На правах рукописи

Ключников Илья Григорьевич

ВЫЯВЛЕНИЕ И ДОКАЗАТЕЛЬСТВО СВОЙСТВ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ МЕТОДАМИ СУПЕРКОМПИЛЯЦИИ

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

Автореферат
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва – 2010

Работа выполнена в Институте прикладной математики им. М.В. Келдыша РАН.

Научный руководитель:	кандидат физико-математических наук Романенко Сергей Анатольевич
Официальные оппоненты:	доктор физико-математических наук, член-корреспондент РАН Абрамов Сергей Михайлович
	кандидат физико-математических наук, старший научный сотрудник Бульонков Михаил Алексеевич
Ведущая организация:	Институт системного программирования РАН

Защита состоится 2 ноября 2010 г. в 11 часов на заседании диссертационного совета Д 002.024.01 в Институте прикладной математики им. М.В. Келдыша РАН по адресу: 125047, Москва, Миусская пл., 4.

С диссертацией можно ознакомиться в библиотеке Института прикладной математики им. М.В. Келдыша РАН.

Автореферат разослан 1 октября 2010 г.

Ученый секретарь диссертационного совета,
доктор физико-математических наук Т.А. Полилова

Общая характеристика работы

Объект исследования и актуальность работы

Одним из распространенных методов формальной верификации программ, обеспечивающих *полноту*, является *проверка на моделях программ* (model checking): для многих формализаций проверка на моделях программ либо доказывает, что модель всегда удовлетворяет спецификации, либо находит ошибку. Главным недостатком проверки на моделях является то, что *проверяется не сама программа, а ее модель*.

Другим распространенным методом проверки правильности программ является *тестирование*: условия корректности компоненты f кодируются в виде предиката (функции) p, и на входных данных X_1, X_2, ..., X_n проверяется истинность $p(X_1, f(X_1))$, $p(X_2, f(X_2))$, ..., $p(X_n, f(X_n))$. *Преимущества* тестирования: рассматривается *реальная система*, условия корректности пишутся *на том же языке*, на котором написана программа. Главным недостатком тестирования является *неполнота*, – как правило, рассматриваются лишь некоторые входные значения, и успешное прохождение тестов *не гарантирует отсутствие ошибок*.

Отметим, что $p(X, f(X))$ является композицией проверяемой функции f и предиката p. Существуют методы преобразования программ, способные упрощать такие композиции, в результате чего получается программа $p'(X) = \ldots$, которая легче поддается анализу, чем исходная композиция $p(X, f(X))$.

Одним из таких методов является суперкомпиляция [9]. В частности, суперкомпиляция была применена А. Немытых для трансформационного анализа кэш-когерентных протоколов (закодированных на языке Рефал) [3]. Суть трансформационного подхода к анализу программ можно сформулировать следующим образом: вместо того, чтобы анализировать исходную программу, вначале преобразуем эту программу в эквивалентную ей, но легче поддающуюся анализу. Если используемый метод преобразования программ способен устранять избыточности исходной программы, то во многих случаях анализ остаточной программы становится тривиальным (например, когда получается программа, выдающая $True$ на всех входных данных: $p'(X) = True$).

Такой подход может быть успешным только при условии, что рассматриваемый язык программирования обладает достаточными *изобразительными средствами* для написания высокоуровневых спецификаций и одновременно позволяет осуществлять *глубокие преобразования* программ *с сохранением семантики*.

Примером языка, удовлетворяющего этим требованиям, является язык Haskell [8]. С одной стороны программы на языке Haskell достаточно хорошо поддаются преобразованиям. С другой стороны Haskell предлагает

мощные изобразительные средства, облегчающие написание спецификаций: функции высших порядков, бесконечные структуры данных, автоматический вывод и проверку типов.

Таким образом, представляется актуальным исследовать возможности трансформационного анализа программ, написанных на языке Haskell.

Цели и задачи работы

Целью работы было исследование возможностей применения суперкомпиляции для трансформационного анализа программ на языке Haskell.

Суперкомпилятор, применяемый для трасформационного анализа программ, должен удовлетворять следующим требованиям:

- *Гарантированно сохранять семантику программы.* В противном случае выводы, сделанные из анализа остаточной программы, могут быть необоснованными или ошибочными.
- *Гарантированно завершаться на любой входной программе.* Логически это требование не является абсолютно необходимым, но весьма важно с прагматической точки зрения.
- *Иметь доступный исходный код.* Без этого отсутствует сама возможность убедиться в корректности реализации суперкомпилятора.

Такого суперкомпилятора на момент начала диссертационной работы (2007 год) не существовало[1]. Поэтому автор поставил перед собой следующие задачи:

1. Разработать метод суперкомпиляции функциональных программ, ориентированный на трансформационный анализ.
2. Разработать работоспособный алгоритм суперкомпиляции программ, написанных на ядре языка Haskell, сохраняющий семантику программ и гарантированно завершающийся на любой входной программе.
3. Реализовать разработанный алгоритм в экспериментальном суперкомпиляторе.
4. Апробировать экспериментальный суперкомпилятор на модельных задачах по выявлению и доказательству свойств программ.

[1] Вообще, а не только для языка Haskell.

Научная новизна работы

В отличие от других работ, где суперкомпилятор рассматривается как фиксированная монолитная конструкция, в данной работе определяется *модульная структура суперкомпилятора* HOSC в виде отношения трансформации и доказывается его корректность.

Впервые полностью и формально описан *алгоритм нахождения тесного обобщения* любых выражений со связанными переменными. Определено *уточненное гомеоморфное вложение* такое, что два сцепленных выражения всегда имеют нетривиальное обобщение. Показано, что уточненное гомеоморфное вложение является *вполне-квазиупорядочением* на множестве выражений, порождаемых суперкомпиляцией.

На базе алгоритма распознавания эквивалентных выражений разработан *алгоритм распознавания улучшающих лемм*, основанный на аннотировании остаточной программы информацией о скорости работы исходной программы.

Предложен *метод многоуровневой суперкомпиляции*, основанный на уклонении от обобщения с помощью применения улучшающих лемм.

Практическая значимость работы

Диссертационная работа дает положительный ответ на вопрос о возможности использования суперкомпиляции для трансформационного анализа программ.

На основе разработанных алгоритмов и методов создан экспериментальный суперкомпилятор HOSC для ядра языка Haskell.

Показано, что *алгоритм распознавания эквивалентности выражений*, основанный на нормализации, позволяет доказывать утверждения, которые даже невозможно сформулировать в терминах подхода, основанного на предикате равенства. Работоспособность алгоритма показана на проверке корректности реализации монад из стандартной библиотеки языка Haskell.

На базе суперкомпилятора HOSC создан многоуровневый суперкомпилятор TLSC, способный производить более глубокие преобразования программ, в частности, улучшать асимптотику программ.

Апробация работы и публикации

Результаты работы докладывались на следующих конференциях и семинарах:

- Международный семинар “First International Workshop on Metacomputation in Russia, META’08”, Россия, Переславль-Залесский, 2008.

- Научный семинар по языкам программирования “Copenhagen Programming Language Seminar (COPLAS)” на факультете информатики Копенгагенского университета, Дания, Копенгаген, 2008.
- Седьмая международная конференция памяти Андрея Ершова “Perspectives of System Informatics, PSI’09”, Россия, Новосибирск, 2009.
- Международный семинар “International Workshop on Program Understanding, PU’09”, Россия, Алтай, 2009.
- Объединенный научный семинар по робототехническим системам ИПМ им. М.В. Келдыша РАН, МГУ им. М.В. Ломоносова, МГТУ им. Н.Э. Баумана, ИНОТиИ РГГУ и отделения “Программирование” ИПМ им. М.В. Келдыша РАН, Россия, Москва, 2009.
- Семинар московской группы пользователей языка Haskell (MskHUG), Москва, 2009.
- Международный семинар “Second International Workshop on Metacomputation in Russia, META’10”, Россия, Переславль-Залесский, 2010.
- Научный семинар ИСП РАН, Россия, Москва, 2010.

По результатам работы имеются четыре публикации, включая одну статью в рецензируемом научном журнале из списка ВАК [1], одну статью в международном периодическом издании [3], две статьи в сборниках трудов международных научных семинаров [2, 4]:

1. *Ключников И.Г., Романенко С.А.* SPSC: Суперкомпилятор на языке Scala // Программные продукты и системы. – 2009. – №2 (86). – С. 74-80.
2. *Klyuchnikov I., Romanenko S.* SPSC: a Simple Supercompiler in Scala // International Workshop on Program Understanding, PU 2009, Altai Mountains, Russia, June 19-23, 2009. – Novosibirsk: A.P. Ershov Institute of Informatics Systems, 2009. – Pp. 5-17.
3. *Klyuchnikov I., Romanenko S.* Proving the Equivalence of Higher-Order Terms by Means of Supercompilation // Perspectives of Systems Informatics. 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. – Vol. 5947 of LNCS. – Springer, 2010. – Pp. 193-205.
4. *Klyuchnikov I., Romanenko S.* Towards Higher-Level Supercompilation // Proceedings of the second International Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 1-5, 2010. – Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2010. – Pp. 82-101.

Структура и объем диссертации

Диссертация состоит из введения, 8 глав, заключения и списка литературы. Содержание работы изложено на 189 страницах, из них 172 страниц основного текста. Список литературы содержит 141 наименование. В работе содержится 90 рисунков и 3 таблицы.

Содержание работы

Во **введении** обоснована актуальность диссертационной работы, сформулированы цели и задачи, аргументирована научная новизна исследований, показана практическая значимость полученных результатов.

В **главе 1** "Позитивная суперкомпиляция и анализ программ" в первом разделе приведен исторический обзор развития методов суперкомпиляции. Особое внимание уделяется работам последних лет, связанных с суперкомпиляцией языков высшего порядка.

Традиционно преобразования программ рассматриваются как последовательность применения к программе некоторого набора локальных правил переписывания, которые сохраняют значение программы (семантику). Чаще всего преобразования программ направлены на их оптимизацию или распараллеливание.

Суперкомпиляция была предложена в 1970-х годах В.Ф. Турчиным как метод преобразования программ, написанных на функциональном языке первого порядка Рефал.

В основе суперкомпиляции находятся следующие процедуры:

- Построение размеченного "дерева процессов", которое представляет все возможны трассы некоторого вычислительного процесса. Метки в дереве (конфигурации) представляют множества состояний вычисления.
- Декомпозиция и обобщение конфигураций с целью превратить (потенциально) бесконечное дерево процессов в конечный граф.
- Преобразование получившегося графа в (остаточную) программу.

Отличительной особенностью суперкомпиляции является рассмотрение программы как единого целого: модель вычисления программы p, представленная графом конфигураций, является самодостаточной и полностью описывает поведение рассматриваемой программы.

Отмечалось, что суперкомпиляцию можно применять не только для оптимизации программ, но и для анализа программ и для верификации. А именно: в результате преобразования программы средствами суперкомпиляции может получиться программа, эквивалентная исходной программе,

Рис. 1 Выражения языка SLL

e	$::=$		$e \in E_{C \cup F \cup V}$, выражение
		v	$v \in V$, переменная
	$\mid$	$c(e_1, \ldots, e_n)$	$c \in C$, конструктор
	$\mid$	$f(e_1, \ldots, e_n)$	$f \in F$, вызов функции

Рис. 2 Гомеоморфное вложение SLL-выражений

$e' \trianglelefteq e''$	если $e' \trianglelefteq_v e''$, $e' \trianglelefteq_d e''$ или $e' \trianglelefteq_c e''$
Вложение переменных	
$v' \trianglelefteq_v v''$	
Сцепление (Coupling)	
$h(e'_1, \ldots, e'_n) \trianglelefteq_c h(e''_1, \ldots, e''_n)$	если $\forall i : e'_i \trianglelefteq e''_i$
Погружение (Diving)	
$e \trianglelefteq_d h(e'_1, \ldots, e'_n)$	если $\exists i : e \trianglelefteq e'_i$

но с более простой структурой, такой, что не совсем очевидные свойства исходной программы могут стать очевидными и легко доказуемыми для остаточной программы.

Цели оптимизации программ и анализа программ в некоторой степени противоречат друг другу. Главная цель оптимизации программы – получить небольшую и быструю программу, которая может быть труднопонимаемой для человека, иметь запутанную и странную структуру. Более того, оптимизатор не обязательно выдает программу, эквивалентную исходной.

Основные усилия были направлены на создание работающего оптимизирующего суперкомпилятора. *Практическим* исследованиям по применению суперкомпиляции для анализа программ не уделялось должного внимания. Появление работающего оптимизирующего суперкомпилятора относится к середине 1990-х годов (суперкомпилятор SCP4). Многие части этого суперкомпилятора описаны неполностью или неформально. При использовании суперкомпилятора для анализа программ необходимо полное и формальное описание его внутреннего устройства.

В 90-е годы выходит серия работ Роберта Глюка, Морте Сёренсена, Нила Джонса, рассматривающих суперкомпиляцию для более простых, нежели Рефал, функциональных языков. Одной из целей этих работ было *полное и формальное* описание простого суперкомпилятора, который сохраняет семантику программы и гарантированно завершается.

Синтаксис выражений языка SLL, рассматриваемого в работах Сёренсена и Глюка [7], приведен на Рис. 1.

Завершаемость суперкомпилятора гарантируется тем, что при построении дерева процессов не допускаются ветки, на которых некоторая “верх-

Рис. 3 Сравнение суперкомпиляторов

	Эквивалентные преобразования	Функции высших порядков	Бесконечные данные	Док-во корректности	Док-во завершаемости	Исходный код
SCP4	–	–	–	–	–	+
Positive SCP	+	–	+	+	+	–
TSG SCP	+	–	–	–	–	+
Jscp	+	–	–	–	–	–
Supero	+	+	+	–	–	+
Timber SCP	+	+	–	+	+	–
HOSC	+	+	+	+	+	+

няя конфигурация" c_1 вкладывается в "нижнюю конфигурацию" c_2: $c_1 \trianglelefteq c_2$, $\trianglelefteq$ – отношение гомеоморфного вложения для SLL-выражений (Рис. 2). При обнаружении таких конфигураций обобщается верхняя конфигурация – вместо нее рассматривается более общая.

Использование отношения $\trianglelefteq$ не допускает построения бесконечных ветвей в частичном дереве процессов в силу следующей теоремы:

Теорема 1 (Крускал, Хигман)**.** *Пусть C и F – конечные множества. В любой бесконечной последовательности SLL-выражений $e_1, e_2, \ldots$ из множества $E_{C \cup F \cup V}$ найдутся e_i и e_j такие, что $i < j$ и $e_i \trianglelefteq e_j$.*

То есть, отношение гомеоморфного вложения $\trianglelefteq$ является вполне-квазиупорядочением на $E_{C \cup F \cup V}$. Также в главе 1 подробно разбирается алгоритм обобщения SLL-выражений.

Суперкомпилятор для языка SLL, описанный Сёренсеном – первый суперкомпилятор, для которого доказаны теоремы корректности и завершаемости. Позитивный суперкомпилятор для языка SLL рассматривается как пример полностью и формально описанного суперкомпилятора. То есть результатам анализа, полученным при использовании суперкомпилятора для языка SLL, в принципе, можно доверять.

В таблице на Рис. 3 сравниваются существовавшие на 2007 год[2] суперкомпиляторы с точки зрения пригодности для трансформационного анализа программ по критериям, указанным в колонках. На момент начала работы над диссертацией суперкомпилятора, удовлетворяющего всем рассматриваемым критериям, не существовало. Одним из результатов данной работы явилось создание экспериментального суперкомпилятора HOSC, удовлетворяющего всем требованиям.

Входным языком суперкомпилятора HOSC является язык HLL, который формально описывается в **глава 2** "Язык HLL: синтаксис и семанти-

[2]Когда автор начал работу над диссертацией.

Рис. 4 Синтаксис HLL-выражений

$e ::= v$	переменная
$\mid c\ \overline{e_i}$	конструктор
$\mid f$	глобальная переменная
$\mid \lambda\ \overline{v_i} \to e$	λ-абстракция
$\mid e_1\ e_2$	аппликация
$\mid$ **case** e_0 **of** $\{\overline{p_i \to e_i;}\}$	case-выражение
$\mid$ **let** $\overline{v_i = e_i;}$ **in** e	let-выражение
$\mid (e)$	выражение в скобках
$p ::= c\ \overline{v_i}$	образец

ка". Язык HLL является подмножеством ядра языка Haskell и выбран таким образом, чтобы минимизировать количество рассматриваемых сущностей в описании синтаксиса, семантики языка и при доказательствах корректности и завершаемости, оставаясь в то же время максимально приближенным к ядру языку Haskell.

Синтаксис выражений языка HLL приведен на Рис. 4 – в HLL-выражениях присутствуют связанные переменные. В последние годы наблюдается интерес к суперкомпиляции языков высшего порядка (со связанными переменными), таким как Haskell, Timber. Однако, опубликованные работы обходят стороной алгоритм обобщения выражений со связанными переменными. Отчасти, это связано с тем, что авторы слишком неформально работают со связанными переменными. В данной работе использование связанных переменных строго формализовано – выбрано соглашение об именовании переменных (соглашение Барендрегта) и вводится понятие допустимой для выражения подстановки.

Это критически важно для определения *алгоритма обобщения HLL-выражений* (глава 5). В случае языка без связанных переменных (такого, как SLL) вопросов относительно именовании переменных не возникает, в случае языка со связанными переменными этот вопрос является ключевым для построения формализма.

Также в главе 2 рассматривается операционная семантика языка с вызовом по имени (call-by-name)[3].

Понятие эквивалентности выражений является основополагающим для всего остального материала. В случае языка первого порядка с вызовом по значению результатом исполнения программы являются конечные данные – поэтому для таких языков программы считаются эквивалентными, если при любых входных данных они выдают одинаковые результаты (или не завершаются). Однако, в случае языка HLL результатом вычис-

[3]Конечные результаты работы программы при семантике вызова по имени и вызова по необходимости совпадают

Рис. 5 Отношение трансформации HOSC: алгоритм построения частичного дерева процессов t для конфигурации e_0

$t = \boxed{e_0}$
while $incomplete(t)$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive^*(t, \beta), generalize(t, \beta), fold(t, \beta)\}$
end

ления могут быть функции или бесконечные данные. Мы рассматриваем в данной работе операционную эквивалентность.

Определение 2 (Эквивалентность)**.** Два HLL-выражения e_1 и e_2 являются эквивалентными, если для любого контекста $C[\,]$, такого, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, вычисления выражений $C[e_1]$ и $C[e_2]$ либо вместе завершаются, либо вместе не завершаются.

Результат работы оптимизирующего суперкомпилятора (да и любого оптимизатора) единственен – выдавать несколько остаточных программ не имеет смысла. Поэтому в работах, посвященных оптимизирующей суперкомпиляции, доказывается корректность конкретного алгоритма суперкомпилятора. Однако, суперкомпилятор, используемый для анализа программ, может выдавать несколько вариантов остаточной программы – может оказаться, что один из вариантов легче поддается анализу, нежели остальные. Поэтому представляет интерес вначале рассмотреть множество программ, которые может выдавать суперкомпилятор.

В **главе 3** "Структура суперкомпилятора HOSC" описывается методологическая основа HOSC в виде отношения трансформации [2] (а не конкретного алгоритма). Недетерминированный алгоритм построения частичного дерева процессов приведен на Рис. 5, – таким образом описывается множество остаточных программ.

В главе 3 также дается алгоритм преобразования частичного дерева процессов в остаточную программу, являющуюся одним самодостаточным выражением с локальными определениями. Построение остаточных программ такого вида положительно сказывается на склонности суперкомпилятора HOSC к нормализации программ.

Оптимизирующий суперкомпилятор очень осторожно работает с локальными определениями (let-выражениями), чтобы не ухудшить скорость выполнения программы. Однако, при трансформационном анализе остаточная программа не предназначена для исполнения, – поэтому вполне допустимыми являются остаточные программы, которые работают медленнее, чем исходная. Суперкомпилятор HOSC устраняет локальные определения (let-выражения) методом λ-лифтинга, что упрощает по-

строение частичного дерева процессов и позволяет более агрессивно распространять позитивную информацию. Также, как показали эксперименты, устранение локальных определений перед суперкомпиляцией положительно сказывается на способности суперкомпилятора к нормализации выражений.

В **главе 4** "Корректность суперкомпилятора HOSC" приводится доказательство корректности суперкомпилятора HOSC. В отличие от большинства работ, где приводится доказательство корректности конкретного алгоритма, приводится доказательство корректности отношения трансформации, определенного в главе 3, т.е. показывается корректность любого алгоритма, удовлетворяющего этому отношению трансформации.

Используется теория операционных улучшений Сэндса [5], с помощью которой доказательства корректности преобразований осуществляются по следующей схеме: нужно показать, что остаточная программа является улучшением исходной программа и выполняются условия теоремы об улучшениях. Теорема Сэндса сформулирована только для случаев, когда локальные определения в остаточной программе не содержат свободных переменных, что неверно для отношения трансформации HOSC. Также, поскольку отношение трансформации HOSC допускает свертку *любых* конфигураций, не всякая остаточная программа является улучшением исходной. Поэтому доказательство корректности проходит в три этапа.

Сперва рассматривается отношение $HOSC_0$, которое допускает свертку только некоторых узлов и не допускает свободных переменных в локальных определениях в остаточной программе. Доказывается теорема, что любая остаточная программа, удовлетворяющая отношению $HOSC_0$, является строгим улучшением исходной программы, а, следовательно, остаточная програма эквивалентна исходной.

Затем рассматривается отношение $HOSC_{1/2}$, которое отличается от $HOSC_0$ тем, что разрешает сворачивать любые конфигурации. Доказывается корректность отношения трансформации $HOSC_{1/2}$ (для доказательства используются суперкомбинаторы с абстракцией максимально свободных выражений и задача сводится к предыдущей).

Наконец, показывается, что результат суперкомпиляции по отношению $HOSC_{1/2}$ и соответствующий результат по отношению $HOSC$ относятся через λ-дроппинг, а поэтому эквивалентны.

Важно отметить, что оптимизирующие суперкомпиляторы никогда не выдают программы, которые могут быть ухудшением (в терминах Сэндса) исходной программы. Суперкомпилятор HOSC может выдавать такие программы. В тексте приводятся примеры, когда ухудшение (в терминах Сэндса) плодотворно сказывается на нормализации программ.

Также рассматривается вопрос о корректности отношения трансфор-

Рис. 6 Уточненное гомеоморфное вложение HLL-выражений

Вложение с учетом таблицы связанных переменных

$e' \trianglelefteq^{**} e'' \mid_\rho$ если $e' \trianglelefteq_v^{**} e'' \mid_\rho$, $e' \trianglelefteq_d^{**} e'' \mid_\rho$ или $e' \trianglelefteq_c^{**} e'' \mid_\rho$

Вложение переменных

$f \trianglelefteq_v^{**} f$

$v' \trianglelefteq_v^{**} v'' \mid_\rho$ если $(v',\ v'') \in \rho$

$v' \trianglelefteq_v^{**} v'' \mid_\rho$ если $v' \notin domain(\rho)$ и $v'' \notin range(\rho)$

Сцепление (Coupling)

$c\ \overline{e'_i} \trianglelefteq_c^{**} c\ \overline{e''_i} \mid_\rho$ если $\forall i : e'_i \trianglelefteq^{**} e''_i \mid_\rho$

$\lambda v' \to e' \trianglelefteq_c^{**} \lambda v'' \to e'' \mid_\rho$ если $e' \trianglelefteq^{**} e'' \mid_{\rho \cup \{(v', v'')\}}$

$e'_0\ \overline{e'_i} \trianglelefteq_c^{**} e''_0\ \overline{e''_i} \mid_\rho$ если $\forall i : e'_i \trianglelefteq^{**} e''_i \mid_\rho$

$case\ e'\ of\ \{c_i\ \overline{v'_{ik}} \to e'_i;\} \trianglelefteq_c^{**} case\ e''\ of\ \{c_i\ \overline{v''_{ik}} \to e''_i;\} \mid_\rho$
если $e' \trianglelefteq^{**} e'' \mid_\rho$ и $\forall i : e'_i \trianglelefteq^{**} e''_i \mid_{\rho \cup \{\overline{(v'_{ik}, v''_{ik})}\}}$

Погружение (Diving) только если $fv(e) \cap domain(\rho) = \emptyset$

$e \trianglelefteq_d^{**} c\ \overline{e_i} \mid_\rho$ если $\exists i : e \trianglelefteq^{**} e_i \mid_\rho$

$e \trianglelefteq_d^{**} \lambda v_0 \to e_0 \mid_\rho$ если $e \trianglelefteq^{**} e_0 \mid_{\rho \cup \{(\bullet, v_0)\}}$

$e \trianglelefteq_d^{**} e_0\ \overline{e_i} \mid_\rho$ если $\exists i : e \trianglelefteq^{**} e_i \mid_\rho$

$e \trianglelefteq_d^{**} case\ e'\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \mid_\rho$
если $e \trianglelefteq^{**} e' \mid_\rho$ или $\exists i : e \trianglelefteq^{**} e_i \mid_{\rho \cup \{\overline{(\bullet, v_{ik})}\}}$

мации *HOSC* с учетом типизации.

Таким образом, в третьей главе был рассмотрен недетерминированный алгоритм суперкомпилятора HOSC, описывающий (в общем случае, бесконечное) множество остаточных программ. В четвертой главе показана корректность этого алгоритма.

Стоит отметить, что не каждая вариация недетерминированного алгоритма завершается. Вопросы завершаемости и корректности в суперкомпиляции в некотором смысле ортогональны. Мы разобрались с корректностью суперкомпилятора HOSC. Перейдем к завершаемости.

В **главе 5** "Схема суперкомпилятора HOSC" излагается несколько конкретных алгоритмов суперкомпилятора HOSC. В главе 6 доказывается их корректность.

Как было отмечено ранее, предыдущие работы по суперкомпиляции языков высшего порядка обходили стороной алгоритм нахождения тесного обобщения выражений со связанными переменными. В главе 5 описывается алгоритм нахождения тесного обобщения для двух любых HLL-выражений. Это первый случай полного и формального описания в литературе по суперкомпиляции алгоритма обобщения выражений со связанными переменными. Ключевую роль в построении алгоритма играют выбранные ранее синтаксические соглашения об именовании переменных

Рис. 7 Результат сравнения суперкомпиляторов

	$\mathbf{Sc}_{---}$	$\mathbf{Sc}_{-+-}$	$\mathbf{Sc}_{--+}$	$\mathbf{Sc}_{-++}$	$\mathbf{Sc}_{+--}$	$\mathbf{Sc}_{++-}$	$\mathbf{Sc}_{+-+}$	$\mathbf{Sc}_{+++}$
(1)	−	−	−	−	−	+	−	+
(2)	−	+	+	+	−	+	+	+
(3)	−	+	−	+	−	+	−	+
(4)	−	−	−	−	−	+	−	+
(5)	−	−	+	+	−	−	+	+
(6)	−	+	+	+	−	+	+	+
(7)	+	+	+	+	+	+	+	+

и допустимости подстановки по отношению к выражению.

Предыдущие работы при рассмотрении суперкомпилятора для языка высшего порядка лишь адаптировали классическое гомеоморфного вложение $\trianglelefteq$: аппликации, λ-абстракции и case-выражения рассматривались как специальные конструкторы и связанные переменные не различались. Как показали эксперименты, в случае языка высшего порядка, использование "наивного" гомеоморфного вложения, подобного $\trianglelefteq$, дает неудовлетворительные результаты в суперкомпиляторе, предназначенном для трансформационного анализа.

В главе 5 рассматривается вложение $\trianglelefteq^*$ – уточнение отношения $\trianglelefteq$, в котором различаются связанные переменные: не допускается вложение связанной переменой через погружение, связанная переменная может быть вложена только в соответствующую связанную переменную, аппликации рассматриваются как бинарные конструкторы. Затем вводится усиление вложения $\trianglelefteq^*$ – вложение $\trianglelefteq^{**}$, в котором аппликации рассматриваются как конструкторы разной арности (Рис. 6). Предложенное вложение обладает следующим свойством: если два HLL-выражения вложены через $\trianglelefteq_c^{**}$, то их обобщение – нетривиальное HLL-выражение.

Как было сказано ранее, для анализа программ может быть полезно иметь несколько результатов суперкомпиляции. В главе 5 описывается параметризованный алгоритм суперкомпилятора HOSC – SC_{ijk}. Алгоритм зависит от трех параметров:

1. i - Какое гомеоморфное вложение использовать: $\trianglelefteq_c^{**}$ $(*)$, $\trianglelefteq_c^{*}$ $(+)$ или $\trianglelefteq_c$ $(-)$?

2. j - Разделять ли узлы на глобальные и локальные при поиске релевантных кандидатов [7] $(+)$ или не разделять $(-)$?

3. k - Разделять ли выражения на классы в соответствие в типом редекса $(+)$ или нет $(-)$?

Все возможные варианты суперкомпилятора корректны, так как соответствуют отношению трансформации HOSC, корректность которого была показана в главе 4.

В некоторых работах, вышедших в последние годы, исследуется, как различные аспекты алгоритма суперкомпилятора влияют на качество оптимизации программ. Однако никто ранее не исследовал на практике как различные детали суперкомпилятора влияют на способность к трансформационному анализу.

Чтобы подчеркнуть важность учета связанных переменных, сравниваются восемь вариантов суперкомпилятора. Для сравнения используется модельная задача: доказательство эквивалентности выражений. Рассматриваемые тестовые примеры используют функции высших порядков и оперируют потенциально бесконечными данными. Результаты эксперимента, представленные на Рис. 7, показывают, что наилучшим сочетанием параметров является использование уточненного гомеоморфного вложения, разделение узлов на локальные и глобальные и разделение выражений на классы в соответствии с типом редекса, – именно такое сочетание параметров выбирается для использования в суперкомпиляторе HOSC (вместо вложения $\trianglelefteq_c^*$ используется его усиление – $\trianglelefteq_c^{**}$) [4].

Таким образом, в главе 5 завершено полное и формальное описание внутреннего устройства суперкомпилятора HOSC.

Осталось доказать, что суперкомпилятор HOSC завершается на любой входной программе. В **главе 6** "Завершаемость суперкомпилятора HOSC" доказывается завершаемость всех алгоритмов, представленных в главе 5.

Для доказательства завершаемости используется инструментарий *абстрактных преобразователей программ*, разработанный Сёренсеном [6], поскольку суперкомпилятор HOSC можно рассматривать как абстрактный преобразователь программ (частичных деревьев процессов).

Самой сложной частью любого суперкомпилятора является алгоритм обобщения, который гарантирует завершаемость суперкомпилятора на любой программе, предотвращая построение бесконечного дерева процессов. Самой сложной задачей является принятие решение о том, какое выражение нужно обобщить. В суперкомпиляции эта часть алгоритма обобщения исторически называется *свистком*.

Не только в суперкомпиляции, но и в других методах преобразования программ в качестве свистка хорошо себя зарекомендовало отношение гомеоморфного вложения. Свисток, основанный на гомеоморфном вложении, сравнивает выражение в текущем узле с выражениями в предках. Если свисток обнаруживает, что два выражения *синтаксически* похожи, то суперкомпилятор обобщает одно из двух выражений, чтобы предотвра-

[4] То есть под суперкомпилятором HOSC понимается суперкомпилятор SC_{*++}.

тить появление бесконечных ветвей в частичном дереве процессов. Таким образом, *существенным* свойством свистка является то, что он будет срабатывать на бесконечной ветви в дереве процессов. Говоря формально, свисток основан на *вполне-квазиупорядочении*.

Как было показано в первой главе (теорема 1), отношение $\trianglelefteq$ является вполне-квазиупорядочением на множестве $E_{C\cup F\cup V}$ SLL-выражений при конечных C и F. То же самое относится и к отношению $\trianglelefteq$ для выражений языка HLL. Однако, как было показано на модельной задаче, отношение $\trianglelefteq$ для HLL-выражений дает неудовлетворительные результаты при использовании суперкомпилятора для трансформационного анализа. Использование предложенного автором уточненного вложения $\trianglelefteq^{**}$, напротив, дает лучшие результаты.

Легко показать, что уточненное вложение $\trianglelefteq^{**}$ не является вполне-квазиупорядочением на множестве $E_{C\cup F\cup V}$ HLL-выражений при конечных C и F. Однако, для завершаемости суперкомпилятора достаточно, чтобы уточненное вложение $\trianglelefteq^{**}$ было вполне-квазиупорядочением не на всем множестве $E_{C\cup F\cup V}$ HLL-выражений, а на множестве выражений, которыми помечаются узлы частичного дерева процессов.

Теорема 3 (Ключников)**.** *Пусть $\mathcal{M}_P$ – множество HLL-выражений, возникающих в узлах частичного дерева процессов для программы P, строящегося по алгоритму на Рис. 5. В любой бесконечной последовательности HLL-выражений $e_1, e_2, \ldots$ из множества $\mathcal{M}_P$ найдутся e_i и e_j такие, что $i < j$ и $e_i \trianglelefteq^{**} e_j$.*

Доказательство теоремы 3 происходит в три этапа:

1. Вначале вводится в рассмотрение отношение $\trianglelefteq^1$, являющееся уточнением отношения $\trianglelefteq$ и допускающее вложение только соответствующих связанных переменных. Показывается, что с помощью кодировки связанных переменных индексами де Брюина, доказательство того, что $\trianglelefteq^1$ – вполне-квазиупорядочение на множестве $\mathcal{M}_P$, сводится к теореме 1.

2. Рассматривается отношение $\trianglelefteq^2$, являющееся уточнением отношения $\trianglelefteq^1$ и гарантирующее нетривиальное обобщения для e_1 и e_2, если $e_1 \trianglelefteq^2_c e_2$. Показывается, что с помощью расширенных индексов де Брюина и факта того, что $\trianglelefteq^1$ – вполне-квазиупорядочение, доказательство того, что $\trianglelefteq^2$ – вполне-квазиупорядочение на множестве $\mathcal{M}_P$, сводится к теореме 1.

3. Доказывается, что $\trianglelefteq^{**}$ – вполне-квазиупорядочение на множестве $\mathcal{M}_P$, – показываем, что в силу типизации исходной программы по Хиндли-Милнеру максимальная арность аппликации выражений в узлах дерева процессов ограничена.

Отношение $\trianglelefteq^{**}$ учитывает свойства связанных переменных, поэтому при использовании отношения $\trianglelefteq^{**}$ в качестве свистка учитывается не только синтаксическая похожесть конфигураций, но и семантическая похожесть, что позволяет в некоторых случаях увеличить глубину преобразований.

Из теоремы 3 следует завершаемость суперкомпилятора HOSC.

Таким образом в диссертации полностью и формально описан суперкомпилятор HOSC (главы 3, 5), доказана его корректность и завершаемость. То есть, суперкомпилятор HOSC удовлетворяет требованиям, предъявляемым к суперкомпилятору, предназначенному для трансформационного анализа (таблица на Рис. 3).

В **главе 7** "Распознавание эквивалентности выражений" описывается алгоритм автоматического распознавания эквивалентных выражений, основанный на следующей идее: если два выражения e_1 и e_2 преобразуются суперкомпилятором в синтаксически одну и ту же конструкцию, то отсюда можно сделать вывод об эквивалентности исходных выражений.

Пусть $A \Rightarrow_{sc} A'$ обозначает, что A' по смыслу эквивалентно A и может быть получено как результат суперкомпиляции A, или, другими словами, $\Rightarrow_{sc}$ есть "отношение суперкомпиляции" (в терминологии Климова [2]).

Пусть $\cong$ обозначает эквивалентность и $\equiv$ означает "равенство текстов". Верно следующее:

$$\frac{A \Rightarrow_{sc} A' \qquad B \Rightarrow_{sc} B' \qquad A' \equiv B'}{A \cong B}$$

Или, проще говоря, если в результате суперкомпиляции A и B получаются программы, совпадающие текстуально, то A и B – эквивалентны.

Таким образом, суперкомпиляцию можно рассматривать как преобразование, которое в некотором смысле нормализует выражения. Некоторые другие методы преобразования также можно рассматривать как нормализующие.

Общая идея доказательства эквивалентности через нормализацию является хорошо известной и является стандартной техникой в таких областях как, например, компьютерная алгебра. Идея использовать суперкомпиляцию для нормализации была высказана Лисицей и Вебстером [4] и была применена для доказательства эквивалентности программ на функциональном языке первого порядка при условии, что программы оперируют только конечными данными и гарантированно завершаются.

В главе 7 показано, что этот подход применим и к программам на языке высшего порядка, даже в случае, когда используются бесконечные структуры данных, и на некоторых входных значениях программа может не завершаться.

Доказательство эквивалентности двух выражений `t1` и `t2` можно свести к доказательству свойства одного выражения. А именно: если `equals`

– функция, проверяющая равенство двух значений, мы можем сконструировать выражение `equals t1 t2` и подвергнуть его суперкомпиляции, чтобы показать, что результатом его вычисления может быть только `True`.

Доказательство эквивалентности выражений, основанное на нормализации суперкомпиляцией, обладает по сравнению с подходом, основанным на равенстве, следующими преимуществами:

1. Не требуется дополнительно определять функцию `equals`, которую в некоторых случаях невозможно определить (например, для равенств функциональных значений).

2. Подход, основанный на нормализации, работает для бесконечных данных.

3. Вычисление `t1` или `t2` может не завершаться. К примеру, если `t1` и `t2` оперируют бесконечными структурами данных и никогда не завершаются, но тем не менее, могут быть эквивалентны (т.е. имеют один и тот же смысл в соответствии с семантикой языка).

Показано, что подход, основанный на нормализации, позволяет доказывать утверждения, которые даже невозможно сформулировать в терминах подхода, основанного на равенстве.

В качестве тестов для проверки способности суперкомпилятора HOSC распознавать эквивалентность выражений рассматриваются примеры из первой главы книги “Алгебра программирования” Ричарда Бёрда, где эквивалентность выражений доказывается ручными преобразованиями. Суперкомпилятор HOSC распознает эквивалентность всех 25 примеров в полностью автоматическом режиме. Стоит отметить, что если использовать в качестве свистка не уточненное вложение $\trianglelefteq^{**}$, а адаптацию классического вложения $\trianglelefteq$, то суперкомпилятор HOSC распознает лишь 6 эквивалентностей из 25. Это показывает плодотворность использования уточненного вложения $\trianglelefteq^{**}$.

В языке Haskell существуют требования к согласованности некоторых операций, которые определяются программистом при реализации монады. Монада – это некоторый алгебраический тип данных и набор функций для операций над значениями типов. Монада определяется следующими операциями (некоторые из них не являются обязательными): `return`, `join`, `bind`, `fmap`, `mzero`. Монадические операции, определяемые программистом, должны удовлетворять, среди прочего, следующим соотношениям:

1. $\forall a, k. join\ (return\ a)\ k \cong k\ a$

2. $\forall m, k, h. join\ m\ (\lambda x \to join\ (k\ x)\ h)\ \cong\ join\ (join\ m\ k)\ h$

3. $\forall f.join\ mzero\ f \cong mzero$

4. $\forall v.bind\ v\ mzero \cong mzero$

Компилятор языка Haskell не способен проверить выполнения этих соотношений для конкретной реализации монады.

Однако, эта задача сводится к распознаванию эквивалентности выражений. Во второй части главы 7 рассматриваются три монады из стандартной библиотеки языка Haskell – List, Maybe, State.

Суперкомпилятор HOSC распознал соотношения 1, 2 и 3 как эквивалентные для всех монад, распознал соотношение 4 как соотношение эквивалентности для монады State. Для монад Maybe и List из анализа остаточных программ, полученных при проверке 4-го соотношения, видно, что оно выполняется только для строгих v (вычисление v завершается и в случае списков представляет из себя конечную структуру).

Таким образом, мы показали *практическую применимость* подхода доказательства эквивалентности выражений с помощью суперкомпиляции.

Многие техники преобразований программ рассчитывают на определенном шаге трансформаций на леммы – простые и относительно легко доказываемые равенства – и применяют их для увеличения глубины преобразований программ.

В **главе 8** "Метод многоуровневой суперкомпиляции" описывается новый метод многоуровневой суперкомпиляции, основанный на применении *улучшающих лемм*.

Недостижимая в большинстве случаев цель суперкомпиляции – построить идеальное дерево процессов [1]. Частичное дерево процессов не является идеальным, если при его построении были произведены обобщения. Отсюда представляется логичным пытаться применять леммы, чтобы избежать обобщения.

На примерах показано, как возможности суперкомпиляции могут быть расширены за счет применения лемм (замены выражений на эквивалентные выражения). С другой стороны, суперкомпилятор сам может быть использован для распознавания эквивалентности двух выражений. Тогда, применяя к этому принцип метасистемного перехода [10], как следствие получаем идею многоуровневой суперкомпиляции: построим башню суперкомпиляторов, – суперкомпиляторы верхнего уровня управляют суперкомпиляторами, расположенными ниже, чтобы получить (и затем применить) леммы. Леммы запрашиваются, когда у суперкомпилятора верхнего уровня обнаруживается ситуация гомеоморфного вложения конфигураций – леммы применяются для избежания обобщения. В главе 8 описан алгоритм такого суперкомпилятора в общем виде.

Однако, в общем случае суперкомпилятор, применяющий леммы, может выдавать некорректную остаточную программу. Дэвидом Сэндсом

Рис. 8 Алгоритм распознавания улучшающих лемм

$$\frac{m \geq n \qquad \forall i : e_i \rhd^*_\sim e'_i}{{}^m\phi(e_1, \ldots, e_k) \rhd^*_\sim {}^n\phi(e'_1, \ldots, e'_k)} \qquad \frac{\mathcal{SC}[\![e_1]\!] \equiv \mathcal{SC}[\![e_2]\!] \qquad \mathcal{SC}[\![e_1]\!] \rhd^*_\sim \mathcal{SC}[\![e_2]\!]}{e_1 \trianglerighteq_s e_2}$$

было показано [5], что замена выражения e_1 на эквивалентное ему выражение e_2 не нарушает корректность преобразования, если выражение e_2 является строгим улучшением выражения e_1. Для удобства будем называть упорядоченную пару таких выражений улучшающей леммой.

Определение 4 (Улучшающая лемма)**.** Упорядоченная пара (e_1, e_2) является улучшающей леммой, $e_1 \trianglerighteq_s e_2$, если выражение e_1 операционно эквивалентно выражению e_2, $e_1 \cong e_2$, и если в любом контексте C, таком, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, если вычисление выражения $C[e_1]$ завершается за n вызовов функций, то вычисление выражения $C[e_2]$ завершается не более чем за n вызовов функций.

Сэндсом было показано, что применение улучшающих лемм дает корректные результаты.

В общем случае задача проверки, является ли пара выражений улучшающей леммой, алгоритмически неразрешима. Однако, с помощью незначительной модификации суперкомпилятора HOSC можно автоматически распознавать некоторые классы улучшающих лемм, представляющих практический интерес.

Достаточно пометить специальным образом дуги частичного дерева процессов, на которых происходит развертка вызова функции. Эта информация легко переносится в остаточную программу в виде аннотаций-звездочек. Концептуально это означает, что с помощью небольшой модификации мы переносим информацию о скорости выполнения исходной программы в остаточную программу.

Вводим простое отношение[5] $\rhd^*_\sim$ на множестве аннотированных выражений (Рис. 8) – здесь ϕ работает как функтор и обозначает любую конструкцию языка.

Доказывается следующая теорема:

Теорема 5 (Распознавание улучшений)**.** *Пусть* $e'_1 = \mathcal{SC}[\![e_1]\!]$ *и* $e'_2 = \mathcal{SC}[\![e_2]\!]$*. Если* $e'_1 \equiv e'_2$ *и* $e'_1 \rhd^*_\sim e'_2$*, то* $e_1 \trianglerighteq_s e_2$

На базе суперкомпилятора *HOSC* и применении улучшающих лемм разработан двухуровневый модельный суперкомпилятор *TLSC*. На мо-

[5] аналогичное гомеоморфному вложению

дельных примерах суперкомпилятор $TLSC$ сравнивается с суперкомпилятором $HOSC$. Показано, что суперкомпилятор $TLSC$ способен строить перфектные частичные деревья процессов там, где суперкомпилятор $HOSC$ делает обобщение.

Сёренсен показал, что позитивный суперкомпилятор для языка с семантикой вычисления вызов по имени не может улучшить асимптотику программ. На примерах показано, что суперкомпилятор $TLSC$ способен *улучшать асимптотику* программ.

Результаты

В данной работе были получены следующие результаты:

- На основе существующих алгоритмов суперкомпиляции для функциональных языков первого порядка был разработан новый алгоритм суперкомпиляции для функционального языка высшего порядка, учитывающий свойства связанных переменных:
  - Сформулировано уточненное отношение гомеоморфного вложения.
  - Расширен алгоритм нахожения тесного обобщения.
  - Доказаны корректность и завершаемость алгоритма.
- Разработанный алгоритм реализован в экспериментальном суперкомпиляторе HOSC для языка Haskell. Суперкомпилятор HOSC является первым суперкомпилятором языка Haskell, для которого формально доказаны теоремы корректности и завершаемости.
- Разработан алгоритм распознавания эквивалентности выражений на основе синтаксического сравнения остаточных программ. Этот алгоритм реализован в суперкомпиляторе HOSC и работает в полностью автоматическом режиме.
- Предложен и реализован алгоритм распознавания улучшающих лемм.
- Предложен новый метод многоуровневой суперкомпиляции, основанный на применении улучшающих лемм для избежания обобщения. Метод был реализован в двухуровневом суперкомпиляторе TLSC. Показано, что суперкомпилятор TLSC способен улучшать асимптотику программ.
- Показана применимость разработанных методов для решения ряда задач по выявлению и доказательству свойств программ, в частности произведена проверка корректности реализации монад для ряда классов из стандартной библиотеки языка Haskell.

Список литературы

[1] *Glück R., Klimov A. V.* Occam's razor in metacompuation: the notion of a perfect process tree // WSA '93: Proceedings of the Third International Workshop on Static Analysis. — Springer, 1993. — Pp. 112–123.

[2] *Klimov A. V.* A program specialization relation based on supercompilation and its properties // First International Workshop on Metacomputation in Russia. — Ailamazyan University of Pereslavl, 2008. — Pp. 54–77.

[3] *Lisitsa A., Nemytykh A.* Verification as a parameterized testing (experiments with the SCP4 supercompiler) // *Programming and Computer Software*. — 2007. — Vol. 33, no. 1. — Pp. 14–23.

[4] *Lisitsa A., Webster M.* Supercompilation for equivalence testing in metamorphic computer viruses detection // First International Workshop on Metacomputation in Russia. — Ailamazyan University of Pereslavl, 2008. — Pp. 113–118.

[5] *Sands D.* Total correctness by local improvement in the transformation of functional programs // *ACM Trans. Program. Lang. Syst.* — 1996. — Vol. 18, no. 2. — Pp. 175–234.

[6] *Sørensen M. H.* Convergence of program transformers in the metric space of trees // Mathematics of Program Construction. — Vol. 1422 of *LNCS*. — Springer, 1998. — Pp. 315–337.

[7] *Sørensen M. H., Glück R.* Introduction to supercompilation // Partial Evaluation. Practice and Theory. — Vol. 1706 of *LNCS*. — Springer, 1998. — Pp. 246–270.

[8] *The GHC Team.* Haskell 2010 language report. — `http://haskell.org/definition/haskell2010.pdf`. — 2010.

[9] *Turchin V. F.* The concept of a supercompiler // *ACM Transactions on Programming Languages and Systems (TOPLAS)*. — 1986. — Vol. 8, no. 3. — Pp. 292–325.

[10] *Turchin V. F.* Metacomputation: Metasystem transitions plus supercompilation // Partial Evaluation. — Vol. 1110 of *LNCS*. — Springer, 1996. — Pp. 481–509.