Московский государственный университет имени М.В. Ломоносова

На правах рукописи

Новикова Татьяна Анатольевна

Математические модели и методы в решении задач реорганизации программ

01.01.09 – Дискретная математика и математическая кибернетика

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
д. ф.-м. н., профессор
Захаров Владимир Анатольевич

Москва – 2016

Оглавление

Введение

1. Задача проверки эквивалентности программ

При проектировании программного обеспечения часто возникают ситуации, когда различные фрагменты программы выполняют схожие действия с поступающими в них данными. Такое поведение отдельных фрагментов программ приводит не только к трудности понимания логики программы, но и к увеличению затрат на ее дальнейшую разработку и сопровождение. Статистические исследования открытых исходных кодов свободно распространяемого программного обеспечения на языках C и Java, результаты которых опубликованы в работах [6,51,79], свидетельствуют о том, что суммарный объем дублируемого кода в крупных программных проектах обычно составляет от 7 до 23% объема исходного кода. Помимо трудностей, возникающих у разработчиков в связи с поддержкой такого кода, дублируемые фрагменты требуют избыточных ресурсов, затрачиваемых компилятором и системой программирования в целом на компиляцию и хранение объектного кода. Поиск таких фрагментов — клонов — даже в небольших программах является очень трудным делом, осуществление же его вручную на промышленных проектах фактически невозможно. В связи с этим возникает потребность в автоматическом средстве обнаружения клонов, которое также позволило бы осуществить проверку того, можно ли путем подстановки вместо входных переменных одного фрагмента программы некоторых инициализирующих выражений, сделать его эквивалентным другому фрагменту программы. В этом случае несколько разных фрагментов можно заменить вызовом одной и той же процедуры с разными значениями ее параметров. Очевидно, что вследствие этого размер программы может значительно сократиться. Но гораздо важнее то, что и логика программы после такого изменения станет гораздо более понятной. Такой подход к трансформации программ широко используется в рефакторинге — изменении структуры программы, не

затрагивающем ее внешнего поведения [9, 27, 62, 81]. Этот метод рефакторинга называется выделением процедуры. На сегодняшний день существует несколько средств автоматического рефакторинга программ (см. обзоры [77, 80]), однако выделение метода по-прежнему требует особого внимания, т.к. существующие средства решения этой задачи опираются по большей части на эвристики, применимые к отдельным языкам программирования, но не универсальные. Некоторые из существующих подходов рассмотрены в конце этого параграфа.

Стоит отметить, что класс задач, в которых важную роль играет отношение эквивалентности, не ограничивается только задачами рефакторинга. Так, необходимость проверки фрагментов кода на эквивалентность возникает при построении оптимизирующих компиляторов как последовательных, так и параллельных программ [1, 49, 53, 71, 100]. Как было сказано выше, обнаружение дубликатов в программном коде позволяет сократить объем памяти, необходимой для хранения объектного файла. Кроме того при разработке оптимизирующих преобразований программ необходимо проверять, что функциональность кода, полученного в результате оптимизации, не изменяется, то есть построенная оптимизированная программа эквивалентна исходному коду.

Аналогичные задачи возникают также при построении суперкомпиляторов [89, 144]. Как известно, одним из этапов работы суперкомпилятора является свертка ветвей вычислений, и чем эффективнее алгоритм проверки “похожести” пары ветвей, тем эффективнее будет работать суперкомпилятор.

Алгоритмы проверки эквивалентности программ также находят применение в области компьютерной безопасности. В частности, потребность в них возникает при решении задачи обфускации программ [22, 23, 32]. Под обфускацией понимается такое преобразование исходного текста программы, при котором ее функциональность сохраняется, но получение информации о специфических особенностях алгоритмов и структур данных, содержащихся в программе, становится значительно более трудоемкой задачей. Чтобы конструировать обфускирующие преобразования программ, необходимо иметь средство для доказа-

тельства того, что обфускация корректна, то есть программа, полученная в результате обфускирующего преобразования, эквивалентна исходной программе. Кроме того, сложность задачи проверки эквивалентности программ может служить мерой качества (стойкости) обфускации программ: если удается легко доказать эквивалентность двух разных обфускированных вариантов одной и той же программы, то это является признаком того, что данный метод обфускации не обладает хорошей стойкостью.

Одним из популярных и эффективных средств поиска вредоносного кода на сегодняшний день является проверка соответствия фрагментов подозрительного кода образцам (сигнатурам) известных вредоносных программ, что также делает задачу поиска вредоносного программного обеспечения зависящей от эффективных алгоритмов проверки эквивалентности программ [20, 21, 93, 95]. Особенно это важно для поиска метаморфных вирусов, способных модифицировать свой код в процессе репликации и не имеющих по этой причине постоянной сигнатуры [117].

Для того, чтобы создать средство автоматического поиска клонов, необходимо в первую очередь выбрать подходящую математическую модель программ, в рамках которой можно было бы не только адекватно формализовать отношение подобия (“похожести”) программ, но также иметь возможность эффективно распознавать это отношение.

Под программой в данном случае может пониматься не только система команд на некотором языке программирования, но и объект любой вычислительной модели, например, машина Тьюринга, машина Поста, конечный автомат, нормальный алгоритм и так далее. При этом в зависимости от выбранной модели и поставленной задачи определяется и отношение подобия, которое изучается на объектах данной модели. Наиболее естественным для задач такого типа является отношение функциональной эквивалентности программ, при котором каждая программа рассматривается как описание функции, преобразующей набор входных данных в набор выходных данных, а эквивалентность

двух заданных программ определяется как совпадение реализуемых ими функций. Однако теорема Райса-Успенского [73] утверждает, что функциональная эквивалентность неразрешима в любой модели вычислений, в которой реализуемы все вычислимые по Тьюрингу функции. Это означает, что для анализа программ необходимо выбирать более строгие, но разрешимые виды эквивалентности, которые аппроксимировали бы функциональную эквивалентность. Фактически именно анализ и исследование разных видов эквивалентности на различных моделях вычислений лежит в основе задач семантического анализа программ. Далее приведены примеры моделей программ и введенных для них видов эквивалентности; некоторые из указаных видов эквивалентности оказались разрешимыми, в отдельных случаях показана возможность решения задач проверки эквивалентности за полиномиальное время.

1.1. Модели программ и задача проверки эквивалентности в рамках этих моделей

Одной из первых математических моделей программ считаются схемы программ Ляпунова-Янова [125, 126]. В шестидесятые годы начинается быстрое развитие программируемой вычислительной техники и языков программирования. Схемы Ляпунова-Янова отражали основные принципы программирования, существовавшие на тот момент. Эти принципы фактически легли в основу парадигмы императивного процедурного программирования. В формализации схем Ляпунова-Янова, предложенной А. П. Ершовым [109], схема представляет собой граф с вершинами двух типов: вершины-преобразователи, соответствующие элементарным операторам, и вершины-распознаватели, соответствующие логическим условиям или тестам. Вершины этого графа отражают инструкции изменения состояния переменных (функциональные символы для операторов присваивания) или проверки некоторых условий (предикатные символы логических операторов) над переменными реальной программы. Дуги графа при этом описывают правила передачи управления между вершинами в зависимости от

значений предиката, приписанного вершине-распознавателю. Перед выполнением схемы задается интерпретация, которая наделяет смыслом функциональные и предикатные символы. Само выполнение схемы состоит в обходе графа и накоплении информации об изменении состояний. При этом путь, по которому совершается обход графа, определяется в соответствии с интерпретацией.

Схемы Ляпунова-Янова считаются эквивалентными, если при любой интерпретации либо обход этих схем бесконечен, либо накопленные в процессе обхода этих схем последовательности операторных символов совпадают. В работе [145] был разработан алгоритм проверки эквивалентности схем Ляпунова-Янова, тем самым была показана разрешимость проблемы эквивалентности для этой модели программ.

В последующие годы было предложено большое количество других разновидностей моделей программ. Рассмотрим некоторые из них.

Теория дискретных преобразователей [107, 108, 120–122] предполагает следующую формализацию понятия программы. Программа представляет собой систему из двух взаимодействующих автоматов. Первый из них описывает поток управления в программе, в то время как второй описывает семантику ее операторов и предикатов. Фактически, второй автомат определяет класс интерпретаций, для которых может рассматриваться задача проверки эквивалентности программ. Оказалось, что в общем случае проблема эквивалентности для дискретных преобразователей неразрешима, но для некоторых автоматов, задающих семантику операторов, алгоритмы проверки эквивалентности дискретных преобразователей могут быть построены. Так, например, в работах [108, 123] установлены необходимые и достаточные условия разрешимости проблемы эквивалентности для случаев, когда семантика базовых операторов программ определяется при помощи полугрупп.

В работе [7] впервые были описаны схемы программ, ориентированные на функциональную парадигму программирования — рекурсивные схемы. Более детально они были изучены в работах [3, 30, 68, 86]. В работе [30] было построен

подкласс рекурсивных схем, по выразительным возможностям подобный схемам Ляпунова-Янова. Для этого подкласса была показана разрешимость проблемы эквивалентности. Кроме того, проблема эквивалентности оказалась разрешимой также для более выразительных классов рекурсивных схем [3, 30, 124] и исследования этой задачи продолжаются [140–142] . Но в общем случае проблема эквивалентности для рекурсивных схем программ оказалась неразрешимой [28].

Алгебраические модели программ, впервые введенные в работе [127], обобщили идеи модели дискретных преобразователей и схем Ляпунова-Янова. Особенностью алгебраических моделей программ по отношению к задаче проверки эквивалентности можно считать тот факт, что они позволяют использовать семантические свойства составляющих программы при решении указанной задачи [128, 129, 131, 137]. Центральной задачей при изучении данной модели было построение системы эквивалентных преобразований, что неизбежно привело к появлению интереса к задаче проверки эквивалентности в этой модели. Эта задача была успешно решена в работе [134].

Динамические модели программ, введенные в работах [111, 112], описывают семантику программ на языке динамической логики. В работе [113] описана методика разрешения эквивалентности в этой модели.

Модель программ, использованная в данной работе, была впервые представлена в 1967 году в работе [109]. Данная модель, названная моделью стандартных схем программ, представляет собой дальнейшее развитие концепции схем программ Ляпунова-Янова посредством использования языка первого порядка для описания операторов и логических условий и теоретико-графового представления схем, что значительно повысило уровень детализации и упростило понимание исходной модели. В дальнейшем методы, предложенные для стандартных схем программ, часто использовались в теории статического анализа программ [119]. Функциональная эквивалентность для данной модели оказалась неразрешимой [58, 67]. Однако для других видов эквивалентности, аппрок-

симирующих функциональную, была показана их разрешимость. Так, например, логико-термальная эквивалентность, введенная в работе [118], оказалась не только разрешимой [106], но для нее удалось построить алгоритмы, требующие полиномиального относительно размеров программ времени [119, 143].

Таким образом, из теоремы Райса-Успенского вовсе не следует невозможность построения эффективно проверяемых достаточных условий функциональной эквивалентности программ. Теорема скорее утверждает, что даже если такие условия будут построены, они будут лишь достаточными, но не будут являться необходимыми. Далее рассматривается логико-термальная эквивалентность — эффективно разрешимый вид эквивалентности, вводимый для стандартных схем программ, изучению которого посвящена эта работа.

1.2. Логико-термальная эквивалентность стандартных схем программ

Стандартные схемы программ, введенные в работе [109], позволили значительно упростить понимание модели программ Ляпунова-Янова благодаря графовому представлению. Приведем краткое описание этой модели, с подробным описанием модели и формальной постановкой задачи проверки логико-термальной эквивалентности можно ознакомиться в главе 1.

Последовательна императивная программа рассматривается как размеченный ориентированный граф π. Каждой вершине этого графа приписано некоторое логическое условие — предикат. Из каждой вершины за исключением одной исходит две дуги, при этом одна из дуг помечена символом 0, а другая — символом 1. Кроме того, каждой дуге графа приписана еще одна пометка, представляющая собой подстановку, то есть отображение, которое ставит в соответствие каждой переменной программы некоторый терм. Отдельно выделены две вершины — вход и выход программы, при этом обе дуги, исходящие из входной вершины, направлены в одну и ту же вершину, а выходная вершина — единственная из вершин программы, из которой не исходит ни одна дуга. Счи-

тается, что через каждую вершину программы проходит некоторый маршрут, который проходит из входа программы в ее выход.

Семантически вершины программы соответствуют операторам ветвления, или условным операторам программы. Дуги программы при этом описывают линейные участки, представляющие собой конечные списки операторов присваивания, изменяющих значения переменных, между операторами ветвления. При это подстановка каждого конкретного линейного участка строится в соответствии с тем эффектом, который оказывает совокупное применение всех операторов присваивания на текущее состояние данных.

Программа осуществляет вычисления над абстрактными данными, которые являются значениями переменных этой программы. Само вычисление заключается в обходе графа программы и начинается с входной вершины графа. Для того, чтобы такой обход был детерминирован, необходимо наделить смыслом не только множество входных переменных, но и каждый преобразователь, которому сопоставляется некоторая функция, и каждый распознаватель, которому сопоставляется некоторая логическая функция. Все это задается при помощи интерпретации, которая также позволяет определить, какая именно из дуг с пометками 0 и 1, исходящих из текущего распознавателя, должна быть выполнена. В зависимости от того, по дугам с какими метками осуществлялся обход, определяется путь в программе. Таким образом интерпретация наделяет семантикой все компоненты программы. Вычисление подразумевает запоминание последовательности состояний данных в соответствии с преобразованиями, происходящими над переменными во время обхода. Обход графа продолжается до тех пор, пока не будет достигнута выходная вершина. Полученное к этому моменту состояние памяти и будет результатом вычисления программы.

Логико-термальная эквивалентность, впервые введенная в работе [118], не использует в своем определении интерпретации функций и предикатов программы, поэтому на нее не распространяется результат статьи [38] о неразрешимости невырожденных интерпретационных отношений эквивалентности на стандарт-

ных схемах программ. Данная особенность логико-термальной эквивалентности стандартных схем программ делает это отношение привлекательным для использования при решении некоторых задач статического анализа программ. Рассмотрим задачу проверки логико-термальной эквивалентности стандартных схем программ подробнее.

Пусть задана программа π. Рассмотрим произвольный маршрут в графе программы π, ведущий из входа программы в ее выход. Под логико-термальной историей маршрута будем понимать информацию о том, в каком порядке этот маршрут обходил вершины графа и каковы были значения аргументов логических условий при прохождении каждого из распознавателей, приписанных вершинам. Формально логико-термальная история маршрута представляет собой последовательность атомарных формул, аннотированных теми тх логическими значениями, которые соответствуют пометкам дуг этого маршрута. Две программы π_1 и π_2 считаются логико-термально эквивалентными, если множества всех логико-термальных историй их маршрутов совпадают. Иными словами, какой бы маршрут из входа в выход мы ни выбрали в одной из программ, в другой обязательно найдется маршрут из входа в выход с точно такой же логико-термальной историей. Важная особенность логико-термальной эквивалентности состоит в том, что она аппроксимирует функциональную эквивалентность программ для любой интерпретации элементов программы.

Первый алгоритм распознавания логико-термальной эквивалентности был представлен в той же работе [118], где она вводилась. Однако этот алгоритм был достаточно сложным. Позже был предложен алгоритм [106], который сводил задачу распознавания логико-термальной эквивалентности к задаче распознавания эквивалентности в классе двухленточных автоматов. Этот алгоритм имел экспоненциальную оценку сложности.

Первый полиномиальный алгоритм был представлен в работе [143] и имел оценку сложности $O(n^7)$, где n — максимальный из размеров исходных фрагментов программ. Этот алгоритм состоит из нескольких шагов. Первый шаг

предполагает преобразование исходных фрагментов программ π_1, π_2 к приведенному виду, то есть к таким фрагментам, в которых каждая вершина достижима из входа, из каждой вершины достижим выход, каждый предикат содержит обращения только к переменным, а не к более сложным термам над ними. Для осуществления этого шага алгоритм предоставляет восемь эквивалентных преобразований. В том случае, если построенные фрагменты π_1' и π_2' не изоморфны, алгоритм объявляет исходные фрагменты не логико-термально эквивалентными. Иначе по имеющимся преобразованным фрагментам строится граф π, по сути являющийся декартовым произведением графов π_1' и π_2', после чего к этому графу применяется одно из восьми правил преобразований, называемое заменой термов; с помощью этого правила строится стационарная разметка графа π. Если же построение правильной стационарной разметки с использованием правила замены термов невозможно, то исходные фрагменты программ π_1, π_2 признаются не логико-термально эквивалентными. Доказательство корректности этого алгоритма опирается на корректность алгоритма Берда [16] распознавания эквивалентности двухленточных автоматов.

2. Задача выделения метода

Задача выделения метода, как было отмечено выше, является одной из наиболее трудоемких в рефакторинге. Код “с душком” [27] в данном случае подразумевает наличие особых фрагментов программы, которые могут неоднократно входить в код, возможно, немного видоизмененными. Такие фрагменты принято называть подозрительными. В простейшем варианте задача выделения метода предполагает следующий набор действий:

1. поиск подозрительного фрагмента программы;
2. поиск вхождений фрагментов, эквивалентных подозрительному, — его клонов;

3. если вхождения обнаружены, то создание отдельной процедуры, код которой соответствует коду подозрительного фрагмента, и замена всех вхождений клонов подозрительного фрагмента вызовом выделенной процедуры.

Существующие на сегодняшний день средства автоматического рефакторинга довольно часто позволяют найти подозрительные фрагменты и составляют список их возможных клонов, предоставляя программисту самостоятельно решать, возможно ли выделение метода в каждом конкретном случае (см., например, [39]). Иногда такой подход оказывается действительно выгодным программисту — это происходит в тех случаях, когда выделение метода нежелательно в связи с повышением модульности проекта, однако при необходимости внесения изменений в схожие фрагменты также решается задача выделения метода. При этом выделяются четыре разновидности потенциально возможных клонов [99]:

- *Тип 1.* Одинаковые фрагменты программного кода, для которых допускается разное форматирование.
- *Тип 2.* Синтаксически одинаковые фрагменты программного кода, для которых допускается переименование переменных и констант.
- *Тип 3.* Функционально эквивалентные фрагменты, которые могут быть получены удалением или добавлением некоторого количества строк из подозрительного фрагмента (например, развертка циклов).
- *Тип 4.* Функционально эквивалентные фрагменты, имеющие абсолютно разный синтаксис.

При этом в большинстве работ, исследующих выделение метода, приводятся алгоритмы поиска только клонов первых трех типов [37, 47, 76, 84, 96].

Отдельный интерес представляют конкретные реализации программных средств, позволяющих проводить поиск программных клонов в исходном коде.

Подходы, которые используются для поиска клонов в такого рода средствах, можно разделить на три основные группы: средства, основанные на синтаксическом анализе, средства, в основе которых лежит семантический анализ кода, а также средства, осуществляющие лексемный анализ кода.

Первая группа средств предполагает использование синтаксического анализатора, который переводит код программы во внутреннее представление. При этом в основе такого представления обычно лежат абстрактные синтаксические деревья. Впоследствии построенные деревья обрабатываются при помощи техник сопоставления деревьев (tree matching) или вводимых на деревьях метрик.

Подход, основанный на сопоставлении деревьев, предполагает поиск наиболее схожих поддеревьев и обычно опирается на синтаксические особенности языка ([14, 18, 26, 45, 90, 97]). Метрический подход ([8, 44, 60, 66]) предполагает введение понятия “расстояние” для фрагментов программного кода. Для каждого фрагмента кода строится вектор расстояний внутри фрагмента, а затем происходит сопоставление векторов, а не самих синтаксических деревьев.

Семантический анализ программного кода ([29, 43, 46, 57]) чаще всего опирается на модель программы, называемую графом программных зависимостей [64]. Вершины этого графа представляют собой выражения и операторы программы, в то время как дуги отражают зависимости одних операторов от других по переменным или же достижимость одних из операторов из других в зависимости от условных выражений в программе. Такой подход позволяет абстрагироваться от того, в каком порядке встречаются выражения в тексте, но при этом учитывает, являются ли они зависимыми друг от друга по смыслу. При таком подходе поиск клонов становится поиском изоморфных подграфов в заданном графе, что может оказаться довольно непростой задачей. Одним из подходов, позволяющих упростить такой поиск, является слайсинг [40, 41, 94] — метод анализа графа, опирающийся на семантические свойства его вершин. Этот метод позволяет разделить граф программных зависимостей на слои — фрагменты кода, в которых между всеми вершинами существует семантиче-

ская связь. При применении слайсинга к графам программных зависимостей двух программ сначала выбирается пара вершин, имеющих одинаковую пометку, после чего осуществляется подъем (backward slicing, [94]) или спуск (forward slicing [34]) по графам программных зависимостей с целью построения изоморфных подграфов. Отличительной особенностью методов слайсинга является возможность нахождения клонов, в которых операторы могут быть переставлены.

Третий подход, опирающийся на анализ текста как набора лексем, фактически ставит каждой конструкции текста в соответствие некий специальный маркер, а затем проверяют программу — последовательность маркеров — на наличие повторяющихся подпоследовательностей ([10, 11]).

Еще один подход, описанный в работах [76,78], также предполагает использование специфического синтаксического анализатора, который помимо анализа строк программного кода позволяет приводить их к некоторому общему виду, а затем осуществляет поиск клонов путем сравнения строк символов. Этот подход не позволяет находить клоны типа 2, является зависимым от конкретного языка программирования, но в то же время является легковесным и выполняет свою работу достаточно быстро по сравнению с другими алгоритмами [78].

Приведенные выше примеры свидетельствуют о том, что задача проверки эквивалентности программ требует систематического подхода, так как на сегодняшний день в основе многих методов поиска клонов лежат либо алгоритмы, требующие экспоненциального времени работы, либо алгоритмы, способные находить только клонов первых трех типов. В то же время, как было показано выше, существуют аппроксимирующие функциональную эквивалентность виды эквивалентностей, которые на соответствующих моделях разрешимы за полиномиальное время, что свидетельствует о потенциальной возможности построения полиномиального алгоритма для проверки эквивалентности.

3. Задачи унификации и антиунификации

Важную роль в исследовании, представленном в данной работе, играют задачи унификации и антиунификации термов и подстановок.

Задача унификации двух выражений E_1 и E_2 заключается в том, чтобы найти для них такие подстановки η_1 и η_2, что верно $E_1\eta_1 = E_2\eta_2$, здесь имеется ввиду синтаксическое совпадение построенных выражений.

Задача синтаксической унификации термов возникла при разработке метода резолюций для систем автоматического доказательства теорем [74]. Именно на основе метода резолюций в дальнейшем разрабатывалась концепция логического программирования, поэтому алгоритмы унификации стали одним из базовых механизмов при вычислении логических программ. Это породило интерес к созданию эффективных алгоритмов синтаксической унификации. Такие алгоритмы были созданы, они имели почти линейную сложность [13,59,69]. Кроме того, были разработаны структуры данных, удобные для построения их программной реализации. Однако для логик высокого порядка задача унификации оказалась неразрешимой [31].

Существует также другая разновидность задач унификации, когда эквивалентность двух выражений предполагает равенство их значений в некоторой аксиоматической теории, то есть при условии, что в алгебре термов введены определяющие тождества, отражающие, например, законы коммутативности, ассоциативности и так далее. Такая унификация получила название E-унификации, или семантической унификации. В области семантической унификации были получены результаты для ассоциативных и коммутативных теорий [35,85,98]. Изучаемые в данной работе задачи унификации программ являются задачами семантической унификации.

Задача унификации впоследствии была широко исследована. Оказалось, что она представляет интерес не только с точки зрения логического программирования и создания систем автоматического доказательства теорем. Похожие

задачи встречаются и в других областях, например, при обработке текстов, как на формальных, так и на естественных языках, а также при построении систем переписывания термов и разработке языков программирования [42].

В данной работе также активно используются существующие решения задачи антиунификации, двойственной к задаче унификации. Эта задача заключается в том, чтобы для двух заданных выражений E_1 и E_2 отыскать такое выражение E_0, что существуют подстановки ν_1 и ν_2, для которых было бы выполнено $E_1 = E_0\nu_1$ и $E_2 = E_0\nu_2$, и при этом E_0 — самое специальное из всех таких выражений. Такое выражение называется наиболее специальным шаблоном. Впервые задача антиунификации выражений была рассмотрена независимо в работах [70] и [72]. В этих работах были разработаны алгоритмы построения наиболее специальных шаблонов с точностью до переименования переменных для термов логики первого порядка. После этого задача была исследована в ряде работ [52,63,65,83,92,103,114], были разработаны алгоритмы антиунификации термов. Однако, в отличие от задачи унификации термов, являющейся P-полной, задача антиунификации оказалась NC -полна [50]. В связи с этим был разработан также ряд параллельных алгоритмов вычисления антиунификатора пары заданных подстановок [24,50]. Кроме того, было показано еще одно отличие задачи антиунификации от задачи унификации. Дело в том, что размер наиболее специального шаблона двух подстановок не превосходит размера самих исходных подстановок, если они представлены деревьями. Но в случае их представления ациклическими ориентированными графами размер наиболее специального шаблона может квадратично зависеть от размеров исходных подстановок [114].

Изучение семантической антиунификации продвигается медленнее, чем изучение семантической унификации. В частности, попытки построения алгоритма антиунификации были предприняты в некоторых работах [2,4,19], но эти алгоритмы представляют собой набор способов улучшения прямого перебора. Была также исследована антиунификация для логик высокого порядка [33,48] и

для логики первого порядка с допущением о неранжировании функциональных символов [12], то есть о том, что местность функционального символа в каждой формуле нефиксирована, он может входить в формулу с разной местностью. Оказалось, что такого рода антиунификация может быть широко применена для анализа текстов с частичной разметкой, например, HTML-страниц.

Идея семантической унификации может быть распространена на стандартные схемы программ, к примеру, следующим образом. Предположим, что две заданные программы π_1, π_2 не являются эквивалентными. Тогда задачу унификации таких программ можно было бы сформулировать следующим образом: найти, если это возможно, пару линейных программ (программ, состоящих лишь из последовательных операторов присваивания) A, B таких, что программы, полученные в результате последовательного выполнения программы A и π_1, а также программ B и π_2, оказались бы эквивалентными.

Еще одним вариантом задачи унификации программ может послужить следующая задача. Пусть заданы две программы π_1 и π_2. Задачей двусторонней унификации программ π_1 и π_2 назовем задачу поиска такой пары линейных программ (A', A''), что программа, полученная в результате последовательного выполнения программ A', π_1, A'', оказалась бы эквивалентна программе π_2.

Эти задачи имеют некоторое прикладное значение. В частности применительно к задаче выделения метода благодаря решению этих задач можно было бы выделить пятый тип клонов: фрагменты программ, которые эквивалентными не являются, но могут быть унифицированы путем применения к состоянию данных ряда преобразований. Тогда процедура выделения метода могла бы оперировать с большим количеством фрагментов автоматически предлагая те правила изменения данных — те подстановки и линейные подпрограммы — которые необходимо произвести, чтобы заменить вхождение фрагмента простым вызовом процедуры.

Обе указанные выше задачи являются фактически задачами о решении уравнений над подстановками: в первом случае эти уравнения представляют

собой, как и классическая задача унификации подстановок, уравнения относительно переменных A и B вида $\theta_1 A = \theta_2 B$. Во втором случае — при двусторонней унификации — вид этих уравнений будет $A\theta_1 B = \theta_2$.

Подобные задачи ранее встречалась и при проверке эквивалентности в некоторых подклассах последовательных программ [101]. Возникающие при этом уравнения над подстановками могут быть описаны так: $A\theta_1 = B\theta_2$.

Описанные задачи имеют интересное прикладное значение. Построенный алгоритм унификации программ по входам позволяет расширить класс фрагментов программ, к которым можно было бы применить выделение метода. Предположим, что задана подозрительная подпрограмма π' и необходимо произвести выделение метода π' из программы π. Можно построить пример такой программы π, которая не будет содержать фрагментов, логико-термально эквивалентных подпрограмме π', но при этом будет содержать фрагменты, логико-термально унифицируемые с π'. Те же идеи могут быть применены к задаче двусторонней унификации программ, при этом задача двусторонней унификации позволяет провести четкую границу между выделяемой подпрограммой и вхождением двусторонне унифицируемого с ней фрагмента в код программы. При этом, как уже было сказано выше, пара двусторонних унификаторов может быть представлена в виде двух последовательных программ; правила, по которому эти программы изменяют состояния памяти, описаны унифицирующими подстановками и также могут быть построены алгоритмом.

4. Цели исследования.

Целью данной диссертационной работы является решение следующих задач.

1. Разработать новый алгоритм проверки логико-термальной эквивалентности программ, который опирался бы на свойства решетки подстановок и превосходил бы по эффективности ранее известные алгоритмы решения

этой задачи.

2. Разработать алгоритм, осуществляющий за полиномиальное время проверку того, могут ли две заданные программы стать логико-термально эквивалентными в результате применения некоторой подстановки к их входным переменным. В случае положительного ответа, алгоритм должен строить такую унифицирующую подстановку.

3. Изучить задачу, которая заключается в проверке того, могут ли две заданные программы стать логико-термально эквивалентными в результате применения к входным и выходным переменным одной из них некоторой пары подстановок. Определить, к какому классу сложности относится эта задача. Разработать алгоритм, разрешающий эту задачу и конструирующий указанную пару подстановок, если это возможно.

5. Результаты исследования

5.1. Формулировка основных результатов

Данная диссертационная работа посвящена решению задачи проверки логико-термальной эквивалентности программ и некоторых смежных с ней задач. Основными целями данной работы являются построение эффективного алгоритма проверки логико-термальной эквивалентности, а также построения алгоритма, позволяющего осуществить проверку логико-термальной унифицируемости двух заданных фрагментов программ, и в случае, если это возможно, представить средства для их унификации. В этом разделе приводится описание понятий, необходимых для формулировки центральных результатов. Более строго эти понятия формулируются в главах 1 и 3, а неформальное описание стандартных схем программ и логико-термальной эквивалентности можно также найти в разделе 1.2 введения.

Результат применения к программе π подстановки θ — программа $\theta;\pi$ —

может быть описан как результат последовательного выполнения программ θ и π. Фактически первая из выполняемых программ просто изменяет состояние переменных, то есть эта программа представляет собой линейный фрагмент кода, состоящий только из инструкций присваивания, для простоты назовем такую программу линейной. Граф программы $\theta;\pi$ отличается от графа программы π тем, что пометки обеих дуг, ведущих из входной вершины, учитывают результат воздействия на переменных программы θ.

Основываясь на понятии логико-термальной эквивалентности, можно привести следующее описание понятия логико-термальной унификации программ. Логико-термальная унификация программ π' и π'' подразумевает поиск такой пары линейных программ (подстановок) θ' и θ'', что программы $\theta';\pi'$ и $\theta'';\pi''$ логико-термально эквивалентны. В том случае, если такие программы θ' и θ'' существуют, будем называть их также унификаторами программ π' и π''.

Результат применения к программе π пары подстановок (η', η'') — программа $\eta';\pi;\eta''$ — может быть описан как результат последовательного выполнения трех программ: η', π, η''. При этом программы, соответствующие подстановкам η' и η'' есть линейные программы. Их суть заключается в том, что они просто изменяют состояние данных до и после запуска программы π. Граф полученной в результате программы будет отличаться от графа программы π тем, что дугам, ведущим из выходной вершины, будет приписана новая подстановка, учитывающая результат воздействия на переменные программой η', а дугам, ведущим в выходную дугу, — подстановка, учитывающая результат воздействия на переменные программой η''.

Под двусторонней унификацией программ π' и π'' подразумевается поиск такой пары линейных программ (η', η''), что программы $\eta';\pi';\eta''$ и π'' логико-термально эквивалентны. В том случае, если такая пара существует, она называется двусторонним унификатором программ π' и π''. Отметим, что отношение двусторонней унифицируемости не является коммутативным.

Перечислим основные результаты, выносимые на защиту:

1. Разработан алгоритм, проверяющий логико-термальную эквивалентность программ за время $O(n^6)$, где n — суммарный размер проверяемых программ. В основу алгоритма положены операции в алгебре подстановок. Ранее известные алгоритмы решения этой задачи имеют сложность $O(n^7)$.

2. Разработан алгоритм, проверяющий унифицируемость программ за время $O(n^{11})$. В случае унифицируемости программ данный алгоритм строит наиболее общий унификатор анализируемых программ.

3. Введено понятие двустороннего унификатора для подстановок и программ. Показано, что задачи проверки двусторонней унифицируемости подстановок и двусторонней логико-термальной унифицируемости программ, являются NP-полными. Разработан алгоритм, проверяющий двустороннюю логико-термальную унифицируемость программ и строящий двусторонний унификатор заданных программ в случае его существования.

Полученные результаты были представлены на следующих конференциях:

- VI Международная научная конференция студентов, магистрантов и молодых ученых «Ломоносов-2010» (Астана, 2010);

- XVI Международная конференция «Проблемы теоретической кибернетики» (Нижний Новгород, 2011);

- VIII Международная научная конференция студентов, магистрантов и молодых ученых «Ломоносов-2012» (Астана, 2012);

- XI Международный семинар «Дискретная математика и ее приложения», посвященный 80-летию со дня рождения академика О.Б. Лупанова (Москва, 2012);

- 27th International Workshop on Unification (UNIF-2013) (Эйндховен, 2013);

- IX Международная научная конференция студентов, магистрантов и молодых ученых «Ломоносов-2013» (Астана, 2013);
- 28th International Workshop on Unification (UNIF-2014) (Вена, 2014);
- XVII Международная конференция «Проблемы теоретической кибернетики» (Казань, 2014);
- Форум ученых СНГ — 2015 (Москва, 2015).

Результаты были опубликованы в работах [104, 105, 146–154], работы [146, 148, 149, 153] были опубликованы в журналах перечня ВАК.

5.2. Методика получения результатов

В основе алгоритмов, представленных в данной работе, лежит метод совместных вычислений. Этот метод неоднократно использовался при исследовании проблемы эквивалентности программ [102, 111, 113, 115, 116], схожие идеи также были представлены в работе [87].

Основная идея метода совместных вычислений может быть коротко сформулирована следующим образом. При описании модели, как было сказано выше, задается интерпретация. В рамках этой интерпретации две программы, производя свои вычисления одновременно, строят пары вычислительных конфигураций. В результате проведения анализа всех возможных построенных пар, метод совместных вычислений позволяет сделать выводы об их эквивалентности.

В данной работе метод совместных вычислений рассматривается в применении к проверке логико-термальной эквивалентности стандартных схем последовательных императивных программ, а также к задаче проверки унифицируемости двух заданных программ. Покажем, как именно конкретизируется метод совместных вычислений при рассмотрении поставленной задачи.

Модель стандартных схем программ предполагает, что каждому вычислению программы может быть поставлен в соответствие некоторый бинарный

вектор, описывающий путь, по которому совершается обход графа в соответствии с заданной интерпретацией. Два пути в схемах программ π_1 и π_2 считаются согласованными, если соответствующие им векторы совпадают. Метод совместных вычислений при этом подразумевает исследование вычислений, проходящих по согласованным путям в программах. Для проведения такого исследования используется специального вида графовая структура Γ_{π_1,π_2}, которая называется графом совместных вычислений или графом согласованных трасс исходных программ. Вершинами графа совместных вычислений являются элементы декартова произведения вершин графов, реализующих программы. Дуги при этом строятся следующим образом: из вершины (v', v'') дуга ведет в вершину (u', u'') в том и только в том случае, если в графе программы π_1 дуга с бинарной пометкой δ ведет из вершину v' в вершину u', а в графе программы π_2 дуга с точно такой же пометкой ведет из вершины v'' в вершину u''. При этом подстановка, приписанная построенной в графе совместных вычислений дуге, будет являться объединением подстановок на соответствующих дугах в исходных программах.

Заметим, что в работе [143] был предложен алгоритм проверки логико-термальной эквивалентности, опирающийся на методику, схожую с методом совместных вычислений. После предварительных эквивалентных преобразований исходных фрагментов программ этот алгоритм проверяет подобие полученных преобразованных фрагментов. Эта проверка подобия реализуется с помощью алгоритма, похожего на алгоритм распознавания эквивалентности двухленточных автоматов, предложенный Бердом [16]. Фактически эта часть алгоритма распознавания строит структуру — согласованную пару фрагментов, — весьма похожую на граф совместных вычислений, но для уже приведенных фрагментов программ. Задавая определенные условия корректности этой структуры, применяя преобразования и строя ее разметку, алгоритм делает выводы об эквивалентности программ. Было доказано [119], что для решения задачи проверки логико-термальной эквивалентности может быть успешно применен аппарат

вычисления неподвижных точек монотонных операторов на решетках. В работе [143] в этом качестве выступает решетка функциональных сетей.

Алгоритм проверки логико-термальной эквивалентности, основанный на методе совместных вычислений, предложенный в данной работе, опирается на алгоритм, предложенный в работе [143]. Существенное отличие заключается в двух моментах. Во-первых, алгоритм, предложенный Сабельфельдом, опирается на систему эквивалентных преобразований, каждое из которых используется на определенном шаге алгоритма. Во-вторых, при построении стационарной разметки графа согласованной пары фрагментов, этот алгоритм опирается на свойства полурешетки функциональных сетей. В данной работе предложен алгоритм проверки логико-термальной эквивалентности программ, который не использует системы эквивалентных преобразований, а в качестве подходящей решетки была выбрана решетка конечных подстановок, подробно исследованная в статьях [25, 65]. В частности, в этих работах было показано, что решетка подстановок обладает свойством обрыва убывающих цепей, а в пункте 1.1 показано, что операция композиции дистрибутивна относительно операции взятия точной нижней грани на множестве подстановок. Именно эти два свойства позволяют использовать указанную решетку для решения задач статического анализа программ.

Применение методики совместных вычислений к поставленной задаче оказалось эффективным, так как с помощью этой методики и процедуры антиунификации подстановок удалось построить полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ. Этот алгоритм также лег в основу алгоритма логико-термальной унификации программ, описанного в третьей главе данной работы. Этот алгоритм, помимо метода совместных вычислений и антиунификации подстановок, также использует операцию унификации подстановок, исследованную в работах [4, 5, 59, 69].

В главе 4 данной работы проводится также исследование нового вида семантической унификации подстановок, названного двусторонней унификацией.

Как было сказано выше, для задачи двусторонней унификации подстановок показано, что она является NP-полной. Для обоснования этого факта используется метод сведения задачи, для которой доказана ее NP-полнота, к задаче, чью трудность необходимо доказать. В данной работе осуществляется сведение NP-полной задачи замощения прямоугольной области домино из определенного конечного набора [54] к задаче двусторонней унификации подстановок.

В общем случае задача домино (или задача тайлинга) состоит в том, чтобы построить корректное покрытие заданной области плоскости квадратами квадратами домино. При этом под квадратами домино подразумеваются квадраты со сторонами единичной длины, каждая из которых окрашена в некоторый фиксированный цвет. Возможных окрасок сторон представлено лишь конечное число, а вращение домино не допускается. Покрытие считается правильным, если смежные стороны соседних квадратов окрашены одинаково. Впервые эта задача была поставлена и изучена в работе [91]. Особенностью этой задачи является сильная зависимость сложности ее решения от вида покрываемой области. Так, например, задача о покрытии плоскости домино из заданного множества алгоритмически неразрешима [15, 75], а задача правильного покрытия прямоугольной области (Bounded Tiling Problem, [54]) оказалась NP-трудной. Этот вид задачи домино широко применяется сегодня для доказательства NP-полноты многих задач, исполняя зачастую роль одной из центральных задач в теории сложности вычислений [55].

5.3. Новизна и значимость

В данном пункте проводится сравнение результатов, полученных в данной диссертационной работе, с результатами в области статического анализа программ, полученными ранее.

Как было сказано выше, первый из выносимых на защиту результатов является продолжением идей, предложенных в статье [143], с использованием операций на решетках подстановок. Стоит отметить, что алгоритм, предложенный

в [143], был первым полиномиальным алгоритмом решения задачи проверки логико-термальной эквивалентности, его сложность составляла $O(n^7)$. Однако замена базовой для алгоритма решетки сетей на решетку подстановок, предложенная в данной работе, позволила получить алгоритм, сложность составляет $O(n^6)$, что улучшает имеющиеся верхние оценки сложности этой задачи. Полученный результат еще раз подтверждает возможность построения эффективных (полиномиальных по времени выполнения) алгоритмов выделения метода, применимых к подмножеству клонов типа 4.

Второе из выносимых на защиту положений расширяет применение предложенного алгоритма. В данной работе впервые исследуется задача семантической унификации программ, которая позволяет расширить понятие программного клона: под клонами в терминах унификации программ могут пониматься не просто эквивалентные фрагменты программ, но и унифицируемые фрагменты, которые могут быть рассмотрены как клоны нового пятого типа. Указанный алгоритм кроме того позволяет решать более глобальную задачу: проверку возможности создания контекстных условий, таких как состояние памяти, для проверки унифицируемости программных модулей.

Наиболее близка к реальности была бы такая постановка задачи изъятия клона, в которой вводилось бы требование построения специальных преобразований состояния памяти, позволяющих осуществить выделение метода с сохранением функционала исходной программы. При этом необходимость изменений может возникнуть не только при входе в тело метода, но и после выхода из него. Такая ситуация соответствует задаче проверки двусторонней унифицируемости программ, которая также была введена впервые и является объектом исследования четвертой главы данной работы. Третий из основных результатов, полученных в данной диссертационной работе, представляет собой алгоритм решения этой задачи. Кроме того, результаты проведенного исследования позволяют оценить сложность данной задачи и относят ее к классу NP-полных задач, что в свою очередь означает, что для ее решения может быть необходимо

использовать практические средства решения вычислительно трудных задач.

Обе задачи семантической унификации программ, представленные в данной работе, используют в качестве унификаторов простые линейные программы. Однако полученные результаты создают предпосылки для исследования более общей задачи семантической унификации, когда в качестве унификаторов могут быть использованы произвольные ациклические программы. Такой подход позволит решить практически значимую задачу выделения общей части двух программ, что в свою очередь позволило бы получить существенные результаты в области построения автоматических средств рефакторинга.

5.4. Структура работы

Данная диссертационная работа состоит из введения, четырех глав основной части, заключения и списка литературы.

В главе 1 основной части формально вводятся все базовые понятия работы, такие как понятие подстановки, операции унификации и антиунификации над подстановками, стандартные схемы программ, логико-термальная эквивалентность программ. В главе 2 приведена постановка задачи проверки логико-термальной эквивалентности программ, алгоритм решения данной задачи, а также обоснование его корректности и оценка его сложности. В главе 3 вводится понятие логико-термальной унификации программ, приводится алгоритм, разрешающий эту задачу, а также обоснование его корректности и оценка сложности. Глава 4 посвящена задачам двусторонней унификации программ и подстановок. Сначала в главе 4 приводится формальная постановка задачи решения уравнений над подстановками, обоснование NP-полноты этой задачи посредством сведения к ней задачи ограниченного домино, а после этого приводится недетерменированный алгоритм двусторонней унификации программ и обоснование его корректности.

Глава 1

Основные понятия

В данной главе вводятся такие базовые для работы понятия, как терм, подстановка, а также правила композиции, унификации и антиунификации подстановок. Также рассматриваются реализации подстановок в виде деревьев и ациклических ориентированных графов и приводится обзор алгоритмов унификации и антиунификации подстановок для разных их представлений.

Далее приводится понятие стандартной схемы программы, понятие вычисления программы в заданной интерпретации. Затем описывается понятие логической истории пути и термальной истории переменной, на основе этого вводятся понятия детерминанта и логико-термальной эквивалентности программ.

1.1. Алгебра подстановок

Рассмотрим конечный алфавит, состоящий из множества функциональных символов $F = \{f_1^{(n_1)}, \ldots, f_m^{(n_m)}\}$, множества предикатных символов $P = \{P_1^{(k_1)}, \ldots, P_\ell^{(k_\ell)}\}$ и множества Var переменных. При этом в записи $f_i^{n_i}$ через n_i будем обозначать местность (валентность) i-го функционального символа f_i, а в записи $P_j^{(k_j)}$ через k_j будем обозначать местность предикатного символа P_j. Нульместный функциональный символ будем называть константой. В дальнейшем будем опускать показатель местности функциональных и предикатных символов, если его значение ясно из контекста.

Под множеством термов $Term(Var, F)$ будем понимать наименьшее множество, удовлетворяющее двум условиям:

- $Var \subseteq Term(Var, F)$;
- если $f^{(n)} \in F$ и $t_1, \ldots, t_n \in Term(Var, F)$, то $f^{(n)}(t_1, \ldots, t_n) \in Term(Var, F)$,

а элементы этого множества $t_i \in Term(Var, F)$ будем называть термами.

Назовем k-местной атомарной формулой (или просто атомом) всякое выражение вида $P^{(k)}(t_1, \ldots, t_k)$, где $P^{(k)}$ — предикатный символ, а $t_1, \ldots, t_k$ — термы. Множество атомарных формул обозначим записью $Atom(Var, F)$. Записи Var_t и Var_A будут обозначать множества переменных, входящих в состав терма t и атома A соответственно.

Пусть X и Y — два конечных множества переменных, $X \subseteq Var, Y \subseteq Var$. $X - Y$-*подстановкой* назовем всякое отображение $\theta : X \to Term(Y, F)$, сопоставляющее каждой переменной из X некоторый терм во множестве $Term(Y, F)$, при этом само множество $Term(Y, F)$ будем называть областью значений подстановки. Множество $Dom_\theta = \{x \in X : \theta(x) \neq x\}$ будем называть областью подстановки. Множество подстановок с конечной областью X и областью значений $Term(Y, F)$ обозначим $Subst(X, Y, F)$. Если $\theta \in Subst(X, Y, F)$, $Dom_\theta = \{x_1, x_2, \ldots, x_n\}$ и $\theta(x_i) = t_i$, $1 \leq i \leq n$, то подстановка θ однозначно определяется множеством пар $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$. Эти пары также будем называть *связками* подстановки θ. Запись Var_θ будет использоваться для обозначения множества переменных $\bigcup_{i=1}^{n} Var_{t_i}$, входящих в состав всех термов подстановки. Результатом применения подстановки θ к терму $t \in Term(X, F)$ называется терм $t\theta$, который получается одновременной заменой в t каждой переменной x_i термом $\theta(x_i)$. Аналогично определяется результат применения подстановки к атому: $A\theta$ есть атом, который получается одновременной заменой каждого вхождения переменной x_i в атом A термом $\theta(x_i)$. Пусть E — атом или терм, тогда выражение $E\theta$, полученное в результате применения к выражению E подстановки θ, будем называть *примером* выражения E, специализированным подстановкой θ.

Композицию $\theta\eta$ подстановок $\theta \in Subst(X, Y, F)$ и $\eta \in Subst(Y, Z, F)$ определим следующим образом: $(\theta\eta)(x) = (\theta(x))\eta$ для каждой переменной $x \in X$, при этом для подстановки $\mu = \theta\eta$ верно $\mu \in Subst(X, Z, F)$. Поскольку для любого терма $t, t \in Term(Var, F)$ справедливо равенство $t(\theta\eta) = (t\theta)\eta$, опера-

ция композиции подстановок является ассоциативной. Подстановка ρ, которая является биекцией Y на Y, называется *переименованием*.

На множестве $Subst(X, Y, F)$ введем отношение предпорядка $\preceq$ и отношение эквивалентности $\sim$: для пары подстановок θ_1, θ_2 условимся считать, что $\theta_1 \preceq \theta_2$ выполняется, если существует такая подстановка $\eta \in Subst(Y, Y, F)$, что $\theta_2 = \theta_1\eta$. В этом случае подстановку θ_1 будем называть *прототипом* или *шаблоном* подстановки θ_2, подстановку θ_2 — *примером* подстановки θ_1. Подстановку η будем называть *пополнением* прототипа θ_1 до примера θ_2. В случае, если для некоторого переименования ρ верно, что $\theta_2 = \theta_1\rho$, будем говорить, что подстановки θ_1 и θ_2 эквивалентны: $\theta_1 \sim \theta_2$.

Как было показано в работе [25], отношение предпорядка $\preceq$ индуцирует на множестве классов эквивалентности $Subst^{\sim}(X, Y, F)$ отношение частичного порядка $\leq$. Частично упорядоченное множество $(Subst^{\sim}(X, Y, F), \leq)$ образует решетку, наименьшим элементом которой является класс эквивалентности, порожденный так называемой *пустой* подстановкой $\varepsilon = \{x_1/y_1, \ldots, x_n/y_n\}$. Данное частично упорядоченное множество было также подробно изучено в работах [52, 65]. Введем на множестве подстановок специальную максимальную подстановку τ, удовлетворяющую равенству $\tau\theta = \theta\tau = \tau$ для любой подстановки θ.

Операция взятия точной верхней грани в этой решетке называется *унификацией* подстановок и обозначается символом $\uparrow$, операция взятия точной нижней грани называется *антиунификацией* подстановок и обозначается символом $\downarrow$. В работе [25] было установлено, что решетка $(Subst^{\sim}(X, Y, F), \leq)$ обладает свойством обрыва убывающих цепей: для всякой подстановки θ длина любой цепи между классами $\theta^{\sim}$ и $\varepsilon^{\sim}$ конечна.

В дальнейшем запись $\theta_1 \downarrow \theta_2$ будем использовать для обозначения произвольной подстановки из класса эквивалентности $\theta_1^{\sim} \downarrow \theta_2^{\sim}$. Также для любой пары подстановок $\theta', \theta' \in Subst(X', Y, F)$ и $\theta'', \theta'' \in Subst(X'', Y, F)$ в том случае, если $X' \cap X'' = \varnothing$, мы будем использовать запись $\theta' \cup \theta''$ для обозначения

подстановки η, которая представляет собой теоретико-множественное объединение связок подстановок θ' и θ''.

Задача антиунификации (построения наименее общих шаблонов) подстановок рассматривалась в ряде работ [63, 65, 70, 72, 83, 92, 114]; в этих работах были предложены различные алгоритмы антиунификации подстановок и термов, а также исследованы свойства этой операции.

Для представления термов в дальнейшем будут использоваться ациклические ориентированные графы (АОГ).

Рассмотрим произвольный конечный ациклический ориентированный граф $G = (V, E)$, где через V обозначено множество вершин, а через E — множество дуг графа. Размером графа будем называть суммарное количество его вершин и дуг, то есть $|G| = |V| + |E|$. Разметкой ациклического ориентированного графа назовем отображение, которое ставит в соответствие каждой вершине графа v символ s_v, где $s_v \in \{F, Var\}$, а каждой дуге графа ставится в соответствие некоторое натуральное число.

Разметка АОГ называется правильной, если она удовлетворяет следующим двум требованиям:

- если вершина v является листовой, то есть не имеет исходящих дуг, то она помечена либо константой, либо переменной;
- если из вершины v исходят n дуг, где $n > 0$, то эта вершина помечена функциональным символом местности n, а пометки всех исходящих дуг соответствуют попарно различным числам из множества $\{1, 2, \ldots, n\}$.

Если в правильно размеченном АОГ G из вершины v_1 в вершину v_2 ведет дуга, помеченная числом m, то вершину v_2 будем называть m-наследником вершины v_1.

Каждой вершине v правильно размеченного АОГ G с приписанным ей символом s_v можно однозначным образом поставить в соответствие терм t_v, который строится согласно следующим правилам:

- если из вершины v не исходит ни одной дуги, то в ней реализуется терм, соответствующий приписанной ей константе или переменной;

- если же из вершины v исходят n дуг, ведущих в вершины $v_1, \ldots, v_n$ соответственно, и при этом для любого i, $1 \leq i \leq n$, дуга, ведущая в вершину v_i, помечена числом i, то в вершине v реализуется терм $s_v(t_{v_1}, t_{v_2}, \ldots, t_{v_n})$.

Будем говорить, что терм t_v *реализуется* в вершине v. При этом правильно размеченный АОГ G реализует конечное множество термов T, $T \subseteq Term(Var, F)$, если для каждого t, $t \in T$, в графе G существует вершина v, для которой $t_v = t$. На рисунке (1.1) представлен граф, соответствующий терму $f(g(x_1, x_2), h(x_2))$.

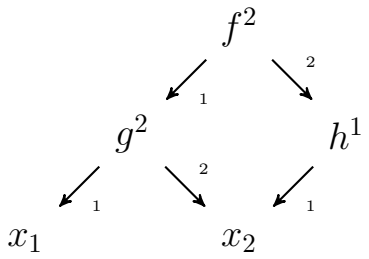


Рис. 1.1. Пример графа, реализующего терм.

Мы будем рассматривать реализующие множество T приведенные АОГ, то есть такие, которые удовлетворяют следующим двум условиям:

- в каждой вершине АОГ G реализуется подтерм какого-либо терма, принадлежащего множеству T;

- в разных вершинах G реализуются разные подтермы.

Описанные таким образом размеченные ациклические ориентированные графы удобно использовать для реализации подстановок. Рассмотрим заданную подстановку $\theta = \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$, где $\theta \in Subst(X, Y, F)$ и АОГ G, реализующий множество термов $T = \{t_1, t_2, \ldots, t_n\}$. Каждой переменной x_i, где $1 \leq i \leq n$, поставим в соответствие ту вершину АОГ G, в

которой реализуется соответствующий терм t_i, при этом переменную x_i назовем *заголовком* этой вершины. Припишем указанную переменную x_i соответствующей вершине t_i. Размеченный таким образом граф G будем называть графовой реализацией подстановки θ. К примеру, подстановку $\theta = \{x/f(g(x_1, x_2), h(x_2)), y/g(x_2, x_3), z/h(g(x_2, x_3))\}$ реализует граф, приведенный на рисунке (1.2).

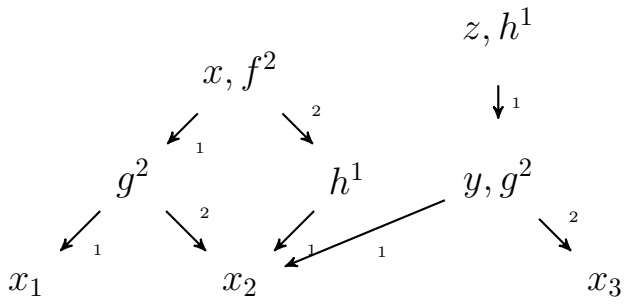


Рис. 1.2. Пример графа, реализующего подстановку θ.

Как уже было сказано выше, основные алгебраические операции над подстановками, используемые в этой работе — это операции унификации и антиунификации подстановок. Задача унификации подстановок возникает во многих разделах математической логики, теории искусственного интеллекта, алгебры, теории вычислений. Более подробно с областями применения процедуры унификации термов можно ознакомиться в работе [42]. Было разработано большое разнообразие эффективных алгоритмов унификации и структур данных, подходящих для их практической реализации ([13, 35, 56, 59, 69, 85]). Здесь мы рассмотрим один из этих алгоритмов, приведенный в работе [59]. Для термов, заданных ациклическими ориентированными графами, данный алгоритм имеет линейную сложность как по времени работы, так и по объему затрачиваемой памяти.

Будем считать, что заданы две подстановки $\theta' = \{x_1/t'_1, x_2/t'_2, \ldots, x_n/t'_n\}$ и $\theta'' = \{x_1/t''_1, x_2/t''_2, \ldots, x_n/t''_n\}$ такие, что $\theta' \in Subst(X, Y', F)$, а $\theta'' \in Subst(X, Y'', F)$. При этом для множеств Y', Y'' должно быть выполнено $Y' \cap$

$Y'' = \emptyset$. Алгоритм нахождения унификации $\theta' \uparrow \theta''$ будет основан на правилах переписывания связанной с исходными подстановками системы термальных уравнений. Перед началом работы алгоритма эта система выглядит так:

$$\begin{aligned} t'_1 &= t''_1 \\ t'_2 &= t''_2 \\ &\dots \\ t'_n &= t''_n. \end{aligned}$$

К этой системе в произвольном порядке, пока это возможно, будем применять одно из следующих 6 правил переписывания уравнений:

1. уравнение вида $f(s'_1, s'_2, \dots, s'_k) = f(s''_1, s''_2, \dots, s''_k)$ заменяется системой уравнений:
$$\begin{aligned} s'_1 &= s''_1 \\ s'_2 &= s''_2 \\ &\dots \\ s'_k &= s''_k; \end{aligned}$$

2. уравнение вида $f(s'_1, s'_2, \dots, s'_k) = g(s''_1, s''_2, \dots, s''_m)$ в случае несовпадения функциональных символов f и g приводит к завершению работы алгоритма с результатом $\theta' \uparrow \theta'' = \tau$;

3. тождество $t = t$ исключается из системы;

4. уравнение $t = y$ в случае, где t — терм, отличный от переменной, а y — переменная, замещается уравнением $y = t$;

5. уравнение $y = t$ в случае, где y — переменная, а t — терм, в состав которого входит переменная y, но отличный от переменной y, приводит к завершению работы алгоритма с результатом $\theta' \uparrow \theta'' = \tau$;

6. уравнение $y = t$ в случае, где y — переменная, а t — терм, в состав кото-

рого не входит переменная y, и при этом верно, что переменная y содержится еще хотя бы в одном уравнении системы, приводит к применению подстановки $\{y/t\}$ к правым и левым частям всех остальных уравнений системы.

Система, сформированная к моменту, когда не применимо ни одно из правил 1-6, будет состоять из уравнений

$$\begin{aligned} y_1 &= t_1 \\ y_2 &= t_2 \\ &\dots \\ y_k &= t_k. \end{aligned}$$

При этом все левые части уравнений y_i попарно различны и не содержатся в термах $t_1, \dots, t_k$. Тогда точной верхней гранью $\theta' \uparrow \theta''$ будет являться подстановка $\theta'\{y_1/t_1, \dots, y_k/t_k\} = \theta''\{y_1/t_1, \dots, y_k/t_k\}$.

Операция унификации также понадобится нам для вычисления наиболее общего примера двух атомарных выражений A', A''. Если $A' = p(t'_1, t'_2, \dots, t'_n)$ и $A'' = p(t''_1, t''_2, \dots, t''_n)$ — две атомарные формулы и при этом верно $Var_{A'} \cap Var_{A''} = \emptyset$, то наиболее общим примером выражений A', A'' будет атом $A = p(x_1, \dots, x_n)\theta$, где $\theta = \{x_1/t'_1, \dots, x_n/t'_n\} \uparrow \{x_1/t''_1, \dots, x_n/t''_n\}$.

В отличие от операции унификации подстановок, операция антиунификации имеет несколько более узкую область применения. Впервые антиунификация была введена в статье [70] и активно использовалась при решении задач суперкомпиляции [61, 83, 144]. Позже было показано, что антиунификация подстановок может быть использована для вычисления инвариантов программ и выделения синтаксических клонов [17]. В работе [114] был описан оптимальный по времени алгоритм антиунификации подстановок, который использует их представление в виде ациклических ориентированных графов. В той же работе показано, что сложность задачи антиунификации подстановок линейно

зависит от размера наименее общего шаблона исходных подстановок. С другой стороны, в той же работе показано, что существуют подстановки, точная нижняя грань которых имеет размер $O(n^2)$, где n — максимальный из размеров исходных подстановок. Здесь мы приводим более простой алгоритм антиунификации подстановок, схожий с приведенным выше алгоритмом унификации: работа алгоритма будет опираться на переписывание системы аннотированных уравнений.

Пусть заданы две подстановки $\theta' = \{x_1/t'_1, x_2/t'_2, \ldots, x_n/t'_n\}$ и $\theta'' = \{x_1/t''_1, x_2/t''_2, \ldots, x_n/t''_n\}$ такие, что $\theta', \theta'' \in Subst(X, Y, F)$. Пусть $Z = Y/(Var_{\theta'} \cup Var_{\theta''})$ — множество вспомогательных переменных, не входящих в состав термов из области значений подстановок θ' и θ''. Антиунификация $\theta' \downarrow \theta''$ строится с помощью итеративного алгоритма. Каждый шаг алгоритма порождает одну из подстановок неубывающей последовательности $\eta_0, \eta_1, \eta_2, \ldots$, преобразуя при этом систему аннотированных уравнений, соответствующую паре подстановок θ', θ''. Каждое уравнение данной системы помечается одной из вспомогательных переменных множества Z. В начале работы алгоритма для пары указанных выше подстановок система имеет вид:

$$\begin{aligned} z_1 :&\quad t'_1 = t''_1 \\ z_2 :&\quad t'_2 = t''_2 \\ &\quad \ldots \\ z_n :&\quad t'_n = t''_n, \end{aligned}$$

а начальная подстановка $\eta_0 = \{x_1/z_1, x_2/z_2, \ldots, x_n/z_n\}$. Далее на каждом i-том шаге работы алгоритма, пока это возможно, будем применять к системе в произвольном порядке одно из следующих правил переписывания уравнений, позволяющих также вычислять последовательность подстановок $\eta_0, \eta_1, \ldots$.

1. Аннотированное уравнение вида $z : f(s'_1, s'_2, \ldots, s'_k) = f(s''_1, s''_2, \ldots, s''_k)$ за-

мещается системой уравнений

$$\begin{aligned} z_{N+1}: \quad s'_1 &= s''_1 \\ z_{N+2}: \quad s'_2 &= s''_2 \\ &\dots \\ z_{N+k}: \quad s'_k &= s''_k, \end{aligned}$$

где $z_{N+1}, z_{N+2}, \dots, z_{N+k}$ — переменные из множества Z, ранее не использовавшиеся для пометки других уравнений. Кроме того, происходит построение новой подстановки последовательности: $\eta_i = \eta_{i-1}\{z/f(z_{N+1}, z_{N+2}, \dots, z_{N+k})\}$.

2. Если в системе есть пара одинаковых аннотированных уравнений $z' : s' = s''$ и $z'' : s' = s''$, то одно из уравнений удаляется из системы, что также сказывается на подстановке. Например, удаление уравнения $z'' : s' = s''$ приведет к изменению подстановки следующим образом: $\eta_i = \eta_{i-1}\{z''/z'\}$.

Алгоритм останавливается, когда к полученной на очередном шаге с номером j системе аннотированных уравнений

$$\begin{aligned} z_{M+1}: \quad s'_1 &= s''_1 \\ z_{M+2}: \quad s'_2 &= s''_2 \\ &\dots \\ z_{M+r}: \quad s'_r &= s''_r, \end{aligned}$$

не применимо ни одно из правил 1-2. Результатом работы алгоритма при этом будем считать подстановку $\eta_j = \theta' \downarrow \theta''$, полученную на последнем шаге. В конце работы алгоритма формируются так называемые аннотирующие подстановки для вспомогательных переменных z_{M+i}, $i = 1, \dots, r$: пары подстановок $\{z_{M+j}/s'_j\}$, $\{z_{M+j}/s''_j\}$, соответствующие уравнениям системы, сформированной в конце работы алгоритма. Заметим, что для аннотирующих подстановок справедливы следующие два равенства: $\theta' = (\theta' \downarrow \theta'')\{z_{M+1}/s'_1, \dots, z_{M+r}/s'_r\}$,

$\theta'' = (\theta' \downarrow \theta'')\{z_{M+1}/s''_1, \ldots, z_{M+r}/s''_r\}$. Таким образом, каждая подстановка из пары аннотирующих является пополнением прототипа $\theta' \downarrow \theta''$ до примера θ' или θ'' соответственно.

Операция антиунификации также понадобится нам для вычисления наименее общего шаблона двух атомарных выражений A_1, A_2. Иными словами, нам необходимо иметь алгоритм, позволяющий вычислить такое наиболее конкретное атомарное выражение A, примерами которого будут являться как A_1, так и A_2. Если $A' = p(t'_1, t'_2, \ldots, t'_n)$ и $A'' = p(t''_1, t''_2, \ldots, t''_n)$ — две атомарные формулы, то наименее общим шаблоном этих формул будет являться атом $A = p(x_1, x_2, \ldots, x_n)\theta_0$, где $\theta_0 = \{x_1/t'_1, x_2/t'_2, \ldots, x_n/t'_n\} \downarrow \{x_1/t''_1, x_2/t''_2, \ldots, x_n/t''_n\}$.

Сложность задач вычисления точной верхней и точной нижней граней на решетке подстановок сильно зависит от графового представления подстановок. Так, в работе [114] было показано, что если подстановки θ_1 и θ_2 имеют реализации в виде приведенных АОГ G_{θ_1} и G_{θ_2} соответственно, то размер приведенного ациклического ориентированного графа $H_{\theta_1\theta_2}$ композиции этих подстановок может быть оценен величиной

$$\Omega(|H_{\theta_1}| + |H_{\theta_2}|). \tag{1.1}$$

В свою очередь размер приведенной реализации АОГ $H_{\theta_1\downarrow\theta_2}$ антиунификации этих подстановок оценивается величиной

$$\Omega(|H_{\theta_1}||H_{\theta_2}|). \tag{1.2}$$

Рассмотрим здесь одно из свойств операции антиунификации, которое активно используется в приведенных далее алгоритмах. Это свойство дистрибутивности операции композиции относительно операции антиунификации.

Теорема 1. *Для подстановок* η, $\eta \in Subst(X, Y, F)$, θ_1, θ_2, $\theta_1, \theta_2 \in$

$Subst(Y, Y, F)$, выполнено соотношение:

$$\eta\theta_1 \downarrow \eta\theta_2 \sim \eta(\theta_1 \downarrow \theta_2).$$

Доказательство. Справедливость соотношения $\eta\theta_1 \downarrow \eta\theta_2 \geq \eta(\theta_1 \downarrow \theta_2)$ следует из определения операции антиунификации, а также из очевидных соотношений $\eta(\theta_1 \downarrow \theta_2) \leq \eta\theta_1$ и $\eta(\theta_1 \downarrow \theta_2) \leq \eta\theta_2$.

Докажем обратное соотношение $\eta\theta_1 \downarrow \eta\theta_2 \leq \eta(\theta_1 \downarrow \theta_2)$.

Для подстановки λ и некоторого произвольного множества переменных Y определим проекцию $\lambda|_Y$ подстановки на множество как подстановку λ', удовлетворяющую следующему условию:

$$\lambda'(z) = \begin{cases} \lambda(z), & z \in Y \\ z, & z \notin Y \end{cases}$$

Для множества Y, состоящего из одной переменной y будем для простоты использовать запись $\lambda|_y$. Пусть $\eta = \{x_1/t_1(y_1, \ldots, y_m), \ldots, x_n/t_n(y_1, \ldots, y_n)\}$. Положим $Y = \cup_{i=1}^{n} Var_{t_i}$.

Поскольку для подстановки η и любых подстановок θ_1, θ_2 выполняются отношения $\eta \leq \eta\theta_1$ и $\eta \leq \eta\theta_2$, в силу свойств операции антиунификации верно и отношение $\eta \leq \eta\theta_1 \downarrow \eta\theta_2$. Последнее неравенство означает, что существует такая подстановка ξ, $\xi \in Subst(Y, Y, F)$, для которой верно равенство $\eta\theta_1 \downarrow \eta\theta_2 = \eta\xi$. Следовательно, существуют такие подстановки ρ_1, ρ_2, $\rho_1, \rho_2 \in Subst(Y, Y, F)$, что $\eta\theta_1 = \eta\xi\rho_1$ и $\eta\theta_2 = \eta\xi\rho_2$. Рассмотрим произвольную переменную y из множества Y. Предположим, что $y \in Var_{t_i}$, где t_i – терм, соответствующий паре $\{x_i/t_i(y_1, \ldots, y_m)\}$ из подстановки η. Тогда для указанных выше подстановок и терма t_i верны две цепочки равенств

$$\begin{aligned} x_i\eta\theta_1 &= t_i(\ldots, y, \ldots,)\theta_1 &= t_i(\ldots, y\theta_1, \ldots), \\ x_i\eta\xi\rho_1 &= t_i(\ldots, y, \ldots,)\xi\rho_1 &= t_i(\ldots, y\xi\rho_1, \ldots). \end{aligned}$$

Из совпадения левых частей этих равенств следует, что $y\theta_1 = y\xi\rho_1$ для переменной y. Поскольку переменная y — произвольная переменная из множества Y, это последнее равенство справедливо для всех $y \in Y$. Аналогично можно показать, что $y\theta_2 = y\xi\rho_2$. Таким образом справедливы неравенства $\xi|_y \leq \theta_1|_y$ и $\xi|_y \leq \theta_2|_y$, и как их следствие неравенство $\xi|_y \leq \theta_1|_y \downarrow \theta_2|_y$.

Из определения операции антиунификации следует, что для проекций двух подстановок на одну и ту же переменную выполнено равенство $\theta_1|_y \downarrow \theta_2|_y \sim (\theta_1 \downarrow \theta_2)|_y$. Таким образом, мы приходим к заключению о том, что $\xi|_y \leq (\theta_1 \downarrow \theta_2)|_y$.

Рассмотрим проекции подстановок на все множество Y. С учетом соотношения $\eta\theta_1 \downarrow \eta\theta_2 = \eta\xi$ получаем следующую цепочку соотношений

$$\eta\theta_1 \downarrow \eta\theta_2 = \eta\xi = \eta(\xi|_Y) \leq \eta((\theta_1 \downarrow \theta_2)|_Y) = \eta(\theta_1 \downarrow \theta_2),$$

которая и служит обоснованием неравенства $\eta\theta_1 \downarrow \eta\theta_2 \leq \eta(\theta_1 \downarrow \theta_2)$. □

1.2. Стандартные схемы программ

Введем понятие стандартной схемы программы.

Рассмотрим два заданных конечных множества переменных $X = \{x_1, \ldots, x_n\}$ и $Y = \{y_1, \ldots, y_m\}$. *Параметрами* или *входными переменными* программы назовем переменные множества X, а *внутренними* или *локальными переменными* программы — переменные множества Y.

Стандартной схемой императивной программы со множеством входных переменных X будем называть конечную систему переходов (размеченный ориентированный граф) $\pi(X)$. Каждой вершине v этого графа приписан некоторый оператор ветвления, соответствующий логическому переходу в программе по условному оператору (предикату). Из каждой вершины исходят две дуги, одна из которых помечена символом 0, а другая — символом 1. Переходы между вершинами этого графа соответствуют линейным участкам в программе —

последовательностям операторов присваивания. Каждому оператору присваивания вида $x := t(...)$ в программе соответствует подстановка $\{x/t(...)\}$; при этом линейному участку операторов присваивания вида $x_1 := t_1(\ldots);\ x_2 := t_2(\ldots);\ \ldots;\ x_n := t_n(\ldots)$ в соответствие может быть поставлена композиция подстановок $\theta = \{x_n/t_n\}\{x_{n-1}/t_{n-1}\}\ldots\{x_1/t_1\}$. Указанная подстановка θ призвана корректно отображать вычислительный эффект линейного участка операторов присваивания в программе в терминах алгебры подстановок.

Будем полагать $\Delta = \{0, 1\}$.

Введем формальное определение программы. При этом в дальнейшем будем опускать указание множества входных переменных программы, если оно понятно из контекста.

Программой над множеством входных переменных $X = \{x_1, \ldots, x_n\}$ и множеством внутренних переменных $Y = \{y_1, \ldots, y_m\}$ назовем размеченную систему переходов (размеченный ориентированный граф) $\pi = \langle X, Y, V, v_{in}, v_{out}, B, \rightarrow, \lambda_0 \rangle$, где:

- V — конечное множество вершин программы;
- $v_{in} \in V$ — единственная входная вершина;
- $v_{out} \in V$ — единственная выходная вершина;
- $B : V \to Atom(Y, F)$ — функция привязки, которая ставит в соответствие каждой вершине v программы π атомарную формулу A_v;
- $\rightarrow$: $(V \setminus \{v_{out}\}) \times \Delta \to V \times Subst(Y, Y, F)$ — функция переходов. Каждой невыходной вершине v, $v \in V \setminus \{v_{out}\}$, программы π в зависимости от истинностного значения приписанного ей предиката функция указывает подстановку, описывающую результат выполнения линейного участка программы, следующего за вершиной v в программе, а также вершину, на которую переходит управление программы после выполнения этого линейного участка;

- $\lambda_0 \in Subst(X, Y, F)$ — подстановка, инициализирующая внутренние переменные программы.

Размером $|\pi|$ графа программы π будем называть мощность множества V вершин этой программы.

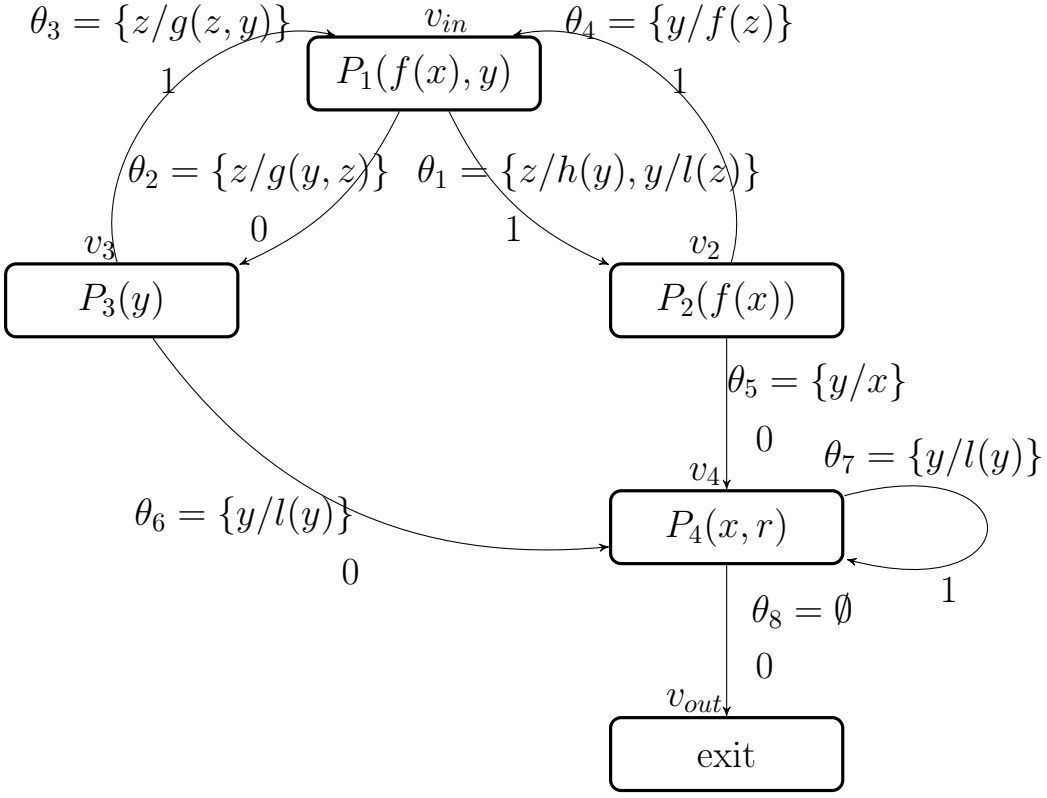
Рассмотрим следующий фрагмент программного кода: Program 1.

```
L1: if P1(f(x), y){
      z = h(y);
      y = l(z);
      if P2(f(x)){
        y = f(z);
        goto L1;
      }
      else y = x;
    }else{
      z = g(y, z);
      if P3(y){
        z = g(z, y);
        goto L1;
      }else y = l(y);
    }
    while P4(x, r)
      y = l(y);
    return;
```

Соответствующая этому коду программа приведена на рисунке (1.3).

Введем несколько обозначений. Договоримся, что если заданная подстановка соответствует линейному участку программы между вершинами u, v, то для краткости такую подстановку будем обозначать $\theta_{(u,v)}$. Кроме того, в дальнейшем при обращении к функции переходов вместо записи $\rightarrow (u, \delta) = (v, \theta_{u,v})$ будем использовать запись $u \xrightarrow{\delta,\theta} v$, опуская у подстановки индекс u, v, если он понятен из контекста. Мы также будем использовать запись $v_j = succ(\pi, v_i, \delta)$ для обозначения того факта, что в графе программы π из вершины v_i исходит дуга с пометкой $\delta \in \Delta$ в вершину v_j, и для удобства в этой записи опускать соответствующую подстановку θ_{v_i,v_j}. При этом вершина v_j называется Δ-наследником

Рис. 1.3. Программа для фрагмента кода 1.



вершины v_i. Так, например, в программе на рисунке (1.3) вершина v_2 является 1-наследником вершины v_1, то есть $v_2 = succ(\pi, v_1, 1)$.

Вершины программы далее будем называть *точками программы*, а дуги — *переходами*. Будем считать, что в программе из каждой ее точки достижим выход. Символом V_π обозначим множество, состоящее из точек входа и выхода программы π, а также всех точек этой программы.

Среди всех переменных множества Y выделим подмножество выходных переменных программы, $Y_{ex} = \{y_{i_1}, y_{i_2}, \ldots, y_{i_k}\}$. Будем считать, что точке выхода программы, вершине v_{out}, приписана атомарная формула вида $OUT(y_{i_1}, y_{i_2}, \ldots, y_{i_k})$. В выходной вершине программы таким образом формируется некая конечная оценка значений переменных программы. Для того чтобы формализовать эту оценку, введем понятие вычисления программы в некоторой интерпретации при заданной оценке входных переменных.

С учетом того, что константами в исходном алфавите считаются нульместные функциональные символы, рассмотрим сигнатуру (F, P). Будем рассмат-

ривать произвольную интерпретацию $I = \langle D, \overline{F}, \overline{P} \rangle$ сигнатуры (F, P), где D — область интерпретации, а F и P — оценки функциональных и предикатных символов соответственно. Оценкой множества переменных Var в интерпретации I назовем всякое отображение $\overline{d} : Var \to D$, ставящее в соответствие каждой переменной элемент из области интерпретации. При этом если $Var = \{z_1, \ldots, z_k\}$, то для обозначения оценки будем использовать запись вида $\{z_1 \leftarrow \overline{d}(z_1), z_2 \leftarrow \overline{d}(z_2), \ldots, z_k \leftarrow \overline{d}(z_k)\}$. Запись $Val(Var, I)$ будем использовать для обозначения множества всех оценок переменных Var в интерпретации I.

Для каждой интерпретации $I = \langle D, \overline{F}, \overline{P} \rangle$ и каждой оценки $\overline{d} \in Val(Var, I)$ обычным образом определим элемент области интерпретации $t[\overline{d}]$, являющийся значением терма t, $t \in Term(Var, F)$, и истинностное значение $A[\overline{d}]$ атомарной формулы A, $A \in Atom(Var, F)$. Всякая подстановка θ, $\theta \in Subst(X, Y, F)$, в интерпретации I преобразует множество оценок переменных $Val(X, I)$ во множество оценок переменных $Val(Y, I)$ следующим образом: если $\theta = \{y_1/t_1, y_2/t_2, \ldots, y_n/t_n\}$, то $\theta(\overline{d}) = \{y_1 \leftarrow t_1[\overline{d}], y_2 \leftarrow t_2[\overline{d}], \ldots, y_n \leftarrow t_n[\overline{d}]\}$.

С учетом сказанного выше, для каждой программы $\pi = \langle X, Y, V, v_{in}, v_{out}, B, \to, \lambda_0 \rangle$ и интерпретации I оценки переменных $Val(X, I)$ выполняют роль входных данных, в то время как оценки переменных $Val(Y, I)$ представляют собой состояние данных в вычислении программы. При этом под вычислением программы π в интерпретации I для оценки входных данных $\overline{d}$ мы будем понимать максимальную последовательность вида

$$comp(\pi, I, \overline{d}) = (v_0, \overline{d_0}), (v_1, \overline{d_1}), \ldots, (v_s, \overline{d_s}), \ldots,$$

которая удовлетворяет следующим условиям: 1) v_0 является входной вершиной, $v_0 = v_{in}$, а оценка $\overline{d_0}$ есть $\lambda_0(\overline{d})$; 2) для любого i, $i \geq 0$, в программе существует переход $v_i \xrightarrow{\delta,\theta} v_{i+1}$, где $\delta = B(v_i)(\overline{d_i})$ и при этом $\overline{d_{i+1}} = \theta(\overline{d_i})$.

Будем считать, что результат бесконечного вычисления не определен. Ес-

ли вычисление $comp(\pi, I, \overline{d})$ не является бесконечным, то оно заканчивается некоторой парой $(v_N, \overline{d}_N)$, при этом $v_N = v_{out}$ — точка выхода, которой приписана атомарная формула $OUT(y_{i_1}, y_{i_2}, \ldots, y_{i_k})$. Тогда результатом вычисления $comp(\pi, I, \overline{d})$ будет являться набор $\overline{d}_N(y_{i_1}), \overline{d}_N(y_{i_2}), \ldots, \overline{d}_N(y_{i_k})$. Для каждой интерпретации I может быть задана частичная функция $\Phi_{\pi,I} : Val(X, I) \to D^k$, которая отображает оценки значений входных переменных в наборы значений переменных на выходе программы. Указанная частичная функция является значением заданной программы π в интерпретации I.

Программы π' и π'' считаются функционально эквивалентными, если для любой интерпретации I их частичные функции совпадают, то есть $\Phi_{\pi',I} = \Phi_{\pi'',I}$.

Но введенная таким образом функциональная эквивалентность оказалась алгоритмически неразрешимой, что было показано в работе [58]. Дальнейшие исследования [38] в этой области привели к выводу о том, что любое отношение эквивалентности, основанное на вычислениях в интерпретации, является алгоритмически неразрешимым даже для очень простых сигнатур. Следовательно, необходимо рассматривать более строгие отношения эквивалентности. Так, в работе [118] была предложена новая неинтерпретационная эквивалентность программ — логико-термальная эквивалентность. Данный вид эквивалентности совмещает в себе как элементы чисто семантических отношений эквивалентности, так и элементы синтаксических отношений. В работах [106], [143] были приведены алгоритмы, подтверждающие разрешимость логико-термальной эквивалентности. В работе [106] приведен алгоритм, который сводит проблему проверки логико-термальной эквивалентности к задаче распознавания эквивалентности в классе двухленточных автоматов и при этом имеет экспоненциальную верхнюю оценку сложности. С другой стороны алгоритм, предложенный в работе [143] имеет полиномиальную сложность. Этот алгоритм основан на использовании эквивалентных преобразований программ и в ходе своей работы приводит фрагменты, эквивалентность которых проверяется, к некой нормальной форме.

Далее в данной работе будет приведена постановка задачи проверки логико-термальной эквивалентности.

1.3. Логико-термальная эквивалентность программ

Путь w в программе π — это последовательность переходов вида:

$$w = v_0 \xrightarrow{\delta_0,\theta_0} v_1 \xrightarrow{\delta_1,\theta_1} v_2 \dots v_n \xrightarrow{\delta_n,\theta_n} v_{n+1},$$

где $\delta_i \in \{0, 1\}$ — пометка, соответствующая переходу из вершины v_i в вершину v_{i+1}. Будем говорить, что путь w — путь из вершины v_0 в вершину v_{n+1}.

Трассой в программе назовем конечный путь, начинающийся в точке входа: $v_0 = v_{in}$. Если трасса tr такова, что:

$$tr = v_{in} \xrightarrow{\delta_0,\theta_0} v_1 \xrightarrow{\delta_1,\theta_1} v_2 \dots v_n \xrightarrow{\delta_n,\theta_n} v_{out},$$

будем называть ее полной трассой.

Примером трассы в программе на рисунке (1.3) является следующая последовательность:

$$tr = v_{in} \xrightarrow{1,\theta_1} v_2 \xrightarrow{1,\theta_4} v_1 \xrightarrow{0,\theta_2} v_3 \xrightarrow{0,\theta_6} v_4 \xrightarrow{0,\theta_8} v_{out}.$$

Термальное значение переменной x для конечного пути w — это терм $t(w, x)$, который строится следующим образом:

- случай $w = v \xrightarrow{\delta,\theta} u$. Тогда $t(w, x) = \theta(x)$;
- случай $w = w' \xrightarrow{\delta,\theta} u$, где w' – путь, ведущий в точку v, такую, что $v \xrightarrow{\delta,\theta} u$. Тогда $t(w, x) = \theta \cdot \{x_1/t(w', x_1), \dots, x_n/t(w', x_n)\}$.

Для описанной выше трассы tr в программе 1 термальным значением переменной z является терм $g(f(h(y)), h(y))$.

Подстановку $\{x_1/t(w', x_1), \ldots, x_n/t(w', x_n)\}$, соответствующую пути w', в дальнейшем будем обозначать символом $\theta_{w'}$.

Логической историей пути w назовем последовательность пар

$$lh(w) = (A_{v_1}, \delta_1), (A_{v_2}, \delta_2), \ldots, (A_{v_k}, \delta_k),$$

где δ_j — пометка перехода $v_j \xrightarrow{\delta_j, \theta_j} v_{j+1}$, а A_{v_j} — атом, приписанный распознавателю v_j. Для логической истории пути, заканчивающегося в выходной вершине, будем считать, что последней парой в последовательности является пара $(v_{out}, 0)$. Для трассы tr будет верно следующее:

$$lh(tr) = (P_1(f(x), y), 1), (P_2(f(x)), 1), (P_1(f(x), y), 0), (P_3(y), 0), (P_4(x, r), 0), (out, 0)$$

Пусть задана некоторая полная трасса в программе π:

$$w = v_{in} \xrightarrow{\delta_0, \theta_0} v_1 \xrightarrow{\delta_1, \theta_1} v_2 \xrightarrow{\delta_2, \theta_2} \ldots v_{n-1} \xrightarrow{\delta_{n-1}, \theta_{n-1}} v_{out}.$$

Логико-термальной историей полной трассы w назовем последовательность пар:

$$lth(w) = (A_{v_{in}}\mu_0, \delta_0), (A_{v_1}\mu_1, \delta_1), \ldots, (A_{v_{n-1}}\mu_{n-1}, \delta_{n-1}), (A_{v_{out}}\mu_n, 1),$$

где $\mu_0 = \varepsilon$, а $\mu_i = \theta_0\theta_1\theta_2 \ldots \theta_{i-1}$ для каждого $i, 1 \leq i \leq n$. Логико-термальная история для трассы задается аналогично с учетом того, что трасса заканчивается некоторой вершиной, возможно, отличной от выходной.

Подстановкой θ_w, соответствующей трассе w, назовем подстановку $\theta_w = \theta_0\theta_1\theta_2 \ldots \theta_{n-1}$.

Так, для приведенной выше трассы tr будет верно:

$$lth(tr) = (P_1(f(x), y), 1), (P_2(f(x)), 1), (P_1(f(x), f(h(y))), 0),$$
$$(P_3(f(h(y))), 0), (P_4(x, r), 0), (exit, 0).$$

Подстановка, соответствующая трассе tr: $\theta_{tr} = \{y/l(f(h(y))), z/g(f(h(y)), h(y))\}$.

Множество всех логико-термальных историй программы π будем называть ее *детерминантом* и обозначать $det(\pi)$.

Будем говорить, что программы π_1 и π_2 *логико-термально эквивалентны* (л-т эквивалентны) и обозначать это отношение записью $\pi_1 \sim \pi_2$ в том и только в том случае, если детерминанты этих программ совпадают: $det(\pi_1) = det(\pi_2)$.

Глава 2

Задача проверки логико-термальной эквивалентности программ

В данной главе приведен и проанализирован алгоритм проверки логико-термальной эквивалентности стандартных последовательных схем программ, основанный на поиске наиболее схожих с точки зрения термальной истории путей в программе. Этот алгоритм использует в качестве опорной структуры граф совместных вычислений или, иначе, граф согласованных трасс программ. Разметка этого графа соответствует преобразованиям термальных историй переменных на всех возможных трассах программы, а сам алгоритм, в свою очередь, сводится к вычислению точных нижних граней на множестве подстановок. Итеративная процедура, осуществляющая это вычисление, и есть процедура построения разметки графа. Результаты, изложенные в этой главе, были опубликованы в работах [146, 148].

В этой главе приведены основные понятия, связанные с графом совместных вычислений программ, а так же сам алгоритм его разметки, на основании которого строится алгоритм проверки логико-термальной эквивалентности программ. Также в этой главе приведены обоснование корректности алгоритма проверки логико-термальной эквивалентности и оценка его сложности.

2.1. Граф совместных вычислений

Пусть заданы две программы $\pi' = \langle X, Y', V', v'_{in}, v'_{out}, B', \to', \lambda'_0 \rangle$ и $\pi'' = \langle X, Y'', V'', v''_{in}, v''_{out}, B'', \to'', \lambda''_0 \rangle$. Считаем, что указанные две программы имеют одинаковое множество входных переменных X и непересекающиеся множества внутренних переменных, $Y' \cap Y'' = \emptyset$.

Графом совместных вычислений $\Gamma_{\pi',\pi''} = \langle V, w_0, \mapsto, \lambda_0 \rangle$ этих двух про-

грамм будем называть граф со следующими компонентами:

- $V = V' \times V''$ — множество вершин графа. Сюда входят всевозможные пары $w = (v', v'')$ точек исходных программ π' и π''. Каждой вершине графа совместных вычислений приписана пара атомарных формул $(B'(v'), B''(v''))$;
- $w_0 = (v'_{in}, v''_{in})$ — выделенная корневая вершина, ей соответствует пара входных вершин исходных программ;
- $\mapsto \subseteq V \times \{0,1\} \times Subst(Y' \cup Y'', Y' \cup Y'', F) \times V$ — отношение переходов, определяющее дуги (переходы) графа. Строгое определение этого отношения приведено ниже;
- $\lambda_0 = \lambda'_0 \cup \lambda''_0$ — инициализирующая подстановка графа совместных вычислений программ.

Для краткости будем вместо записи $(v', v'', \delta, \theta, u', u'') \in \mapsto$ записывать отношение переходов более естественно: $(v', v'') \overset{\delta,\theta}{\mapsto} (u', u'')$.

Отношение $\mapsto$ определяется следующим образом: для каждой пары точек (v', v''), (u', u'') графа $\Gamma_{\pi',\pi''}$ и подстановки $\theta, \theta \in Subst(Y' \cup Y'', Y' \cup Y'', F)$, должно быть выполнено

$$(v', v'') \overset{\delta,\theta}{\mapsto} (u', u'') \Leftrightarrow \exists \delta \in \{0,1\} : v' \xrightarrow{\delta,\theta'} u' \quad \& \quad v'' \xrightarrow{\delta,\theta''} u'' \quad \& \quad \theta = \theta' \cup \theta''. \tag{2.1}$$

При этом, если верно $(v', v'') \overset{\delta,\theta}{\mapsto} (u', u'')$, будем говорить, что вершина (u', u'') — наследник вершины (v', v'') в графе $\Gamma_{\pi',\pi''}$ и будем обозначать это отношение следующим образом: $(u', u'') = succ((v', v''), \delta)$.

По аналогии с трассой в программе, введем понятие маршрута в графе совместных вычислений. Всякую последовательность дуг

$$path = (v'_{in}, v''_{in}) \overset{\delta_0,\theta_0}{\mapsto} (v'_1, v''_1) \overset{\delta_1,\theta_1}{\mapsto} \ldots \overset{\delta_{n-1},\theta_{n-1}}{\mapsto} (v'_n, v''_n) \overset{\delta_n,\theta_n}{\mapsto} (v'_{n+1}, v''_{n+1}),$$

будем называть *маршрутом в графе совместных вычислений* $\Gamma_{\pi',\pi''}$. Для каждого маршрута $path$ в графе $\Gamma_{\pi',\pi''}$ введем понятие *подстановки маршрута*, θ_{path}, представляющей собой композицию инициализирующей подстановки λ_0 и всех подстановок, приписанных дугам этого маршрута: $\theta_{path} = \lambda_0\theta_1\theta_2\ldots\theta_{n-1}\theta_n$.

Две трассы tr' и tr'' в программах π' и π'' соответственно будем называть *логически согласованными* или просто *согласованными*, если верно, что

$$tr' = v'_{in} \xrightarrow{\delta_0,\theta'_0} v'_1 \xrightarrow{\delta_1,\theta'_1} \ldots \xrightarrow{\delta_n,\theta'_n} v'_{n+1}$$

и

$$tr'' = v''_{in} \xrightarrow{\delta_0,\theta''_0} v''_1 \xrightarrow{\delta_1,\theta''_1} \ldots \xrightarrow{\delta_n,\theta''_n} v''_{n+1}.$$

Вектор $\widetilde{\delta} = (\delta_0, \delta_1, \ldots, \delta_n)$ будем называть *общим вектором согласованных трасс* tr', tr''. Маршрут в графе совместных вычислений, проходящий по вершинам

$$path = (v'_{in}, v''_{in}) \overset{\delta_0,\theta'_0\cup\theta''_0}{\mapsto} (v'_1, v''_1) \overset{\delta_1,\theta'_1\cup\theta''_1}{\mapsto} \ldots \overset{\delta_n,\theta'_n\cup\theta''_n}{\mapsto} (v'_{n+1}, v''_{n+1})$$

будем называть *соответствующим* паре согласованных трасс tr' и tr'' по вектору $\widetilde{\delta}$.

Для всех вершин w, $w \in V$, графа совместных вычислений G_{π_1,π_2} будем использовать обозначение $Path(w)$ для множества всех маршрутов графа, ведущих из входной вершины w_0 в вершину w.

Граф совместных вычислений программ π', π'' будем также называть графом логически согласованных трасс этих программ.

Из введенных выше определений графа совместных вычислений, согласованных трасс и соответствующих им маршрутов, а также из условия (2.1) следует справедливость следующих двух лемм.

Лемма 1. *Пусть tr' и tr'' — согласованные трассы в программах π' и π'' соответственно. Тогда в графе совместных вычислений $\Gamma_{\pi',\pi''}$ существует един-*

ственный маршрут, соответствующий паре трасс tr', tr''.

Лемма 2. *Пусть* $path$ *— маршрут в графе совместных вычислений* $\Gamma_{\pi',\pi''}$ *программ* π' *и* π''. *Тогда в исходных программах существует единственная пара трасс* tr', tr'', *которой соответствует этот маршрут.*

Граф совместных вычислений назовем *корректным*, если в нем

- из входной вершины недостижимы вершины вида (v'_{out}, v''_i) и вершины вида (v'_j, v''_{out}), где $v''_i \neq v''_{out}$ и $v'_j \neq v'_{out}$ соответственно;
- из каждой вершины графа достижима вершина (v'_{out}, v''_{out}).

Теорема 2. *Две программы* $\pi' = \langle X, Y', V', v'_{in}, v'_{out}, B', \to', \lambda'_0 \rangle$ *и* $\pi'' = \langle X, Y'', V'', v''_{in}, v''_{out}, B'', \to'', \lambda''_0 \rangle$ *логико-термально эквивалентны тогда и только тогда, когда их граф совместных вычислений корректен и для любого маршрута в нем*

$$path = (v'_{in}, v''_{in}) \overset{\delta_0, \theta'_0 \cup \theta''_0}{\mapsto} (v'_1, v''_1) \overset{\delta_1, \theta'_1 \cup \theta''_1}{\mapsto} \dots \overset{\delta_n, \theta'_n \cup \theta''_n}{\mapsto} (v'_{n+1}, v''_{n+1})$$

выполнено равенство $B'(v'_{n+1})\theta_{tr'} = B''(v''_{n+1})\theta_{tr''}$, *где* $\theta_{tr'}$ *и* $\theta_{tr''}$ *— подстановки трасс* tr' *и* tr'' *соответственно таких, что путь* $path$ *с ними согласован.*

Доказательство. Справедливость прямого утверждения можно показать, используя определение логико-термальной эквивалентности программ. Из того, что программы π_1 и π_2 логико-термально эквивалентны, следует, что для каждой трассы

$$tr' = v'_{in} \xrightarrow{\delta_0, \theta'_0} v'_1 \xrightarrow{\delta_1, \theta'_1} \dots \xrightarrow{\delta_n, \theta'_n} v'_{n+1}$$

в программе π_1 найдется единственная согласованная с ней трасса

$$tr'' = v''_{in} \xrightarrow{\delta_0, \theta''_0} v''_1 \xrightarrow{\delta_1, \theta''_1} \dots \xrightarrow{\delta_n, \theta''_n} v''_{n+1}$$

в программе π_2 и по лемме 1 этой паре будет соответствовать единственный путь $path$ в графе Γ_{π_1,π_2}. Корректность графа совместных вычислений в данном случае следует из согласованности трасс tr' и tr''.

Пусть для вершин v'_{n+1} и v''_{n+1} выполнено $A' = B'(v'_{n+1})$ и $A'' = B''(v''_{n+1})$ соответственно. Последними парами в соответствующих этим трассам логико-термальных историях будут являться пары $(A'\theta_{tr'}, \delta_n)$ и $(A''\theta_{tr''}, \delta_n)$. Поскольку программы π_1 и π_2 логико-термально эквивалентны, то совпадают логико-термальные истории согласованных трасс этих программ, а это означает, что совпадают также и их частичные логико-термальные истории, то есть $A'\theta_{tr'} = A''\theta_{tr''}$.

Справедливость обратного утверждения следует из леммы 2, то есть из единственности пары согласованных трасс в программах π_1 и π_2, соответствующих заданному пути в графе совместных вычислений. Тогда если для всех путей графа совместных вычислений в вершину (v'_{n+1}, v''_{n+1}), которой приписаны атомы $A' = B'(v'_{n+1})$ и $A'' = B''(v''_{n+1})$, верно $A'\theta_{tr'} = A''\theta_{tr''}$, то для пары согласованных путей в программах π_1 и π_2 это будет означать совпадение их логико-термальных историй. Нетрудно заметить, что это же выполнено и для всех путей, ведущих в выходные точки программ, то есть полные логико-термальные истории всех трасс первой программы будут совпадать с точностью до переименования с полными логико-термальными историями всех трасс второй программы, откуда следует их логико-термальная эквивалентность. □

Таким образом, проверка логико-термальной эквивалентности может быть сведена к анализу пар согласованных трасс исходных программ, или анализу графа их совместных вычислений. Но в графе совместных вычислений могут существовать вершины, в которые ведет бесконечно много маршрутов, что затрудняет проведение проверки эквивалентности на основании одной лишь теоремы 2. Поэтому нам потребуется следующая вспомогательная лемма.

Лемма 3. *Пусть* θ_1 *и* θ_2, $\theta_1, \theta_2 \in Subst(X, Y, F)$, *– некоторые подстановки, а*

атомы A' *и* A'', $A', A'' \in Atom(Y, F)$, *– условные выражения. Тогда*

$$\begin{cases} A'\theta_1 &= A''\theta_1 \\ A'\theta_2 &= A''\theta_2 \end{cases} \Leftrightarrow A'(\theta_1 \downarrow \theta_2) = A''(\theta_1 \downarrow \theta_2).$$

Доказательство. Докажем сначала прямое утверждение:

$$\begin{cases} A'\theta_1 &= A''\theta_1 \\ A'\theta_2 &= A''\theta_2 \end{cases} \Rightarrow A'(\theta_1 \downarrow \theta_2) = A''(\theta_1 \downarrow \theta_2). \tag{2.2}$$

Для доказательства этого факта воспользуемся теоремой 1 о дистрибутивности композиции относительно операции антиунификации, заметив, что все свойства композиции подстановок могут быть перенесены на операцию применения подстановки к предикату. Тогда $A'(\theta_1 \downarrow \theta_2) = A'\theta_1 \downarrow A'\theta_2 = A''\theta_1 \downarrow A''\theta_2 = A''(\theta_1 \downarrow \theta_2)$, значит, утверждение (2.2) верно.

Покажем теперь справедливость обратного утверждения:

$$\begin{cases} A'\theta_1 &= A''\theta_1 \\ A'\theta_2 &= A''\theta_2 \end{cases} \Leftarrow A'(\theta_1 \downarrow \theta_2) = A''(\theta_1 \downarrow \theta_2). \tag{2.3}$$

Пусть $\eta = \theta_1 \downarrow \theta_2$. Тогда по определению точной нижней грани для некоторых подстановок ρ_1, ρ_2 справедливы равенства $\theta_1 = \eta\rho_1$ и $\theta_2 = \eta\rho_2$. Значит, верна следующая цепочка заключений:

$$A'(\theta_1 \downarrow \theta_2) = A''(\theta_1 \downarrow \theta_2) \;\Rightarrow\; A'\eta = A''\eta \;\Rightarrow\; A'\eta\rho_1 = A''\eta\rho_1 \;\Rightarrow\; A'\theta_1 = A''\theta_1.$$

Аналогичная цепочка может быть построена и для подстановки θ_2, что завершает доказательство леммы. □

Доказанная лемма 3 позволяет переформулировать необходимое и достаточное условие логико-термальной эквивалентности программ, приведенное в

теореме 2, следующим образом:

Теорема 3. *Две программы π' и π'' логико-термально эквивалентны тогда и только тогда, когда граф их совместных вычислений корректен и для каждой его вершины $w = (v', v'')$ выполнено равенство $B'(v')\theta_w = B''(v'')\theta_w$, где $\theta_w = \downarrow_{path \in Path(w)} \theta_{path}$.*

Доказательство. Справедливость данного утверждения следует из теоремы 2, леммы 3 и устройства графа совместных вычислений программ. □

Приведенная выше теорема сводит задачу о проверке эквивалентности программ к вычислению точных нижних граней подстановок, соответствующих всевозможным маршрутам в графе совместных вычислений, ведущим в каждую из вершин w графа. Далее приведена процедура глобальной разметки графа, позволяющая решать эту задачу.

2.2. Процедура разметки графа совместных вычислений

В данном разделе описан алгоритм разметки графа логически согласованных трасс, который, в соответствии с теоремой 3, последовательно строит приближение сверху к искомой точной нижней грани подстановок, соответствующих путям в графе в каждую конкретную вершину. Алгоритм работает с графом $\Gamma_{\pi',\pi''}$. Не нарушая общности рассуждений, будем считать, что граф $\Gamma_{\pi',\pi''}$ корректен, т.к. в противном случае, в соответствие с теоремой 3, его анализ не требуется.

- **Переименование переменных.** Перед началом работы осуществляется переименование вхождений всех переменных из множества X в первую программу переменными $\{x'_1, x'_2, \ldots, x'_n\}$, отличными от переменных множества X. Также осуществляется переименование вхождений всех переменных из множества X во вторую программу переменными

$\{x''_1, x''_2, \ldots, x''_n\}$, также отличными от переменных множества X. Условимся обозначать получившиеся в результате указанных переименований переменных программы π_1 и π_2 соответственно.

- **Граф совместных вычислений.** Для программ π_1, π_2 строится их граф совместных вычислений Γ_{π_1,π_2}.

- **Начальная разметка графа.** Корневой вершине $w_0 = (v'_{in}, v''_{in})$ приписывается подстановка $\eta_{w_0} = \{x'_1/x_1, \ldots, x'_n/x_n, x''_1/x_1, \ldots, x''_n/x_n\}$, а все остальные вершины помечаются максимальной в решетке подстановок мнимой подстановкой τ: $\eta_w = \tau$ для всех $w \neq w_0$. Кроме того, корневой вершине присваивается пометка *.

- **Итерации алгоритма.** До тех пор пока в графе Γ_{π_1,π_2} есть хотя бы одна вершина, помеченная символом "*" (активная вершина), алгоритм произвольным образом выбирает одну из таких вершин (v', v''). Предположим, что выбранной вершине приписана некоторая подстановка $\eta_{(v',v'')}$. Тогда символ "*" снимается с вершины (v', v''), и для каждой дуги $(v', v'') \overset{\delta,\theta}{\mapsto} \Rightarrow(u', u'')$, ведущей из вершины (v', v'') в некоторую вершину $(u', u'') = succ((v', v''), \delta)$ графа совместных вычислений, выполняются следующие действия

  1. вычисляется подстановка $\eta'_{(u',u'')} = \theta\eta_{(v',v'')} \downarrow \eta_{(u',u'')}$, где $\eta_{(u',u'')}$ — подстановка, которой помечена вершина (u', u'');

  2. если $\eta_{(u',u'')} \neq \eta'_{(u',u'')}$, то вершине (u', u'') приписывается пометка $\eta'_{(u',u'')}$. При этом вершина (u', u'') помечается символом "*". Если же $\eta_{(u',u'')} = \eta'_{(u',u'')}$, то замена подстановки не происходит.

  Эти действия повторяются до тех пор, пока в графе Γ_{π_1,π_2} существуют активные вершины, помеченные "*".

- **Правило верификации.** Осуществляется проверка следующего условия: существует ли в графе Γ_{π_1,π_2} хотя бы одна вершина (u', u''), помеченная подстановкой $\eta_{(u',u'')}$ такая, что $A_{u'}\eta_{(u',u'')} \neq A_{u''}\eta_{(u',u'')}$. В том случае, если такая вершина существует, программы признаются не логико-термально эквивалентными. Если же таких вершин не обнаружено, то программы логико-термально эквивалентны.

Далее будут приведены утверждения, обосновывающие корректность алгоритма.

Условимся через $\eta_v^{[n]}$ обозначать подстановку, приписанную вершине $v = (v', v'')$ на n-ом шаге алгоритма. Тогда справедливо следующее утверждение:

Лемма 4. *Для любой вершины $v = (v', v'')$ и для любого шага n работы алгоритма существует такое подмножество P множества $Path(v)$, что для подстановки $\eta_v^{[n]}$, вычисленной на шаге n для вершины v, выполнено равенство:*

$$\eta_v^{[n]} = \downarrow_{path \in P} \theta_{path}$$

Доказательство. Докажем утверждение индукцией по номеру шага алгоритма.

Базис. При $n = 0$ утверждение справедливо, т.к. всем вершинам графа совместных вычислений приписана подстановка $\eta_v^{[0]} = \tau$, для которой выполнено равенство $\tau = \downarrow_{path \in \emptyset} \theta_{path}$.

Индуктивный переход. Пусть утверждение справедливо для некоторого шага n. Покажем, что оно верно и для $n + 1$. Пусть u – вершина-предшественник вершины v в графе совместных вычислений, $v = succ(u, \delta)$, и пусть дуге, идущей из u в v, приписана подстановка θ, то есть в графе совместных вычислений найдется путь $path \overset{\delta,\theta}{\mapsto} v$. Заметим, что согласно описанию алгоритма в этом случае имеет место равенство $\eta_v^{[n+1]} = \eta_v^{[n]} \downarrow \theta\eta_u^{[n]}$.

По индуктивному предположению существуют множества $P_u \subseteq Path(u)$ и

$P_v \subseteq Path(v)$, для которых $\eta_v^{[n]} = \downarrow_{path \in P_v} \theta_{path}$ и $\eta_u^{[n]} = \downarrow_{path \in P_u} \theta_{path}$. Рассмотрим множество $P'_v = P_v \cup \{path \overset{\delta,\theta}{\mapsto} v : path \in P_u\}$. Покажем, что это множество и является искомым. Для этого можно воспользоваться законом левой дистрибутивности антиунификации подстановок относительно операции композиции, установленным в теореме 1:

$$\begin{aligned} \downarrow_{path \in P'_v} \theta_{path} &= \left(\downarrow_{path \in P_v} \theta_{path}\right) \downarrow \left(\downarrow_{path \in P_u} \theta\theta_{path}\right) = \\ &= \left(\downarrow_{path \in P_v} \theta_{path}\right) \downarrow \left(\theta \downarrow_{path \in P_u} \theta_{path}\right) = \\ &= \eta_v^{[n]} \downarrow \theta\eta_u^{[n]} \end{aligned}$$

Значит, множество P'_v и является искомым. □

Введем еще одно обозначение. Для каждой вершины v будем обозначать $In(v)$ множество всех вершин u таких, что $v = succ(u, \delta)$. Условимся также подстановку, приписанную переходу $u \overset{\delta,\theta}{\mapsto} v$, обозначать θ_{uv}. Тогда из предыдущей леммы следует

Лемма 5. *Для любой вершины v*

$$\downarrow_{path \in Path(v)} \theta_{path} = \downarrow_{u \in In(v)} \theta_{uv}\left(\downarrow_{path \in Path(u)} \theta_{path}\right)$$

Воспользуемся записью $\theta(v)$ для обозначения точной нижней грани композиций подстановок, вычисленных по всем путям из множества $Path(v)$ в графе совместных вычислений, т. е. $\theta(v) = \downarrow_{path \in Path(v)} \theta_{path}$. Как показывает лемма 5, множество подстановок $\{\theta(v), v \in V\}$, где V – множество вершин графа совместных вычислений, является решением системы уравнений

$$\bigcup_{v \in V \setminus entry} \left[X_v = \downarrow_{u \in In(v)} \theta_{uv} X_u\right] \cup (X_{entry} = \lambda), \tag{2.4}$$

где λ — подстановка констант первого шага алгоритма.

Лемма 6. *Предположим, что на некотором шаге с номером N алгоритм завершил свою работу. Тогда $\{\eta_v^{[N]} : v \in V\}$ – решение системы уравнений (2.4).*

Доказательство. Поскольку по условию леммы на N-м шаге работы алгоритма не осталось активных вершин, равенство $\eta_v^{[N]} = \eta_v^{[N-1]}$ выполняется для всех вершин графа совместных вычислений.

Рассмотрим произвольную вершину v графа совместных вычислений программ. Пусть для этой вершины последним шагом, на котором происходили изменения приписанной ей подстановки был шаг N_1, то есть N_1 – наименьший номер, для которого $\eta_v^{[N_1]} = \eta_v^{[N_1+1]} = \ldots = \eta_v^{[N]}$. Как уже было показано выше, подстановка $\eta_v^{[N_1]}$ представима в виде $\downarrow_{u \in In(v)} \theta_{uv}\eta_u^{[N_1-1]}$. Но тогда, в силу выбора шага N_1, для любого номера i такого, что $i \geq N_1$, выполнено $\theta_v^{[i]} = \downarrow_{u \in In(v)} \theta_{uv}\eta_u^{[i]}$. А поскольку $N_1 \leq N$, то из этого равенства следует $\eta_v^{[N]} = \downarrow_{u \in In(v)} \theta_{uv}\eta_u^{[N]}$ для любой вершины v графа совместных вычислений программ. Таким образом, $\{\eta_v^{[N]}\}$ - решение системы (2.4). □

Лемма 7. *Допустим, что алгоритм остановился на шаге N. Тогда для любой вершины v графа совместных вычислений верно равенство*

$$\eta_v^{[N]} = \downarrow_{path \in Path(v)} \theta_{path}.$$

Доказательство. Допустим, что существует такая вершина v, для которой

$$\eta_v^{[N]} \neq \downarrow_{path \in Path(v)} \theta_{path}. \tag{2.5}$$

Тогда существует такой путь $path'_v$ из входной вершины графа совместных вычислений в вершину v, что $\eta_v^{[N]} \downarrow \theta_{path'_v} \neq \eta_v^{[N]}$. Выберем кратчайший из таких путей $path'_v$ по всем вершинам, удовлетворяющим соотношению (2.5). Рассмотрим вершину u_0 такую, что она является предшественником вершины

v в пути $path'_v$: $path'_v = path_{u_0} \xrightarrow{\delta\theta_{u_0 v}} v$. Путь $path_{u_0}$ не может быть кратчайшим из рассматриваемых путей, так как тогда этот путь был бы короче $path'_v$, что противоречит выбору $path'_v$. Значит, для него не выполнено условие (2.5), то есть $\eta_{u_0}^{[N]} = \eta_{u_0}^{[N]} \downarrow \theta_{path_{u_0}}$. Воспользовавшись леммой 6, получаем следующую цепочку равенств:

$$\begin{aligned}
\eta_v^{[N]} = \downarrow_{u \in In(v)} \theta_{uv}\eta_u^{[N]} &= \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} \theta_{uv}\eta_u^{[N]}\right) \downarrow \theta_{u_0 v}\eta_{u_0}^{[N]} = \\
&= \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} \theta_{uv}\eta_u^{[N]}\right) \downarrow \left(\theta_{u_0 v}\left(\eta_{u_0}^{[N]} \downarrow \theta_{path_{u_0}}\right)\right) = \\
&= \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} \theta_{uv}\eta_u^{[N]}\right) \downarrow \theta_{u_0 v}\eta_{u_0,v} \downarrow \theta_{u_0 v}\theta_{path_{u_0}} = \\
&= \left(\downarrow_{u \in In(v)} \theta_{uv}\eta_u^{[N]}\right) \downarrow \theta_{path'_v} = \\
&= \eta_v^{[N]} \downarrow \eta_{path'_v},
\end{aligned}$$

противоречащую сделанному ранее предположению $\theta_v^{[N]} \neq \theta_v^{[N]} \downarrow \theta_{path'_v}$. □

Воспользуемся приведенными выше леммами для обоснования корректности алгоритма проверки логико-термальной эквивалентности программ, приведенного в этой главе.

Теорема 4. *Для любых двух программ π_1 и π_2 таких, что их граф совместных вычислений корректен, алгоритм построения разметки графа совместных вычислений всегда завершает свою работу.*

Доказательство. Покажем, что вычисление новой подстановки для каждой вершины не может повторяться бесконечное число раз. Рассмотрим вершину графа совместных вычислений v. Допустим, что в ней изменение приписанной подстановки происходит бесконечно долго. Пусть на i-ом шаге работы алгоритма ей была приписана некоторая подстановка η_v, а на шаге с номером $i+1$ был осуществлен переход из вершины u с приписанной ей подстановкой η_u по дуге с подстановкой θ в вершину v. По описанию алгоритма в вершине v должна быть вычислена точная нижняя грань подстановок η_v и $\eta_u\theta$. Из предположе-

ния о том, что изменение приписанной этой вершине подстановки происходит бесконечно, следует, что $\eta'_v \neq \eta_v \downarrow \eta_u\theta$, но тогда $\eta'_v < \eta_v$.

Как уже было сказано, решетка подстановок $(Subst^{\sim}(X, Y, F), \leq)$ обладает свойством обрыва убывающих цепей, то есть для всякой подстановки η длина любой цепи между классами $\eta^{\sim}$ и $\varepsilon^{\sim}$ конечна. Но тогда изменение длин подстановок, приписанных вершине v, не может происходить бесконечно долго, а значит, спустя конечное число шагов в каждой вершине графе совместных вычислений помечающая эту вершину подстановка перестает изменяться. □

Теорема 5. *Для двух программ π_1 и π_2, чей граф совместных вычислений корректен, остановка алгоритма проверки логико-термальной эквивалентности программ с положительным результатом происходит тогда и только тогда, когда программы π_1 и π_2 логико-термально эквивалентны.*

Доказательство. Остановка алгоритма с положительным результатом означает, что в графе совместных вычислений больше не осталось активных вершин(вершин, имеющих пометку "*") и для каждой вершины, которой приписана пара атомов (A', A''), на одном из шагов алгоритма была вычислена подстановка $\eta_v^{[N]}$ такая, что $A'\eta_v^{[N]} = A''\eta_v^{[N]}$. Эта подстановка, по лемме 7, представляет собой точную нижнюю грань подстановок всех путей, ведущих в вершину v, то есть $\eta_v^{[N]} = \downarrow_{path \in Path(v)} \theta_{path}$. Это, в соответствии с теоремой 3, означает, что для любого пути $path$ в графе совместных вычислений из входной вершины в вершину v выполнено равенство $A'\theta_{path} = A''\theta_{path}$. Тогда по теореме 3 программы π_1 и π_2 логико-термально эквивалентны.

Предположим теперь, что для двух логико-термально эквивалентных программ π_1 и π_2 алгоритм остановился с отрицательным результатом. Пусть стабилизация подстановок произошла на шаге с номером n. Из описания алгоритма следует, что для некоторой вершины v с приписанной ей парой атомов (A', A'') и подстановкой $\eta_v^{[n]}$ выполнено $A'\eta_v^{[n]} \neq A''\eta_v^{[n]}$. По лемме 4 $\eta_v^{[n]} = \downarrow_{path \in P} \theta_{path}$, где P — это некоторое подмножество путей, ведущих в вершину v. Тогда,

учитывая утверждение леммы 3, существует путь $path, path \in P$, такой, что для соответствующей ему подстановки θ_{path} равенство нарушается, то есть $A'\theta_{path} \neq A''\theta_{path}$. Но отсюда в силу теоремы 3 следует, что программы π_1 и π_2 не являются логико-термально эквивалентными, что противоречит исходному предположению. □

В основе приведенного в этом разделе алгоритма лежит итеративная процедура, которая на каждом шаге вычисляет подстановку: каждый раз для некоторой тройки подстановок $\theta_1, \theta_2, \theta_3$ вычисляется подстановка $(\theta_1\theta_2) \downarrow \theta_3$. Но в соответствии с формулой 1.2 размер ациклического ориентированного графа, реализующего подстановку $(\theta_1\theta_2) \downarrow \theta_3$ можно оценить величиной, пропорциональной размерам графов исходных подстановок. В худшем случае приведенный в этом разделе алгоритм должен осуществить количество итераций, пропорциональное размерам исходных программ. С учетом упомянутой выше сложности вычисления точной нижней грани и с учетом возможного роста размера ациклического ориентированного графа подстановок получается, что алгоритм имеет экспоненциальную сложность. Для того, чтобы устранить этот недостаток алгоритма далее рассмотрена операция, представляющая замену антиунификации подстановок. Эта операция также вычисляет нижнюю грань двух подстановок, при этом она не противоречит теоремам о корректности алгоритма, но является более простой: эта операция осуществляется за линейное время, а ациклический ориентированный граф, представляющий ее результат, по размеру не превосходит минимальный из графов исходных подстановок.

2.3. Редуцированные подстановки и алгоритм редукции

Мы будем рассматривать множество подстановок $Subst(X, Y, F)$, определенных на конечном множестве переменных $X = \{x_1, x_2, \ldots, x_n\}$ и бесконечном множестве переменных $Y = \{y_1, y_2, \ldots\}$. Будем называть переменные множества X *главными переменными*, а переменные множества Y, используемые

для построения термов, — *вспомогательными переменными.* Для каждой подстановки θ, $\theta \in Subst(X, Y, F)$, разобьем множество переменных Var_θ на два подмножества: $HVar_\theta = \{y : \exists x(x \in X \quad \& \quad \theta(x) = y)\}$ и $NVar_\theta = Var_\theta \backslash V$. Заметим, что в графовой реализации подстановки θ все вершины, помеченные переменными из множества $HVar_\theta$ имеют заголовки. Введенное разбиение множества переменных позволяет выделить во множестве $Subst(X, Y, F)$ подмножество редуцированных подстановок.

Определение 1. *Подстановка θ, $\theta \in Subst(X, Y, F)$ называется редуцированной, если для нее выполнено равенство: $Var_\theta = HVar_\theta$.*

Из определения 1 следуют следующие характеристические свойства редуцированных подстановок:

- в АОГ, реализующем редуцированную подстановку, все листовые вершины озаглавлены;
- каждая подстановка θ имеет хотя бы один редуцированный прототип. Так, например, пустая подстановка $\{x_1/y_1, x_2/y_2, \ldots, x_n/y_n\}$ является редуцированной.

Пусть задана некоторая подстановка $\theta = \{x_1/t_1(\ldots), x_2/t_2(\ldots), \ \ldots, \ x_n/t_n(\ldots)\}$, $\theta \in Subst(X, Y, F)$. Процессом сокращения подстановки θ назовем следующую последовательность действий. Выберем из множества Y переменную y такую, что $y \notin Var_\theta$. Предположим, что входящий в l-ую связку подстановки θ терм t_l содержит подтерм $f(y_1, y_2, \ldots, y_n)$ такой, что хотя бы для одного j, $1 \leq j \leq k$, верно, что $y_j \notin HVar_\theta$. Для каждого t_i, $1 \leq i \leq n$, построим терм t'_i в соответствии с одним из следующих правил:

- если t_i не содержит подтерма $f(y_1, y_2, \ldots, y_n)$, то $t'_i = t_i$;

- если t_i содержит подтерм $f(y_1, y_2, \ldots, y_n)$, то t'_i получается из t_i синхронной заменой всех вхождений подтерма $f(y_1, y_2, \ldots, y_n)$ на переменную y.

Рассмотрим новую подстановку $\theta' = \{x_1/t'_1(\ldots), x_2/t'_2(\ldots), \ldots, x_n/t'_n(\ldots)\}$. Из построения t'_i видно, что $\theta = \theta'\{y/f(y_1, \ldots, y_k)\}$. Кроме того, $Var_{\theta'} = Var_\theta \cup \{y\}$.

Покажем, что для построенной таким образом пары подстановок θ и θ' и произвольной редуцированной подстановки η справедливо следующее утверждение.

Лемма 8. *Редуцированная подстановка η является прототипом подстановки θ в том и только том случае, когда η является прототипом подстановки θ'.*

Доказательство. Рассмотрим произвольный редуцированный прототип η подстановки θ. Ввиду того, что η — прототип θ, существует некоторая подстановка ρ, $\rho \in Subst(Y, Y, F)$ такая, что $\eta\rho = \theta = \theta'\{y/f(y_1, \ldots, y_k)\}$. Без ограничения общности будем считать, что подстановка ρ не содержит связок-переименований. Но тогда одна из подстановок η или ρ должна содержать в одной из своих связок подтерм $f(y_1, y_2, \ldots, y_n)$. Но ввиду того, что подстановка η является редуцированной, а переменная y_j не входит во множество $HVar_\theta$, терм $f(y_1, y_2, \ldots, y_n)$ не может входить в состав ни одной из связок подстановки η. Значит, он является подтермом некоторого терма одной из связок подстановки ρ. Но тогда подстановку ρ можно представить в виде $\rho = \rho'\{y/f(y_1, y_2, \ldots, y_n)\}$, $\rho' \in Subst(Y, Y, F)$, где ρ' получена в результате тех же действий, что и θ' из подстановки θ. Получается, выражение $f(y_1, y_2, \ldots, y_n)$ не входит в состав ни одного подтерма подстановок ρ', θ'. Но тогда $\eta\rho' = \theta'$, то есть η является прототипом θ'.

Обратное утверждение следует из того факта, что $\theta = \theta'\{y/f(y_1, \ldots, y_k)\}$, то есть каждый прототип θ' является прототипом θ. □

Определение 2. *Наиболее специальной редукцией постановки θ назовем такую редуцированную подстановку θ', что θ — пример подстановки θ' и любая редуцированная подстановка, являющаяся прототипом θ, будет прототипом подстановки θ'.*

Введем также для наиболее специальной редукции обозначение: $\theta' = msr(\theta)$.

Лемма 9. *Для каждой подстановки θ, $\theta \in Subst(X, Y, F)$ существует ее наиболее специальная редукция $msr(\theta)$. Наиболее специальная редукция единственна с точностью до переименования переменных.*

Доказательство. Доказательство этого утверждения следует непосредственно из леммы 8. □

Опишем алгоритм построения наиболее специальной редукции подстановки θ. Этот алгоритм работает с ациклическим ориентированным графом G_θ подстановки θ, проводя его раскраску.

1. Вершины графа, имеющие заголовки, окрашиваются в синий цвет. Все вершины, которым приписаны вспомогательные переменные из множества $NVar_\theta$, окрашиваются в красный цвет.

2. Строим раскраску графа:

   - каждая вершина красного цвета окрашивает в красный цвет все входящие в нее дуги;
   - если из вершины исходит хотя бы одна красная дуга, то все исходящие из нее дуги окрашиваются в красный цвет;
   - если вершина не синего цвета, а все исходящие из нее дуги — красные, то эта вершина окрашивается в красный цвет;

- действия повторяются до тех пор, пока возможно применение хотя бы одного из указанных выше правил раскраски;

3. Для тех вершин, у которых есть заголовки и из которых исходят только красные дуги, вводится в качестве пометки новая вспомогательная переменная.

4. Все красные дуги и вершины удаляются из графа.

Замечание. Данный алгоритм может работать с графами, не являющимися корректной реализацией подстановки: такие графы могут содержать вершины, недостижимые из озаглавленных вершин. Для этого перед началом работы первого шага алгоритма все такие вершины должны быть окрашены в красный цвет.

Полученный в результате описанной выше процедуры размеченный ориентированный граф является графовой реализацией подстановки $msr(\theta)$. Пример работы алгоритма для подстановки $\theta = \{x_1/f^2(h^1(f^2(y_1, y_2)), g^1(y_2)), x_2/g^1(y_2), x_3/h^1(f(y_1, y_2)), x_4/y_1\}$ приведен на рисунке 2.1. Результатом является подстановка $msr(\theta) = \{x_1/f(y_3, y_4), x_2/y_4, x_3/y_3, x_4/y_1\}$. Время работы описанного алгоритма пропорционально размеру графа G_θ исходной подстановки.

Опишем далее несколько важных свойств наиболее специальных редукций подстановок. Обоснование этих свойств опирается на предложенный алгоритм построения наиболее специальной редукции подстановки.

Лемма 10. *Для любых двух подстановок θ_1 и θ_2 таких, что $\theta_1 \leq \theta_2$ верно, что их редукции находятся в соотношении $msr(\theta_1) \leq msr(\theta_2)$.*

Доказательство. Заметим, что из факта $\theta_1 \leq \theta_2$ следует, что существует такая подстановка ρ, $\rho \in Subst(Y, Y, F)$, что выполнено равенство: $\theta_2 = \theta_1\rho$.

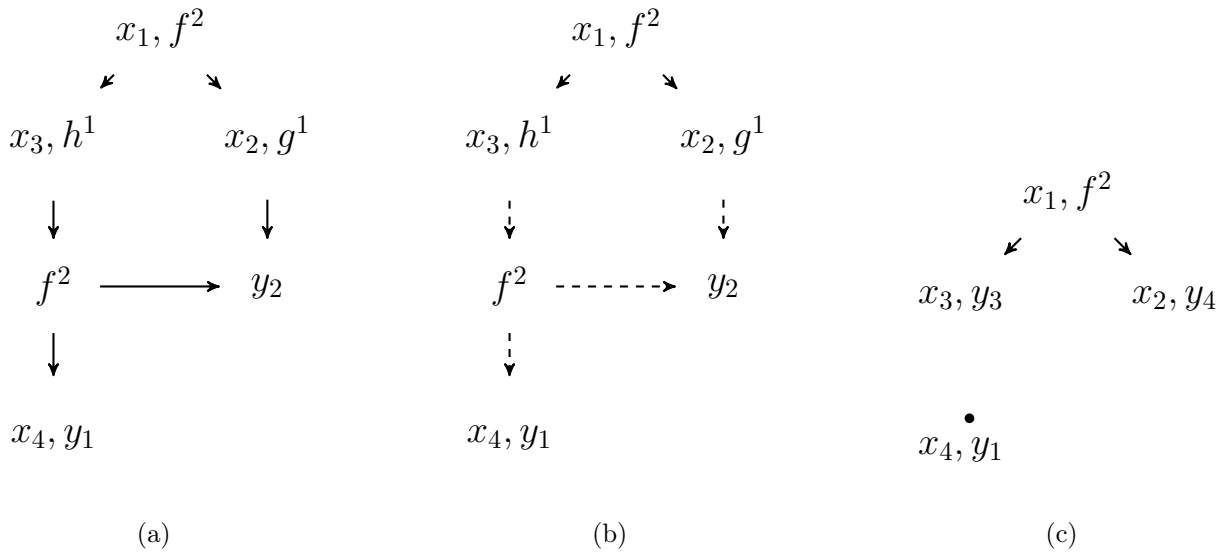


Рис. 2.1. Пример построения наиболее специальной редукции: (a) АОГ исходной подстановки θ; (b) результат покраски дуг; (c) АОГ наиболее специальной редукции подстановки θ

Тогда для обоснования справедливости леммы достаточно показать неравенство

$$msr(\theta_1) \leq msr(\theta_1\rho).$$

Рассмотрим процесс построения редукции $msr(\theta_1\rho)$. Множество "синих" вершин в АОГ θ_1 по построению будет то же, что и в АОГ $\theta_1\rho$. Это в, свою очередь, означает, что полученные в результате редукции ациклические ориентированные графы будут отличаться только наборами дуг. При этом в редуцированном графе, соответствующем θ_1, таких дуг не может быть больше, чем в АОГ $msr(\theta_1\rho)$. А это, в свою очередь, означает, что $msr(\theta_1\rho)$ будет примером $msr(\theta_1)$. □

Следующая лемма устанавливает свойства редукции относительно операции композиции подстановок.

Лемма 11. *Пусть $\theta \in Subst(X, X, F)$, $\eta \in Subst(X, Y, F)$. Тогда справедливо следующее соотношение:*

$$msr(\theta\eta) \sim msr(\theta\ msr(\eta)).$$

Доказательство. Заметим, что существует такая подстановка ρ, $\rho \in$

$Subst(Y, Y, F)$, что $\eta = msr(\eta)\rho$, то есть выражение $msr(\theta\eta)$ представимо в виде $msr(\theta\ msr(\eta)\rho)$. Будем рассматривать только тот случай, когда ρ не является переименованием, т.к. иначе соотношение очевидно. Рассмотрим для подстановки $\theta\ msr(\eta)\rho$ процедуру сокращения: сократим в ней все дуги и вершины, соответствующие подстановке ρ. Тогда по лемме 8 получаем, что каждый редуцированный прототип подстановки $\theta\eta = \theta\ msr(\eta)\rho$ есть редуцированный прототип подстановки $\theta\ msr(\eta)$, а каждый редуцированный прототип $\theta\ msr(\eta)$ в свою очередь является редуцированным прототипом $\theta\eta$, из чего и следует утверждение леммы. □

Для дальнейшего использования редукции подстановок нам понадобится также следующее свойство редукций относительно атомарных выражений.

Теорема 6. *Для любой пары атомарных формул A, B, $A, B \in Atom(X, F)$ и для любой подстановки θ, $\theta \in Subst(X, Y, F)$ соотношение $A\theta = B\theta$ справедливо тогда и только тогда, когда $A\ msr(\theta) = B\ msr(\theta)$.*

Доказательство. ($\Leftarrow$) По определению подстановка $msr(\theta)$ является прототипом подстановки θ, что подтверждает $A\ msr(\theta) = B\ msr(\theta) \Rightarrow A\theta = B\theta$.

($\Rightarrow$) Покажем, что если $A\theta = B\theta$, то и $A\ msr(\theta) = B\ msr(\theta)$. По условию теоремы, атомы A и B унифицируемы, их унификатором является подстановка θ. А это означает, что для них может быть построен наиболее общий унификатор η, $\eta \in Subst(X, X, F)$ такой, что справедливо, равенство $Dom_\eta \cap Var_\eta = \emptyset$. Пусть $Var_\theta = \{x_{j_1}, x_{j_2}, \ldots, x_{j_m}\}$. Рассмотрим переименование этих переменных $\lambda = \{x_{j_1}/y_1, x_{j_2}/y_2, \ldots, x_{j_m}/y_m\}$. Тогда подстановка $\theta' = \eta\lambda$ также будет являться наиболее общим унификатором той же пары атомов, то есть существует такая подстановка ρ, $\rho \in Subst(Y, Y, F)$, что $\theta = \theta'\rho$ и, вместе с тем, $A\theta' = B\theta'$. При этом θ' будет также являться редуцированной подстановкой из множества $Subst(X, Y, F)$, а значит, будет являться редуцированным прототипом подстановки θ. Это в свою очередь означает, что для некоторой подстановки ρ', $\rho' \in Subst(Y, Y, F)$ будет выполнено равенство $msr(\theta) = \theta'\rho'$ из чего следует

$A\ msr(\theta) = B\ msr(\theta)$. □

2.4. Редуцированная антиунификация подстановок

В этом разделе вводится операция редуцированной антиунификации подстановок, которая позволяет применять аппарат редуцированных подстановок к решению задачи проверки логико-термальной эквивалентности программ в дальнейшем.

Определение 3. *Редуцированной антиунификацией подстановок* θ_1, θ_2, $\theta_1, \theta_2 \in Subst(X, Y, F)$, *назовем такую подстановку* θ, *что* $\theta = msr(\theta_1 \downarrow \theta_2)$.

Введем для обозначения редуцированной антиунификации подстановок следующую запись: $\theta = \theta_1 \Downarrow \theta_2$. Выделим важное свойство операции редуцированной антиунификации.

Теорема 7. *Для любой пары подстановок* η_1, η_2 *таких, что* $\eta_1, \eta_2 \in Subst(Y, Y, F)$ *справедливо равенство:*

$$\eta_1 \Downarrow \eta_2 \sim msr(\eta_1) \Downarrow\ msr(\eta_2).$$

Доказательство. Так как $\eta_1 \geq msr(\eta_1)$ и $\eta_2 \geq msr(\eta_2)$, справедливо следующее неравенство: $msr(\eta_1) \downarrow msr(\eta_2) \leq \eta_1 \downarrow \eta_2$. Тогда с учетом леммы 10 справедливо соотношение $msr(\eta_1) \Downarrow msr(\eta_2) \leq \eta_1 \Downarrow \eta_2$.

Теперь покажем, что $msr(\eta_1) \Downarrow msr(\eta_2) \geq \eta_1 \Downarrow \eta_2$, из чего и будет следовать утверждение леммы. Нам необходимо показать, что каждый редуцированный прототип подстановки $\eta_1 \downarrow \eta_2$ является редуцированным прототипом подстановки $msr(\eta_1) \downarrow msr(\eta_2)$. Для этого рассмотрим произвольную редуцированную подстановку θ из множества $Subst(X, Y, F)$, которая является прототипом подстановки $\eta_1 \downarrow \eta_2$. Но тогда выбранная подстановка θ также является

прототипом самих подстановок η_1 и η_2. Поскольку она является редуцированной, то, кроме этого, она также является прототипом подстановок $msr(\eta_1)$ и $msr(\eta_2)$, то есть $\theta \leq msr(\eta_1)$ и $\theta \leq msr(\eta_2)$. А из этого следует справедливость следующего соотношения: $\theta \leq msr(\eta_1) \downarrow msr(\eta_2)$. Но раз подстановка θ — произвольный редуцированный прототип $\eta_1 \downarrow \eta_2$ — является редуцированным прототипом подстановки $msr(\eta_1) \downarrow msr(\eta_2)$, то и $\eta_1 \Downarrow \eta_2 \leq msr(\eta_1) \Downarrow msr(\eta_2)$.

Поскольку справедливы неравенства $\eta_1 \Downarrow \eta_2 \leq msr(\eta_1) \Downarrow msr(\eta_2)$ и $msr(\eta_1) \Downarrow msr(\eta_2) \leq \eta_1 \Downarrow \eta_2$, подстановки $msr(\eta_1) \Downarrow msr(\eta_2)$ и $\eta_1 \Downarrow \eta_2$ эквивалентны. □

Следующие две теоремы представляют собой аналоги теоремы 1 и леммы 3 для редуцированной антиунификации подстановок.

Теорема 8. *Для любой подстановки θ, $\theta \in Subst(X, X, F)$, и пары подстановок η_1, η_2, $\eta_1, \eta_2 \in Subst(X, Y, F)$ справедливо $\theta\eta_1 \Downarrow \theta\eta_2 = msr(\theta(\eta_1 \Downarrow \eta_2))$.*

Доказательство. Из леммы 11 и определения редуцированной антиунификации следует справедливость следующей цепочки равенств: $\theta\eta_1 \Downarrow \theta\eta_2 = msr(\theta\eta_1 \downarrow \theta\eta_2) = msr(\theta(\eta_1 \downarrow \eta_2)) = msr(\theta\ msr(\eta_1 \downarrow \eta_2)) = msr(\theta(\eta_1 \Downarrow \eta_2))$. □

Теорема 9. *Для любой пары атомов A', A'' таких, что $A', A'', \in Atom(X, F)$ и любой пары подстановок η_1, η_2, $\eta_1, \eta_2 \in Subst(X, Y, F)$ справедливо следующее соотношение:*

$$\begin{cases} A'\eta_1 = A''\eta_1 \\ A'\eta_2 = A''\eta_2 \end{cases} \Leftrightarrow A'(\eta_1 \Downarrow \eta_2) = A''(\eta_1 \Downarrow \eta_2).$$

Доказательство. По условию леммы 3 справедливо:

$$\begin{cases} A'\eta_1 = A''\eta_1 \\ A'\eta_2 = A''\eta_2 \end{cases} \Leftrightarrow A'(\eta_1 \downarrow \eta_2) = A''(\eta_1 \downarrow \eta_2).$$

Но тогда с учетом теоремы 6 равенство $A'(\eta_1 \downarrow \eta_2) = A''(\eta_1 \downarrow \eta_2)$ выполняется тогда и только тога, когда справедливо $A'(\eta_1 \Downarrow \eta_2) = A''(\eta_1 \Downarrow \eta_2)$, что доказывает утверждение леммы. □

Определение 3 редуцированной антиунификации термов формально задает алгоритм построения редуцированной нижней грани подстановок как последовательного применения сперва операции антиунификации, а затем алгоритма построения наиболее специальной редукции подстановки. Но тогда вычисление редуцированной нижней грани и размер результирующей подстановки будут иметь квадратичную зависимость от размеров ациклических ориентированных графов исходных подстановок. Но в п. 2.2 уже было сказано, что такая зависимость приведет к тому, что алгоритм проверки логико-термальной эквивалентности окажется экспоненциальным. Поэтому далее в этом разделе приводится алгоритм построения редуцированной нижней грани подстановок, или алгоритм вычисления редуцированной антиунификации,имеющий линейную сложность.

Пусть заданы две подстановки θ_1, θ_2 такие, что $\theta_1, \theta_2 \in Subst(X, Y, F)$. Пусть ациклические ориентированные графы G_{θ_1} и G_{θ_2} реализуют соответственно подстановки θ_1 и θ_2. Для подстановки $\theta = \theta_1 \Downarrow \theta_2$ будет построен ациклический ориентированный граф G_θ, вершинами которого будут упорядоченные пары $w = (u, v)$, где u есть вершина графа G_{θ_1}, а v — соответственно вершина графа G_{θ_2}. Построение этого графа будет осуществляться в три этапа.

1. Построение множества помеченных вершин. На этом этапе осуществляется выбор таких пар $w_i = (u_i, v_i)$, что обе вершины u_i и v_i имеют одинаковые заголовки x_i в соответствующих графах. Для каждой из таких пар в графе G_θ строится соответствующая паре w_i вершина, которая помечается тем же заголовком x_i, что и исходные вершины.

2. Пополнение множества вершин. На этом этапе осуществляется добавление вершин, входящих в граф, но не имеющих заголовков. Рассмотрим пару

вершин (u_0, v_0) таких, что они помечены одним и тем же функциональным символом f^n, $f^n \in F$. Предположим, что для любого натурального i, $1 \leq i \leq n$ верно, что пара (u_i, v_i) удовлетворяет следующим двум требованиям: 1) вершина u_i является i-наследником вершины u_0 в графе G_{θ_1}, а вершина v_i является i-наследником вершины v_0 в графе G_{θ_2}; 2) пара $w_i = (u_i, v_i)$ уже внесена во множество вершин графа G_θ. Тогда, если вершина $w_0 = (u_0, v_0)$ еще не была внесена во множество вершин графа G_θ, то она в него вносится. Вершина w_0 помечается соответствующим функциональным символом f^m, а для каждого i, $1 \leq i \leq n$ проводится дуга от (u_0, v_0) в вершину (u_i, v_i). Этот шаг повторяется до тех пор, пока к графу G_θ можно добавить хотя бы одну дугу.

3. Все листовые вершины, озаглавленные переменной из множества X, помечаются попарно различными вспомогательными переменными из множества Y. После этого применяется процедура редукции с учетом замечания, позволяющего работать с избыточными графами.

Лемма 12. *Алгоритм построения редуцированной точной нижней грани двух подстановок корректен.*

Доказательство. Для доказательства утверждения леммы необходимо доказать следующие три факта:

1. построенная в результате работы алгоритма подстановка θ является редуцированной;

2. построенная подстановка θ является прототипом подстановок θ_1 и θ_2;

3. указанная подстановка является наиболее специальной редукцией подстановки $\theta_1 \downarrow \theta_2$.

Первый пункт выполняется в соответствии с третьим шагом алгоритма построения редуцированного антиунификатора. Выполнение второго пункта также следует из описания алгоритма. Остается показать, что полученная в результате

работы алгоритма подстановка θ является не просто редуцированным антиунификатором, но наиболее специальным из них. Справедливость этого утверждения в свою очередь следует из правил добавления дуг в граф G_θ. $\square$

В п. 1.1 была приведена оценка размера ациклического ориентированного графа, реализующего точную нижнюю грань двух подстановок. Приведем также оценку размера ациклического ориентированного графа, реализующего редуцированную антиунификацию двух подстановок, а также оценку времени работы этого алгоритма.

Лемма 13. *Количество шагов алгоритма построения редуцированной антиунификации не более, чем* $O(|X| + min(|G_{\theta_1}|, |G_{\theta_2}|))$.

Доказательство. На первом шаге своей работы указанный алгоритм осуществляет не более $|X|$ действий. Второй шаг работы алгоритма осуществляет пополнение множества вершин, и количество итераций пополнения, происходящих на этом шаге, не превосходит числа вершин в каждом из исходных графов G_{θ_1} и G_{θ_2}, то есть $min(|G_{\theta_1}|, |G_{\theta_2}|)$. Процедура редукции, проводимая на третьем шаге, имеет линейную сложность. Таким образом, количество шагов алгоритма может быть сверху оценено величиной $O(|X| + min(|G_{\theta_1}|, |G_{\theta_2}|))$. $\square$

Но для того, чтобы оценить размер графа редуцированного антиунификатора, необходимо ввести еще одну сложностную характеристику АОГ для редуцированных подстановок. Пусть η — произвольная редуцированная подстановка из множества $Subst(X, Y, F)$, а ациклический ориентированный граф G_η — ее реализация. *Сложностью редуцированной подстановки* назовем величину $l(\eta) = N_F(\eta) + |X| - N_X(\eta)$, где $N_F(\eta)$ — количество вершин в графе G_η, помеченных функциональными символами из множества F, а $N_X(\eta)$ — количество вершин графа G_η, которым приписаны переменные из множества X.

Для введенной таким образом сложности редуцированной подстановки справедливо следующее утверждение.

Лемма 14. *Пусть* $\theta = \theta_1 \Downarrow \theta_2$. *Тогда для нее справедливо:* $l(\theta) \leq min(l(\theta_1), l(\theta_2))$.

Доказательство. Поскольку $l(\theta) = N_F(\theta) + |X| - N_X(\theta)$, достаточно рассмотреть оценки величин $N_F(\theta)$ и $N_X(\theta)$. Из описания алгоритма следует, что каждая вершина, добавляемая в граф G_θ на втором этапе работы алгоритма, взаимно однозначно соответствует ровно одной вершине графа G_{θ_1} и одной вершине графа G_{θ_2}, из чего следует неравенство $N_F(\theta) \leq min(N_F(\theta_1), N_F(\theta_2))$. Кроме того, величина $N_X(\theta)$ точно не меньше, чем количество озаглавленных переменных в каждой из подстановок θ_1 и θ_2, то есть $N_X(\theta) \geq max(N_X(\theta_1), N_X(\theta_2))$. Из этого уже следует утверждение леммы. □

Лемма 15. *Пусть* $\theta = \theta_1 \Downarrow \theta_2$. *Тогда равенство* $l(\theta) = l(\theta_i)$, $i \in \{1, 2\}$, *означает, что* $\theta \sim \theta_i$.

Доказательство. Исходя из того, что $N_F(\theta) \leq min(N_F(\theta_1), N_F(\theta_2))$ и $N_X(\theta) \geq max(N_X(\theta_1), N_X(\theta_2))$, из совпадения размеров $l(\theta) = l(\theta_i)$, $i \in \{1, 2\}$ следует, что графы, реализующие подстановки θ и θ_i изоморфны. □

Приведенные выше утверждения позволяют применить алгоритмы, работающие с редуцированными подстановками, к проверке логико-термальной эквивалентности программ. В следующем пункте данной главы приведена модификация алгоритма, предложенного в п. 2.2, которая использует редуцированные подстановки.

2.5. Модифицированный алгоритм проверки логико-термальной эквивалентности, его корректность и сложность

Описанные в предыдущем разделе свойства редуцированной антиунификации подстановок позволяют внести в алгоритм проверки логико-термальной эквивалентности следующее изменение.

Будем рассматривать алгоритм проверки логико-термальной эквивалентности фрагментов программ, заменив в нем операцию взятия точной нижней грани подстановок на операцию редуцированной антиунификации подстановок. Полученный в результате такой замены операций алгоритм будем далее называть модифицированным алгоритмом проверки логико-термальной эквивалентности программ.

Теорема 10. *Модифицированный алгоритм проверки логико-термальной эквивалентности корректен.*

Доказательство. Заметим, что леммы 4, 5, 6, 7, а также теоремы 4, 5 из всех свойств операции взятия точной нижней грани подстановок опираются только на свойства, показанные в леммах 1 и 3. Но справедливость этих же свойств для операции редуцированной антиунификации показана в лемме 11 и теоремах 6, 7, 8, 9. Это означает, что указанные леммы сохраняют справедливость всех утверждений, обосновывающих корректность алгоритма проверки логико-термальной эквивалентности. □

Приведенные в леммах 13 и 14 оценки времени выполнения редуцированной антиунификации и размера ее результата позволяют привести оценку трудоемкости модифицированного алгоритма проверки логико-термальной эквивалентности программ.

Теорема 11. *Существует алгоритм, который решает задачу проверки логико-термальной эквивалентности произвольной пары программ* π_1 *и* π_2 *за* $O(n^6)$ *шагов, где* $n = max(|\pi_1|, |\pi_2|)$.

Доказательство. Для приведенного в этом разделе модифицированного алгоритма проверки логико-термальной эквивалентности программ уже была обоснована его корректность. Приведем теперь оценку его сложности.

Этот алгоритм работает с графом совместных вычислений исходных программ, для которого количество его вершин заведомо не превосходит величины

n^2. Размер каждой приписанной его вершине подстановки $\eta_{(v',v'')}$ может быть оценен сверху величиной n^2. Из леммы 14 и замечания к ней следует, что на каждом шаге размер хотя бы одной из подстановок, приписанных вершинам графа совместных вычислений, уменьшается. Это означает, что количество итераций алгоритма может быть оценено величиной $O(n^4)$.

На каждой итерации алгоритм осуществляет процедуру вычисления редуцированной антиунификации подстановок $\theta\eta_{(v',v'')} \Downarrow \eta_{(u',u'')}$. Лемма 13 показывает, что сложность осуществления этой процедуры линейно зависит от размеров подстановок. Тогда сложность всего алгоритма может быть оценена величиной $O(n^6)$, что и требовалось доказать.

Глава 3

Логико-термальная унифицируемость программ

Как уже было сказано во введении, задача проверки логико-термальной эквивалентности программ непосредственно связана с задачами рефакторинга программного кода. В частности, задача выделения метода подразумевает, что изъятый фрагмент кода действительно является эквивалентным коду той функции, вызовом которой заменяется его вхождение. При этом, однако, каждый фрагмент программного кода подразумевает наличие "предыстории": перед входом во фрагмент каждая переменная имеет некоторую термальную историю. В связи с этим оправданным является подход, когда происходит не просто изъятие фрагмента и замена его на вызов некоторой процедуры, не делающей никаких предположений о значении переменных, а замена фрагмента вызовом процедуры, принимающей на вход переменные в их текущем состоянии.

В данной главе вводится понятие логико-термальной унификации программ. Предложенный алгоритм проверки унифицируемости программ позволяет проверить, может ли один фрагмент кода быть логико-термально эквивалентен другому фрагменту при условии, что перед входом в каждый из них осуществляется некоторая инициализация переменных. Данный алгоритм позволяет также в случае, если программы являются унифицируемыми, получать унифицирующую подстановку.

Полученные в данной главе результаты были изложены в статьях [104], [149] и частично в статье [146].

3.1. Задача унификации программ

Для того, чтобы формально поставить задачу проверки логико-термальной унифицируемости программ и задачу нахождения унификатора, введем предварительно операцию применения подстановки к программе.

Пусть заданы программа $\pi(X) = \langle X, Y, V, v_{in}, v_{out}, B, \to, \lambda_0 \rangle$ и подстановка α, $\alpha \in Subst(X, X, F)$. Тогда *результатом применения подстановки* α *к программе* π назовем программу $\alpha;\pi$, полученную в результате следующих действий:

- в программе π переименовывается входная вершина: вершина v_{in} становится вершиной v_0;
- в программу вносится новая вершина v, исходящие из нее дуги направляются в вершину v_0. Вершине v приписывается нульместный атом A_0. В программу вводятся две дуги, ведущие из вершины v в вершину v_0, одна из которых помечается 0, а другая — 1. Каждой из этих дуг приписывается подстановка α;
- введенная вершина v становится входной в новой программе.

Семантически таким образом определенное применение подстановки к программе соответствует задаче, возникающий при выделении метода в рефакторинге. Подстановка α играет роль набора подготовительных присваиваний перед передачей управления в тело функции, представляемой программой π. Это позволяет изымать из программы не только логико-термально эквивалентные фрагменты кода, но, как будет показано дальше, некоторые виды фрагментов, которые эквивалентными не являются, но тем не менее имеют сходное поведение. Для формальной постановки указанной задачи введем понятие логико-термально унифицируемых программ.

Пусть конечное множество переменных X разбито на два непересекающихся подмножества X' и X'' . Подстановку α, $\alpha \in Subst(X, X, F)$, назовем *логико-термальным унификатором* программ $\pi'(X')$ и $\pi''(X'')$, если программы $\alpha;\pi'$ и $\alpha;\pi''$ логико-термально эквивалентны. Далее для краткости будем называть логико-термальный унификатор просто унификатором.

Две программы $\pi'(X')$ и $\pi''(X'')$ назовем логико-термально унифицируемыми, если для них существует унификатор.

Унификатор α программ $\pi'(X')$ и $\pi''(X'')$ назовем *наиболее общим унификатором* этих программ, если для любого унификатора этих программ α' выполнено $\alpha \leq \alpha'$. Наиболее общий унификатор двух программ $\pi'(X')$ и $\pi''(X'')$ будем обозначать $MGU(\pi'(X'), \pi''(X''))$.

Для унификатора программ $\pi'(X')$ и $\pi''(X'')$ справедливо следующее свойство.

Теорема 12. *Подстановка* α, $\alpha \in Subst(X' \cup X'', X' \cup X'', F)$ *является унификатором программ* $\pi' = \langle X', Y', V', v'_{in}, v'_{out}, B', \to', \lambda'_0 \rangle$ *и* $\pi'' = \langle X'', Y'', V'', v''_{in}, v''_{out}, B'', \to'', \lambda''_0 \rangle$ *в том и только в том случае, когда граф их совместных вычислений* $\Gamma_{\pi',\pi''}$ *корректен и для любого маршрута*

$$path = (v'_{in}, v''_{in}) \overset{\delta_0,\theta_0}{\mapsto} (v'_1, v''_1) \overset{\delta_1,\theta_1}{\mapsto} \dots \overset{\delta_n,\theta_n}{\mapsto} (v'_{n+1}, v''_{n+1}),$$

где $\theta_i = \{\theta'_i \cup \theta''_i\}$, *в нем выполнено равенство* $B'(v'_{n+1})\theta\alpha = B''(v''_{n+1})\theta\alpha$, *где* $\theta = \theta_n\theta_{n-1}\dots\theta_0$.

Доказательство.

Пусть программы π' и π'' логико-термально унифицируемы, при этом унифицирующая их подстановка есть α. Тогда по определению унифицируемых программ из теоремы 2 следует, что каждый путь в графе совместных вычислений $\Gamma_{\alpha;\pi',\alpha;\pi''}$ программ $\alpha;\pi'$ и $\alpha;\pi''$ удовлетворяет указанному в теореме равенству. Рассмотрим граф совместных вычислений $\Gamma_{\pi',\pi''}$ программ π' и π''. Выберем в нем произвольный путь $path$. Ему соответствует единственный путь $path'$ в графе $\Gamma_{\alpha;\pi',\alpha;\pi''}$, который, как показано выше, удовлетворяет равенству в условии теоремы. Отсюда получаем справедливость утверждения теоремы для пути $path$.

Обратное утверждение доказывается схожим образом. Пусть для каждого

пути графа совместных вычислений $\Gamma_{\pi',\pi''}$ выполнено приведенное в теореме равенство. Рассмотрим произвольный путь этого графа $path$. В графе совместных вычислений $\Gamma_{\alpha;\pi',\alpha;\pi''}$ существует единственный путь $path'$, соответствующий пути $path$, для которого выполнено равенство $B'(v'_{n+1})\theta\alpha = B''(v''_{n+1})\theta\alpha$, где θ — соответствующая пути $path$ подстановка. Тогда из определения унификатора программ, произвольного выбора пути $path$ и из теоремы 2 следует, что для каждого пути $path'$ графа $\Gamma_{\alpha;\pi',\alpha;\pi''}$ выполнено равенство из условия теоремы, а значит, программы π' и π'' логико-термально унифицируемы и α — их унификатор. □

Пусть $\Gamma_{\pi',\pi''} = \langle V, w_0, \mapsto, \lambda_0\rangle$ — граф совместных вычислений программ π' и π''. Рассмотрим для каждой вершины $w = (u, v)$ графа $\Gamma_{\pi',\pi''}$ путь $path$, ведущий из вершины w_0 в вершину w, а также подстановку этого пути, θ_{path}. Обозначим через $H_{(u,v)}$ множество пар $(B'(u)\theta_{path}, B''(v)\theta_{path})$. Для введенного таким образом множества справедлива следующая теорема.

Теорема 13. *Для логико-термального унификатора программ π' и π'' справедливо следующее утверждение:*

$$MGU(\pi', \pi'') = \uparrow_{(u,v)\in V} \uparrow_{(B',B'')\in H_{(u,v)}} (B' \uparrow B'').$$

Доказательство. Справедливость этого утверждения следует из теорем 3 и 12. □

Указанные выше свойства унификатора программ позволяют использовать методы проверки логико-термальной эквивалентности программ, описанные в предыдущей главе, для проверки унифицируемости программ и построения унификаторов.

3.2. Алгоритм проверки логико-термальной унифицируемости программ

В этом разделе приведен алгоритм, который проверяет, унифицируемы ли заданные программы, и если это так, то строит наиболее общий унификатор. В основу этого алгоритма положен алгоритм проверки логико-термальной эквивалентности, описанный в предыдущей главе. Сначала описывается упрощенная версия этого алгоритма, удобная для доказательства его корректности. Эта версия, однако, требует экспоненциальной памяти для хранения вспомогательных структур. В следующем разделе предложена модификация алгоритма, дающая полиномиальную оценку сложности.

Указанный алгоритм на каждом i-ом шаге своей работы строит одну подстановку ρ_i, $\rho_i \in Subst(X, X, F)$. Подстановка ρ_0, с которой начинает работу алгоритм, является тождественной, а последовательность подстановок, построенных на каждом из шагов алгоритма, как будет позже показано, является монотонно возрастающей:

$$\rho_0 < \rho_1 < \rho_2 < \ldots < \rho_n.$$

Каждая i-ая подстановка соответствует i-ому шагу работы алгоритма: на этом шаге начальная вершина графа совместных вычислений помечается текущей подстановкой ρ_i, после чего запускается процедура вычисления стационарной разметки графа подстановками. По окончании работы процедуры очередная подстановка ρ_{i+1} вычисляется как композиция подстановки ρ_i и наиболее общего унификатора некоторого конечного множества пар атомарных выражений. Алгоритм завершает свою работу в одном из следующих двух случаев: когда происходит стабилизация подстановки, то есть $\rho_i \sim \rho_{i+1}$, или когда невозможно построение унификатора соответствующего множества пар атомарных выражений. Второй вариант соответствует ситуации.ю когда для программ не

существует унификатора.

Далее будет описан сам алгоритм проверки логико-термальной унифицируемости программ $\pi'(X')$ и $\pi''(X'')$. Будем рассматривать их граф совместных вычислений $\Gamma_{\pi',\pi''} = \langle V, w_0, \mapsto, \lambda_0 \rangle$.

Как и алгоритм проверки логико-термальной эквивалентности программ, алгоритм проверки логико-термальной унифицируемости осуществляет разметку графа совместных вычислений подстановками. При этом процедура вычисления стационарной разметки графа приписывает каждой вершине $w = (u', u'')$ подстановку η_w, $\eta_w \in Subst(X, X \cup Y, F)$, где $Y = \{y_1, y_2, \ldots, y_n\}$ — бесконечное множество переменных, отличных от переменных множества X, переменные-заглушки. Кроме того, вершине w приписывается также некоторое множество подстановок S_w из класса $Subst(X, X, F)$. Далее приведено описание n-ого этапа работы алгоритма.

- **Начальная разметка графа.** Вершине $w_0 = (u'_{in}, u''_{in})$ приписывается подстановка ρ_n (подстановка ρ_0 — тождественная). Все остальные вершины графа совместных вычислений помечаются максимальным в квазирешетке подстановок элементом τ. Также вершине w_0 приписывается множество подстановок $S_{w_0} = \{\rho_n\}$. Всем остальным вершинам w графа совместных вычислений приписывается пустое множество подстановок: $S_w = \emptyset$.

- **Вычисление стационарной разметки графа.** На этом этапе происходит вычисление стационарной разметки графа, аналогичное соответствующим действиям модифицированного алгоритма проверки логико-термальной эквивалентности программ. Отличие заключается в следующем. На каждом шаге выбираются две вершины $w' = (u', u'')$, $w'' = (v', v'')$ такие, что $w'' = succ(w', \delta)$ для некоторого произвольного δ, вершинам w' и w'' соответственно приписаны подстановки $\eta_{w'}$ и $\eta_{w''}$, а дуге с пометкой δ между ними приписана подстановка θ. Для каждой такой пары, как и

раньше, вычисляется подстановка $\eta' = \theta\eta_{w'} \Downarrow \eta_{w''}$. Если при этом не выполняется соотношение $\eta' \sim \eta_{w''}$, то подстановка вершины w'' заменяется подстановкой η'. Множество подстановок $S_{w''}$ также заменяется множеством $S'_{w''} = S_{w''} \cup \{\theta\mu : \mu \in S_{w'}\}$ или $S'_{w''} = S_{w''} \cup \{\theta\}$ в случае, когда $S_{w'}$ пусто. Итерации алгоритма повторяются до тех пор, пока для графа совместных вычислений не будет построена такая разметка, что для любой пары его вершин $w' = (u', u'')$ и $w'' = (v', v'')$, соединенных дугой с пометкой θ, выполнено $\eta_{w''} \sim \theta\eta_{w'} \Downarrow \eta_{w''}$.

- **Устранение неравенств.** После стабилизации разметки графа совместных вычислений на некотором шаге с номером n для каждой вершины $w = (v', v'')$ графа совместных вычислений проверяется равенство $B'(v')\eta_w = B''(v'')\eta_w$. В случае, если для всех вершин указанное равенство выполнено, подстановка ρ_n объявляется унификатором программ π' и π'' и алгоритм завершает свою работу. В противном случае осуществляется вычисление наиболее общего унификатора:

$$\rho' = \uparrow \bigcup_{w=(v',v'')\in V} \bigcup_{\mu\in S_w} \{(B'(v')\mu, B''(v'')\mu)\}.$$

Если указанное множество пар атомов неунифицируемо, то алгоритм останавливается и объявляет, что программы также не являются унифицируемыми. В противном случае вычисляется подстановка $\rho_{n+1} = \rho_n\rho'$ и алгоритм унификации программ переходит к следующему $n + 1$ этапу своего выполнения.

В предыдущей главе было показано, что процедура построения стационарной разметки графа всегда завершается. Покажем, что у данного алгоритма конечное количество шагов, т.е. последовательность подстановок $\rho_0, \rho_1, \ldots, \rho_n \ldots$ также является конечной. Для этого рассмотрим следующие три утверждения.

Лемма 16. *Пусть на n-ом шаге работы алгоритма вершине* $w = (v', v'')$ *приписано множество подстановок* S_w*. Тогда для любой подстановки* μ, $\mu \in S_w$ *существует путь* $path$ *в графе совместных вычислений из начальной вершины* $w_{in} = (v'_{in}, v''_{in})$ *в вершину* w *такой, что для него выполнено равенство* $\mu = \theta_{path}\rho_n$.

Доказательство. Доказательство утверждения леммы можно провести индукцией по количеству итераций процедуры построения стационарной разметки графа на шаге работы с фиксированным номером n. Для этого достаточно рассмотреть правила формирования множества S_w. □

Лемма 17. *Пусть на шаге работы алгоритма с номером* n *вершине* $w = (v', v'')$ *приписана подстановка* η_w *и множество подстановок* S_w*. Тогда для подстановки* η_w *справедливо равенство:*

$$\eta_w = \Downarrow_{\mu \in S_w} \mu.$$

Доказательство. Как и в предыдущей лемме, доказательство проводится индукцией по числу шагов процедуры вычисления стационарной разметки графа совместных вычислений. Для входной вершины справедливость утверждения очевидна. Рассмотрим индуктивный переход. Предположим, что для вершин $w' = (v', v'')$ и $w'' = (u', u'')$ таких, что $w'' = succ(w', \delta)$ и дуге с пометкой δ приписана подстановка θ, справедливо утверждение леммы, то есть $\eta_{w'} = \Downarrow_{\mu \in S_{w'}} \mu$ и $\eta_{w''} = \Downarrow_{\mu \in S_{w''}} \mu$. Заметим, что на основании леммы 8 справедливо соотношение :

$$msr(\theta\eta_{w'}) = msr(\theta \Downarrow_{\mu \in S_{w'}} \mu) = \Downarrow_{\mu \in S_{w'}} \theta\mu.$$

Но тогда с учетом теоремы 7 справедлива следующая цепочка равенств:

$$\theta\eta_{w'} \Downarrow \eta_{w''} = msr(\theta\eta_{w'}) \Downarrow msr(\eta_{w''}) = (\Downarrow_{\mu \in S_{w'}} \theta\mu) \Downarrow (\Downarrow_{\mu \in S_{w''}} \mu) = \Downarrow_{\mu \in S_{w''} \cup \theta\mu : \mu \in S_{w'}} \mu,$$

которая доказывает утверждение леммы. □

Для подстановки θ, $\theta \in Subst(X, X, F)$, обозначим через $Var_{free}(\theta)$ множество переменных x_i из X таких, что для $\theta(x_i) = x_i$. Тогда имеет место следующее утверждение.

Лемма 18. *Пусть $\rho_0, \rho_1, \ldots, \rho_n, \rho_{n+1}, \ldots$ — последовательность подстановок, построенная алгоритмом унификации программ. Тогда для любого n, $n \geq 0$, справедливо соотношение $Var_{free}(\rho_{n+1}) \subseteq Var_{free}(\rho_n)$, причем равенство $Var_{free}(\rho_{n+1}) = Var_{free}(\rho_n)$ выполняется тогда и только тогда, когда $\rho_{n+1} \sim \rho_n$.*

Доказательство. Рассмотрим n-ый этап работы алгоритма. Из леммы 16 следует, что для любой вершины w и для каждой подстановки $\mu \in S_w$ существует путь $path$ в графе совместных вычислений такой, что μ представима в виде композиции $\mu = \theta_{path}\rho_n$, откуда следует, что $Var_{free}(\mu) \subseteq Var_{free}(\rho_n)$.

Рассмотрим случай, когда алгоритм логико-термальной унификации программ не останавливает свою работу на шаге с номером n. Это означает, что для некоторой вершины $w = (u, v)$ выполнено неравенство $B'(u)\eta_w \neq B''(v)\eta_w$. По утверждению леммы 17 для подстановки η_w верно равенство $\eta_w = \Downarrow_{\mu \in S_w} \mu$. Из этих двух фактов на основании теоремы 9 следует, что существует такая подстановка μ, $\mu \in S_w$, что для нее выполнено соотношение $B'(u)\mu \neq B''(v)\mu$. Это, в свою очередь, означает, что на данном этапе будет вычислен наиболее общий унификатор множества $\rho' = \uparrow \bigcup_{w=(u,v)\in V} \cup_{\mu \in S_w} \{(B'(u)\mu, B''(v)\mu)\}$ и он будет отличен от тождественной подстановки. Но тогда множество переменных $Var_{free}(\rho')$ является собственным подмножеством множества переменных $Var_{free}(\mu)$ хотя бы для одной подстановки μ из множества S_w. А значит, это же множество является собственным подмножеством множества переменных $Var_{free}(\rho_n)$, но тогда справедливо строгое включение $Var_{free}(\rho_{n+1}) = Var_{free}(\rho_n\rho') \subset Var_{free}(\rho_n)$. При этом выполнение равенства $Var_{free}(\rho_{n+1}) = Var_{free}(\rho_n)$ означает, что алгоритм унификации уже закончил свою работу. □

Следствием трех приведенных лемм является теорема о конечном числе шагов алгоритма.

Теорема 14. *Для любой пары программ* $\pi' = \langle X', Y', V', v'_{in}, v'_{out}, B', \to', \lambda'_0 \rangle$ *и* $\pi'' = \langle X'', Y'', V'', v''_{in}, v''_{out}, B'', \to'', \lambda''_0 \rangle$ *алгоритм проверки логико-термальной унифицируемости завершает свою работу.*

Доказательство. Рассмотрим последовательность $\rho_0, \rho_1, \ldots, \rho_n, \ldots$ подстановок, построенных алгоритмом проверки логико-термальной унифицируемости программ. В соответствии с леммой 18 для этой последовательности выполнена цепочка строгих вложений: $X = X' \cup X'' = Var_{free}(\rho_0) \supset Var_{free}(\rho_1) \supset \ldots \supset Var_{free}(\rho_n) \supset \ldots$. Поскольку множество X конечно, указанная последовательность также является конечной, что влечет за собой утверждение теоремы. □

Кроме завершаемости алгоритма необходимо также показать его корректность. Для этого понадобятся следующие две леммы.

Лемма 19. *Пусть алгоритм проверки логико-термальной унифицируемости программ* π' *и* π'' *завершил свою работу, выдав в качестве результата подстановку* ρ*. Тогда эта подстановка является унификатором программ* π' *и* π''*.*

Доказательство. Справедливость этого утверждения следует из теорем 2 и 12. Действительно, если алгоритм выдал в качестве результата своей работы подстановку ρ, то по теореме 2, лемме 16 и описанию алгоритма это означает, что для любой вершины $w = (u, v)$ и любого пути $path$ в графе совместных вычислений, ведущего из входной вершины в вершину w выполнено $B'(u)\theta_{path}\rho = B''(v)\theta_{path}\rho$. Но тогда по теореме 12 это означает, что подстановка ρ действительно является унификатором программ π' и π''. □

Лемма 20. *Если программы π' и π'' логико-термально унифицируемы, то для каждой подстановки ρ_i, вычисленной на i-ом шаге работы алгоритма, верно соотношение*

$$\rho_i \leq MGU(\pi', \pi'').$$

Доказательство. Доказательство проведем индукцией по количеству шагов алгоритма проверки логико-термальной унифицируемости программ. Для подстановки ρ_0 утверждение справедливо. Покажем, что если утверждение справедливо для подстановки ρ_i, вычисленной на i-ом шаге работы алгоритма, то оно справедливо и для подстановки ρ_{i+1}. Подстановка ρ_{i+1} по алгоритму является композицией $\rho'\rho_i$, где подстановка ρ' есть унификатор множества $\bigcup_{w=(v',v'')\in V} \cup_{\mu\in S_w}\{(B'(v')\mu, B''(v'')\mu)\}$ для некоторой вершины w.

Для наиболее общих унификаторов произвольных множеств пар атомов H_1 и H_2 справедливо следующие утверждения:

- если H_2 — унифицируемое множество пар и при этом $H_1 \subseteq H_2$, то наиболее общие унификаторы этих множеств ν_1 и ν_2 соответственно находятся в соотношении $\nu_1 \leq \nu_2$. Справедливость этого утверждения следует из определения наиболее общего унификатора атомов;

- если наиболее общий унификатор множества H_1 есть подстановка ν, то наиболее общие унификаторы множеств $H_1 \cup H_2$ и $H_1\nu \cup H_2\nu$ совпадают с точностью до переименования переменных. Справедливость этого утверждения следует из определения наиболее общего унификатора атомов и свойств операции композиции.

Но тогда, с учетом сформулированных свойств, представления $\rho_{i+1} = \rho'\rho_i$ и теоремы 13 получаем, что для подстановки ρ_{i+1} утверждение леммы также справедливо. Значит, каждая из подстановок $\rho_0, \rho_1, \ldots, \rho_n$, полученных в ходе работы алгоритма проверки логико-термальной унифицируемости программ, является прототипом наиболее общего унификатора этих программ. □

Приведенные выше утверждения позволяют сформулировать и обосновать следующую теорему о корректности алгоритма проверки логико-термальной унифицируемости программ.

Теорема 15. *Алгоритм проверки логико-термальной унифицируемости программ π' и π'' останавливается с положительным результатом и объявляет подстановку ρ унификатором программ π' и π'' в том и только том случае, когда программы действительно являются унифицируемыми и при этом для подстановки ρ выполнено соотношение $\rho = MGU(\pi', \pi'')$.*

Это утверждение непосредственно следует из справедливости теоремы 14, а также лемм 19 и 20.

Приведенный в этом разделе алгоритм требует явного построения на каждом шаге множеств ассоциированных с вершинами подстановок S_w. При этом на каждом шаге вычислений количество подстановок во множестве S_w может увеличиваться в экспоненциальной зависимости от размеров входных программ. Это делает приведенный в этом пункте алгоритм неэффективным с точки зрения использованной памяти. В следующем пункте приведен алгоритм, который позволяет задавать множества подстановок S_w неявно. Такой подход позволит использовать полиномиальный относительно размеров программ объем памяти, а не экспоненциальный, как в текущем варианте алгоритма.

3.3. Полиномиальный алгоритм проверки логико-термальной унифицируемости программ и его корректность

Как показано в предыдущем разделе, каждый шаг алгоритма проверки унифицируемости программ вычисляет стационарную разметку графа совместных вычислений при помощи операции редуцированной унификации. Рассмотрим подробнее эту процедуру. На каждом шаге для пары вершин w' и w'' таких,

что $w'' = succ(w', \delta)$, а дуге с пометкой δ приписана подстановка θ, вычисляется новая подстановка $\eta' = \theta\eta_{w'} \Downarrow \eta_{w''}$. При этом в случае, когда $\eta' \neq \eta_{w''}$, подстановка η' заменяет подстановку $\eta_{w''}$.

Рассмотрим подстановки-пополнения, образующиеся при вычислении $\eta' = \theta\eta_{w'} \Downarrow \eta_{w''}$. Обозначим через ξ' пополнение подстановки η' до подстановки $\theta\eta_{w'}$, а через ξ'' — пополнение η' до подстановки $\eta_{w''}$. Отметим также, что эти подстановки имеют в качестве области определения множество вспомогательных переменных $Y = \{y_1, \ldots, y_n, \ldots\}$. В том случае, если подстановка η' в результате была приписана вершине w'', то для подстановок η', ξ', ξ'' можно ввести отношение *предшествования*: подстановки ξ' и ξ'' объявляются предшественниками подстановки η'. Обозначим это отношение следующим образом: $\xi' \lhd \eta'$ и $\xi'' \lhd \eta'$. При этом для подстановок ξ' и ξ'' также определены предшественники: будем считать, что предшественниками ξ' являются все предшественники подстановки $\eta_{w'}$, а предшественниками ξ'' — все предшественники подстановки $\eta_{w''}$. Отметим, что суммарный размер всех подстановок-пополнений, вычисляемых алгоритмом, ограничен сверху величиной $O(n^6)$, где n — суммарный размер входных программ. Данная оценка может быть получена следующим образом. Заметим, что в ходе работы алгоритма в каждой вершине w происходит замена подстановки η_w. Такая замена может происходить не более, чем $O(n^2)$ раз, как было показано в предыдущем разделе. При этом всего вершин в графе не более $O(n^2)$, значит, количество порождаемых подстановок-пополнений не превосходит $O(n^4)$. При этом размер каждой подстановки-пополнения не превосходит размера той подстановки, чьим предшественником она является. Таким образом, суммарный размер всех подстановок-пополнений, порождаемых в ходе работы процедуры построения разметки графа совместных вычислений, оценивается величиной $O(n^6)$.

Теперь покажем связь между подстановкой η_w, приписанной некоторой вершине w в ходе построения стационарной разметки графа, и подстановками-пополнениями, связанными с этой подстановкой отношением предшествования.

Лемма 21. *Пусть на некотором шаге работы процедуры вычисления стационарной разметки графа совместных вычислений вершине w приписана подстановка η_w и множество подстановок S_w. Тогда для любой подстановки μ, $\mu \in S_w$, существует такая последовательность подстановок-пополнений $\xi_1, \xi_2, \ldots, \xi_m$, связанных с подстановкой η_w отношением предшествования $\xi_m \lhd \xi_{m-1} \lhd \ldots \lhd \xi_1 \lhd \eta_w$, для которой справедливо равенство $\mu = \eta_w \xi_1 \xi_2 \ldots \xi_m$.*

Доказательство. Доказательство осуществляется индукцией по числу шагов процедуры вычисления стационарной разметки графа совместных вычислений. На первом шаге множество ξ_i пустое, а для пустого множества утверждение справедливо.

Рассмотрим произвольную вершину w графа совместных вычислений исходных программ. Не нарушая общности рассуждений, будем считать, что вершина w такова, что $w = succ(w', \delta)$ и при этом дуге с пометкой δ приписана подстановка θ. Предположим, что на некотором шаге с номером m утверждение леммы в некоторой вершине w справедливо: для любой подстановки μ из S_w верны равенства $\mu = \eta_w \xi_1 \xi_2 \ldots \xi_m$ для некоторой последовательности подстановок-пополнений. Пусть на шаге с номером $m+1$ в вершине w происходит изменение разметки: подстановка η_w заменяется подстановкой $\eta' = \theta \eta_{w'} \Downarrow \eta_w$, а множество подстановок S_w заменяется множеством $S_w \cup \{\theta\mu' : \mu' \in S_{w'}\}$. Рассмотрим подстановки-пополнения ξ' и ξ'', которые образуются на этом шаге. Для них справедливы равенства $\theta\eta_{w'} = \eta'\xi'$ и $\eta_w = \eta'\xi''$. По индуктивному предположению каждая подстановка μ, $\mu \in S_w$, может быть представлена в виде композиции $\mu = \eta_w \xi_1 \xi_2 \ldots \xi_m = \eta' \xi'' \xi_1 \xi_2 \ldots \xi_m$. Также по индуктивному предположению для каждой подстановки μ из множества $\{\theta\mu' : \mu' \in S_{w'}\}$ верны равенства $\mu = \theta\mu' = \theta\eta_{w'}\xi_1\xi_2 \ldots \xi_m = \eta'\xi'\xi_1\xi_2 \ldots \xi_m$, что подтверждает справедливость леммы. □

Лемма 22. *Пусть на некотором шаге работы процедуры вычисления стационарной разметки графа совместных вычислений вершине w приписана подста-*

новка η_w и множество подстановок S_w. Тогда для любой последовательности подстановок-пополнений, построенной в ходе работы алгоритма, в которой подстановки связаны соотношением $\xi_m \lhd \xi_{m-1} \lhd \ldots \lhd \xi_1 \lhd \eta_w$, существует такая подстановка μ, $\mu \in S_w$, для которой справедливо равенство $\mu = \eta_w \xi_1 \xi_2 \ldots \xi_m$.

Доказательство. Доказательство данного утверждения, как и доказательство предыдущего, осуществляется по индукции. Для обоснования индуктивного перехода в данном случае осуществляются все те же действия, что и в доказательстве предыдущей леммы, но в обратном порядке. □

Доказанные выше леммы означают, что вся информация о подстановках множеств S_w, ассоциированных с вершинами w графа совместных вычислений исходных программ, содержится в подстановках-пополнениях, связанных отношением предшествования. Далее приведем описание тех изменений, которые нужно внести в алгоритм проверки логико-термальной унифицируемости для того, чтобы сократить объемы занимаемой им рабочей памяти. Для этого опишем новую процедуру унификации, которая позволяет вычислить $\rho = \uparrow \bigcup_{w=(v',v'')\in V} \cup_{\mu\in S_w} \{(B'(v')\mu, B''(v'')\mu)\}$ за полиномиальное время с использованием введенного неявного описания подстановок.

Данная процедура состоит в поочередном выполнении действий алгоритма унификации Мартелли-Монтанари, описанного в главе 1, и анализа подстановок-пополнений на каждом шаге. Пусть на очередном шаге достигнута стационарная разметка всех вершин графа совместных вычислений. Процедура начинает свою работу с применения алгоритма Мартелли-Монтанари к системе уравнений $\{B'(v')\eta_w = B''(v'')\eta_w : w = (v', v'') \in V_{\Gamma_{\pi',\pi''}}\}$. Как видно из описания в главе 1, алгоритм построит равносильную приведенную систему уравнений, обозначим эту систему E. Если построение такой системы невозможно, то унификация исходного множества пар атомов также невозможна. Но если алгоритму Мартелли-Монтанари удается построить приведенную систему уравнений, то дальнейшая работа ведется с подстановками-пополнениями.

Рассмотрим множество вспомогательных переменных, введенных в результате построения редуцированных антиунификаций, $Y = \{y_1, y_2, \ldots, y_m\}$. Рассмотрим подстановки-пополнения, полученные на этом шаге. Напомним, что для каждой такой подстановки определен ее предшественник. Выберем из всех подстановок-пополнений максимальную по транзитивному замыканию отношения предшествования пару ξ', ξ''. Эта пара определяет хотя бы одну переменную y_i из множества Y. После этого ко всем уравнениям системы применяются подстановки ξ' и ξ'', что означает, что исходная система E заменяется системой уравнений $E\xi' \cup E\xi''$, в которой переменная y_i будет отсутствовать. Далее процедура снова вызывает алгоритм Мартелли-Монтанари для новой системы и все описанные выше действия повторяются. Отметим, что переменная y_i не появится при следующих итерациях алгоритма, так как каждая вспомогательная переменная множества Y возникает в результате редуцированной антиунификации только один раз, а следовательно определяется только одной парой подстановок-пополнений ξ', ξ''.

Выполнение алгоритма продолжается до тех пор, пока на очередном шаге не будет получена приведенная система уравнений E, не содержащая связок для вспомогательных переменных множества Y: $E = \{x_1 = t_1(\ldots);\ x_2 = t_2(\ldots);\ \ldots;\ x_k = t_k(\ldots)\}$. Тогда подстановка $\rho' = \{x_1/t_1(\ldots),\ x_2/t_2(\ldots),\ \ldots,\ x_k/t_k(\ldots)\}$ объявляется результатом работы данной процедуры. Далее будем называть описанную процедуру *процедурой быстрой унификации множества пар атомов.*

Лемма 23. *Процедура быстрой унификации пар атомов вычисляет подстановку* $\rho' = \{x_1/t_1(\ldots),\ x_2/t_2(\ldots),\ \ldots,\ x_k/t_k(\ldots)\}$ *тогда и только тогда, когда существует наиболее общий унификатор* $\rho = \uparrow \bigcup_{w=(v',v'')\in V} \cup_{\mu\in S_w} \{(B'(v')\mu, B''(v'')\mu)\}$. *При этом* $\rho' = \rho$.

Доказательство. Справедливость этого утверждения следует из лемм 17, 21 и 22, а также из корректности алгоритма Мартелли-Монтанари $\square$

Лемма 24. *Вычисление подстановки* $\rho = \uparrow \bigcup_{w=(v',v'')\in V} \cup_{\mu\in S_w}\{(B'(v')\mu, B''(v'')\mu)\}$ *может быть осуществлено за время* $O(n^{10})$*, где* n *— суммарный размер исходных программ.*

Доказательство. Оценим время выполнения описанной выше процедуры быстрой унификации множества пар атомов. Алгоритм унификации Мартелли-Монтанари в этой процедуре запускается не более $O(n^4)$ раз, т.к. на каждом шаге из системы убывает хотя бы одна вспомогательная переменная, общее количество которых ограничено этой величиной. При этом алгоритм Мартелли-Монтанари применяется к системам, в которых также не более $O(n^4)$ уравнений. Размер термов в каждом уравнении оценивается величиной $O(n^2)$. Таким образом, справедлива оценка леммы, вычисление унификатора может быть осуществлено за время $O(n^{10})$. □

Описанный в предыдущем пункте алгоритм проверки логико-термальной унифицируемости программ требует построения в явном виде множества подстановок S_w. Изменим алгоритм так, чтобы подстановки этого множества вводились в неявном виде, как это было показано выше. Это позволит применять вместо явной унификации множеств пар атомов процедуру быстрой унификации. Для такого алгоритма справедлива следующая теорема.

Теорема 16. *Алгоритм проверки логико-термальной унифицируемости программ* π' *и* π'' *завершает работу за время, полиномиальное относительно суммарного размера исходных программ.*

Доказательство. Из утверждения леммы 18 следует, что алгоритм унификации завершает работу за $O(n)$ этапов работы. Каждый из этапов начинается с вычисления стационарной разметки графа совместных вычислений. Как уже было показано в доказательстве теоремы 11, вычисление стационарной разметки завершает работу за $O(n^4)$ шагов, а с учетом проведения редуцированной антиунификации на каждом шаге, каждый такой этап завершается за $O(n^2)$.

При этом после вычисления стационарной разметки графа предпринимается попытка унификации множества пар атомов, которая осуществляется быстрым алгоритмом за время $O(n^{10})$. Эта величина превосходит оценку времени построения стационарной разметки графа. Следовательно, общее время работы алгоритма проверки логико-термальной унифицируемости программ оценивается величиной $O(n^{11})$. $\square$

Глава 4

Двусторонняя унификация программ

В предыдущих разделах данной работы были рассмотрены алгоритмы, которые позволяют за полиномиальное от размеров входных программ время осуществлять проверку программ на эквивалентность и унифицируемость. Однако эти два вида задач на отвечают полностью требованиям, которые можно было бы предъявить к автоматическому средству рефакторинга. В частности, унификация программ, которая описана в предыдущем разделе, предполагает только поиск такой инициализации входных переменных, которая унифицирует программы. В реальных же случаях выделения метода зачастую необходимо выяснить, какие преобразования программ необходимо осуществить уже после выполнения фрагмента изъятого метода, сохраняя при этом функциональные свойства исходных программ.

При рассмотрении этой задачи возникает задача, связанная с уравнениями над подстановками. Под задачей унификации двух подстановок θ_1 и θ_2 принято понимать поиск таких подстановок η' и η'', что $\theta_1\eta' \sim \theta_2\eta$, что фактически означает решение линейных уравнений вида $\theta_1 X = \theta_2 Y$ в полугруппе подстановок. В данной же работе рассмотрен другой тип уравнений, названный *двусторонней унификацией подстановок*: для двух заданных подстановок θ_1 и θ_2 решается уравнение вида $X\theta_1 Y = \theta_2$. Как показано в последнем разделе этой главы, решение такого рода уравнений находит непосредственное применение в описанной выше задаче рефакторинга. Далее в этой главе подробно рассмотрена задача двусторонней унификации подстановок, а также ее обобщение — задача двусторонней унификации программ. Для данных задач показано, что обе она являются NP-полными. NP-трудность этих задач доказана путем сведения к ним одного из вариантов задачи ограниченного домино (bounded tiling problem) [**?**, Lewis]

Полученные в этой главе результаты были описаны в статьях [105], [153], [154].

4.1. Задача двусторонней унификации подстановок

Задача двусторонней унификации подстановок формально определяется следующим образом. Рассмотрим пару подстановок (θ_1, θ_2) таких, что $\theta_1 \in Subst(X, Y, F)$, а $\theta_2 \in Subst(Z, U, F)$, где множества X, Y, Z, U являются конечными подмножествами множества переменных Var. Пару подстановок (η', η'') таких, что $\eta' \in Subst(Z, X, F)$, $\eta'' \in Subst(Y, U, F)$, назовем *двусторонним унификатором пары* (θ_1, θ_2) тогда и только тогда, когда выполнено равенство $\eta'\theta_1\eta'' \sim \theta_2$. Тогда задача двусторонней унификации заключается в проверке того, существует ли для пары подстановок заданного типа (θ_1, θ_2) двусторонний унификатор (η', η''), и если он существует, то построении такого унификатора. С прикладной точки зрения задача поиска двустороннего унификатора является фактически задачей о решении уравнения над подстановками вида $A\theta_1 B = \theta_2$, где области значений переменных A и B есть множества $Subst(Z, X, F)$ и $Subst(Y, U, F)$ соответственно. Нужно отметить, что такого рода уравнения над подстановками ранее встречались в некоторых задачах. Так, классическая задача унификации подстановок может быть рассмотрена как задача о решении уравнений вида $\theta_1 A = \theta_2 B$. Уравнения вида $A\theta_1 = B\theta_2$ также возникают при проверке эквивалентности в некоторых классах последовательных программ [101].

В отличие от задачи односторонней унификации, задача двусторонней унификации является асимметричной: семантически подстановки θ_1 и θ_2 играют разную роль, также как и подстановки η' и η''. Все четыре задействованных в задаче подстановки действуют над разными множествами переменных. Как показано в четвертом разделе данной главы, подстановки θ_1 и θ_2 описывают вычисления программ, в то время как подстановка η' инициализирует входные

переменные, а подстановка η'' описывает модификацию значений переменных на выходах. В связи с этим важно отметить, что термы подстановки η'' не могут непосредственно влиять на выходные переменные подстановки η': подстановка θ_1 в данном случае обязана выполнять роль посредника.

Еще одной важной особенностью данной задачи является возможность существования для заданной пары подстановок (θ_1, θ_2) более чем одного двустороннего унификатора. Рассмотрим пару подстановок $\theta_1 = \{x/f(y_1, y_2)\}$, $\theta_2 = \{z/f(f(a, b), c)\}$ и еще четыре подстановки следующего вида: $\eta'_1 = \{z/x\}$, $\eta''_1 = \{y_1/f(a, b), y_2/c\}$, $\eta'_2 = \{z/f(x, c)\}$, $\eta''_2 = \{y_1/a, y_2/b\}$. Двусторонним унификатором пары (θ_1, θ_2) может являться как пара (η_1, η_2), так и пара (η'_1, η'_2), при этом подстановки в указанных парах не являются эквивалентными. Деревья, соответствующие указанным в примере подстановкам, приведены на рисунках 4.1, 4.2.

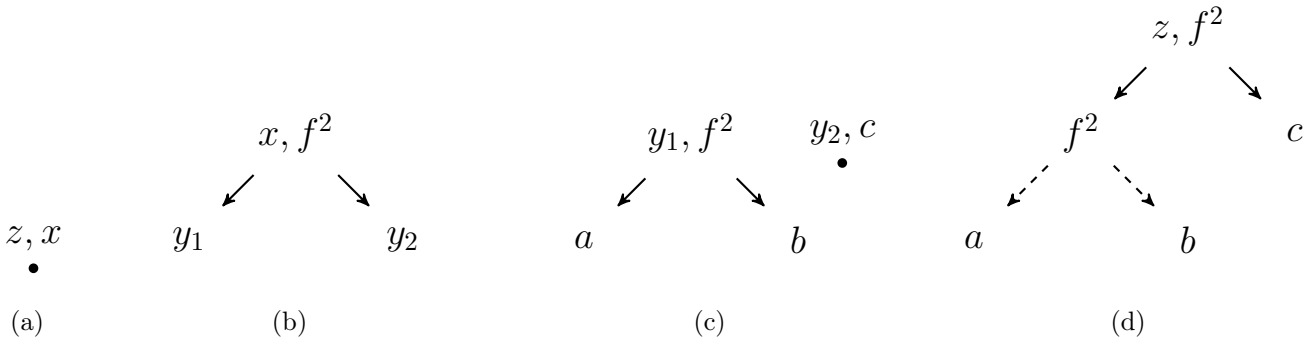


Рис. 4.1. Деревья подстановок: (a) η'_1; (b) θ_1; (c) η''_1; (d) θ_2. Пунктирными линиями в дереве подстановки θ_2 изображены дуги, соответствующие связкам подстановок двустороннего унификатора.

Тем не менее, имеет место следующая лемма.

Лемма 25. *Для произвольной заданной пары подстановок (θ_1, θ_2) количество двусторонних унификаторов этой пары конечно.*

Доказательство. Для каждого двустороннего унификатора (η', η'') пары подстановок (θ_1, θ_2) справедливо, что подстановка η' является шаблоном подстановки θ_2. А как следует из устройства подстановки, число попарно неэквива-

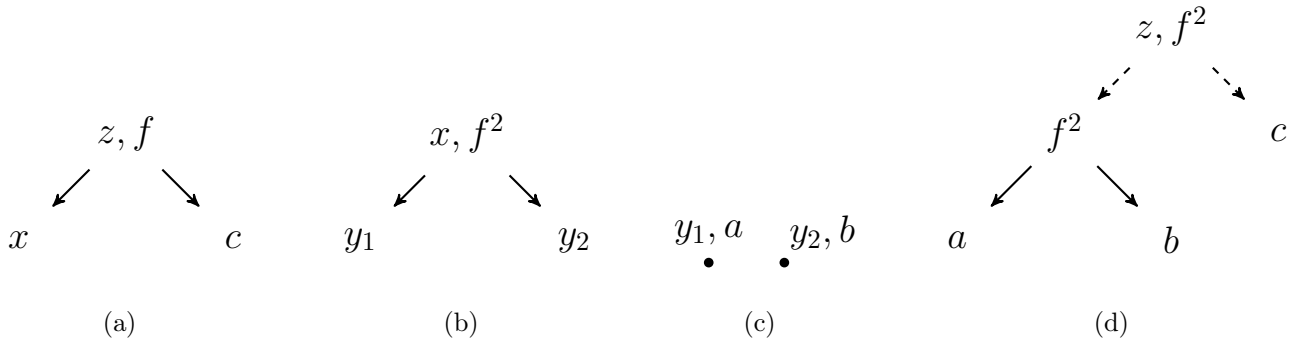


Рис. 4.2. Деревья подстановок: (a) η'_2; (b) θ_1; (c) η''_2; (d) θ_2. Пунктирными линиями в дереве подстановки θ_2 изображены дуги, соответствующие связкам подстановок двустороннего унификатора.

лентных шаблонов каждой подстановки конечно. Аналогично, подстановка η'' является пополнением подстановки θ_2, а число попарно неэквивалентных пополнений подстановки также конечно. Отсюда следует, что множество попарно неэквивалентных двусторонних унификаторов конечно. □

Как уже было отмечено в главе 1, сложность многих алгоритмов работы с подстановками зависит от структуры данных, используемой для представления подстановок. В данном случае будет использовано древесное представление подстановок: каждому терму t_i подстановки $\theta = \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ в соответствие поставлено размеченное дерево. Условимся далее обозначать T_θ лес, соответствующий подстановке θ. По-прежнему считаем, что выполнено условие: $\theta_1 \in (X, Y, F)$, $\theta_2 \in Subst(Z, U, F)$, $\eta' \in Subst(Z, X, F)$, и $\eta'' \in Subst(Y, U, F)$.

Лемма 26. *Задача проверки двусторонней унифицируемости подстановок принадлежит классу сложности NP.*

Доказательство. Покажем, каким образом можно проверить двустороннюю унифицируемость подстановок за полиномиальное время недетерминированным алгоритмом. Алгоритм работает с лесом T_{θ_2} подстановки θ_2 и состоит из следующих шагов:

1. в каждом из деревьев леса T_{θ_2} недетерминированным образом проводится по два вершинных сечения, в результате чего лес T_{θ_2} разделяется на три

части: T', T, T''. При этом в результате сечений листовые вершины части T' совмещаются с корневыми вершинами части T, а листовые вершины T — с корневыми вершинами части T'';

2. осуществляется согласованная разметка вершин. Переменные из множеств X и Y приписываются всем листовым вершинам фрагментов T' и T соответственно, но при этом одинаковая переменная может быть приписана двум разным вершинам u и v только в том случае, если поддерево следующего слоя (слоя T для вершин T' и слоя T'' для вершин T соответственно), корнем которого является вершина v, совпадает с поддеревом следующего слоя, корнем которого является вершина u;

3. проверяется, верно ли, что все построенные таким образом деревья слоя T представляют термы подстановки θ_1.

Последний шаг алгоритма осуществляется за полиномиальное относительно размеров исходных подстановок время, т.к. на нем достаточно осуществить проверку согласованности разметки листовых вершин переменными и вхождение деревьев из леса T в древесное представление подстановки T_{θ_1}. Что и доказывает справедливость леммы. □

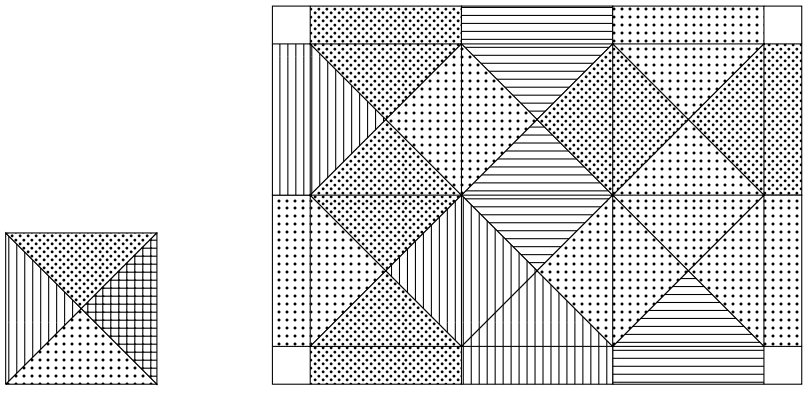
Но задача проверки двусторонней унифицируемости подстановок не просто принадлежит классу NP. Она является также NP-трудной, но для обоснования этого факта необходимо сначала сформулировать задачу ограниченного домино.

4.2. Задача ограниченного домино

Задача ограниченного домино заключается в проверке того, можно ли замостить некоторую ограниченную область плоскости (например, прямоугольник) элементами домино из заданного множества. Сложность этой задачи существенно зависит от вида области, о которой идет речь. В этом пункте будет

рассмотрена задача о покрытии прямоугольной области, которая в англоязычной литературе известна под названием Bounded tiling problem. В статье [54] было показано, что задача ограниченного домино с прямоугольной областью является NP-полной. Именно этот факт позволяет использовать эту задачу для доказательства NP-трудности задачи двусторонней унификации подстановок.

Задача ограниченного домино неформально может быть описана следующим образом. Под элементом домино будем понимать квадрат, каждая сторона которого имеет длину 1 и окрашена в некоторый цвет. Пусть задано конечное множество элементов домино $T = \{T_1, T_2, \ldots, T_L\}$. Рассмотрим прямоугольник размера $m \times n$, каждая сторона которого разделена на сегменты единичной длины. При этом каждый сегмент окрашен в некоторый цвет. Задача ограниченного домино в таком случае заключается в том, чтобы проверить, можно ли разместить в заданном прямоугольнике $m \times n$ элементов домино из множества T таким образом, чтобы смежные стороны каждых двух соседних элементов домино имели одинаковый цвет, и при этом цвет стороны домино, примыкающей к границе, был таким же, что и цвет сегмента границы, к которому он примыкает.



(a) (b)

Рис. 4.3. Задача ограниченного домино: (a) пример элемента домино ; (b) пример правильного покрытия прямоугольной области 2×3.

Формально же задача ограниченного домино ставится следующим образом. Пусть задано конечное множество цветов $Colours = \{1, 2, \ldots, K\}$. *Эле-*

ментом домино назовем четверку $\langle a_1, a_2, a_3, a_4\rangle$, где $a_1, a_2, a_3, a_4 \in Colours$. При этом будем считать, что цвета в четверке упорядочены в соответствии с обходом элемента домино по часовой стрелке: цвету a_1 соответствует окраска верхней грани, цвету a_2 — правой, a_3 — нижней, a_4 — соответственно левой граней домино. Тогда для элементов домино удобно определить следующую индексацию: будем обозначать $tile[0,-1] = a_1$, $tile[-1,0] = a_2$, $tile[0,1] = a_3$, $tile[1,0] = a_4$. Прямоугольная область $n \times m$ задается множеством пар $Area = \{(i,j) : 0 \le i \le n+1, 0 \le j \le m+1\}$. Элементы этого множества — пары (i,j) — называются *квадратами*. Заметим, что в каждом квадрате размещен экземпляр одного из элементов домино. *Внутренней областью* прямоугольной области назовем множество квадратов $Interior = \{(i,j) : 1 \le i \le n, 1 \le j \le m\}$. *Границей* прямоугольной области назовем множество пар $Border = Area \setminus Interior$. Два заданных квадрата (i_1, j_1) и (i_2, j_2) будем называть *соседними*, если для них выполнено соотношение $|i_1 - i_2| + |j_1 - j_2| = 1$. Это отношение в графическом смысле описывает ситуацию, когда два квадрата имеют пару смежных сторон: тогда координаты соответствующих квадратов по одному измерению совпадают, а по другому отличаются ровно на единицу. *Граничным условием* назовем отображение, которое ставит в соответствие каждому квадрату границы некоторый цвет: $B : Border \to Colours$. Это отношение интерпретируется следующим образом: соотношение $B(i,j) = a$ означает, что та сторона квадрата границы (i,j), которая примыкает к внутренней области, окрашена в цвет a. Не нарушая общности рассуждений, можем полагать, что все стороны граничных домино окрашены одинаково.

Пусть задано конечное множество элементов домино: $Tiles = \{tile_1, tile_2, \ldots, tile_L\}$. *Покрытием* прямоугольника $Area$ назовем всякое отображение $T : Interior \to Tiles$, которое задает замощение внутренней части прямоугольника экземплярами элементов домино из множества $Tiles$. Отметим, что подобное замощение по сути является окраской сторон каждого квадрата в соответствии с окраской расположенного в этом квадрате элемента домино.

Пусть задано граничное условие B прямоугольника $Area$. Покрытие T назовем *B-правильным*, если оно удовлетворяет следующим двум требованиям:

1. *внутренняя корректность.* Для каждой пары соседних внутренних квадратов (i_1, j_1), (i_2, j_2) выполнено соотношение $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = T(i_2, j_2)[i_2 - i_1, j_2 - j_1]$;

2. *внешняя корректность.* Для каждого квадрата (i_1, j_1), $(i_1, j_1) \in Interior$, граничащего с квадратом (i_2, j_2), $(i_2, j_2) \in Border$, справедливо равенство $T(i_1, j_1)[i_1 - i_2, j_1 - j_2] = B(i_2, j_2)$.

Отметим, что пункт 1 означает, что смежные стороны соседних квадратов окрашены одинаково, а пункт 2 — что сторона домино, примыкающая к границе, должна иметь ту же окраску, что и смежный с ней участок границы. Обозначим цвет общей стороны пары соседних квадратов (i_1, j_1), (i_2, j_2) при B-правильном покрытии через $c(i_1, j_1, i_2, j_2)$.

Примером задачи ограниченного домино назовем четверку $BT = \langle n, m, Tiles, B\rangle$. Пример будем называть *допустимым*, если существует B-правильное покрытие прямоугольной области $Area$ размера $n \times m$ экземплярами домино из множества $Tiles$. Задача ограниченного домино тогда заключается в том, чтобы для заданного произвольного примера задачи ответить на вопрос, является ли он допустимым.

В такой постановке задача домино является разрешимой, но, как было показано в работе [54], это NP-полная задача. Тем не менее, существуют постановки задачи домино, которые являются алгоритмически неразрешимыми. Так, например, в работе [15] было показано, что в случае замены прямоугольной области плоскостью и даже квадрантом плоскости задача становится неразрешимой.

В следующем разделе показано, каким образом можно использовать NP-полноту задачи ограниченного домино для обоснования NP-трудности задачи двусторонней унификации подстановок.

4.3. Обоснование NP-полноты задачи двусторонней унификации подстановок

Для обоснования свойств двусторонней унификации подстановок опишем сведение задачи ограниченного домино к задаче проверки двусторонней унифицируемости подстановок.

Рассмотрим множества элементов домино $Tiles = \{tile_1, tile_2, \ldots, tile_L\}$ и цветов $Colours = \{1, 2, \ldots, K\}$. Пусть на этих множествах задан произвольный пример задачи ограниченного домино $BT = \langle n, m, Tiles, B\rangle$. Опишем процесс построения для заданного примера BT такой пары подстановок (θ_1, θ_2), которая является двусторонне унифицируемой с унификатором (η', η'') в том и только том случае, когда пример BT является допустимым. Идея сведения заключается в следующем: каждой из подстановок будет подставлена в соответствие какая-то часть задачи BT. При этом связки подстановки θ_1 будут описывать перечень элементов домино и заданные граничные условия, а связки подстановки θ_2 — идеальную монохромную окраску всех сторон всех квадратов области в цвет K. Связки подстановки η' будут соответствовать некоторому покрытию исходной области, а подстановка η'' будет использоваться для модификации имеющейся окраски следующим образом: у смежных сторон, имеющих одинаковую окраску, эта подстановка будет увеличивать номер поставленного этой паре в соответствие цвета. Таким образом, в случае, если исходный пример допустим, то в результате такого перекрашивания должна получиться монохромно окрашенная в цвет K область.

Определим семантику каждой из подстановок формально. Для этого необходимо ввести специальным образом множества X, Y, Z и U переменных и множество F функциональных символов.

Множества переменных, на которых будут определены подстановки θ_1 и θ_2, определяются следующим образом.

- Множество переменных X: $X = X' \cup X''$, где
  - X' — множество переменных $x'_{i,j}$, соответствующих квадратам (i,j) границы области $Area$;
  - X'' — множество переменных $x''_{i,j,l}$, каждая из которых ассоциирована с размещением одного экземпляра домино $tile_l$ во внутреннем квадрате (i,j) области $Area$.

  Формально это можно записать так: $X' = \{x'_{i,j} : (i,j) \in Border\}$, $X'' = \{x''_{i,j,l} : (i,j) \in Interior, 1 \leq l \leq L\}$.

- Множество переменных Y: $Y = \{y_0\} \cup Y'$, где y_0 — вспомогательная переменная-“заглушка” , а множество Y' представляет собой множество переменных y_{i_1,j_1,i_2,j_2}, каждая из которых ассоциирована с парой соседних квадратов области $Area$: $Y' = \{y_{i_1,j_1,i_2,j_2} : 0 \leq i_1, i_2 \leq n+1, 0 \leq j_1, j_2 \leq m+1, |i_1 - i_2| + |j_1 - j_2| \leq 1\}$.

- Множество переменных $Z = \{z\}$.

- Множество переменных $U = \{u\}$.

Множество функциональных символов F состоит из трех элементов со следующей семантикой:

- двухместный функциональный символ $g^{(2)}$, необходимый для построения области $Area$;

- шестиместный символ $h^{(6)}$, с его помощью строятся термы, описывающие границу области $Border$ и термы, представляющие экземпляры домино;

- одноместный функциональный символ $f^{(1)}$, который позволяет строить *нумералы*. Под нумералами понимаются термы, глубина которых соответствует числу, обозначающему номер цвета некоторого квадрата области покрытия.

Все эти функциональные символы используются для построения связок каждой из подстановок θ_1 и θ_2. Начнем с построения связок для подстановки θ_1.

Покажем, каким именно образом функциональные символы из множества F позволяют осуществить намеченные цели. Для начала определим следующие рекурсивные правила построения нумералов f_n с помощью функционального символа $f^{(1)}$:

$$f_n(y) = \begin{cases} y \ , & n = 0, \\ f(f_{n-1}(y)) \ , & n > 0 \end{cases} \tag{4.1}$$

Для таким образом введенных нумералов и произвольного целого неотрицательного числа n верно соотношение: $f_n(y) = \underbrace{f(f(\ldots f(y)\ldots))}_{n}$. Для нумерала $f_n(y)$ число n будем называть *показателем* нумерала.

Опишем термы, которые представляют граничные условия и все возможные расстановки элементов домино из множества $Tiles$ во внутренних квадратах исходной области. Для этого понадобится три типа связок, отражающих три типа возможного взаимного расположения экземпляров домино.

1. Квадрат находится в одном из углов прямоугольной области $Area$. Для квадратов (i, j) таких, что $(i, j) \in \{(0, 0), (n+1, 0), (0, m+1), (n+1, m+1)\}$, в подстановке θ_1 вводится связка $x_{i,j}/t_{i,j}$, где

   $$t_{i,j} = h(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_K(y_0)).$$

   Трактовать значение этого терма следует следующим образом: все стороны углового квадрата окрашены в цвет K.

2. Квадрат не лежит ни в одном из углов прямоугольной области $Area$, но является граничным. Для каждого квадрата (i, j) такого, что $(i, j) \in Border \setminus \{(0, 0), (n+1, 0), (0, m+1), (n+1, m+1)\}$, окрашенного в цвет k,

$B(i,j) = k$, $1 \leq k \leq K$, существует единственный смежный с ним квадрат (i', j') из внутренней области $Interior$. Рассмотрим переменную $y_{i,j,i',j'}$ из множества Y', ассоциированную с парой квадратов (i,j), (i',j'). Тогда для квадрата (i,j) вводится связка $x_{i,j}/t_{i,j}$, где

$$t_{i,j} = h(f_i(y_0), f_j(y_0), f_K(y_0), f_K(y_0), f_K(y_0), f_k(y_{i,j,i',j'}))$$

Значение этого терма таково: каждая сторона граничного квадрата, не смежная ни с одним внутренним квадратом, имеет окраску K, в то время как сторона, смежная с некоторым внутренним квадратом, имеет окраску k.

3. Квадрат принадлежит области $Interior$. Каждый квадрат (i,j), лежащий в области $Interior$, имеет ровно четырех соседей — это квадраты, расположенные соответственно сверху, слева, снизу и справа от квадрата (i,j). Предположим, что в квадрат (i,j) помещен экземпляр домино $tile_l = \langle k_1, k_2, k_3, k_4 \rangle$. Рассмотрим четыре переменные множества Y', каждая из которых ассоциирована с одной из возможных пар смежных сторон, в которых принимает участие квадрат (i,j): y_{i,j,i_1,j_1} — пара смежных сторон с верхним соседом, y_{i,j,i_2,j_2} — с левым соседом, y_{i,j,i_3,j_3} — с нижним соседом, и y_{i,j,i_4,j_4} — пара смежных сторон с правым соседом квадрата (i,j). Тогда для каждого экземпляра домино $tile_l$ набора $Tiles$ введем в подстановку θ_1 связку $x_{i,j,l}/t_{i,j,l}$, где

$$t_{i,j,l} = h(f_i(y_0), f_j(y_0), f_{k_1}(y_{i,j,i_1,j_1}), f_{k_2}(y_{i,j,i_2,j_2}), f_{k_3}(y_{i,j,i_3,j_3}), f_{k_4}(y_{i,j,i_4,j_4})).$$

Этот терм вводится для того, чтобы обозначить возможность размещения в квадрате (i,j) элемента домино $tile_l$ в соответствии с его раскраской.

Введенные таким образом связки позволяют сформировать подстановку

θ_1 следующим образом:

$$\theta_1 = \{x_{i,j}/t_{i,j} : (i,j) \in Border\} \cup \{x_{i,j,l}/t_{i,j,l} : (i,j) \in Interior, 1 \leq l \leq L\}.$$

Как и было сказано выше, эта подстановка моделирует задачу ограниченного домино BT, для которой далее необходимо будет проверить, является ли она допустимой, то есть можно ли разместить плитки домино во внутренних квадратах области так, чтобы удовлетворить требованиям правильного покрытия.

Теперь покажем, каким образом строится подстановка θ_2. Напомним, что она призвана отобразить корректную монохромную окраску исходной прямоугольной области $Area$. Для этого введем вспомогательные термы, описывающие монохромную окраску каждого из квадратов области: для каждого квадрата (i,j), $0 \leq i \leq n+1$, $0 \leq j \leq m+1$, строится терм $\widehat{t}_{i,j} = h(f_i(u), f_j(u), f_K(u), f_K(u), f_K(u), f_K(u))$. Этот терм описывает ситуацию, когда все стороны квадрата (i,j) окрашены в цвет K. Рассмотрим терм $t_{Area}(\widehat{t}_{0,0}, \ldots, \widehat{t}_{n+1,m+1}) = g(\widehat{t}_{0,0}, g(\widehat{t}_{0,0}, g(\ldots, g(\widehat{t}_{n+1,m}, \widehat{t}_{n+1,m+1}))))$. С использованием этого терма можно ввести подстановку θ_2:

$$\theta_2 = \{z/t_{Area}(\widehat{t}_{0,0}, \widehat{t}_{0,1} \ldots, \widehat{t}_{n+1,m+1})\}$$

Эта подстановка моделирует ситуацию, когда вся область заполнена монохроматическими экземплярами домино цвета K.

Построенную таким образом пару подстановок (θ_1, θ_2) назовем *ассоциированной* с примером ограниченной задачи домино $BT = \langle n, m, Tiles, B\rangle$.

Справедлива следующая лемма.

Лемма 27. *Пример ограниченной задачи домино $BT = \langle n, m, Tiles, B\rangle$ допустим тогда и только тогда, когда ассоциированная с ним пара подстановок (θ_1, θ_2) двусторонне унифицируема.*

Доказательство.

($\Rightarrow$). Пусть пример ограниченной задачи домино $BT = \langle n, m, Tiles, B\rangle$ допустим, то есть существует B-правильное покрытие области $Area$ элементами $Tiles$. Пусть с данным примером ассоциирована пара подстановок (θ_1, θ_2). С учетом того, что запись $T(i, j)$ означает тип домино, расположенного в квадрате (i, j), рассмотрим подстановку η', устроенную следующим образом:

$$\eta' = \{z/t_{Area}(x_{0,0}, \ldots, x_{0,m+1}, x_{1,0}, x_{1,1,T(1,1)} \ldots, x_{1,m,T(1,m)}, x_{1,m+1}, \ldots, x_{n+1,m+1})\}.$$

Подстановка η' использует терм t_{Area}, подставляя на место каждого аргумента этого терма для квадрата (i, j) либо переменную $x_{i,j}$, если $(i, j) \in Border$, либо переменную $x_{i,j,T(i,j)}$, если $(i, j) \in Interior$. Теперь для того, чтобы построить подстановку η'', необходимо унифицировать имеющуюся подстановку θ_2 и композицию $\eta'\theta_1$. Из описания правил построения нумералов f_i следует, что во всех связках подстановки θ_1 все вхождения переменных y_{i_1,j_1,i_2,j_2} как в термы $t_{i_1,j_1,T(i_1,j_1)}$, так и в термы $t_{i_2,j_2,T(i_2,j_2)}$ имеют одинаковую глубину $c(i_1, j_1, i_2, j_2)$. Рассмотрим подстановку η'', устроенную следующим образом:

$$\eta'' = \{y_0/u, y_{0,1,1,1}/f_{K-c(0,1,1,1)}(u), \ldots, y_{i_1,j_1,i_2,j_2}/f_{K-c(i_1,j_1,i_2,j_2)}, \ldots\}.$$

Эта подстановка вместо каждой переменной, ассоциированной с парой смежных сторон квадратов (i_1, j_1), (i_2, j_2) области $Area$, подставляет нумерал, дополняющий исходный цвет пары сторон $c(i_1, j_1, i_2, j_2)$ до максимального цвета K. Фактически, данная подстановка осуществляет синхронное перекрашивание пар смежных сторон. Принимая во внимание, что исходное покрытие является B-правильным, подобное синхронное перекрашивание приводит к построению монохромно окрашенной области, которую как раз и описывает подстановка θ_2.

($\Leftarrow$). Пусть существует пара подстановок (η', η''), таких, что $\eta'\theta_1\eta'' = \theta_2$. Для того, чтобы выяснить состав подстановок η', η'', рассмотрим устройство подстановок θ_1 и θ_2. Каждая нелистовая вершина каждого размеченного дерева

леса, представляющего геометрическую реализацию подстановки θ_1, помечена одним из символов $f^{(1)}$ или $h^{(6)}$, причем в каждом пути единственный функциональный символ $h^{(6)}$ предшествует функциональным символам $f^{(1)}$. В то же время, размеченные деревья, представляющие подстановку θ_2, по каждому пути от корня к листу содержат сначала вершины, помеченные функциональным символом $g^{(2)}$, затем единственную вершину, помеченную символом $h^{(6)}$, затем вершины, помеченные символом $f^{(1)}$, после чего по каждой ветви следует листовая вершина. Кроме того, каждое дерево терма подстановки θ_1 для каждого квадрата (i, j) области $Area$ содержит единственное поддерево, начинающееся с функционального символа $h^{(6)}$, и такое, что его терм имеет вид $h(f_i(y_0), f_j(y_0), \ldots)$. Из приведенных фактов следует, что подстановка η' имеет следующий вид:

$$\eta' = \{z/t_{Area}((x_{0,0}, \ldots, x_{0,m+1}, x_{1,0}, x_{1,1,T(1,1)} \ldots, x_{1,m,T(1,m)}, x_{1,m+1}, \ldots, x_{n+1,m+1}))\},$$

а η'' определяется так:

$$\eta'' = \{y_0/u, y_{0,1,1,1}/f_{K-c(0,1,1,1)}(u), \ldots, y_{i,j,i',j'}/f_{K-c(i,j,i',j')}(u), \ldots\}.$$

Рассмотрим покрытие T области $Area$, в котором для каждого квадрата (i, j) выполнено равенство $T(i, j) = l_{i,j}$ тогда и только тогда, когда в некотором терме подстановке η' содержится переменная $x_{i,j,l_{i,j}}$. Покажем, что такое покрытие является B-правильным.

Предположим, что это не так. Это означает, что существуют два соседних квадрата (i, j), (i', j') таких, что окраска сторон домино, примыкающих к их общей границе, разная, то есть выполнено одно из неравенств: $T(i, j)[i - i', j - j'] \neq T(i', j')[i' - i, j' - j]$, если $(i, j), (i', j') \in Interior$, или $T(i, j)[i - i', j - j'] \neq B(i', j')$, если $(i, j) \in Interior$, а $(i', j') \in Border$.

Рассмотрим сначала первый вариант. Не ограничивая общности, будем

считать, что $i \leq i'$, $j \leq j'$. Поскольку глубина вхождения переменной $y_{i,j,i',j'}$ в термы $t_{i,j,T(i,j)}$ и $t_{i',j',T(i',j')}$ определяется показателем нумерала f_n, входящего в состав терма, а показатель нумерала в свою очередь является номером цвета соответствующей стороны домино, то глубина вхождения этой переменной в термы $t_{i,j,T(i,j)}$ и $t_{i',j',T(i',j')}$ будет разной. Но из устройства подстановок η' и θ_1 следует, что в разных связках композиции $\eta'\theta_1$ вхождения переменной $y_{i,j,i',j'}$ также будут иметь разную глубину. Рассмотрим композицию $\eta'\theta_1\eta''$. В ней нумералы, соответствующие подтермам $t_{i,j,l_{i,j}}$ и $t_{i',j',l_{i',j'}}$ будут разными. Но терм подстановки θ_2 моделирует монохромную окраску, то есть показатели всех нумералов, связанных с окраской домино, должны быть в нем одинаковы. Получаем противоречие с тем, что $\theta_2 = \eta'\theta_1\eta''$.

Второй вариант, когда отличается цвет соседних сторон домино в квадратах, один из которых расположен на границе, опровергается аналогично.

Следовательно, предложенное покрытие является B-правильным. □

Лемма 28. *Проблема ограниченного домино $log-space$-сводима к задаче двусторонней унификации программ.*

Доказательство. Рассмотрим описанный в этом разделе процесс построения пары подстановок (θ_1, θ_2), ассоциированных с примером задачи ограниченного домино $BT = (n, m, Tiles, B)$. Этот процесс является детерминированным алгоритмом, который строит термы $t_{i,j}$, $\widehat{t}_{i,j}$, $t_{i,j,l_{i,j}}$ и $\widehat{t}_{i,j,l_{i,j}}$. Для построения древесного представление каждого из таких термов алгоритму необходим объем рабочей памяти размера $O(logN)$, где N — размер исходной задачи. Таким образом, пара подстановок (θ_1, θ_2) требует объема рабочей памяти, пропорционального логарифму размера описания примера задачи ограниченного домино. □

Таким образом, из лемм 27 и 28, а также из результатов, полученных в работе [54], следует справедливость следующей теоремы.

Теорема 17. *Задача двусторонней унификации подстановок является NP-полной.*

4.4. Задача проверки двусторонней унифицируемости программ

Задача проверки двусторонней унифицируемости программ фактически является обобщением задачи двусторонней унификации подстановок. Рассмотрим эту задачу для введенной в первой главе модели последовательных императивных программ и для введенного над этой моделью отношения логико-термальной эквивалентности программ. Для этого необходимо описать операцию применения подстановок к программе. Отметим, что описание этой операции хоть и повторяет частично описание применения подстановки ко входам программы, описанное в предыдущей главе, но является более широким: в данном случае подстановки применяются не только ко входным переменным программы, но и к переменным, поданным на выход.

Пусть задана программа $\pi = \langle X, Y, V, v_{in}, v_{out}, B, \to, \lambda_0 \rangle$. Будем считать, что множество выходных переменных этой программы совпадает со множеством внутренних переменных Y, что не противоречит определению программы. При этом обозначим $input_\pi$ множество входных переменных программы, то есть $input_\pi = X$, также обозначим $output_\pi$ множество выходных переменных программы, то есть $output_\pi = Y$. Пусть также задана пара подстановок (η', η'') таких, что $\eta' \in Subst(Z, X, F)$, $\eta'' \in Subst(Y, U, F)$, где $U = \{u_1, u_2, \ldots, u_k\}$, а множество $Z = \{z_1, z_2, \ldots, z_t\}$. *Результатом применения пары подстановок* (η', η'') *к программе* π назовем программу $\eta'; \pi; \eta''$, полученную из программы π путем следующих преобразований:

- множество входных переменных программы π заменяется множеством Z: $input_{\eta';\pi;\eta''} = Z$;
- подстановка θ_0, приписанная дуге в программе π, исходящей из входной вершины, заменяется подстановкой $\theta_0\eta'$;
- множество выходных переменных программы π заменяется множеством

U: $output_{\eta';\pi;eta''} = U$;

- для каждой дуги, ведущей в выходную вершину программы π, подстановка θ, приписанная этой дуге, заменяется подстановкой $\eta''\theta$.

Графически применение пары подстановок к программе изображено на рисунке 4.4.

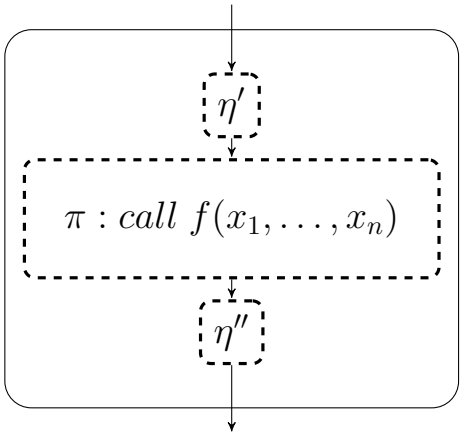


Рис. 4.4. Задача двусторонней унификации программ: программа $\eta'; \pi; \eta''$.

Применение пары подстановок (η', η'') к программе π можно трактовать как вызов процедуры с телом π при условии предварительной инициализации переменных подстановкой η' и последующей специализации выходных значений подстановкой η''.

Задачей двусторонней унификации программ π_1 *и* π_2 назовем задачу поиска такой пары подстановок η' и η'', что программа $\eta'; \pi_1; \eta''$ оказывается логико-термально эквивалентной программе π_2. При этом пару (η', η'') будем называть двусторонним унификатором программ π_1 и π_2. Для двустороннего унификатора справедливо следующее утверждение.

Лемма 29. *Пусть* (η', η'') — *двусторонний унификатор программ* π_1 *и* π_2. *Рассмотрим произвольную трассу программы* π_1, *соответствующую вектору* $\widetilde{\delta} = \{\delta_0, \delta_1, \ldots, \delta_n\}$:

$$tr = v_{in} \xrightarrow{\delta_0,\theta_0} v_1 \xrightarrow{\delta_1,\theta_1} \ldots \xrightarrow{\delta_n,\theta_n} v_{out}.$$

Тогда в программе π_2 *существует трасса*

$$tr' = v'_{in} \xrightarrow{\delta_0,\mu_0} v'_1 \xrightarrow{\delta_1,\mu_1} \dots \xrightarrow{\delta_n,\mu_n} v'_{out},$$

соответствующая тому же самому вектору $\widetilde{\delta} = \{\delta_0, \delta_1, \dots, \delta_n\}$ *и удовлетворяющая равенству:* $\eta''\theta_n\theta_{n-1}\dots\theta_0\eta' = \mu_n\mu_{n-1}\dots\mu_0$.

Доказательство. По определению логико-термальной эквивалентности, из эквивалентности программ $\eta'; \pi_1\eta''$ и π_2 следует, что совпадают их детерминанты: $det(\eta'; \pi_1; \eta'') = det(\pi_2)$. Из совпадения детерминантов и теоремы 2 следует, что если детерминант программы $\eta'; \pi_1; \eta''$ содержит историю $lth(tr)$, то в программе π_2 существует некоторая трасса tr' такая, что их логико-термальные истории совпадают, то есть $lth(tr) = lth(tr')$. Из этого следует, что трасса tr' соответствует тому же самому логическому вектору $\widetilde{\delta}$, что и трасса tr, и при этом должны совпадать последние пары этих двух последовательностей, из чего и следует утверждение теоремы. □

Лемма 29 позволяет использовать результаты, полученные в предыдущем разделе, для изучения сложности задачи двусторонней унификации программ.

Теорема 18. *Задача двусторонней унификации программ является* NP*-полной.*

Доказательство. Из теоремы 17 следует NP-трудность данной задачи, так как задача двусторонней унификации подстановок по лемме 29 является частным случаем задачи двусторонней унификации программ. Рассмотрим вопрос о принадлежности данной задачи классу NP. Для этого предложим следующий недетерминированный алгоритм, разрешающий эту задачу за полиномиальное время. Выберем в программе π_1 кратчайшую трассу

$$tr = v_{in} \xrightarrow{\delta_0,\theta_0} v_1 \xrightarrow{\delta_1,\theta_1} \dots \xrightarrow{\delta_n,\theta_n} v_{out}.$$

Вычислим подстановку θ такую, что $\theta = \theta_n\theta_{n-1}\dots\theta_0$. Вычисление этой подстановки осуществляется за линейное относительно размера программы π_1 время. Согласно лемме 29 в программе π_2 можно единственным способом выбрать трассу, согласованную с трассой tr в программе π_1,

$$tr' = v'_{in} \xrightarrow{\delta_0,\mu_0} v'_1 \xrightarrow{\delta_1,\mu_1} \dots \xrightarrow{\delta_n,\mu_n} v'_{out}.$$

Для трассы tr' также вычислим подстановку $\mu = \mu_n\mu_{n-1}\dots\mu_0$, это вычисление также требует времени, полиномиального относительно размера программы π_2. Теперь, имея пару подстановок θ и μ, вспомним, что по лемме 29 двусторонний унификатор пары исходных программ должен также являться двусторонним унификатором заданной пары подстановок. При этом необходимо учитывать, что может существовать не единственный двусторонний унификатор пары подстановок, но их обязательно конечное число. Недетерминированным образом выберем произвольный из них — построим пару (η', η''). Согласно лемме 26 такое вычисление осуществляется недетерминированным алгоритмом за полиномиальное время. После этого необходимо проверить, являются ли программы $\eta'; \pi_1; \eta''$ логико-термально эквивалентными. Но по теореме 11 такая проверка также осуществляется за время, полиномиальное относительно размеров исходных программ. Таким образом, поиск двустороннего унификатора заданной пары программ требует полиномиального времени. □

Полученный результат подтверждает возможность применения подхода, изучаемого в данной работе, в современных средствах автоматического рефакторинга для выделения фрагментов программ и замены их вызовами некоторой процедуры. При этом фрагментом программы для выделения в контексте обозначений этого раздела является программа π_2, фрагментом кода, соответствующим вызываемой функции, является программа π_1, а вспомогательная инициализация переменных осуществляется простыми линейными фрагментами последовательных программ, соответствующими подстановкам η' и η''.

Заключение

В рамках данной диссертационной работы было проведено исследование, позволившее получить следующие результаты.

1. Разработан полиномиальный алгоритм проверки логико-термальной эквивалентности последовательных императивных программ, использующий метод совместных вычислений и свойства решетки подстановок. Для данного алгоритма строго обоснована его корректность. Доказано, что сложность этого алгоритма составляет $O(n^6)$, где n — суммарный размер исходных программ.

2. Разработан полиномиальный алгоритм проверки логико-термальной унифицируемости последовательных императивных программ, который позволяет строить унифицирующие исходные программы подстановки. Для этого алгоритм приведено строгое обоснование его корректности. С использованием этого алгоритма показано, что задача проверки логико-термальной унифицируемости может быть решена за время $O(n^{11})$, где n — суммарный размер исходных программ.

3. Для задачи двусторонней унификации подстановок показана ее NP-полнота с помощью сведения к ней задачи ограниченного домино, а также разработан недетерминированный алгоритм решения этой задачи, работающий за полиномиальное время. На основании этого результата показана NP-полнота задачи двусторонней унификации программ, для которой также построен недетерминированный алгоритм, строящий двусторонний унификатор программ за полиномиальное относительно их суммарных размеров время. Корректность всех указанных алгоритмов обоснована.

Список литературы

1. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: principles, techniques, and tools. Second edition. — Addison-Wesley, 2007.
2. Alpuente M., Escobar S., Espert J., Meseguer J. A modular order-sorted equational generalization algorithm // Information and Computation, 235(0). — 2014. — p. 98–136.
3. Ashcroft E., Manna Z., Pnueli A. Decidable properties of monadic functional schemes // Journal of the ACM. — 1973. — Vol. 20, N 3. — P. 489–499.
4. Baader F. Unification, weak unification, upper bound, lower bound, and generalization problems // Ronald V. Book, editor, RTA, Springer. — v. 488 of Lecture Notes in Computer Science. — 1991. — p. 86–97.
5. Baader F., Snyder W. Unification theory // J. A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. — 2001. — v. 1. — p.447–533.
6. Baker B. On finding duplication and near-duplication in large software systems // Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995. — 1995 — p. 86-–95.
7. De Bakker J. W., Scott D. A. Theory of programs. Unpublished notes. — Vienna: IBM Seminar. — 1969.
8. Balazinska M., Merlo E., Dagenais M., Lague B., Kontogiannis K. Measuring clone based reengineering opportunities // Proceedings of the IEEE Symposium on Software Metrics, METRICS 1999. — 1999. — pp. 292–303
9. Bannwart F., Müller P. Changing Programs Correctly: Refactoring with Specifications // Proceedings of the 14th International Symposium on Formal Methods. — 2006. — P. 492–507
10. Baker B. A program for identifying duplicated code // Proceedings of Computing Science and Statistics: 24th Symposium on the Interface. — vol. 24. — 1992. — pp. 49–57.
11. Baker B. On finding duplication and near-duplication in large software systems

// Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995. — 1995. — pp. 86–95.

12. Baumgartner A. Anti-Unification Algorithms: Design, Analysis, and Implementation. Technical report no. 15-17 in RISC Report Series, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria. September 2015. PhD Thesis.
13. Baxter L.D. An efficient unification algorithm // Technical Report CS-73-23, Dep. of Analysis and Comp. Sci., University of Waterloo, Ontario, Canada, 1973.
14. Baxter I, Yahin A., Moura L., Anna M. Clone detection using abstract syntax trees // Proceedings of the 14th International Conference on Software Maintenance, ICSM1998. — 1998. — p. 368-397.
15. Berger R. The undecidability of domino problem // Memoirs of American Mathematical Society. — v. 66. — 1966. — p. 1–72.
16. Bird R. The equivalence problem for deterministic two-tape automata // J. of Computer and System Science. — 1973. — v. 7, N 4. — p. 218–236.
17. Bulychev P., Kostylev E., Zakharov V. Anti-unification algorithms and their applications in program analysis // Proceedings of the 7th International Conference “Perspectives of System Informatics”, June 15–19, 2009, Novosibirsk. — 2009. — p. 82–89.
18. Bulychev P., Minea M., Duplicate code detection using anti-unification // Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008. — 2008. — p. 51–54.
19. Burghardt J. E-generalization using grammars // Artif. Intell. — v. 165(1). — 2005. — p. 1–35.
20. Christodorescu M., Jha S. Static Analysis of Executables to Detect Malicious Patterns // Proceedings of the 12th USENIX Security Symposium. — 2003. — P. 169–186.
21. Christodorescu M., Jha S., Seshia S., Song D., Bryant R. E. Semantics-Aware

Malware Detection // Proceedings of the 2005 IEEE Symposium on Security and Privacy. — 2005. — P. 32–46.

22. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, 1997.
23. Collberg C., Thomborson C., Low D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs // Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — 1998. — P. 184–196.
24. Delcher A. L., Kasif S. Efficient parallel term matching and anti-unification. Journal of Automated Reasoning. — v. 9, N 3. — 1992. — pp. 391–406.
25. Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. — v. 1. — 1985. — p. 31–46.
26. Falke R., Koschke R, Frenzel P. Empirical evaluation of clone detection using syntax suffix trees // Empirical Software Engineering. — 2008. — Vol. 13. — pp. 601–643.
27. Fowler M., Beck K., Brant J., Opdyke W. Refactoring: Improving the Design of Existing Code. — Addison-Wesley, 1999.
28. Friedman E. P. Equivalence problems for deterministic languages and monadic recursion schemes // Journal of Computer and System Sciences. — 1977. — Vol. 14, N 3. — P. 342–359.
29. Gabel M., Jiang L., Su Z. Scalable detection of semantic clones // Proceedings of the 30th International Conference on Software Engineering, ICSE 2008. — 2008. — pp. 321–330.
30. Garland S. J., Luckham D. C. Program schemes, recursion schemes and formal languages // Journal of Computer and System Sciences. — 1973. — Vol. 7, N 2. P. 119–160.
31. Goldfarb W. D.The undecidability of the second-order unification problem // Theoretical Computer Science. — 1981. — v. 13, N 2. — p. 225–230.

32. Goldwasser S., Rothblum G. N. On best-possible obfuscation // Journal of Cryptology. — 2014. — Vol. 27, N 3. — P. 480–505.
33. Hagiya M. Generalization from partial parametrization in higher-order type theory // Theor. Comput. Sci. — v. 63(2) — 1989. — p. 113–139.
34. Hamid A., Zaytsev V. Detecting Refactorable Clones by Slicing Program Dependence Graphs // Post-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution. —2015. — p. 37–48.
35. Herold A., Sieckmann J. Unification in Abelean semigroups // Journal of Automated Reasoning. — 1983. — p. 247–283.
36. Hopcroft J. E., Motwani R., Ullman J. D. Introduction to automata theory, languages, and computation — international edition (2. ed). — Addison-Wesley. — 2003.
37. Hotta K., Higo Y., Kusumoto S. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph // Proceedings of the 16th European Conference on Software Maintenance and Reengineering. — 2012. — pp. 53–62.
38. Itkin V.E., Zwinogrodski Z. On program schemata equivalence // Journal of Computer and System Science. — 1972. — v. 6, N 1. — p. 88–101.
39. Komondoor R., Horwitz S. Tool Demonstration: Finding duplicated code using program dependences // Proceedings of ESOP 2001: European Symposium on Programming, Genoa, Italy, April 2-6, 2001.
40. Komondoor R., Horwitz S. Eliminating Duplication in Source Code via Procedure Extraction // UW-Madison Dept. of Computer Sciences Technical Report 1461. — 2002.
41. Komondoor R., Horwitz S. Effective, Automatic Procedure Extraction // Proceedings 11th IEEE International Workshop on Program Comprehension, Portland, Oregon. — 2003.
42. Knight K. Unification: a multidisciplinary survey // ACM Computing Surveys. — 1989. — v. 21, N 1 — p. 93–124.

43. Komondoor R., Horwitz S. Using slicing to identify duplication in source code // Proceedings of the 8th International Symposium on Static Analysis, SAS 2001. — 2001. — pp. 40–56.
44. Kontogiannis K., DeMori R., Merlo E., Galler M., Bernstein M. Pattern matching for clone and concept detection // Journal of Automated Software Engineering. — 3 (1–2). — 1996. — pp. 77–108.
45. Koschke R., Falke R., Frenzel P. Clone detection using abstract syntax suffix trees // Proceedings of the 13th Working Conference on Reverse Engineering, WCRE 2006. — 2006. — pp. 253–262.
46. Krinke J. Identifying similar code with program dependence graphs // Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001. — 2001. — pp. 301–309.
47. Krishnan G. P., Tsantalis N. Unification and Refactoring of Clones // Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week. — 2014. — pp. 104–113.
48. Krumnack U., Schwering A., Gust H., Kuhnberger K. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, Australian Conference on Artificial Intelligence, Springer. — v. 4830 of Lecture Notes in Computer Science. — 2007. — p. 273–282.
49. Kundu S., Tatlock Z., Lerner S. Proving Optimizations Correct Using Parameterized Program Equivalence // Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2009. — P. 327–337.
50. Kuper G. M., McAloon K. W., Palem K. V., Perry K. J. A note on the parallel complexity of anti-unification. Journal of Automated Reasoning. — v. 9, N 3. — 1992. — pp. 381–389.
51. Lague B., Proulx D., Mayrand J et al. Assessing the benefits of incorporating function clone detection in a development process // Proceedings of the

International Conference on Software Maintenance. — Washington, DC, USA: IEEE Computer Society. — 1997. — p. 314–321.

52. Lassez J.I., Maher M.J., Marriot K. Unification revisited// Foundations of Deductive Databases and Logic Programming. — 1988.
53. Lerner S., Millstein T. D., Chambers C. Automatically proving the correctness of compiler optimizations // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. — 2003. — P. 220–231.
54. Lewis C.H. Complexity of solvable cases of the decision problem for predicate calculus // Proceedings of the 19-th Annual Symposium on Foundations of Computer Science. — 1978. — p. 35–47.
55. Lewis H. R., Papadimitriou C. H. Elements of the Theory of Computation — Prentice Hall, Englewood Cliffs. — 1981.
56. Lincoln P., Christian J. Adventures in associative-commutative unification // Journal of Symbolic Computation. — 1989. — v. 8. — p. 393–416.
57. Liu C., Chen C., Han J., Yu P. GPLAG: Detection of software plagiarism by program dependence graph analysis // Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006. — 2006. — pp. 872–881.
58. Luckham D.C., Park D.M., Paterson M.S. On formalized computer programs // Journal of Computer and System Science. — 1970. — v.4, N 3. — p.220–249.
59. Martelli A., Montanari U. An Effective Unification Algorithm // Transactions on Programming Langages and Systems (TOPLAS). — 1982. — v. 4., N. 2. — p. 258–282.
60. Mayrand J., Leblanc C., Merlo E. Experiment on the automatic detection of function clones in a software system using metrics // Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996. — 1996. — pp. 244–253.
61. Mendel-Gleason J. Types and Verification for Infinite State Systems. PhD thesis, Dublin City University. — 2012.

62. Mitsch S., Quesel J.-D., Platzer A. Refactoring, Refinement, and Reasoning - A Logical Characterization for Hybrid Systems // Proceedings of the 19th International Symposium on Formal Methods. — 2014. — P. 481–496.
63. Oancea C.E., So C., Watt S.M. Generalization in Maple // *Maple Conference*. — 2005. — p. 277–382.
64. Ottenstein K., Ottenstein L.. The program dependence graph in a software development environment // Proceedings of ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments. — 1984. — p. 177–184.
65. Palamidessi C. Algebraic properties of idempotent substitutions // *Lecture Notes in Computer Science*. — v. 443. — 1990. — p. 386–399.
66. Patenaude J., Merlo E., Dagenais M., Lague M. Extending software quality assessment techniques to java systems // Proceedings of the 7th International Workshop on Program Comprehension, IWPC 1999. — 1999. — pp. 49–56.
67. Paterson M. S. Program schemata // Machine Intelligence. — 1968. — Vol. 3. — P. 19–31.
68. Paterson M. S., Hewitt C. T. Comparative Schematology // Proceedings of the ACM Conference on Concurrent Systems and Parallel Computation. — 1970. — P. 119–127.
69. Paterson M.S., Wegman M.N. Linear unification // The Journal of Computer and System Science. — 1983. — v. 16, N 2 —p. 158–167
70. Plotkin G.D. A note on inductive generalization. *Machine Intelligence*. — 1970. — v. 5, N 1. — p.153–163.
71. Rahli V., Bickford M., Anand A. Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types // Proceedings of the 4th Conference on Interactive Theorem Proving. — 2013. — P. 261–278.
72. Reynolds J.C. Transformational systems and the algebraic structure of atomic formulas // Machine Intelligence. — 1970. — v. 5, N 1 — p. 135—-151.
73. Rice H. G. Classes of Recursively Enumerable Sets and Their Decision

Problems // Transactions of the American Mathematical Society. — 1953. — Vol. 74. — P. 358–366.

74. Robinson J. A.A machine-oriented logic based on the resolution principle // Journal of the ACM. — 1965. — v. 12, N 1. — p. 23–41.
75. Robinson R. Undecidability and Nonperiodicity for Tilings of the Plane // Inventiones Mathematicae. — v. 12. — 1971. — p. 177–209.
76. Roy C. K. Detection and analysis of near-miss software clones. A thesis submitted to the School of Computing in conformity with the requirements for the degree of Doctor of Philosophy. Canada. — 2009.
77. Roy C. K., Cordy J. R. A survey on software clone detection research // Technical report TR 2007-541, School of Computing, Queen's University. — 2007. — v. 115.
78. Roy C. K., Cordy J. R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization // Proceedings of the 16th IEEE International Conference on Program Comprehension. — 2008. — p. 172–181.
79. Roy C. K., Cordy J. R. An empirical study of function clones in open source software systems // Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008. — 2008. — p. 81-–90.
80. Roy C.K., Zibran M.F., Koschke R. The vision of software clone management: past, present and future // Proc. CSMR- 18/WCRE-21 - SEW'14, Belgium. — 2014. — pp. 16
81. Schäfer M., Ekman T., de Moor O. Challenge proposal: verification of refactorings // Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification. — 2009. — P. 67–72.
82. Smith R., Horwitz S., Detecting and Measuring Similarity in Code Clones // Proceedings of the 3rd International Workshop on Software Clones (IWSC'2009) — 2009.
83. Sorensen M. H., Gluck R. An algorithm of generalization in positive

supercompilation // Proceedings of the 1995 International Symposium on Logic Programming. — MIT Press:1995. — p. 465–479.

84. Speicher D., Bremm A. Clone removal in java programs as a process of stepwise unification // Proceedings of the 26th Workshop on Logic Programming — 2012.
85. Stickel E.M. A unification algorithm for associative-commutative functions // Journal of the association for Computing Machinary. — 1981. — v. 28, N 5 — p.423–434.
86. Strong H. R. Translating recursive equations into flow-charts // Journal of Computer and System Science. — 1971. — Vol. 5, N 3. — P. 254–285.
87. de Souza R. On the Decidability of the Equivalence for k-Valued Transducers // Proceedings of the 12th International Conference on Developments in Language Theory. — 2008. — P. 252–263.
88. Turchin V. Metacomputation: Metasystem transitions plus supercompilation // Partial Evaluation. — Springer Berlin/Heidelberg. — 1996. — p.481–509.
89. Turchin V. F.. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. — 1986. — V. 8. — p. 292–325.
90. Wahler V., Seipel D., Gudenberg J., Fischer G. Clone detection in source code by frequent itemset techniques // Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, SCAM 2004. — 2004. — pp. 128–135.
91. Wang Hao. Proving theorems by pattern recognition // Bell System Technical Journal. — 1961. — v. 40, N 1. — p. 1-41.
92. Watt S.M. Algebraic generalization. *ACM SIGSAM Bulletin.* — v. 39 — N 3. — 2005. — p. 93–94.
93. Webster M., Malcolm G. Detection of metamorphic and virtualization-based malware using algebraic specification // Journal in Computer Virology. — 2009. — Vol. 5, N 3. P. 221–245.
94. Weiser M. Program slicing. IEEE Trans. on Software Eng. — v. 10, N 4. — 1984. — p. 352—357.

95. Wong W., Stamp M. Hunting for metamorphic engines // Journal in Computer Virology. — 2006. — Vol. 2, N 3. P. 211–229.

96. Xia L., Tiantian W., Xiaohong S., Peijun M. Functionally equivalent clone detection using IOT-Behavior algorithm // Proceedings of the 1st International Conference on Artificial Intelligence and Software Engineering. — 2013. — p. 165–170.

97. Yang W. Identifying syntactic differences between two programs // Software Practice and Experience. — Vol. 7, N 21. — 1991. — pp. 739–755.

98. Yelick K. A. Generalized approach to equational unification // Technical Report, Massachusetts Institute of Technology Cambridge, MA, USA, 1990.

99. Yixin B., Gunes K., Xiaohong S., Peijun M. SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones // Journal of Systems and Software. — v. 86, N 8. — 2013. — p. 2077–2093.

100. Young P. Optimization Among Provably Equivalent Programs // Journal of the ACM. — 1977. — Vol. 2, N 4. P. 93–700.

101. Zakharov V. On the decidability of the equivalence problem for orthogonal sequential programs // Grammars. — 1999. — v. 2 — N 3. — p. 271–281.

102. Zakharov V. A. An efficient and unified approach to the decidability of equivalence of propositional program schemes // Lecture Notes in Computer Science. — 1998. — Vol. 1443. — P. 247–258.

103. Zakharov V. A. On the refinement of logic programs by means of anti-unification // Proceedings of the 2-nd Panhellenic Logic Symposium, Delphi, Greece. — 1999. — pp. 219–224.

104. Novikova T.A., Zakharov V.A. Is it possible to unify programs? // Proceedings of the 27-th International Workshop on Unification, Epic Series. — 2013. — v. 19 — p. 35–45.

105. Novikova T.A., Zakharov V.A. Two-sided unification is NP-complete. // Proceedings of the 28th International Workshop on Unification (UNIF-2014), RISC-Linz Report Series. — 2014. — JKU Linz, Vienna, Austria. — vol. 6. —

p. 55–61.

106. Буда А.О., Иткин В.Э. Сводимость эквивалентности программ к термальной эквивалентности // Системное и теоретическое программирование. Сборник статей. Т. 1. — Кишенев, 1974. — с. 293–324.

107. Глушков В. М. Теория автоматов и формальные преобразования микропрограмм // Кибернетика. — 1965. — № 5. — С. 1–9.

108. Глушков В. М., Летичевский А. А. Теория дискретных преобразователей // Избранные вопросы алгебры и логики. — 1973. — С. 5–39.

109. Ершов А. П. Об операторных схемах Янова // Проблемы кибернетики. — 1967. — Вып. 20. — С. 181—200.

110. Ершов А. П. Современное состояние теории схем программ // Проблемы кибернетики. — 1973. — Вып. 27. — С. 87–110.

111. Захаров В. А. Быстрые алгоритмы разрешения эквивалентности операторных программ на уравновешенных шкалах // Математические вопросы кибернетики. — 1998. — Вып. 7. — С. 303–324.

112. Захаров В. А. Быстрые алгоритмы разрешения эквивалентности пропозициональных операторных программ на упорядоченных полугрупповых шкалах // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. — 1999. — № 3. — С. 29–35.

113. Захаров В. А. Проверка эквивалентности программ при помощи двухленточных автоматов // Кибернетика и системный анализ. — 2010. — N 4. — С. 39–48.

114. Захаров В.А., Костылев Е.В. О сложности задачи антиунификации // Дискретная математика. — 2008. — т. 20, вып. 1. — с. 131–144.

115. Захаров В. А., Подымов В. В. Об одной полугрупповой модели программ, определяемой двухленточным автоматом // Научные ведомости Белгородского государственного университета. Серия История, экономика, политология, информатика. — 2010. — т. 14, N 7. — С. 94–101.

116. Захаров В. А., Подымов В. В. Об эквивалентности металинейных унарных

рекурсивных программ // Материалы XI Международного семинара “Дискретная математика и ее приложения”, посвященного 80-летию со дня рождения академика О. Б. Лупанова. — 2012. — С. 157–159.

117. Захаров В., Кузюрин Н., Подловченко Р., Щербина В. Использование алгебраических моделей программ для обнаружения метаморфного вредоносного кода // Фундаментальная и прикладная математика. — 2009. — Т. 15. — № 5. — С. 181–198.

118. Иткин В. Э. Логико-термальная эквивалентность схем программ // Кибернетика. — 1972. — N 1. — с. 5–27.

119. Котов В. Е., Сабельфельд В. К. Теория схем программ. — М.:Наука. — 1991. — 348 с.

120. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей I // Кибернетика. — 1969. — № 2. — С. 5–15.

121. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей II // Кибернетика. — 1970. — № 2. — С. 14–28.

122. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей III // Кибернетика. — 1972. — № 1. — С. 1–4.

123. Летичевский А. А. Эквивалентность автоматов относительно полугрупп с сокращением // Проблемы кибернетики. Вып.27. — М.:Наука. — 1973. — с. 195–212.

124. Лисовик Л. П. Метайлинейные схемы с засылками констант // Программирование. — 1985. — № 2. — с. 29–38.

125. Ляпунов А. А. О логических схемах программ // Проблемы кибернетики. — 1958. — Вып. 1. — С. 46–74.

126. Ляпунов А. А., Янов Ю. И. О логических схемах программ // Труды конференции "Пути развития советского математического машиностроения и приборостроения". — 1956. — часть 3. — с. 5–8.

127. Подловченко Р. И. Иерархия моделей программ // Программирование. — 1981. — № 2. — С. 3–14.

128. Подловченко Р. И. Полугрупповые модели программ // Программирование. — 1981. — № 4. — С. 3–13.

129. Подловченко Р. И. Рекурсивные программы и иерархия их моделей // Программирование. — 1991. — № 6. — С. 44–51.

130. Подловченко Р. И. Абстрактные программы с процедурами и конечные автоматы с магазином // Интеллектуальные системы. — 1997. — Т. 2, вып. 1–4. — С. 275–295.

131. Подловченко Р. И. От схем Янова к теории моделей программ // Математические вопросы кибернетики. — 1998. — Вып. 7. — С. 281–302.

132. Подловченко Р. И. О схемах программ с перестановочными и монотонными операторами // Программирование. — 2003. — № 5. — С. 46–54.

133. Подловченко Р. И. Алгебраические модели программ и автоматы // Математические вопросы кибернетики. — 2006. — Вып. 15. — С. 47–56.

134. Подловченко Р. Техника следов в разрешении проблемы эквивалентности в алгебраических моделях программ // Кибернетика и системный анализ. — 2009.

135. Подловченко Р. И. К вопросу о полиномиальной сложности проблемы эквивалентности в алгебраических моделях программ // Кибернетика и системный анализ. — 2012. — № 5. — С. 17–24.

136. Подловченко Р. И. Об одном классе алгебраических моделей программ, представляющем практический интерес // Программирование. — 2013. — № 3. — С. 15–28.

137. Подловченко Р. И., Долгих Б. А. Двухступенчатое моделирование программ с процедурами // Математические вопросы кибернетики. — 2004. — Вып. 12. — С. 47–56.

138. Подловченко Р. И., Захаров В. А. Полиномиальный по сложности алгоритм, распознающий коммутативную эквивалентность схем программ // Доклады РАН, серия Информатика. — 1998. — Т. 362, № 6. — С. 27–31.

139. Подловченко Р. И., Молчанов А. Э. Разрешимость эквивалентности в пере-

городчатых моделях программ // Моделирование и анализ информационных систем. — 2014. — Т. 21, № 2. — С. 56–70.

140. Подымов В. В. О проверке эквивалентности последовательных и рекурсивных программ на упорядоченных полугрупповых шкалах // Материалы X Международной конференции “Интеллектуальные системы и компьютерные науки”. — 2011. — С. 295–298.

141. Подымов В. В. Алгоритм проверки эквивалентности линейных унарных рекурсивных программ на упорядоченных полугрупповых шкалах // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. — 2012. — № 4. — С. 37–43.

142. Подымов В. В. О проверке сильной эквивалентности металинейных унарных рекурсивных программ // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. — 2013. — № 1. — С. 21–27.

143. Сабельфельд В. К. Полиномиальная оценка сложности распознавания логико-термальной эквивалентности // ДАН СССР. — 1979. — т.249, N 4. — с. 793–796.

144. Турчин В. Ф. Metacomputation: Metasystem Transitions + Supercompilation, http://www.refal.org/doc/turchin/dag/dag.html

145. Янов Ю. И. О логических схемах алгоритмов // Проблемы кибернетики. — 1958. — Вып. 1. — С. 75–127.

146. Захаров В. А., Новикова Т. А. Применение алгебры подстановок для унификации программ. // Труды Института системного программирования РАН. — 2011. — т. 21 — с. 141–166.

147. Новикова Т. А. Применение операции антиунификации подстановок для проверки логико-термальной эквивалентности программ // Математика и прикладная математика. Исследования студентов, магистрантов и выпускников. — Астана, 2011. — с. 205–228.

148. Захаров В. А., Новикова Т. А. Полиномиальный по времени алгоритм проверки логико-термальной эквивалентности программ. // Труды Института

системного программирования РАН. — 2012. — т. 22 — с. 435–455.

149. Захаров В. А., Новикова Т. А. Унификация программ. // Труды Института системного программирования РАН. — 2012. — т. 23 — с. 455–476.

150. Новикова Т. А. Использование редуцированной антиунификации подстановок для проверки логико-термальной эквивалентности фрагментов программ // Ломоносов-2012: Междунар. науч. конф. студентов, магистрантов и молодых ученых. г. Астана, 13-14 апр. 2012: тез.докл. — ч.I. — Астана, 2012.

151. Новикова Т. А. Об одном алгоритме полной унификации программ // Ломоносов-2013: Междунар. науч. конф. студентов, магистрантов и молодых ученых. г. Астана, 12-13 апр. 2013: тез.докл. — ч.I. — Астана, 2013. — с. 143–144.

152. Новикова Т. А. Об одном алгоритме унификации фрагментов программ // Наука молодых: сборник научных трудов выпускников Казахстанского филиала МГУ им. М. В. Ломоносова. — Астана, 2014. — с. 71–80.

153. Новикова Т. А., Захаров В. А. Двусторонняя унификация программ и ее применение для задач рефакторинга. // Труды Института системного программирования РАН. — 2014. — т. 26 — с. 245–268.

154. Новикова Т.А., Захаров В.А. О сложности задачи решения линейных уравнений над конечными подстановками. // Материалы XVII международной конференции «Проблемы теоретической кибернетики». — Отечество Казань, 2014. — с. 221–223.