сразу очевидно, что применимо правило с ϕ, действующим на функции так:

$$\phi(v) = id_1 v$$
$$\phi(f) = id_{4 \to N} f$$
$$\phi(g) = \{1 \to 2, 2 \to 1, 3 \to 4\}_{3 \to N} g$$

Получаем в итоге следующую полипрограмму (не в канонической форме):

$$id_4 f \equiv id_4 \mathbf{Id}(\{1 \to 2, 2 \to 1, 3 \to 4\} g)$$
$$id_1 v \equiv id_1 \mathbf{Var}$$

Или:

$$f(x, y, z, u) \equiv g(y, x, u)$$
$$v(x) \equiv x$$

△

Пример 18. Рассмотрим пример применения правил из Главы 4 для слияния функций, представленных изоморфными деревьями.

$$\left\{\begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ f_2(x) \equiv C(g_2(), h(x)) \\ g_2() \equiv Z() \end{array}\right\}$$
$$\Rightarrow_{(\text{trans})} \left\{\begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ f_2(x) \equiv C(g_2(), h(x)) \\ g_2() \equiv Z() \\ g_1() \equiv g_2() \end{array}\right\}$$
$$\Rightarrow_{g_2 = g_1} \left\{\begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ f_2(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ g_1() \equiv g_1() \end{array}\right\}$$

$$\Rightarrow_{(\text{trans})} \left\{ \begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ f_2(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ g_1() \equiv g_1() \\ f_1(x) \equiv f_2(x) \end{array} \right\}$$

$$\Rightarrow_{f_2=f_1} \left\{ \begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ g_1() \equiv g_1() \\ f_1(x) \equiv f_1(x) \end{array} \right\}$$

$$\Rightarrow^*_{(\text{remove-dups})} \left\{ \begin{array}{l} f_1(x) \equiv C(g_1(), h(x)) \\ g_1() \equiv Z() \\ g_1() \equiv g_1() \\ f_1(x) \equiv f_1(x) \end{array} \right\}$$

△

Пример 19. Слияние по бисимуляции способно приводить к понижению арности функций. Рассмотрим следующую полипрограмму G:

$$f(x) \equiv S(h(x))$$
$$h(x) \equiv f(g(x))$$

Эта программа вычисляет бесконечное число $S(S(S(\ldots)))$, но не очень эффективно, накапливая бесполезный вызов некоторой функции g:

$$f(x) \to S(f(g(x))) \to S(S(f(g(g(x))))) \to \ldots$$

Рассмотрим следующую полипрограмму H:

$$f(x,y) \equiv S(h(x,y))$$
$$h(x,y) \equiv f(g(x),g(y))$$

Эта полипрограмма имеет единственную модель для каждой интерпретации g. Морфизмы $\phi_{1,2} : H \to G$ возьмём такие:

$$\phi_1(f)(x,y) = f(x)$$
$$\phi_1(h)(x,y) = h(x)$$
$$\phi_1(g) = g$$

$$\phi_2(f)(x,y) = f(y)$$
$$\phi_2(h)(x,y) = h(y)$$
$$\phi_2(g) = g$$

Тогда после применения последнего утверждения мы получим следующие дополнительные определения:

$$f(x) \equiv f(y)$$
$$h(x) \equiv h(y)$$

Определения имеют такой вид, что можно применить понижение арности, в результате полипрограмма будет выглядеть так:

$$f() \equiv S(h())$$
$$h \equiv \mathbf{Call}(f)$$
$$f() \equiv f()$$
$$h() \equiv h()$$

Второе определение записано при помощи нотации бесточечного представления, чтобы подчеркнуть, что это всё ещё **Call**, а не **Id**. В $h \equiv \mathbf{Id}(f)$ оно превращается при помощи преобразования (call-to-permutation). Далее, после применения конгруэнтности и удаления дубликатов, полипрограмма прини-

мает следующий вид:

$$f() \equiv S(f())$$
$$f() \equiv f()$$

Таким образом, слияние по бисимуляции позволило понизить арность рекурсивной функции и удалить бесполезный вызов. $\triangle$

Приложение B
Доказательства некоторых утверждений

Доказательство Утверждения 12, с. 75

Утверждение. Категория *конечных* множеств функциональных символов конечно кополна, т.е. в ней есть начальный объект, бинарные копроизведения и коуравнители.

Доказательство. Начальным объектом является пустое множество функциональных символов, а копроизведениями — обычные копроизведения (размеченные объединения) двух множеств функциональных символов как множеств.

Докажем существование коуравнителей. Пусть $\alpha, \beta : F \to G$. Построим для этой пары морфизмов коуравнитель $\delta : G \to H$. Пусть $\delta(g) = (\eta_g, h_g)$. Во-первых, надо обеспечить $\delta \circ \alpha = \delta \circ \beta$. Пусть $f \in F$, введём обозначения:

$$\alpha(f) = (\theta_f, g_f)$$
$$\beta(f) = (\xi_f, g'_f)$$

В этих обозначениях нам надо обеспечить

$$\theta_f \eta_{g_f} = \xi_f \eta_{g'_f}$$
$$h_{g_f} = h_{g'_f}$$

Сначала построим коуравнитель только для самих функциональных символов, без учёта перестановок, т.е. для функций $\pi_2 \circ \alpha$ и $\pi_2 \circ \beta$ в категории множеств — получим функцию $\varepsilon : G \to H$ такую, что

$$\varepsilon \circ \pi_2 \circ \alpha = \varepsilon \circ \pi_2 \circ \beta$$

И назначим $h_g = \varepsilon(g)$, что обеспечит нам второе равенство ($h_{g_f} = h_{g'_f}$).

Для обеспечения первого равенства нам надо решить конечную систему уравнений $\theta_f \eta_{g_f} = \xi_f \eta_{g'_f}$ относительно $\eta_{...}$. Её можно разбить на непересекающиеся по искомым $\eta_{...}$ подсистемы уравнений, для каждой из которых $\varepsilon(g_f) = \varepsilon(g'_f) = h$ для некоторого h. Для каждой из таких подсистем $\eta_{...}$ должны иметь общий домен $arity(h)$ (арность h мы ещё не знаем), а значит каждую такую подсистему можно решить, построив предел соответствующей диаграммы в категории перестановок параметров (конечных множеств с инъективными функциями в качестве морфизмов). К сожалению, в категории перестановок параметров нет произведений, хотя есть уравнители и декартовы квадраты, однако по крайней мере в конечном случае уравнителей и декартовых квадратов достаточно для построения данного предела. В итоге мы получим все необходимые перестановки параметров $\eta_{...}$ и арности всех $h \in H$ (т.к. всем h соответствует некоторая подсистема уравнений).

Теперь необходимо доказать универсальность. Пусть существует некоторый δ' такой, что $\delta' \circ \alpha = \delta' \circ \beta$. Нужно доказать существование и единственность морфизма функциональных символов ω такого, что $\delta' = \omega \circ \delta$.

Для δ' будет выполнено

$$\pi_2 \circ \delta' \circ \pi_2 \circ \alpha = \pi_2 \circ \delta' \circ \pi_2 \circ \beta$$

А значит существует единственная функция τ такая, что $\pi_2 \circ \delta' = \tau \circ \varepsilon$, т.к. ε — коуравнитель $\pi_2 \circ \alpha$ и $\pi_2 \circ \beta$ в категории множеств. τ и будет являться второй компонентой искомого морфизма, т.е. $\pi_2 \circ \omega = \tau$.

Теперь для каждого $h \in H$ нам надо найти такую перестановку σ_h, чтобы для всех g выполнялось $\pi_1(\delta'(g)) = \eta_g \circ \sigma_{\varepsilon(g)}$. Про $\pi_1(\delta'(g))$ мы знаем, что

$$\theta_f \circ \pi_1(\delta'(g_f)) = \xi_f \circ \pi_1(\delta'(g'_f))$$

Т.е. они удовлетворяют рассматриваемой ранее системе уравнений, а значит

и рассматриваемым ранее подсистемам. Т.к. решение подсистем строилось как предел, то для каждой подсистемы (а каждая подсистема соответствует своему $h \in H$) существует единственная σ_h такая, что $\pi_1(\delta'(g)) = \eta_g \circ \sigma_h$, если $\varepsilon(g) = h$, что нам и нужно. Таким образом утверждение доказано. □

Доказательство Утверждения 13, с. 76

Утверждение. Морфизмы функциональных символов хорошо взаимодействуют с понятием канонической формы, а именно если $p_1 \approx p_2$, то $\alpha(p_1) \approx \alpha(p_2)$.

Доказательство. Для доказательства утверждения достаточно доказать, что для любых p и α выполняется $canon(\alpha(p)) = canon(\alpha(canon(p)))$. Рассмотрим случай, когда p имеет вид $\theta_{-1} f_{-1} \equiv \theta\, \mathbf{Call}(\theta_0 f_0, \theta_1 f_1, \ldots)$, а $\alpha(f_i) = (\zeta_i, g_i)$. Тогда, расписывая определения *canon*, *canon-call* и *decompose*, получаем:

$$
\begin{aligned}
&canon(p) = (\xi_{-1} f_{-1} \equiv \xi\, \mathbf{Call}(id\ f_0, \overline{\xi_{\theta_0(i)} f_{\theta_0(i)}})),\ \text{где}\\
&\quad (\delta, \overline{\xi_{\theta_0(i)}}) = nsplit(\overline{\theta_{\theta_0(i)}})\\
&\quad (\_, \xi, \xi_{-1}) = bisplit(\theta \circ \delta, \theta_{-1})\\
&\alpha(canon(p)) = (\xi_{-1}\zeta_{-1} g_{-1} \equiv \xi\, \mathbf{Call}(\zeta_0 g_0, \overline{\xi_{\theta_0(i)}\zeta_{\theta_0(i)} g_{\theta_0(i)}}))\\
&canon(\alpha(canon(p))) = (\sigma_{-1} g_{-1} \equiv \sigma\, \mathbf{Call}(id\ g_0, \overline{\sigma_{\theta_0\zeta_0(i)} g_{\theta_0\zeta_0(i)}})),\ \text{где}\\
&\quad (\delta', \overline{\sigma_{\theta_0\zeta_0(i)}}) = nsplit(\overline{\xi_{\theta_0\zeta_0(i)}\zeta_{\theta_0\zeta_0(i)}})\\
&\quad (\_, \sigma, \sigma_{-1}) = bisplit(\xi \circ \delta', \xi_{-1}\zeta_{-1})\\
&\alpha(p) = (\theta_{-1}\zeta_{-1} g_{-1} \equiv \theta\, \mathbf{Call}(\theta_0\zeta_0 g_0, \overline{\theta_i \zeta_i g_i}))\\
&canon(\alpha(p)) = (\chi_{-1} g_{-1} \equiv \chi\, \mathbf{Call}(id\ g_0, \overline{\chi_{\theta_0\zeta_0(i)} g_{\theta_0\zeta_0(i)}})),\ \text{где}\\
&\quad (\delta'', \overline{\chi_{\theta_0\zeta_0(i)}}) = nsplit(\overline{\theta_{\theta_0\zeta_0(i)}\zeta_{\theta_0\zeta_0(i)}})\\
&\quad (\_, \chi, \chi_{-1}) = bisplit(\theta \circ \delta'', \theta_{-1}\zeta_{-1})
\end{aligned}
$$

Воспользовавшись определением *nsplit* и Утверждением 6, перепишем некоторые из равенств в следующем виде:

$$
\begin{aligned}
&(\alpha, \overline{\xi \circ \xi_{\theta_0(i)}}, \xi_{-1}) = nsplit(\overline{\theta \circ \theta_{\theta_0(i)}}, \theta_{-1})\\
&(\_, \overline{\sigma \circ \sigma_{\theta_0\zeta_0(i)}}, \sigma_{-1}) = nsplit(\overline{\xi \circ \xi_{\theta_0\zeta_0(i)} \circ \zeta_{\theta_0\zeta_0(i)}}, \xi_{-1} \circ \zeta_{-1})\\
&(\_, \overline{\chi \circ \chi_{\theta_0\zeta_0(i)}}, \chi_{-1}) = nsplit(\overline{\theta \circ \theta_{\theta_0\zeta_0(i)} \circ \zeta_{\theta_0\zeta_0(i)}}, \theta_{-1} \circ \zeta_{-1})
\end{aligned}
$$

Фактически мы просто внесли перестановку при правой части и добавили перестановку при функции слева к остальным перестановкам. Из первого равенства следует, что

$$\theta \circ \theta_{\theta_0(i)} = \alpha \circ \xi \circ \xi_{\theta_0(i)}$$
$$\theta_{-1} = \alpha \circ \xi_{-1}$$

А значит, если переставить индексы, то

$$\theta \circ \theta_{\theta_0\zeta_0(i)} = \alpha \circ \xi \circ \xi_{\theta_0\zeta_0(i)}$$
$$\theta_{-1} = \alpha \circ \xi_{-1}$$

Эти следствия из первого равенства используем для переписывания третьего равенства:

$$(\_, \overline{\chi \circ \chi_{\theta_0\zeta_0(i)}}, \chi_{-1}) = \mathit{nsplit}(\overline{\alpha \circ \xi \circ \xi_{\theta_0\zeta_0(i)} \circ \zeta_{\theta_0\zeta_0(i)}}, \alpha \circ \xi_{-1} \circ \zeta_{-1})$$

В результате правая часть этого равенства становится такой же, как и правая часть второго из вышеуказанных равенств с точностью до α, поэтому по Утверждению 6 будет выполнено $\sigma_{-1} = \chi_{-1}$ и $\sigma \circ \sigma_{\theta_0\zeta_0(i)} = \chi \circ \chi_{\theta_0\zeta_0(i)}$. А значит, т.к. σ и χ на самом деле являются тривиальными вложениями, верно $\sigma_i = \chi_i$, а поэтому $\mathit{canon}(\alpha(p)) = \mathit{canon}(\alpha(\mathit{canon}(p)))$.

Остальные случаи доказываются аналогично. □

Доказательство Утверждения 34, с. 132

Утверждение. Если для узла b пребисимуляции вида

$$b = \mathbf{Cong}(f_L, f_R, (\xi_L f_L \equiv \mathit{rhs}_L), (\xi_R f_R \equiv \mathit{rhs}_R), \overline{b_i})$$

выполняются неравенства вида

$$\mathit{perm}(b) \sqsupseteq (\xi_L^{op} \circ \mathit{iperm}(b) \circ \xi_R)$$
$$\mathit{iperm}(b) \sqsupseteq (\xi_L \circ \mathit{perm}(b) \circ \xi_R^{op}) \oplus \mathit{lift\text{-}perms}(\mathit{rhs}_L, \mathit{rhs}_R, \overline{\mathit{perm}(b_i)})$$
$$\mathit{perm}(b_i) \sqsupseteq \mathit{push\text{-}perm}(\mathit{iperm}(b), \mathit{rhs}_L, \mathit{rhs}_R)[i]$$

то выполняется равенства

$$perm(b) = \xi_L^{op} \circ iperm(b) \circ \xi_R$$
$$perm(b_i) = push\text{-}perm(iperm(b), rhs_L, rhs_R)[i]$$

Доказательство. Обозначим $\theta = perm(b)$ и $\theta_i = perm(b_i)$, $\tau = iperm(b)$. Начнём с первого равенства. По условию выполняется следующее соотношение:

$$\tau \sqsupseteq (\xi_L \circ \theta \circ \xi_R^{op}) \oplus lift\text{-}perms(rhs_L, rhs_R, \overline{\theta_i})$$

А значит

$$\tau \sqsupseteq \xi_L \circ \theta \circ \xi_R^{op}$$

По монотонности операции композиции получаем

$$\xi_L^{op} \tau \xi_R \sqsupseteq \xi_L^{op} \xi_L \circ \theta \circ \xi_R^{op} \xi_R = \theta$$

Таким образом, $\theta \sqsubseteq \xi_L^{op} \tau \xi_R$, а по условию также выполняется соотношение $\theta \sqsupseteq \xi_L^{op} \tau \xi_R$, поэтому $\theta = \xi_L^{op} \tau \xi_R$.

Теперь перейдём ко второму равенству. По условию выполняется соотношение $\theta_i \sqsupseteq push\text{-}perm(\tau, rhs_L, rhs_R)[i]$, поэтому таким же образом остаётся доказать, что $\theta_i \sqsubseteq push\text{-}perm(\tau, rhs_L, rhs_R)[i]$.

Для определённости будем считать, что конструкция в определениях имеет вид $\mathbf{Case}_{\overline{C_i}}$ (остальные случаи аналогичны), т.е.

$$rhs_L = \mathbf{Case}_{\overline{C_i}}(\overline{\zeta_i g_i})$$
$$rhs_R = \mathbf{Case}_{\overline{C_i}}(\overline{\sigma_i h_i})$$

Обозначим $k_i = arity(C_{i-1})$ и $k_1 = 0$, чтобы работать со всеми дочерними узлами одинаковым образом. Далее будем сокращённо писать $push\text{-}perm(\theta)$ вместо $push\text{-}perm(\theta, rhs_L, rhs_R)[i]$, $lift\text{-}perms(\overline{\theta_i})$ вместо $lift\text{-}perms(rhs_L, rhs_R, \overline{\theta_i})$, $shift(\theta)$ вместо $shift(\theta, k_i)$ и $unshift(\theta)$ вместо $unshift(\theta, k_i)$. Отметим также, что по дистрибутивности $\circ$ относительно $\oplus$ и по Утверждению 33 выполняется $push\text{-}perm(\theta \oplus \theta') = push\text{-}perm(\theta) \oplus push\text{-}perm(\theta')$ для любых θ, θ'.

Будем преобразовывать правую часть:

$$
\begin{aligned}
&\mathit{push\text{-}perm}(\tau) \sqsupseteq \\
&\quad \text{(по выполняющимся по условию соотношениям и монотонности)} \\
&\quad \sqsupseteq \mathit{push\text{-}perm}(\xi_L \theta \xi_R^{op} \oplus \mathit{lift\text{-}perms}(\overline{\theta_i})) \sqsupseteq \\
&\quad \sqsupseteq \mathit{push\text{-}perm}(\mathit{lift\text{-}perms}(\overline{\theta_i})) = \\
&\quad = \mathit{push\text{-}perm}(\ldots \oplus \mathit{unshift}(\zeta_i \theta_i \sigma_i^{op}) \oplus \ldots) \sqsupseteq \\
&\quad \sqsupseteq \mathit{push\text{-}perm}(\mathit{unshift}(\zeta_i \theta_i \sigma_i^{op})) = \\
&\quad = \zeta_i^{op} \circ \mathit{shift}(\mathit{unshift}(\zeta_i \theta_i \sigma_i^{op})) \circ \sigma_i \sqsupseteq \\
&\quad \sqsupseteq \zeta_i^{op} \circ \zeta_i \theta_i \sigma_i^{op} \circ \sigma_i = \\
&\quad = \theta_i
\end{aligned}
$$

Таким образом, $\theta_i \sqsubseteq \mathit{push\text{-}perm}(\tau)$, откуда и следует требуемое равенство. $\square$

Приложение C Дополнительные экспериментальные результаты

В Главе 4 была предложена система правил. Правила разделены на две группы: упрощающие правила и насыщающие правила. Некоторые правила являются деструктивными. Возникает вопрос: почему система правил именно такая? В частности, очень интересен вопрос, почему одни правила деструктивные, а другие нет (если все правила недеструктивные, то у системы автоматически появляются некоторые хорошие свойства, поэтому очень желательно избавиться от деструктивных правил, если они не дают никаких преимуществ).

В этом приложении приведены дополнительные результаты тестирования нашего прувера Graphsc с модифицированными системами правил. Результаты приведены в Таблицах C.1, C.2 и C.3, в которых указано соответственно потребовавшееся время в секундах, количество поколений, и количество определений в полипрограмме по окончанию доказательства. Для простоты мы запускали прувер в режиме без лемм и только на тех примерах, которые взял Graphsc в базовой конфигурации. В столбце (0) приведены результаты работы немодифицированного прувера, в остальных столбцах приведены результаты для систем правил со следующими модификациями:

- **A** — правила преобразования вызова в перестановку

(call-to-permutation) и редукции вызова тождественной функции (red-call-var) применяются недеструктивно. Для этого отключается их применение в момент добавления определения и они вместо этого применяются уже после добавления.

- **B** — правила редукции сопоставлений с образцом (red-case-cons) и (red-case-cons-bot) применяются недеструктивно, что реализовано тем же образом.
- **AB** — все вышеуказанные правила применяются недеструктивно.
- **P** — правило распространения позитивной информации (propagation-case-var) применяется как недеструктивное упрощающее правило.
- **PD** — правило распространения позитивной информации (propagation-case-var) применяется как упрощающее правило, причём **деструктивно**.
- **noG** — правило (distribute-case-case) применяется только в том случае, когда выражение в разбираемой позиции внутреннего сопоставления с образцом представляет из себя переменную. Это уменьшает возможность системы правил производить некоторые дополнительные обобщения.
- **noP** — правило распространения позитивной информации вообще не применяется.

Можно сделать следующие выводы:

- Отключение деструктивности преобразований (call-to-permutation) и (red-call-var) (столбец A) не влечёт катастрофических последствий, хотя наблюдается некоторое замедление прувера и увеличение числа определений.
- Отключение деструктивности преобразований (red-case-cons) и (red-case-cons-bot) (столбец B) приводит к тому, что несколько примеров перестают браться, однако интересно, что прувер начинает брать некоторые примеры, которые не брались в базовой конфигурации.

Таблица С.1 Сравнение модификаций graphsc, время в секундах

	(0)	A	B	AB	P	PD	noG	noP
Взято примеров	56	56	53	47	57	54	53	40
inf	1.6	1.7	1.6	1.8	1.6	1.6	1.8	1.7
idle-simple	1.7	2.0	1.7	2.0	1.8	1.7	1.7	1.8
shuffled-let	1.7	1.9	1.8	2.1	1.8	1.8	1.8	1.7
idnat-idemp	1.8	2.0	1.9	2.0	1.9	1.8	1.7	2.0
bool-eq	1.8	1.8	1.7	1.8	1.8	1.7	1.7	1.7
add-assoc	1.8	2.1	2.0	2.4	1.8	2.0	2.1	1.8
append-assoc	1.9	2.3	2.0	2.4	1.9	2.0	2.2	1.9
sadd-comm	2.2	2.3	2.3	2.4	2.1	2.1	2.1	2.1
ho/map-append	2.1	2.5	2.2	3.0	2.1	2.1	2.1	2.2
even-double	2.1	2.4	1.9	2.3	1.8	2.1	2.0	1.8
double-add	2.0	2.5	2.3	2.7	2.0	2.2	2.0	2.1
ho/fold-append	2.1	2.7	2.1	2.6	2.4	2.2	2.1	2.1
length-concat	2.3	2.9	2.6	3.5	2.4	2.3	2.3	2.2
shifted-cycle	2.7	4.2	3.2	4.8	3.7	3.1	2.8	2.2
ho/church-id	3.8	5.8	3.7	6.3	5.5	4.0	3.4	2.6
ho/map-comp	2.9	4.1	3.2	4.8	3.0	2.9	3.0	2.9
even-slow-fast	1.9	2.3	1.9	2.1	2.1	1.9	fail	1.9
deepseq-idemp	2.1	2.4	2.2	2.7	2.1	2.0	2.1	fail
add-assoc-bot	2.2	2.4	8.8	fail	2.0	2.0	2.1	2.1
deepseq-s	2.1	2.7	2.5	3.0	2.1	2.3	2.1	fail
take-drop	2.0	2.5	2.2	3.1	2.3	2.3	2.2	fail
take-length	2.1	2.4	2.0	2.5	2.1	2.1	2.1	fail
ho/filter-append	2.5	3.6	2.7	3.8	2.9	2.6	fail	2.4
mul-distrib	3.2	4.4	3.7	5.9	3.8	fail	3.3	2.1
or-even-odd	2.7	3.1	5.4	9.3	2.5	2.7	2.4	fail
ho/map-filter	2.6	3.7	3.9	5.8	3.0	2.9	fail	2.4
ho/map-concat	2.3	3.0	fail	fail	2.4	2.5	2.5	2.6
ho/concat-concat	2.4	3.4	fail	fail	2.9	2.4	2.6	2.5
append-take-drop	2.6	4.4	47.1	fail	2.8	3.0	2.6	fail
mul-assoc	6.0	9.4	fail	fail	8.9	fail	5.7	2.5
mul-double	3.9	5.8	fail	fail	4.7	fail	3.7	2.3
double-half	2.8	3.8	fail	fail	3.0	3.2	2.7	fail
exp-idle	fail	fail	264.6	fail	fail	fail	fail	fail
prop-11	1.8	1.8	1.6	2.1	1.7	1.7	1.7	1.8
prop-40	1.7	1.9	1.7	1.8	1.7	1.7	1.6	1.8
prop-44	1.8	1.8	1.7	1.9	1.7	1.8	1.7	1.6
prop-46	1.7	1.9	1.7	2.0	1.7	1.7	1.7	1.6
prop-13	1.7	2.0	1.7	2.2	1.8	1.8	1.7	1.8
prop-42	1.7	2.0	1.7	2.2	2.0	1.8	1.7	1.8
prop-45	1.9	2.1	1.7	2.2	1.7	1.8	1.8	1.8
list-weird-is-normal	2.1	2.3	2.1	2.3	2.0	1.9	2.1	2.0
prop-17	2.0	2.2	2.0	2.3	1.9	1.9	2.0	1.8
prop-31	2.0	2.4	4.7	43.2	2.0	2.3	2.2	2.0
weird-nat-add3-assoc1	2.1	3.5	2.2	3.8	2.2	2.2	2.1	2.1
weird-nat-add3-assoc3	2.2	4.0	2.4	3.9	2.3	2.3	2.3	2.3
weird-nat-add3-assoc2	2.3	3.8	2.4	4.1	2.4	2.3	2.2	2.2
prop-39	3.0	4.2	4.5	8.4	3.4	3.2	3.7	2.5
prop-02	3.5	5.1	3.7	5.8	5.1	3.4	8.8	2.6
prop-09	2.6	3.0	2.6	3.5	2.7	2.1	2.4	fail
prop-22	9.2	100.2	9.7	66.6	30.7	2.4	9.6	fail
bin-plus-comm	2.4	2.7	2.5	3.1	2.3	2.2	2.3	fail
prop-67	2.2	3.1	3.8	19.8	2.6	2.2	2.5	fail
prop-33	2.3	2.9	2.7	3.3	2.4	2.2	2.2	fail
heap-minimum	3.2	5.8	4.4	8.0	3.4	2.9	3.3	fail
prop-80	4.2	7.3	4.2	9.8	12.1	5.7	3.7	fail
prop-50	4.5	7.0	91.1	fail	6.9	2.8	3.4	fail
prop-82	2.6	3.6	fail	fail	2.6	2.5	2.4	fail
prop-55	fail	fail	9.5	fail	fail	8.3	fail	fail
int-add-ident-left	fail	fail	16.2	fail	fail	fail	fail	fail
prop-47	fail	fail	fail	fail	55.3	fail	fail	fail

Таблица С.2 Сравнение модификаций graphsc, количество “поколений”

	(0)	A	B	AB	P	PD	noG	noP
Взято примеров	56	56	53	47	57	54	53	40
inf	1	1	1	1	1	1	1	1
idle-simple	2	2	2	2	2	2	2	2
shuffled-let	1	1	1	1	1	1	1	1
idnat-idemp	3	3	3	3	3	3	3	3
bool-eq	1	1	1	1	1	1	1	1
add-assoc	3	3	3	3	3	3	3	3
append-assoc	3	3	3	3	3	3	3	3
sadd-comm	2	2	2	2	2	2	2	2
ho/map-append	4	4	4	4	4	4	4	4
even-double	3	3	3	3	3	3	3	3
double-add	6	6	6	6	6	6	6	6
ho/fold-append	3	3	3	3	3	3	3	3
length-concat	6	6	6	6	6	6	6	6
shifted-cycle	7	7	7	7	7	7	7	7
ho/church-id	5	5	5	5	5	5	5	5
ho/map-comp	5	5	5	5	5	5	5	5
even-slow-fast	4	4	4	4	4	4	fail	4
deepseq-idemp	4	4	4	4	3	3	4	fail
add-assoc-bot	6	6	6	fail	6	6	6	6
deepseq-s	5	5	5	5	4	4	5	fail
take-drop	5	5	5	5	4	4	5	fail
take-length	6	6	6	6	6	6	6	fail
ho/filter-append	5	5	5	5	5	5	fail	5
mul-distrib	6	6	6	6	6	fail	6	6
or-even-odd	7	7	7	7	6	6	7	fail
ho/map-filter	4	4	4	4	4	4	fail	4
ho/map-concat	6	6	fail	fail	6	6	6	6
ho/concat-concat	7	7	fail	fail	7	7	7	7
append-take-drop	7	7	7	fail	7	7	7	fail
mul-assoc	7	7	fail	fail	7	fail	7	7
mul-double	8	8	fail	fail	8	fail	8	8
double-half	9	9	fail	fail	9	9	10	fail
exp-idle	fail	fail	6	fail	fail	fail	fail	fail
prop-11	1	1	1	1	1	1	1	1
prop-40	1	1	1	1	1	1	1	1
prop-44	1	1	1	1	1	1	1	1
prop-46	1	1	1	1	1	1	1	1
prop-13	2	2	2	2	2	2	2	2
prop-42	2	2	2	2	2	2	2	2
prop-45	2	2	2	2	2	2	2	2
list-weird-is-normal	2	2	2	2	2	2	2	2
prop-17	2	2	2	2	2	2	2	2
prop-31	5	5	5	5	5	5	5	5
weird-nat-add3-assoc1	5	5	5	5	5	5	5	5
weird-nat-add3-assoc3	6	6	6	6	6	6	6	6
weird-nat-add3-assoc2	6	6	6	6	6	6	6	6
prop-39	5	5	5	5	5	5	6	5
prop-02	5	5	5	5	5	5	6	5
prop-09	4	4	4	4	3	3	4	fail
prop-22	5	5	5	5	5	5	5	fail
bin-plus-comm	3	3	3	3	2	2	3	fail
prop-67	5	5	5	5	5	5	5	fail
prop-33	5	5	5	5	4	4	5	fail
heap-minimum	5	5	5	5	4	4	5	fail
prop-80	7	7	5	5	7	7	7	fail
prop-50	8	8	6	fail	8	8	8	fail
prop-82	8	8	fail	fail	7	7	8	fail
prop-55	fail	fail	5	fail	fail	6	fail	fail
int-add-ident-left	fail	fail	7	fail	fail	fail	fail	fail
prop-47	fail	fail	fail	fail	9	fail	fail	fail

Таблица С.3 Сравнение модификаций graphsc, количество определений

	(0)	A	B	AB	P	PD	noG	noP
Взято примеров	56	56	53	47	57	54	53	40
inf	2	9	2	9	2	2	2	2
idle-simple	6	12	7	13	6	6	6	5
shuffled-let	6	13	6	13	6	6	6	6
idnat-idemp	8	16	10	18	8	8	8	7
bool-eq	10	23	10	23	10	10	10	10
add-assoc	13	25	14	26	13	13	13	11
append-assoc	13	25	14	26	13	13	13	11
sadd-comm	20	37	24	41	20	20	20	19
ho/map-append	24	49	38	70	24	24	24	22
even-double	25	53	30	58	25	25	25	24
double-add	29	54	49	82	29	29	29	28
ho/fold-append	34	66	35	67	44	35	34	28
length-concat	59	130	77	157	59	59	59	56
shifted-cycle	179	342	207	388	282	242	179	59
ho/church-id	474	783	493	811	736	446	474	142
ho/map-comp	169	261	208	309	169	169	169	167
even-slow-fast	12	23	16	27	12	12	fail	11
deepseq-idemp	12	27	20	49	15	13	12	fail
add-assoc-bot	24	47	206	fail	24	24	24	21
deepseq-s	21	49	31	81	22	21	21	fail
take-drop	36	77	63	160	32	26	36	fail
take-length	28	56	36	77	28	27	28	fail
ho/filter-append	75	136	105	178	83	74	fail	32
mul-distrib	251	362	393	548	331	fail	251	37
or-even-odd	82	142	306	445	89	81	82	fail
ho/map-filter	130	239	272	576	159	119	fail	94
ho/map-concat	51	95	fail	fail	51	51	51	49
ho/concat-concat	57	122	fail	fail	57	57	57	55
append-take-drop	104	344	2764	fail	113	103	104	fail
mul-assoc	526	771	fail	fail	854	fail	526	48
mul-double	467	616	fail	fail	536	fail	467	52
double-half	175	263	fail	fail	181	178	187	fail
exp-idle	fail	fail	581	fail	fail	fail	fail	fail
prop-11	9	20	10	21	9	9	9	7
prop-40	10	23	11	24	10	10	10	8
prop-44	8	19	9	20	8	8	8	8
prop-46	9	21	10	22	9	9	9	8
prop-13	10	22	12	25	10	10	10	9
prop-42	10	24	12	26	10	10	10	10
prop-45	10	24	12	26	10	10	10	10
list-weird-is-normal	12	28	13	29	12	12	12	12
prop-17	18	38	22	42	18	18	18	17
prop-31	21	39	217	1148	21	21	21	20
weird-nat-add3-assoc1	24	50	28	54	24	24	24	23
weird-nat-add3-assoc3	28	58	35	62	28	28	28	27
weird-nat-add3-assoc2	29	59	38	66	29	29	29	28
prop-39	152	289	777	1235	218	173	221	71
prop-02	213	354	271	428	337	219	338	74
prop-09	59	118	85	165	51	19	59	fail
prop-22	188	281	229	332	267	33	188	fail
bin-plus-comm	57	103	64	110	42	34	57	fail
prop-67	52	140	265	461	63	44	52	fail
prop-33	57	101	80	142	47	45	57	fail
heap-minimum	354	591	498	853	287	158	343	fail
prop-80	283	437	265	614	586	342	237	fail
prop-50	304	478	26	fail	478	82	183	fail
prop-82	81	134	fail	fail	77	76	81	fail
prop-55	fail	fail	22	fail	fail	408	fail	fail
int-add-ident-left	fail	fail	7	fail	fail	fail	fail	fail
prop-47	fail	fail	fail	fail	95	fail	fail	fail

- Когда все правила применяются недеструктивно (AB), это уже достаточно ощутимо — эта конфигурация берёт суммарно на 9 примеров меньше, на остальных наблюдается замедление.

- Правило распространения позитивной информации имеет смысл включить в число упрощающих, т.к. это ведёт к тому, что берётся один новый пример (столбец P).

- Если правило распространения позитивной информации применять деструктивно (столбец PD), то перестают браться несколько примеров. Отметим, что ещё один пример, когда распространения позитивной информации лучше избежать, приведён в [88] (он не вошёл в наш набор примеров, но он также перестаёт браться нашим прувером, если правило распространения позитивной информации применяется деструктивно).

- Если ослабить правило (distribute-case-case) (столбец noG), то также перестают браться некоторые примеры, в частности Пример 14.

- Если отключить распространение позитивной информации (столбец noP), то сразу много примеров перестают браться (тем не менее, многие примеры его не требуют).