На правах рукописи

Гречаник Сергей Александрович

Доказательство свойств функциональных программ методом насыщения равенствами

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

Автореферат
диссертации на соискание учёной степени кандидата физико-математических наук

Москва
2017

Работа выполнена в Федеральном государственном учреждении "Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша Российской академии наук"(ИПМ им. М.В. Келдыша РАН)

Научный руководитель:	Романенко Сергей Анатольевич, кандидат физико-математических наук, ведущий научный сотрудник отдела №12 “Инструментальное и прикладное программное обеспечение” института прикладной математики им. М.В. Келдыша РАН
Официальные оппоненты:	Захаров Владимир Анатольевич, доктор физико-математических наук, профессор кафедры математической кибернетики факультета ВМК МГУ имени М.В. Ломоносова, старший научный сотрудник отдела теоретической информатики ИСП РАН, ведущий научный сотрудник НИУ ВШЭ
	Бульонков Михаил Алексеевич, кандидат физико-математических наук, доцент, заместитель заведующего кафедрой программирования механико-математического факультета НГУ, заведующий лабораторией смешанных вычислений ИСИ СО РАН
Ведущая организация:	ФГБУН “Институт программных систем имени А.К. Айламазяна Российской академии наук”

Защита состоится 20 февраля 2018 г. в 11 часов на заседании диссертационного совета Д 002.024.01 в ИПМ им. М.В. Келдыша РАН по адресу: 125047, Москва, Миусская пл., 4.

С диссертацией можно ознакомиться в библиотеке ИПМ им. М.В.Келдыша РАН, `http://keldysh.ru`

Автореферат разослан «____» _____ 2017 г.

Учёный секретарь диссертационного совета
кандидат физико-математических наук А.Е. Бондарев

Общая характеристика работы

Объект исследования и актуальность работы

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

Среди функциональных языков программирования наилучшим образом для выявления и доказательства равенств подходят тотальные языки, поскольку в них выполняется больше равенств. В тотальных языках все функции тотальны, т.е. определены для всех возможных значений параметров (подходящих по типу), в частности невозможно зацикливание, а значит они не могут быть тьюринг-полными. Такие языки, однако, меньше используются на практике для обычного программирования. Среди оставшихся (нетотальных) функциональных языков можно выделить два класса: строгие языки и нестрогие языки. В строгих языках все определённые пользователем функции являются строгими, то есть значение выражения вызова функции $f(a, b, c)$ не определено (зацикливается), если значение хотя бы одного из аргументов a, b или c не определено. Операционный смысл строгости состоит в том, что значения аргументов вычисляются перед вызовом функции. Большинство языков являются строгими, однако такая семантика вызовов функций считается менее удобной для анализа и преобразования программ. В нестрогих же языках функции по умолчанию нестрогие. При вызове нестрогой функции даже если среди аргументов есть такой, что его вычисление зацикливается, значение всего вызова $f(a, b, c)$ может быть определено, если этот аргумент не используется (простейший пример — константная функция $f(x) = C$). Обычно это реализуется при помощи вычислений по необходимости (ленивых вычислений). Примером нестрогого языка является Haskell. Нестрогие языки считаются более удобными для анализа и преобразования, поэтому в данной диссертации в качестве входного языка был выбран нестрогий функциональный язык первого порядка.

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

- Верификация: многие желательные свойства функций над алгебраическими структурами данных можно сформулировать в таком виде. Широко известным примером являются законы, которые должны выпол-

няться для реализаций различных классов типов в языке Haskell, таких, как монады, аппликативные функторы и многие другие.

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

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

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

Среди методов решения данной задачи отметим такой класс методов, которые применимы не только непосредственно к задаче доказательства эквивалентности, но и к задаче преобразования программ, например, с целью оптимизации или анализа для выявления и доказательства свойств другого вида. Одним из таких методов является суперкомпиляция. Суперкомпиляция состоит в построении программы, эквивалентной исходной (иногда в некотором слабом смысле, например, на общей области определения) при помощи правил переписывания (прогонка, обобщение) в сочетании со свёрткой. Часто суперкомпиляция рассматривается как метод оптимизации, однако суперкомпиляцию можно использовать и для других задач, в частности для доказательства эквивалентности (например, для этого можно сравнить на синтаксическое совпадение результат суперкомпиляции левой и правой частей). Одно из направлений развития суперкомпиляции — многорезультатная суперкомпиляция, позволяющая строить не одну программу, а целое множество различных программ, эквивалентных исходной. Это позволяет использовать вместо эвристик, управляющих процессом переписывания, более точные метода выбора подходящей программы, работающие уже после суперкомпиляции и имеющие в своём распоряжении уже полностью построенные программы (например, в этом случае для выбора наиболее быстрой программы можно просто измерить производительность всех полученных программ).

Ещё один метод, применимый как к задаче доказательства эквивалентности, так и к задаче преобразования программ — насыщение равенствами (equality saturation). Насыщения равенствами — метод преобразования программ, заключающийся в применении преобразований к структуре данных, представляющей не одну программу, а целое множество семантически эквивалентных программ. Сам термин был введён в работе “Equality Saturation: A New Approach to Optimization” Тейта и других[1], где этот метод использовался

[1]Equality saturation: a new approach to optimization / R. Tate, M. Stepp, Z. Tatlock, S. Lerner // SIGPLAN Not. New York, NY, USA, 2009. Jan. Vol. 44, issue 1. Pp. 264–276. ISSN 0362-1340. URL: `http://doi.acm.org/10.1145/1594834.1480915`.

для преобразования императивных программ, хотя схожие идеи применялись и задолго до этого. В насыщении равенствами каждое преобразование применяется недеструктивно, то есть исходная программа остаётся во множестве, и ко множеству добавляется новая преобразованная программа. При этом потребление памяти уменьшается за счёт совмещения общих частей программ. В итоге строится множество всех возможных программ, полученных из исходной применением преобразований в различном порядке. После этого из построенного множества выделяется одна наилучшая программа, если целью было преобразование программы. Если же целью было доказательство эквивалентности, то построение одной программы не требуется, т.к. информацию об эквивалентности функций можно извлечь гораздо более простым способом.

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

Цели и задачи работы

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

Были поставлены следующие задачи:

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

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

Научная новизна работы

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

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

Кроме того, мы исключаем дублирование в полипрограмме функций, отличающихся только порядком аргументов, что делается впервые в рамках работы со структурами, подобными E-графам (самый близкий из предшествующих результатов — исключение дублирования вершин, обозначающих значения, отличающиеся некоторым целочисленным смещением[2]).

Мы также вводим специальное преобразование, которое называем слиянием по бисимуляции, работающее как доказательство равенств по индукции или коиндукции. Его предшественником можно считать специализированную версию слияния по конгруэнтности, способную сливать одинаковые циклы, реализованную в системе насыщения равенствами Peggy, созданной в рамках оригинальных работ по насыщению равенствами, однако она менее мощная и не упоминается в публикациях. Слияние по бисимуляции играет роль ана-

[2] *Nieuwenhuis R., Oliveras A.* Congruence Closure with Integer Offsets // Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings. Vol. 2850 / ed. by M. Y. Vardi, A. Voronkov. Springer, 2003. Pp. 78–90. (Lecture Notes in Computer Science). ISBN 3-540-20101-7. URL: http://dx.doi.org/10.1007/978-3-540-39813-4_5.

логичную выявлению, доказательству и применению лемм в многоуровневой суперкомпиляции, однако для обеспечения корректности в данной диссертации используются признаки структурной и защищённой рекурсии вместо теории улучшения Сэндса.

Практическая значимость работы

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

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

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

Апробация работы

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

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

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

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

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

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

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

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

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

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

Структура и объём диссертации

Диссертация состоит из введения, 6 глав, заключения, списка литературы и трёх приложений. Работа изложена на 215 страницах, из них 173 страницы основного текста. Список литературы содержит 88 наименований. В работе содержится 16 рисунков и 9 таблиц.

Содержание работы

Введение Во введении даётся обоснование актуальности, формулируются цели и задачи работы, аргументируется научная новизна и показывается практическая значимость.

Глава 1 В Главе 1 “Смежные работы” приведён исторический обзор методов доказательства свойств программ по индукции и методов преобразования программ на основе идей, близких к насыщению равенствами.

Насыщение равенствами — термин, введённый в работе “Equality Saturation: A New Approach to Optimization” Тейта и других, где этот метод использовался для преобразования императивных программ. Суть метода насыщения равенствами заключается в использовании структуры данных, компактно представляющей множество программ. В работах Тейта и др. данная структура данных называется E-PEG и представляет из себя размеченный E-граф. E-граф — это граф, вершины которого разбиты на классы эквивалентности. E-графы и подобные им структуры данных эффективно представляют множества равенств над термами и часто используются в средствах доказательства теорем. Обычно E-графы применяются в сочетании с алгоритмом конгруэнтного замыкания, позволяющим поддерживать представляемое E-графом множество равенств замкнутым относительно аксиомы монотонности ($\overline{a_i = b_i} \Rightarrow f(\overline{a_i}) = f(\overline{b_i})$). Кроме задачи доказательства эквивалентности E-графы могут быть использованы для других задач, например для оптимизации программ. Данное применение основано на том, что при помощи E-графа можно компактно представить множество эквивалентных программ, компактность представления осуществляется за счёт объединения общих подвыражений (sharing). При этом над E-графом можно осуществлять различные преобразования, что даёт эффект одновременного преобразования всего множества программ.

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

Суперкомпиляция — метод преобразования программ, который был разработан В.Ф. Турчиным. Суперкомпиляция похожа на автоматическое доказательство по индукции, однако эти области развивались довольно независимо. Изначально суперкомпиляция была сформулирована для языка Рефал, но потом появились формулировки и для других языков. В общих чертах суперкомпиляция заключается в применении правил переписывания к исходному выражению для построения так называемого графа конфигураций, который потом может быть превращён в остаточную программу, семантически эквивалентную исходному выражению. Одно из развитий идеи суперкомпиляции — многорезультатная суперкомпиляция, предложенная Ключниковым и Романенко[3]. Традиционно суперкомпилятор выдаёт на выходе только одну программу, но многорезультатный суперкомпилятор выдаёт целое множество остаточных программ. Впрочем для большинства приложений многорезультатной суперкомпиляции выдавать непосредственно множества остаточных программ не очень эффективно. Эффективнее генерировать некоторое компактное представление множества остаточных программ, которое потом можно обрабатывать также более эффективно. Например, в работе Гречаника, Ключникова и Романенко "Staged Multi-Result Supercompilation: Filtering by Transformation"[4] описано такое представление множеств остаточ-

[3] *Klyuchnikov I. G.*, *Romanenko S. A.* Multi-Result Supercompilation as Branching Growth of the Penultimate Level in Metasystem Transitions // Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011. Vol. 7162 / ed. by E. Clarke, I. Virbitskaite, A. Voronkov. Springer, 2012. Pp. 210–226. (Lecture Notes in Computer Science). ISBN 978-3-642-11485-4.

[4] *Grechanik S.*, *Klyuchnikov I.*, *Romanenko S.* Staged Multi-Result Supercompilation: Filtering by Transformation // Proceedings of the Fourth International Valentin Turchin Work-

ных программ в виде деревьев, компактность при этом достигается за счёт совмещения общих начальных частей программ, а дальнейшая фильтрация этого множества по некоторому условию может быть произведена эффективнее за счёт того, что в процессе разом могут выбрасываться целые подклассы программ. В работе Гречаника “Overgraph Representation for Multi-Result Supercompilation” используется ещё более компактное представление множества программ в виде графа, подобного E-графу, но при этом сохраняется общая схема суперкомпиляции. Фактически эта работа является промежуточной между суперкомпиляцией и насыщением равенствами. Настоящая диссертационная работа является следующим шагом по направлению к насыщению равенствами — фактически мы используем преобразования из суперкомпиляции, но в рамках насыщения равенствами.

Глава 2 В Главе 2 “Основная идея метода: полипрограммы и преобразования над ними” даётся описание нестрогого нетипизированного функционального языка первого порядка, который используется в качестве входного, и вводится понятие *полипрограммы* — центрального понятия диссертации. Также в этой главе даётся семантика полипрограмм и описывается общая схема предлагаемого метода.

Синтаксис входного языка описан на Рис. 1. Для данного языка вводится понятие программы и полипрограммы.

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

---

shop on Metacomputation / ed. by A. Klimov, S. Romanenko. Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", July 2014. Pp. 54–78. ISBN 978-5-901795-31-6.

Рис. 1 Синтаксис входного языка

Определение:

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

Выражение:

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

Определение 2. *Полипрограммой* (на нашем языке) назовём набор определений (как на Рис. 1).

Таким образом, полипрограмма отличается от программы главным образом тем, что в ней разрешены множественные определения одной и той же функции. Если рассматривать определение как уравнение (относительно неизвестных-функций), то полипрограмма является системой уравнений, т.е. семантические функции-решения должны удовлетворять всем определениям. Полипрограммы полезны при анализе и преобразовании программ, поскольку позволяют одновременно работать с несколькими представлениями одной и той же функции.

Полипрограммы также представляют множества обычных программ. Действительно, если для каждой функции полипрограммы, имеющей определение, выбрать по одному определению, то получится обычная программа. Однако такая программа может не быть эквивалентна исходной полипрограмме, поэтому необходимо добавить ограничение, что программа должна быть семантически эквивалентна исходной полипрограмме. Таким образом, мы получим множество программ, представляемое полипрограммой.

Традиционная денотационная семантика программ, определяющаяся через наименьшую неподвижную точку, не очень подходит для полипрограмм, потому что не обладает композиционностью на уровне определений: замена фрагмента полипрограммы на другой фрагмент полипрограммы с той же самой наименьшей неподвижной точкой может изменить семантику всей полипрограммы. Поэтому вместо рассмотрения только наименьшей неподвижной точки, мы определяем семантику полипрограммы как совокупность всех неподвижных точек (моделей) полипрограммы — в этом случае композиционность на уровне определений выполняется.

Аналогично логике предикатов первого порядка вводится понятие интерпретации и модели полипрограммы. Интерпретация отображает каждый функциональный символ из полипрограммы в непрерывную функцию типа $[\mathbb{A}^n \to \mathbb{A}]$, где $\mathbb{A}$ — множество данных первого порядка, заданное как наибольшая неподвижная точка следующего определения:

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

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

Определение 3. Пусть G — полипрограмма, а F — множество функциональных символов данной полипрограммы.

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

Моделью полипрограммы G будем называть такую интерпретацию μ, что $\mu \models G$, где $\mu \models G \stackrel{\text{def}}{=} \bigwedge_{p \in G} [\![p]\!]_\mu$, т.е. в μ выполняются все определения p полипрограммы G. $[\![p]\!]_\mu$ определено на Рис. 2.

Рис. 2 Семантика определений

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

$$[\![t_1 \equiv t_2]\!]_\mu \Leftrightarrow \forall \sigma : V \to \mathbb{A} \,.\, (\!|t_1|\!)_{\mu\sigma} = (\!|t_2|\!)_{\mu\sigma}$$

(σ — контекст, отображающий имена переменных из V в значения)

Определение 4. Две полипрограммы G_1 и G_2 со множествами функциональных символов F_1 и F_2 будем называть *семантически эквивалентными* на подмножестве функциональных символов $F \subseteq F_1 \cap F_2$, если для любой модели $\mu_1 \models G_1$ существует модель $\mu_2 \models G_2$ такая, что $\forall f \in F \,.\, \mu_1(f) = \mu_2(f)$, и наоборот, для любой модели $\eta_2 \models G_2$ существует модель $\eta_1 \models G_1$ такая, что $\forall f \in F \,.\, \eta_1(f) = \eta_2(f)$.

Основная задача преобразования полипрограмм звучит следующим образом:

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

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

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

Действительно, чтобы решить задачу эквивалентности, достаточно рассмотреть программу P как полипрограмму, преобразовать её в полипрограмму P' алгоритмом, решающим задачу преобразования, и проверить, существует ли в полипрограмме P' функция h и два определения $f(\overline{x_i}) \equiv h(\overline{x_i})$ и $g(\overline{x_i}) \equiv h(\overline{x_i})$.

Рис. 3 Простейшие выражения

Элементарный вызов функции:

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

Простейшее выражение:

$$
\begin{aligned}
s ::= {} & r \\
| {} & x \\
| {} & f(\overline{r_i}) \\
| {} & C(\overline{r_i}) \\
| {} & \textbf{case } r_0 \textbf{ of } \{ \overline{C_i(\overline{x_{ij}}) \ \rightarrow \ r_i;} \}
\end{aligned}
$$

Для решения задачи преобразования полипрограмм в диссертации предлагается использовать локальные правила преобразования полипрограмм вида $P_1 \mapsto P_2$ и преобразование, называемое слиянием по бисимуляции.

В Главе 2 рассмотрены приведены некоторые возможные правила преобразования полипрограмм, однако они удобны для ручного преобразования, но не очень удобны для программной реализации (правила, удобные для программной реализации, приведены в Главе 4). Также в Главе 2 вкратце рассмотрено слияние по бисимуляции, которое более подробно описано в Главе 5.

Глава 3 В Главе 3 “Расчленённая форма и бесточечное представление полипрограмм” вводятся понятия расчленённой формы и бесточечного представления полипрограммы и излагается теория, лежащая в основе применения локальных правил преобразований.

Понятие полипрограммы и правила, введённые в Главе 2 диссертации, не очень удобны для программной реализации. Главная проблема состоит в том, что если применять правила в ширину, то происходит слишком быстрое разрастание полипрограммы. Для борьбы с этой проблемой мы используем, во-первых, стандартный приём слияния общих подвыражений (используемый, в частности, в насыщении равенствами), а во-вторых, применяем некоторые правила деструктивно.

Для того, чтобы производить слияние общих подвыражений и упростить программную реализацию, в Главе 3 вводится понятие полипрограммы в расчленённой форме.

Определение 5. *Полипрограммой в расчленённой форме* назовём полипрограмму, в которой каждое определение имеет вид $f(\overline{x_i}) \equiv s$, где s — простейшее выражение как на Рис. 3.

Для слияния общих подвыражений в полипрограмме необходимо при-

вести её к расчленённой форме (это делается за конечное число шагов), после чего применить правила транзитивности, конгруэнтности и удаления определений-дубликатов, описанные в Главе 4.

При применении правил преобразования из Главы 2 диссертации у расчленённой формы есть одна проблема: если левая часть правила находится не в расчленённой форме, то перед применением правила необходимо будет применить правило раскрытия вызова функции. Это разрушит расчленённость программы, что нежелательно, однако после применения правила такое раскрытое определение можно будет удалить. Также если правая часть правила не находится в расчленённой форме, то после его применения расчленённость, опять же, будет нарушена. В этом случае после применения правила необходимо применить расчленение полипрограммы.

Однако есть более изящное решение — переписать все правила так, чтобы обе части находились в расчленённой форме. При этом свойство расчленённости будет сохраняться применением правил. Правила, переписанные в таком виде, приведены в Главе 4. Такие переписанные правила получаются проще и удобнее для реализации, однако они не очень удобны для ручного преобразования полипрограмм.

Также в Главе 3 вводится понятие бесточечно-расчленённого представления полипрограммы, позволяющее избавиться от имён переменных в полипрограммах, что ещё больше упрощает программную реализацию.

Определение 6. *Перестановкой параметров* назовём инъективное отображение $\{1,\dots,n\} \hookrightarrow \{1,\dots,m\}$ (для простоты будем писать $n \hookrightarrow m$), где $m, n \in \mathbb{N}$. Множество всех перестановок параметров обозначим Θ.

Определение 7. Множеством *функциональных символов* назовём произвольное множество F, на котором определена функция $arity : F \to \mathbb{N}$.

Определение 8. *Полипрограммой в бесточечно-расчленённом представлении* будем называть тройку $\langle H, F, v\rangle$, где H — просто некоторое множество (индексирующее определения), F — множество функциональных символов, а $v : H \to P$, где P — множество формул вида $\theta_0 f_0 \equiv \xi\, L(\overline{\theta_i f_i})$ как на Рис. 4, где $f_i \in F$, $\theta_i \in \Theta$, $\xi \in \Theta$, причём обе части формулы должны быть согласованы по арности.

Далее в Главе 3 напрямую вводится семантика полипрограмм в бесточечно-расчленённом представлении, вводится категория полипрограмм и формально описывается, что такое правило и как применять правило к полипрограмме.

Глава 4 В Главе 4 “Локальные правила преобразования” приводятся правила преобразования полипрограмм в бесточечно-расчленённом представлении, используемые в нашем методе. Правила делятся на две группы — упрощающие правила, для которых выполняется завершаемость (хотя и с некоторыми

Рис. 4 Виды формул полипрограммы

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

оговорками), и насыщающие правила, для которых не выполняется завершаемость, и которые всегда применяются недеструктивно.

Список упрощающих правил приведён в Таблице 1.

Кроме правил вводится отдельное преобразование — конгруэнтность. Оно применяется вместе с упрощающими правилами, но оно не может быть сформулировано в виде правила преобразования полипрограмм в рамках определений, данных в Главе 3, поэтому формулируется в виде отдельного преобразования.

Далее доказывается, что без коммутативности упрощающие правила вместе с конгруэнтностью являются завершающейся системой переписывания с точностью до удаления дубликатов. Правило коммутативности же разрушает завершаемость при наличии деструктивных правил (которые крайне важны). Однако можно перемежать применения коммутативности и остальных правил следующим образом: сначала применяем все упрощающие правила (без коммутативности) и конгруэнтность до нормальной формы, потом применяем коммутативность до нормальной формы, потом снова упрощающие правила без коммутативности и конгруэнтность и т.д. Доказывается, что такая процедура переписывания завершается.

Наконец, приводится набор насыщающих правил. Насыщающие правила могут приводить к незавершаемости. Все они применяются следующим образом: сначала полипрограмма упрощается при помощи применения упрощающих правил, конгруэнтности и коммутативности описанным выше способом, потом находятся все возможности применения насыщающих правил, и все эти применения осуществляются, потом полипрограмма снова упрощается, потом снова применяются насыщающие правила и т.д. Таким образом, происходит применение насыщающих правил “в ширину”, слой за слоем, с полным упрощением полипрограммы между слоями. Отметим, что большинство насыщающих правил и некоторые из упрощающих по сути реализуют прогонку, важную составляющую суперкомпиляции.

Глава 5 В главе 5 “Слияние по бисимуляции” рассматривается преобразование, реализующее возможность доказательства эквивалентности функций

Таблица 1 Упрощающие правила

Обозначение	Смысл
(trans)	Транзитивность равенства $f \equiv E, g \equiv E \Rightarrow f \equiv g$
(remove-dups)	Удаление дубликатов $f \equiv E, f \equiv E \Leftrightarrow f \equiv E$
(clone)	Дублирование определения $f \equiv E \Leftrightarrow f \equiv E, f \equiv E$
(eq-symm)	Симметричность равенства $f \equiv g \Rightarrow g \equiv f$
(eq-refl)	Рефлексивность равенства $f \equiv f$
(arity-reduction)	Редукция арности $f(x,y) \equiv E\langle x \rangle \overset{f(x,y)\to g(x)}{\Leftrightarrow} g(x) \equiv E\langle x \rangle$
(call-to-permutation)	Преобразование **Call** в **Id** $f(x,y) \equiv h(id(x), id(y)) \Leftrightarrow f \equiv h$
(commutativity)	Коммутативность $h \equiv L(f), f \equiv \theta f \Rightarrow h \equiv L(\theta f)$
(inj-cons)	Инъективность **Cons** $f \equiv C(g), f \equiv C(h) \Rightarrow g \equiv h$
(inj-case)	Инъективность **Case** $f \equiv \textbf{case}\ x\ \textbf{of}\{C(y) \to g(y)\},$ $f \equiv \textbf{case}\ x\ \textbf{of}\{C(y) \to h(y)\}$ $\Rightarrow g \equiv h$
(red-call-var)	Редукция **Call-Var** $f \equiv id(g) \Leftrightarrow f \equiv g$
(red-case-cons)	Редукция **Case-Cons** $f \equiv \textbf{case}\ c\ \textbf{of}\{C(y) \to h(y)\},$ $c \equiv C(g)$ $\Rightarrow f \equiv h(g)$
(red-case-cons-bot)	Незнакомый конструктор $f \equiv \textbf{case}\ c\ \textbf{of}\{C \to \ldots\}, c \equiv D(\ldots)$ $\Rightarrow f \equiv \bot$

по индукции и коиндукции. Это преобразование не может быть выражено правилом, т.к. требует довольно сложного анализа всей полипрограммы.

Идея слияния по бисимуляции в том, чтобы искать фрагменты полипрограммы, находящиеся в отношении бисимуляции и сливать соответствующие функции. Бисимуляция полипрограмм является обобщением понятие бисимуляции для размеченных систем переходов, однако интуитивно они сильно отличаются (в частности, недетерминизм в размеченных системах переходов **не** соответствует недетерминизму, связанному с множественностью определений). Отметим, что если полипрограммы изоморфны (превращаются друг в друга переименованием функций и переменных), то они находятся в отношении бисимуляции.

В главе приводится алгоритм поиска бисимуляции между двумя функциями и доказывается, что он действительно находит бисимуляции. Основная трудность при этом состоит в работе с перестановками параметров, т.к. необходимо искать бисимуляции с точностью до перестановок параметров.

Слияние по бисимуляции позволяет в некоторых случаях доказывать эквивалентность функций, когда этого нельзя сделать при помощи локальных правил преобразований, например как в этом примере:

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

Вообще говоря, одного отношения бисимуляции недостаточно для доказательства эквивалентности, нужны некоторые дополнительные условия, например следующие два определения нельзя слить в одно, хотя они представляют из себя два изоморфных фрагмента:

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

Действительно, они могут быть частью некоторой большей полипрограммы, в которой f и g не могут быть равны:

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

В качестве дополнительных условий, обеспечивающих эквивалентность, в диссертации предлагается использовать синтаксические признаки структурной и защищённой рекурсии, которые обычно используются для обеспечения тотальности функций. В случае нашего языка, однако, эти признаки обеспечивают не тотальность, а единственность модели фрагмента полипрограммы, что доказывается в диссертации.

Теорема 1. Если для конечной полипрограммы выполнено достаточное условие структурной рекурсии и каждая её функция имеет не более одного определения, то для каждой интерпретации $\eta : W \to \mathbb{D}$ её функций без определений она имеет ровно одну модель $\mu(\eta)$.

Следствие 1. Если для конечной полипрограммы выполнено достаточное условие *структурной-защищённой* рекурсии и каждая её функция имеет не более одного определения, то для каждой интерпретации $\eta : W \to \mathbb{D}$ её функций без определений она имеет ровно одну модель $\mu(\eta)$.

Из единственности модели для каждого из двух фрагментов, находящихся в отношении бисимуляции, сразу следует, что соответствующие функции в полипрограмме можно слить в одну.

Глава 6 В Главе 6 "Реализация и экспериментальные результаты" описаны некоторые подробности практической реализации изложенного метода и приведены результаты тестирования на наборе примеров, в том числе в сравнении с похожими инструментами.

Практическая реализация представляет собой систему для решения задачи эквивалентности функций на входном языке. Она написана на языке Scala и имеет открытый исходный код[5]. Работает она следующим образом:

- Сначала над программой производятся некоторые упрощающие преобразования, самое главное из которых — дефункционализация, позволяющая избавиться от функций высшего порядка. Это позволяет решать задачу эквивалентности для языка высшего порядка не смотря на то, что сам прувер внутри работает с языком первого порядка.

- Затем программа преобразуется в бесточечно-расчленённое представление. Утверждение, которое требуется доказать преобразуется в форму $\theta f \equiv \xi g$, где f и g являются функциями полипрограммы.

- Производится преобразование полипрограммы (насыщение) до тех пор, пока не будет доказано целевое утверждение.

Прувер написан в императивном стиле, и преобразования производятся при помощи модификации полипрограммы, представленной мутабельным множеством функций и определений.

Для экспериментальной оценки нашего прувера и сравнения его с похожими инструментами мы использовали два набора простых эквивалентностей, порядка 50 примеров каждый. Первый набор — наш собственный. Для второго набора мы использовали проект TIP (Tons of Inductive Problems), включающий как собственные примеры, так и некоторые более старые наборы, оставив только примеры, верные в нашей семантике.

[5] Graphsc source code and the test suite. `https://github.com/sergei-grechanik/supercompilation-hypergraph`.

Таблица 2 Суммарное количество взятых примеров

Набор	graphsc	graphsc без лемм	hipspec	hipspec без лемм	zeno	hosc	всего
Наш набор	36	32	36	31	32	33	49
TIP	23	24	37	28	34	18	50
Всего	59	56	73	59	66	51	99

В качестве других инструментов для сравнения мы использовали HipSpec, HOSC и Zeno. Сравнение количества взятых примеров приведено в Таблице 2, наш инструмент приведён под названием graphsc. В целом наш прувер работает хуже современных инструментов Zeno и HipSpec, главным образом за счёт того, что не умеет производить нетривиальные обобщения, однако при отключении лемм наш прувер проигрывает три примера HipSpec'у без лемм, что является неплохим результатом.

Результаты

В настоящей работе были получены следующие результаты:

- На основе насыщения равенствами и суперкомпиляции был разработан метод преобразования множеств программ на нестрогом функциональном языке первого порядка.
  - Было предложено понятие полипрограммы, позволяющее представлять множества функциональных программ, и была определена семантика полипрограмм.
  - Был разработан набор локальных правил эквивалентных преобразований полипрограмм.
  - Было предложено глобальное преобразование слияние по бисимуляции, соответствующий доказательству эквивалентности функций по индукции и коиндукции.
- Разработанный метод был реализован в системе доказательства эквивалентности функций на нестрогом функциональном языке.
- Разработанная система доказательства эквивалентности была протестирована на наборе модельных задач.

Публикации автора по теме диссертации

1. *Grechanik S. A.* Overgraph Representation for Multi-Result Supercompilation // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. Pereslavl-Zalessky,

Russia : Pereslavl-Zalessky: Ailamazyan University of Pereslavl, July 2012. Pp. 48–65. ISBN 978-5-901795-28-6. URL: http://pat.keldysh.ru/~grechanik/doc/overgraph.pdf

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

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

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

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

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

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

Подписано в печать xx.xx.xx.
Формат 60x84/16. Усл. печ. л. 1,1. Тираж 65 экз. Заказ А-22.
ИПМ им. М.В. Келдыша РАН. 125047, Москва, Миусская пл., 4