ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М.В. КЕЛДЫША
РОССИЙСКОЙ АКАДЕМИИ НАУК

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

Диссертация на соискание учёной степени кандидата физико-математических наук

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

Москва 2010

Оглавление

Введение

Объект исследования и актуальность работы

Компьютерные программы часто содержат ошибки. В современном программирование при написании программы избежать ошибок практически невозможно. Размер промышленных программ начинается от десятков тысяч строк кода, а большие промышленные системы достигают миллионов строк кода. Мысленно охватить работу больших систем ни один человек не в состоянии. Сложность разрабатываемого программного обеспечения подошла к границе его понимания и, следовательно, управляемости.

Материальные последствия ошибок программного обеспечения могут быть весьма значительными, порой приводящими к полному провалу дорогостоящих проектов и потере репутации.

Ошибки в сложных программных и аппаратных системах не являются чем-то особенным – они регулярно появляются в сложных системах. Их причинами являются и некорректные спецификации, и неправильное понимание спецификаций разработчиками, непредвиденные условия работы, несогласованность и многое другое. Правильность и корректность поведения систем являются более важными свойствами, чем производительность, модифицируемость, скорость разработки и т.д.

> *Существует потребность в инструментах анализа программ на предмет ошибок и соответствие спецификации.*

Наиболее очевидным и широко распространенным методом проверки правильности систем является тестирование - проверка построенной

системы в различных ситуациях, при различных исходных данных. Наиболее распространено модульное тестирование (unit testing). Инструменты для модульного тестирования существуют практически для любого языка программирования [38]. При модульном тестировании рассматривается некоторый компонент, например некоторый модуль (функция) f, принимающий на вход некоторый параметр X. Результатом работы модуля являются некоторые данные Y:

$$Y = f(X)$$

Проверяется, обладает ли ответ Y предполагаемыми свойствами, заложенными в систему. Часто проверка этих свойств кодируется на том же самом языке, на котором написана программа, в виде предиката (функции) p и считается, что на данных входных значениях модуль f работает корректно, если предикат верен:

$$Y = f(X)$$
$$p(Y)$$

Или:

$$p(f(X))$$

Если при тестировании проверяется работа модуля на входных параметрах $X_1, X_2, \ldots, X_n$, то, соответственно, рассматривается выполнение предикатов

$$p(f(X_1)), p(f(X_2)), \ldots, p(f(Xn))$$

Зачастую предикат может сам зависеть от значений входного параметра:

$$p(X, f(X))$$

Тогда рассматриваются выполнения предикатов

$$p(X_1, f(X_1)), p(X_2, f(X_2)), \ldots, p(X_n, f(X_n))$$

Тестирование имеет множество преимуществ:

- Рассматривается реальная система.
- Проверка может выполняться в реальной среде с реальными интерфейсами.
- Проверять можно наиболее опасные или часто используемые режимы работы системы.
- Программист сам пишет тесты на том же языке, на котором написана программа.

В то же время у тестирования есть и недостатки:

- Тестированием можно проверить лишь немногие траектории вычисления системы (их обычно бесконечное множество).
- Тестированием сложно проверить редко выявляющиеся ошибки, особенно ошибки в системах реального времени.
- Тестирование не может гарантировать правильность системы: “тестированием можно доказать только наличие ошибок”. (Дейкстра).

Другим подходом к проверке корректности программ является формальная верификация продукта. Формальная верификация программ – приемы и методы формального доказательства (или опровержения) того, что программа удовлетворяет заданной формальной спецификации. Одним из распространенных методов формальной верификации программ является проверка на моделях программ (model checking) [18]. Доказать, что конкретная реализация продукта (программа) удовлетворяет некоторым формальным требованиям очень сложно, поэтому при проверке на моделях проверяется не сама программа, а ее формальная модель. Формальная модель строится вручную. Обычно такая модель значительно проще проверяемой системы, – это абстракция, которая отражает наиболее существенные характеристики системы. Как и в случае тестирования, при проверке на моделях при описании условий корректности записывается некоторый предикат (логическая формула) относительно модели, и проверяется, будет ли заданная формула выполняться всегда на данной

модели. Соответственно, есть различные языки описаний моделей и логических формул (темпоральные логики, структуры Крипке, бинарные решающие диаграммы, и т.д.).

Проверка на моделях программ обладает следующими преимуществами.

- Полнота. Для многих формализаций проверка на моделях программ либо доказывает, что модель всегда удовлетворяет формуле, либо находит ошибку.
- Проверка на моделях программ может находить логические ошибки, так как работает с моделью программы, а не с реализацией.
- Можно проверять модель программы еще до написания самой программы.

Однако и у формальной верификации есть ряд недостатков:

- Проверяется не сама программа, а ее модель. Такая модель может быть неадекватной. Таким образом, ценность верификации зависит от адекватности модели. Модель - некоторое неисполняемое представление исполняемой программы. Модель в некотором смысле чужда программисту.
- Язык спецификации может быть неполным, недостаточным для формулировки всех требований.
- Процесс верификации автоматизирован, но модель программы пишет человек. Требуется очень высокая квалификация персонала, чтобы изготовить адекватную модель.

Существуют методы, в которых модель программы (вычислений) строится автоматически, – несущественные с точки зрения системы детали отбрасываются, и затем происходит моделирование вычислений. Здесь уместно упомянуть такие методы как символьное выполнение [55] и абстрактную интерпретацию [20]. В некоторых случая утверждения о корректности программ записываются практически на том же самом языке, что и верифицируемая программа [131].

Как было отмечено, главным недостатком тестирования предиката $p(X, f(X))$ при конкретных условиях

$$p(X_1, f(X_1)), p(X_2, f(X_2)), \ldots, p(X_n, f(X_n))$$

является неполнота. Успешное выполнение тестов не гарантирует отсутствия ошибок.

Предикат p пишется обычно (и это правильно для тестирования) без учета внутренней структуры программы f. Если приглядеться к проверяемому выражению $p(X, f(X))$, то легко увидеть, что оно является композицией двух функций – предиката и проверяемой функций f. Существуют методы преобразований программ, способные упрощать композицию двух функций. В случае выражения $p(X, f(X))$ преобразование специализирует предикат p для конкретной функции f. Результатом преобразования будет некоторая функция p'

$$p'(X) = \ldots,$$

зависящая только от входного параметра X, но учитывающая особенности композиции конкретного предиката p и конкретной функции f. Если преобразованная программа обладает более простой и ясной структурой по сравнению с исходными программами p и f, то с помощью очень простого анализа текста программы удается показать, что преобразованная программа выдает только $True$, или найти такие входные параметры X, когда она выдает $False$.

Одним из методов преобразований программ, способных упрощать композицию функций, является суперкомпиляция. Одним из успешных применений суперкомпиляции для проверки корректности программ является верификация с помощью суперкомпиляции реализаций (на языке РЕФАЛ) ряда протоколов кэш-когерентности, выполненная Андреем Немытых [71] (при этом, в некоторых протоколах удалось найти ошибки). Реализация протокола принимала в качестве входа конечную цепочку команд, и проверялось состояние памяти после выполнения этих команд. Требования, которым должна удовлетворять реализация протокола, кодировались в виде предиката, проверяющего состояния ячеек.

Суть трансформационного анализа программ можно сформулировать следующим образом: вместо того, чтобы анализировать исходную программу, эта программа вначале преобразуется в эквивалентную ей программу, и затем анализируется уже преобразованная программа. Если используемый метод преобразования программ способен устранять многие избыточности исходной программы, то в некоторых случаях анализ остаточной программы становится тривиальным [127, 40].

Подчеркнем, что если утверждения о корректности программы (спецификация) закодированы на том же языке, что и сама программа, то можно преобразовывать программу с учетом спецификации, – больше шансов на то, что несущественные с точки зрения спецификации детали реализации будут отсутствовать в остаточной программе. Для плодотворности такого подхода необходимо, чтобы рассматриваемый язык программирования обладал достаточными изобразительными средствами для написания высокоуровневых спецификаций и одновременно позволял осуществлять глубокие преобразования программ с сохранением семантики.

Функциональный язык программирования Haskell можно рассматривать как язык исполняемых спецификаций [46]. Программы на языке Haskell имеют, как правило, простую и ясную структуру и в то же время достаточно хорошо поддаются упрощающим преобразованиям (с сохранением семантики). Семантика языка Haskell детально формализована [110]. Язык Haskell обладает мощными изобразительные средствами, которые облегчают написание на нем спецификаций:

- Функции высших порядков.
- Бесконечные структуры данных.
- Автоматический вывод и проверка типов.

Таким образом, существуют предпосылки для трансформационного анализа программ, написанных на языке Haskell.

> *В данной работе исследовались возможности применения суперкомпиляции для трансформационного анализа программ на языке Haskell.*

Цели и задачи работы

Исторически главной целью суперкомпиляции была оптимизация программ. Работы последних лет по созданию суперкомпиляторов также ориентированы на использование суперкомпиляции как средства оптимизации программ.

Целью данной работы является изучение возможностей применения суперкомпиляции для трансформационного анализа программ.

Стоит отметить, что 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! Если исходная успешно программа завершается на некоторых входных данных и выдает осмысленный результат, то, несомненно, оптимизированная программа также должна завершаться и выдавать тот же результат. Однако, если исходная программа не завершается, или завершается аварийно, оптимизированной программе позволительно завершаться и выдавать некоторый произвольный результат (особенно, если это позволяет ускорить программу или уменьшить ее размер). Например, суперкомпилятор SCP4 [71, 72] часто осуществляет преобразования функций с расширением области определения.

Если же некоторый метод преобразования программ используется не для оптимизации программ, а для анализа программ, то не предполагается выполнения преобразованных программ. Таким образом, размер и скорость выполнения преобразованной программы больше не играют главной роли. В частности, при преобразованиях позволительно дублировать код. Например, следующее выражение

```
let
  p = f x y
in
  g p q p r
```

определенно можно преобразовать в

```
g (f x y) q (f x y) r
```

С другой стороны, желательно, чтобы преобразованная программа имела тот же смысл (ту же семантику), что и исходная программа, когда метод преобразования программ используется для анализа.

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.* Иначе сложно распознавать ситуации, когда для суперкомпилятора требуется дополнительное время и когда суперкомпилятор зациклился.

- *Иметь доступный исходный код.* Иначе нет возможности формально убедиться в корректности реализации суперкомпилятора.

Как показал анализ положения дел в суперкомпиляции (см. следующую главу), такого суперкомпилятора на момент начала диссертационной работы (2007 год) не существовало[1]. В тоже время, как отмечено в предыдущем разделе, существуют предпосылки для трансформационного анализа программ, написанных на языке Haskell.

Поэтому автор поставил перед собой следующие задачи:

1. Разработать метод суперкомпиляции функциональных программ, ориентированный на трансформационный анализ.

2. Разработать на базе метода 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

Структура суперкомпилятора, предложенная в работе, допускает более глубокие преобразования программ, чем методы суперкомпиляции, ориентированные на оптимизацию, с помощью двух приемов:

1. Рассматривается операционная семантика вызова по имени (call-by-name), а не семантика вызова по необходимости (call-by-need)[2]. Непосредственно перед суперкомпиляцией все локальные определения (`let`-выражения) поднимаются на верхний уровень методом λ-лифтинга, что позволяет существенно упростить дальнейший конфигурационный анализ и агрессивно распространять информацию о текущей конфигурации вниз по дереву процессов.

2. Частичное дерево процессов, напротив, преобразуется в программу без глобальных определений, – одно самодостачное выражение, где рекурсивные функции определяются в том же месте, где и используются. Вместе с первым приемом это позволяет приводить одинаковые по смыслу, но текстуально разные программы к одной и той же синтаксической форме.

В отличие от других работ, где суперкомпилятор рассматривается как фиксированная монолитная конструкция, в данной работе вначале определяется структура суперкомпилятора HOSC в виде отношения трансформации HOSC, – фиксируются *методы* (без деталей реализации) построения частичного дерева процессов и преобразования частичного дерева процессов в остаточную программу, доказывается корректность отношения трансформации HOSC. То есть доказывается корректность не конкретного алгоритма, а *множества* алгоритмов, удовлетворяющих отношению трансформации HOSC.

Доказательство корректности отношения трансформации HOSC опирается на операционную теорию улучшений Дэвида Сэндса. Стандартный способ доказать корректность трансформации – показать, что остаточная программа является улучшением исходной (в терминологии Сэндса).

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

В работе показана корректность трансформации HOSC в общем случае – включая те случаи, когда остаточная программа не является улучшением (в терминологии Сэндса) исходной программы.

Все определенные далее алгоритмы суперкомпиляции удовлетворяют отношению трансформации HOSC и, следовательно, корректны.

Язык Haskell является функциональным языком высшего порядка. На момент начала диссертационной работы методы и алгоритмы суперкомпиляции для языков высшего порядка были слабо разработаны, – в существовавших работах использовались алгоритмы суперкомпиляции для языка первого порядка, *адаптированные* для языков высшего порядка, не учитывающие всех синтаксических и семантических особенностей языков высших порядков.

В данной работе при разработке алгоритмов для суперкомпиляции ядра языка Haskell была произведена ревизия классических алгоритмов суперкомпиляции с учетом работы с функциями высших порядков:

1. Сделана ревизия обработки ситуации зацикливания, – использование λ-абстракций и функций как данных позволяет писать рекурсивные “функции” без явного использования рекурсии. Как результат во время построения дерева процессов *необходимо* чаще делать проверку на возможное зацикливание.

2. В выражениях присутствуют связанные переменные: сделаны ревизии гомеоморфного вложения и обобщения, учитывающие свойства связанных переменных.

Классическое гомеоморфное вложение является вполне-квазиупорядочением на всем множестве выражений языка при использовании конечного числа конструкторов и функций. Уточненное гомеоморфное вложение не является вполне-квазиупорядочением на всем множестве выражений языка. Однако, как показано в работе, уточненное гомеоморфное вложение является вполне-квазиупорядочением множестве выражений, возникающих при построении дерева процессов, что достаточно для доказательства завершаемости алгоритмов суперкомпилятора HOSC.

В результате выбранные методы и алгоритмы обеспечивают способность суперкомпилятора HOSC к нормализации выражений, что позво-

ляет свести задачу распознавания эквивалентных программ к автоматическому распознаванию синтаксической эквивалентности текстов преобразованных суперкомпилятором программ.

В результате обобщения в дереве процессов появляются лишние трассы и как результат в остаточной программе появляется недостижимый код, который, как правило, затрудняет анализ остаточной программы. Чем меньше в остаточной программе недостижимого кода, тем легче она поддается анализу. Цель предложенного метода многоуровневой суперкомпиляции – получить в результате суперкомпиляции программу с как можно меньшим объемом недостижимого кода. Средство достижения цели – избежать (или как минимум отложить) обобщения конфигураций ("уход из под свистка") с помощью замены конфигурации, вынуждающей суперкомпилятор сделать обобщение на эквивалентную ей, которая позволяет продвинуться дальше без обобщений. Эквивалентные конфигурации (для замены) распознаются методом нормализацией через суперкомпиляцию. Корректность многоуровневой суперкомпиляции обеспечивается тем, что конфигурация может заменяться только на "улучшение" (в терминологии Сэндса).

Предложен алгоритм распознавания улучшающих лемм, основанный на аннотировании остаточной программы дополнительной информацией о том, насколько быстро исполнялась исходная программа. В результате задача распознавания улучшающих лемм сводится к простому синтаксическому сравнению двух остаточных программ.

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 распознавание эквивалентных выражений методом нормализации суперкомпиляцией может происходить в полностью автоматическом режиме. Распознавание улучшающих лемм с помощью суперком-

пиляции происходит также в полностью автоматическом режиме.

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.

Глава 1

Позитивная суперкомпиляция и анализ программ

В данной работе исследуется возможности применения суперкомпиляции для трансформационного анализа программ.

Суперкомпиляция является одним из методов *метавычислений*. Теория метавычислений занимается построением конструктивных и эффективных метапрограмм, способных осуществлять вычисление, преобразование и анализ программ.

Метапрограмма (суперкомпилятор) является метасистемой над программами. Процитируем В.Ф. Турчина:

> Представим себе некоторую систему S. И представим, что какое-то число систем S или систем типа S соединены в единое целое и снабжены вдобавок какой-то управляющей ими системой. Образованную таким образом систему $S+$ назовем *метасистемой* по отношению к системе S. Метасистема $S+$ содержит ряд систем S в качестве своих подсистем и содержит также средства, позволяющие *управлять* этими подсистемами – в самом широком смысле: координировать их работу, модифицировать их, генерировать и т. д. Переход от системы S к метасистеме $S+$ мы называем *метасистемным переходом*.

В книге “Феномен науки” [124] утверждается, что метасистемный переход является квантом эволюции и любое качественное изменение явля-

ется метасистемным переходом. Таким образом, предметом рассмотрения метавычислений является автоматическая эволюция программ.

1.1 Исторический обзор

В данном кратком обзоре мы затрагиваем только работы, посвященные суперкомпиляции и оставляем в стороне работы, посвященные другим методам метавычислений.

1.1.1 Суперкомпиляция Рефала

Творческое наследие В.Ф. Турчина по суперкомпиляции (а более обще – по метавычислениям) необычайно многогранно. Как показали дальнейшие исследования, в работах В.Ф. Турчина рассыпано множество плодотворных идей, которые требует переосмысления. Многие концепции описаны недостаточно формально или частично, и чаще всего в терминах языка Рефал. Чрезвычайно интересно отделить эти идеи от языка Рефал и рассмотреть их в контексте современных языков программирования. Эти концепции требуют дальнейшей разработки и ждут своего исследователя. Поскольку для Турчина суперкомпиляция была неразрывно связана с языком Рефал, в данном разделе мы рассматриваем только основные вехи работы В.Ф. Турчина по суперкомпиляции, не затрагивая связь с Рефалом.

Первым шагом на пути к метасистемным переходам в программировании, по словам Турчина [125], явилось создание языка программирования, предназначенного для определения семантик других языков. Первая версия такого языка называлась метаалгоритмическим языком [136, 138], который затем стал называться Рефал [137]. Первый эффективный интерпретатор языка Рефал был реализован в 1968 [141]. Компиляторы с языка Рефал были созданы для большинства вычислительных машин того времени. Библиография работ, посвященных разработке и использованию Рефала насчитывает более двухсот единиц [125].

В 1972 году выходят работы, описывающие преобразование Рефал-программ [139, 140], в частности, в работе [140] описывается *прогонка*, – главная составляющая суперкомпиляции.

В конце 70-х - начале 80-х выходят публикации, описывающие идею суперкомпиляции [112, 114].

В 1982 эксперименты с суперкомпилятором были описаны Турчиным и его коллегами [123].

В 1986 году выходят статьи, описывающие в подробностях прогонку [116, 115].

В работе [117] Турчин формулирует основания математики в терминах теории метасистемных переходов.

В 1988 году выходит работа, посвященная проблеме завершаемости суперкомпилятора, и описывается автоматический алгоритм обобщения [118].

В 1989 году Турчин (совместно с Робертом Глюком) добился самоприменения суперкомпилятора [36].

В 1990 году Турчин (совместно с Робертом Глюком) в работе [37] продемонстрировал, что суперкомпилятор способен автоматически генерировать из наивного распознавателя подстроки в строке КМР-алгоритм.

В 1993 Турчин описывает расширение суперкомпиляции [119]. В этом расширении суперкомпилятор применяется не напрямую к функции, а к метафункции (интерпретатору, который вычисляет функцию по ее определению и абстрактным данным).

В отчете [122] описывается работа с метакодом, поднятыми переменными[1] и метасистемными прыжками, в отчете также вводится язык MST-схем[2]. Рассмотрение плоского[3] Рефала и метасистемных прыжков позволяет добиться самоприменения суперкомпилятора [84].

В отчете [120] описываются различия между списками (данными языка LISP) и строками (данными языка Рефал) с точки зрения распознавания вложения и обобщения.

В работе [121] описывается ориентированный на программирование суперкомпилятора язык SPCL[4]. Язык SPCL позволяет более ясно и модульно разрабатывать суперкомпиляторы.

[1] elevated variables
[2] MST = MetaSystem Transition
[3] Разрешается использовать только хвостовую рекурсию
[4] a supercompilation language

В [125] рассматривается совмещение суперкомпиляции и многократных метасистемных переходов.

В работе [83] продемонстрированы эксперименты по увеличению возможностей суперкомпилятора методом, описанным в [119], – добавить интерпретатор между суперкомпилятором и преобразуемой программой.

Суперкомпилятор SCP4 [80, 82] для языка Рефал-5, использует свойства языка Рефал (ассоциативность конкатенации) и помимо методов суперкомпиляции включает дополнительные инструменты: распознавание частично рекурсивных константных функций, распознавание частично рекурсивных мономов конкатенации, нахождение и анализ выходных форматов. Мономы конкатенации описываются в [81]. Стоит отметить, что суперкомпилятор SCP4 может расширять область определения программы в следующем смысле: если на каком-то входном данном исходная программа завершалась с ошибкой или зацикливалась, то преобразованная программа может завершаться и выдавать некоторое значение.

В работах [69, 71, 70, 72] задача верификации рассматривается как задача параметризованного тестирования, и показывается, как параметризованное тестирование может быть осуществлено средствами суперкомпиляции.

1.1.2 Суперкомпиляция функциональных языков первого порядка

Изначально суперкомпиляция была неотделима от языка Рефал и формулировалась в его терминах.

Рефал является чистым функциональным языком с сопоставлением по образцу. Для операций с символьной информацией Рефал использует *R-выражения*. Исследователи отмечают, что по сравнению с другими функциональными языками в языке Рефал образец определяется достаточно сложным образом, одной из причин этой сложности является неортогональность образцов, – алгоритм сопоставления с образцом зависит от порядка их перечисления. Как следствие, алгоритмы сопоставления с образцом и любые метаалгоритмы, рассматривающие Рефал-программы как входные данные, сложно сформулировать и объяснить.

К сожалению, как отмечает Сёренсен, механизмы конкретизации и

развертки, лежащие в основе прогонки, сильно зависят от сопоставления с образцом. Таким образом, даже прогонка для Рефала формулировалась достаточно сложным образом, не говоря об обобщении. Также, ни в одной работе, посвященной суперкомпиляции Рефала, алгоритм суперкомпиляции не приведен полностью. По этим причинам, несмотря на значительное количество опубликованных работ, к началу 1990-х суперкомпиляция не обрела признания за пределами узкого круга экспертов. Более того, практически все основные составляющие суперкомпиляции описывались без должной доли формализма – зачастую неформально и расплывчато,

По мнению многих (в том числе западных) исследователей, основным трудом, описывающим идеи суперкомпиляции является статья Турчина 1986 года “The concept of a supercompiler” [116]. Статья обобщает идеи суперкомпиляции в достаточно сжатом виде. И, к сожалению, затрудняет восприятие из-за отсутствия хорошо проработанной терминологии (и формализации) для представления новых понятий[5].

В упомянутой классической работе Турчина [116] используется достаточно сложный язык для представления конфигураций (в силу сложности алгоритмов сопоставления с образцом в Рефале). Из-за неортогональности образцов сильно усложняется и конфигурационный анализ, – из графа конфигураций сложно вычленить конфигурацию как таковую, – конфигурация определяется не только путем от начального узла до текущего узла, – необходимо учитывать и структуру всего графа (следствие того, что важен порядок задания образцов). В результате определяется достаточно сложный *обобщенный алгоритм сопоставления с образцом*, лежащий в основе прогонки.

Работа [33] Андрея Климова и Роберта Глюка о сущности прогонки рассматривала в качестве входного языка S-Graph, который можно рассматривать как подмножество языка программирования LISP, – по сути, эта была одна из первых работ, нацеленных на понимание суперкомпиля-

[5]Например, в терминологии Турчина в результате прогонки получается граф состояний и переходов. В результате суперкомпиляции также получается граф состояний и переходов. Дальнейшие исследователи более тщательно проработали терминологию, – в современных понятиях в результате прогонки получается *дерево процессов*, а целью суперкомпиляции является превращение потенциально бесконечного дерева процессов в конечное *частичное дерево процессов* (иногда таже называемое графом конфигураций).

ции как общего метода – без привязки к языку Рефал. В работе Климова и Глюка показано, что если взять язык с более очевидным сопоставлением с образцом (полный интерпретатор языка S-Graph занимает треть страницы[6]), то работа с конфигурациями сильно упрощается, – (1) можно вычленить рассмотрение конфигурации (обобщенного состояния) от дерева процессов, (2) можно достаточно просто определить язык для конфигураций и (3) обобщенный алгоритм сопоставления с образцом становится тривиальным. В работе явным образом разделяются понятия дерева процессов и графа процессов (частичного дерева процессов). Статья [33] является первой работой, где прогонка и суперкомпиляция (без обобщения) описаны формально (в виде алгоритмов на языке Haskell). Конфигурация представлена как место в программе (program point) и обобщенная среда. Обобщенная среда состоит из связей (позитивной информации) и рестрикций (негативной информации, ограничений на переменные).

В магистерская диссертация Морте Х. Сёренсена [103], рассматривается простой функциональный язык первого порядка (язык Miranda без функций высшего порядка с семантикой вызова по имени). В работе Сёренсена впервые целиком и формально описываются *все* составляющие суперкомпиляции – прогонка, обобщение, генерация остаточной программы, и приводятся доказательство корректности суперкомпилятора и доказательство его завершаемости на любой входной программе. Особое внимание уделяется языку M_0, в котором при описании условий должны быть разобраны все случаи (нет if-выражений). В этом случае понятие негативной информации не имеет смысла, и достаточно распространять только позитивную информацию. По сравнению с работой Климова и Глюка конфигурации представляются еще более простым образом – конфигурация является просто выражением языка со свободными переменными. Также в работе Сёренсена дается хороший обзор исторических взаимосвязей с другими методами метавычислений. Сёренсоном рассматриваются языки $M_{1/2}$ и M_1, и суперкомпиляция описывается для них только с распространением позитивной информации, – чтобы сохранить простоту конфигурационного анализа. За таким видом суперкомпиляции закрепилось название позитивная суперкомпиляция.

[6]Сложно представить себе интерпретатор языка Рефал такого размера

Работы [33] и [103] открывают целую серию работ, направленных на лучшее понимание суперкомпиляции и ее связь с другими методами метавычислений (преобразований программ).

В статьях [31, 32] иллюстрируется применение интерпретационного подхода, описанного Турчиным в работе [119]: показано, как с помощью добавления слоя интерпретации можно из частичного вычислителя получить дефорестатор и суперкомпилятор.

В работе [34] на основе анализа определений и принципов построения показывается, что частичная дедукция логических программ аналогична прогонке функциональных программ и обе техники имеют общий принцип – распространение информации.

В статье [48] прогонка рассматривается с фундаментальной точки зрения – с точки зрения семантики и без привязки к конкретному языку программирования и структурам данным.

В работе [106] суперкомпиляция сравнивается с частичными вычислениями, дефорестацией и обобщенными частичными вычислениями с точки зрения количества информации, накапливаемой и используемой во время преобразований.

В статье [104] подробно описывается использование гомеоморфного вложения как синтаксического критерия для принятия решения о необходимости обобщения конфигураций. Использования отношения гомеоморфного вложения сравнивается со стековым обобщением, предложенным в [118].

Работа 1996 “A positive supercompiler” [107] обобщает суть позитивной суперкомпиляции – рассматривается модельный суперкомпилятор для простого функционального языка первого порядка. Рассматривается KMP-тест – автоматический вывод эффективного алгоритма нахождения подстроки в строке из наивного алгоритма.

Статья 1996 года “A Roadmap to Metacomputation by Supercompilation” [35] рассматривает суперкомпиляцию в перспективе метавычислений и три задачи метавычислений: специализацию программ, композицию программ и инверсное вычисление программ. Также суперкомпиляция сравнивается другими методами преобразования программ.

Работа 1998 “Introduction to Supercompilation” [105] помимо введения

в суперкомпиляцию обозначивает, что разбиение узлов на локальные и глобальные и последующая обработка такого разбиения повышает глубину преобразования программ и во многих случаях помогает избежать слишком поспешного обобщения.

В статье 1998 года “Convergence of Program Transformers in the Metric Space of Trees” [108] рассматривается задача доказательства завершаемости преобразователя и разрабатывается соответствующий инструментарий. Приводится пример использования разработанного инструментария для доказательства завершаемости простого суперкомпилятора.

В 1999 Йенс Зехер завершает магистерскую диссертацию, посвященную перфектной суперкомпиляции – “Perfect Supercompilation” [100]. Перфектный суперкомпилятор распространяет не только позитивную, но и негативную информацию при прогонке. Отличия данной работы от предыдущих в следующем: (1) рассматривается типизированный (с поддержкой полиморфных данных и функций) язык первого порядка, (2) конфигурация теперь является выражением со свободными переменными и рестрикциями – вводится система рестрикций и алгебра работы с ними, (3) явным образом определяется алгоритм генерации остаточной программы из частичного дерева процессов. Магистерская диссертация Зехера является первой работой, в которой описан суперкомпилятор, учитывающий негативную информацию и доказана его завершаемость. Обзор диссертации опубликован в виде статьи – [102].

В работе Зехера 2001 года “Driving in the Jungle” описывается вариант позитивной прогонки со стратегией вычислений в схлопнутых джунглях (collapsed jungle evaluation). Конфигурация представляет из себя уже не выражение (дерево), а направленный ациклический граф (DAG).

Докторская диссертация Зехера “Driving-Based program transformation in theory and practice” [101] рассматривает методы преобразования программ (в том числе и суперкомпиляцию) для случаев, когда выражения представляются в виде направленного ациклического графа. Особого внимания заслуживают определение вложения и обощения таких выражений.

В книге Абрамова и Пармёновой “Метавычисления и их применения. Суперкомпиляция” [134] рассматривается суперкомпилятор для плоского

функционального языка первого порядка TSG[7], варианта языка S-Graph, используемого в работе [33]. Приведены алгоритмы обнаружения вложения и обобщения. Можно сказать, что работа [134] является в некотором смысле завершением работы [33].

В работе [56] Андрей Климов рассматривает суперкомпиляцию в общем виде – как отношение специализации и исследует его свойства.

1.1.3 Суперкомпиляции императивных языков

Традиционно суперкомпиляция рассматривается как метод преобразавания функциональных программ.

Прогонка для языка простого императивного языка Flowchart рассматривается в работе Нила Джонса [48].

Интересны работы Андрея Климова по применению идей суперкомпиляции к специализации объектно-ориентированного языка Java [57], [58].

Отдельно стоит упомянуть работу Димитура Крустева [65], в которой рассматривается вопрос механической верификации суперкомпилятора для простого императивного языка.

1.1.4 Суперкомпиляция функциональных языков высшего порядка

В конце 2000-х годов возникает большой интерес к применению методов суперкомпиляции для оптимизации программ на функциональных языках первого порядка.

Питер Джонсон и Йохан Нордландер разрабатывают суперкомпилятор для языка Timber (язык высшего порядка с семантикой вызова по имени), алгоритмы суперкомпилятора представлены в серии работ, озаглавленных “Positive Supercompilation for a Higher Order Call-By-Value Language” [50, 51, 49, 52]. Главной задачей этих работ является обеспечить полную эквивалентность исходной и остаточной программ (сохранить свойства завершаемости) и одновременно обеспечить глубину преобразования, близкую к глубине преобразования программ с ленивой се-

[7] TSG = Typed S-Graph

мантикой вычислений. Основное средство – использовать анализ завершаемости при анализе конфигураций.

В 2008 году Митчел сообщил о суперкомпиляторе для языка Haskell Supero [78], особенностью которого являлась работа с let-выражениями (сохранение семантики вызова по необходимости). Другой особенностью являлся отказ от тесного обобщения в случае, если верхняя конфигурация вложена в нижнюю через погружение.

В статье “Rethinking Supercompilation” [77] изложено переосмысление частей суперкомпилятора с целью создания быстро работающего и робастного суперкомпилятора. (1) Для представления конфигураций выбрано специальное синтаксическое подмножество ядра языка Haskell – “Simplified Core”. На каждом шаге осуществляется преобразование текущей конфигурации в “Simplified Core”[8]. (2) Отказ от использования гомеоморфного вложения как синтаксического критерия для свистка, – каждому выражению (конфигурации) ставится в соответствие мультимножество меток (tag-bag) и в качестве свистка используется сравнения этих мультимножеств.

На практичность рассчитан и CHSC[9], описанный в работе “Supercompilation by Evaluation” [15]. Конфигурация представляется обобщенным состоянием абстрактной машины Кривина - кучей, активным подвыражением и стеком. В качестве свистка также используется упорядочение над мультимножествами меток. Особенностью CHSC является отсутствие обобщения с учетом истории вычислений – конфигурация обобщается безотносительно истории вычислений. Также CHSC поддерживает суперкомпиляцию рекурсивных let-выражений с сохранением семантики вызова по необходимости.

Стоит также упомянуть работы [53], [54], [89], [75].

При суперкомпиляции делается однократная прогонка. Идея дистилляции – осущетвлять двойную прогонку. Прогонка (построение дерева процессов) в дистилляции происходит также как и в суперкомпиляции. А

[8]Хотя “Simplified Core” и является алгоритмически полным языком, не каждое Haskell-выражение можно перевести в “Simplified Core” – поэтому реально рассматриваются не любые программы. К счастью, непереводимые в “Simplified Core” Haskell-программы экзотичны и достаточно редки.

[9]CHSC = Cambridge Haskell SuperCompiler

вот при обработке возможного вложения и зацикливания используются не сами конфигурации, а их суперкомпилированные версии. В статье [39] показано, как дистилляция может изменять асимптотику программ. Формализм описания дистилляции находится пока еще в крайне сыром виде и претерпевает существенные изменения от публикации к публикации. Например, в работе [39] перед каждым шагом прогонки выражение суперкомпилировалось и дальше шаг прогонки осуществлялся относительно суперкомпилированного выражения. В работе [41] зацикливание и обобщение осуществляется не над выражениями, а над соответствующими им (в общем случае бесконечными) деревьями процессов. В работе [42] сравниваются не выражения, а графы процессов. В докладе по материалам работы [42], представленном Гамильтоном на семинаре по метавычислениям META'2010 сравнивались и обобщались системы переходов. Дистилляция – многообещающее развитие идей суперкомпиляции, требующее пока соответствующей формализации.

1.1.5 Другие работы

В 1970-х годах при разработке методов суперкомпиляции был предложен метод окрестностного анализа для аппроксимации алгоритмически неразрешимой проблемы останова. Суть окрестностного анализа – определить какая информация о тексте *text* была использована, и какая информация о тексте *text* не была использована в некотором процессе обработки текста *text*. Окрестностный анализ основан на прогонке.

Окрестностный анализ разрабатывается в работах Сергея Абрамова "Metacomputation and program testing" [2], "Метавычисления и их применение" [133]. В частности, показано, как окрестностный анализ может использоваться для организации "предельно надежного тестирования".

Задачу инвертирования программ (вычислений) можно определить как нахождения входных данных (множества входных) по результату вычисления. Проблемами инверсного вычисления детально занимались Александр Романенко, Сергей Абрамов и Роберт Глюк, придерживаясь идеи синтаксической инверсии программ. А. Романенко в работах "The generation of inverse functions in Refal" [91], "Inversion and metacomputation" [92] разрешает некоторые сложности текстуальной инверсии программ на

Рефале.

Инверсный Рефал рассматривается в работе Абрамова “Метавычисления и логическое программирование” [132] как язык логического программирования и сравнивается с другими языками логического программирования.

Технические вопросы организации инверсных вычислений для языка TSG исследуются в работах Сергея Абрамова и Роберта Глюка “The Universal Resolving Algorithm: Inverse Computation in a Functional Language” [5], “Inverse Computation and the Universal Resolving Algorithm” [7], “Principles of inverse computation and the universal resolving algorithm” [8], “Faster Answers and Improved Termination in Inverse Computation of Non-Flat Languages” [9].

Инверсное вычисления и окрестностный анализ, по сути, для данной (стандартной) семантики языка определяют другие (нестандартные) семантики вычислений. Нестандартные семантики и принципы работы с ними исследуются в работах Сергея Абрамова и Роберта Глюка “Semantics modifiers: an approach to non-standard semantics of programming languages” [3], “Combining Semantics with Non-standard Interpreter Hierarchies” [4], “From standard to non-standard semantics by semantics modifiers” [6].

1.2 Суперкомпиляция функционального языка первого порядка

В качестве отправной точки рассмотрим, как устроена позитивная суперкомпиляция для простого функционального языка SLL. Материал этого разделан основан на работах [105, 107, 103, 62, 135].

Данные языка SLL – константы и простые структуры. Простая структура – это именованный конструктор с фиксированным и упорядоченным набором полей. Каждое поле в свою очередь, – некоторые данные (конструктор или константа). Константа есть нуль-арный конструктор.

Вычисления в данном языке организованы через вызовы функций. Каждая функция имеет строго заданное количество аргументов. Для реализации ветвления функциям разрешается заглядывать в первый аргумент и, в зависимости от того, в какой конструктор обернут первый

аргумент, – выполнять различные вычисления. Функция конструирует результат вычислений из констант, конструкторов, вызовов функций и аргументов. При заглядывании в первый аргумент функции становятся доступны части этого аргумента. Тело функции – одно выражение.

Например, если рассматривать алфавит из двух букв (констант) – `A` и `B`, то слова этого алфавита логично представлять в виде списка. Пустой список - константа `Nil`, непустой список - структура из двух частей, первая часть - первый элемент списка, вторая часть - остальные элементы (хвост). Непустые списки создаются с помощью бинарного конструктора `Cons(head, tail)`.

Тогда слова алфавита из двух букв представляются так:
`Nil()` - пустое слово “”
`Cons(A(), Nil())` - слово “A”
`Cons(A(), Cons(B(), Nil()))` - “AB”
`Cons(B(), Cons(B(), Cons(B, Nil())))` - “BBB”

Например функция, приписывающий элемент `x` к списку `xs`, записывается так:

```
prepend(x, xs) = Cons(x, xs);
```

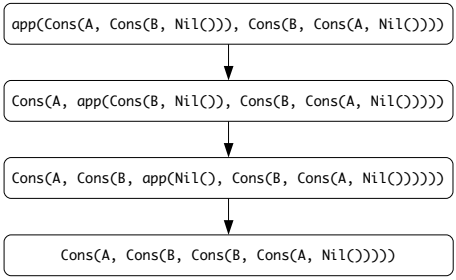
Функция `app`, конкатенирующая списки, записывается так:

```
app(Nil(), vs)       = vs;
app(Cons(u, us), vs) = Cons(u, app(us, vs));
```

Здесь константа `Nil` используется для представления пустых списков. Конструктор `Cons(x, xs)` представляет непустой список, где `x` – первый элемент списка, а `xs` – остальная часть.

Для того, чтобы синтаксически различать функции и конструкторы, принято следующее соглашение: имена конструкторов начинаются с заглавной буквы, а функций – с прописной. Можно не писать скобки для нуль-арных конструкторов (констант).

Конкатенация строк `Cons(A, Cons(B, Nil))` и `Cons(B, Cons(A, Nil))` происходит так:

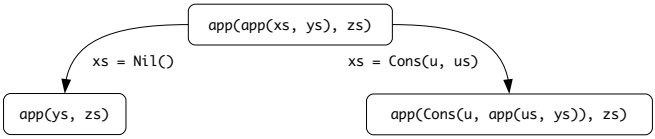


Состояние вычисления – выражение. Вычисление – последовательность переходов между состояниями в соответствии с правилами, описанными в программе. Порядок вычислений – ленивый (снаружи-внутрь).

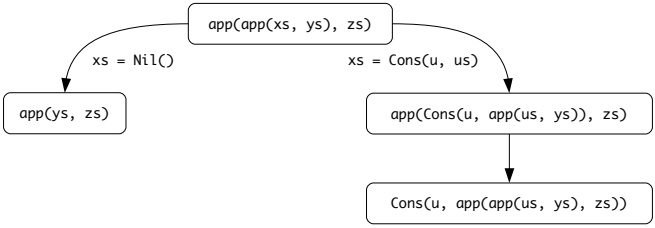
1.2.1 Примеры суперкомпиляции

Пример 1

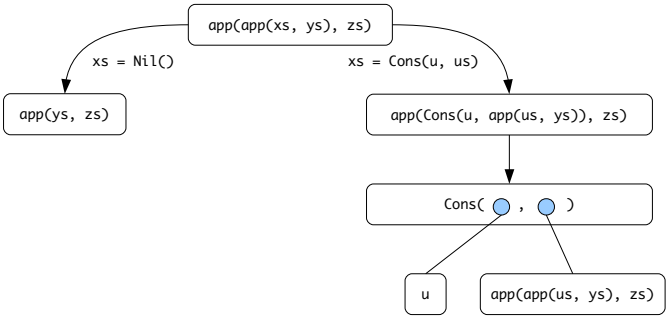
Допустим, что нам требуется получить результат конкатенации трёх списков. Это можно сделать, вычислив выражение `app(app(xs, ys), zs)`. Однако, при этом элементы списка `xs` будут анализироваться два раза (сначала - внутренним вызовом `app`, а потом - наружным). Покажем, как с помощью суперкомпиляции можно получить более эффективное вычисление. Начнем "символически" интерпретировать выражение (конфигурацию) `app(app(xs, ys), zs)`. Сначала попробуем раскрыть внутренний вызов `app(xs, ys)`. Однако, из-за того, что `app` проверяет, какой вид имеет её первый аргумент - `Nil()` или `Cons(u, us)`, нам придётся рассмотреть два случая (или, выражаясь языком, принятым среди специалистов по суперкомпиляции, "расщепить конфигурацию"):



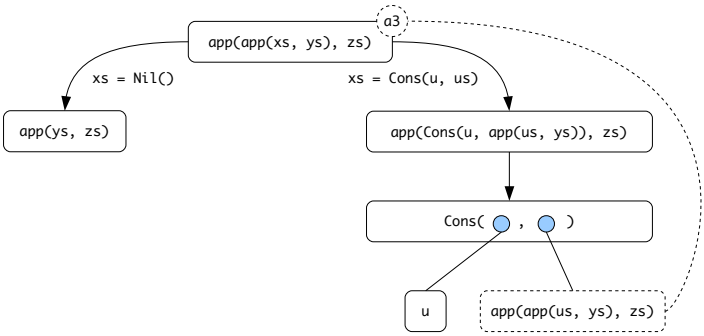
Правую нижнюю конфигурацию теперь можно однозначно развернуть – раскрыть наружный вызов app:



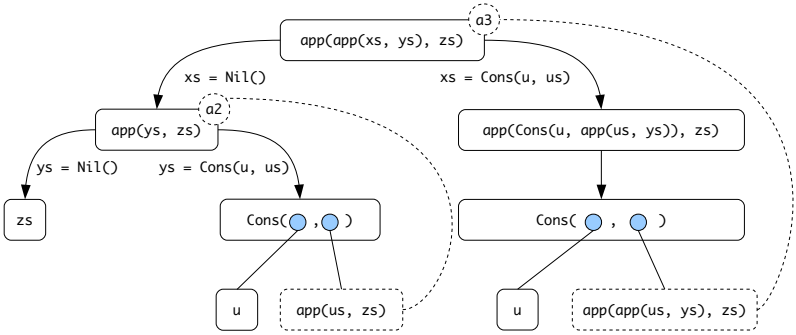
Новое выражение содержит конструктор на верхнем уровне, – мы переходим к “символическому” вычислению аргументов конструктора:



В этот момент обнаруживается, что конфигурация в нижнем правом узле получившегося дерева совпадает с конфигурацией в корне дерева (с точностью до переименования переменных). Понятно, что нет смысла второй раз пытаться вычислить ту же самую конфигурацию. Обозначим конфигурацию в верхнем узле a3 и с помощью обратной дуги изобразим совпадение конфигураций:



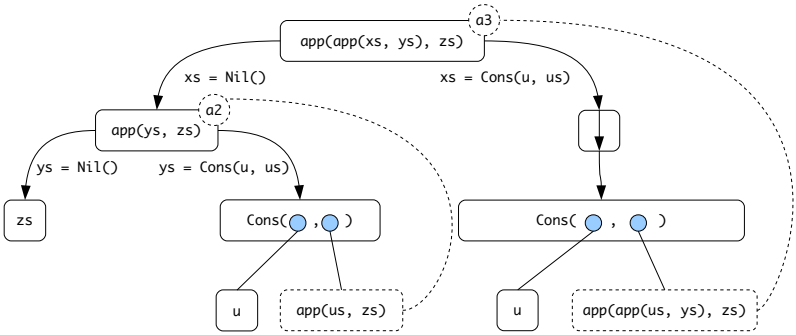
Повторяем те же операции для левого узла. После нескольких шагов также обнаруживаем совпадение конфигураций в листе дерева с конфигурацией в одном из предков. Обозначаем верхнюю конфигурация `a2` и также проводим обратную дугу:



Построение “дерева конфигураций” завершено: каждый лист дерева содержит либо уже ранее встречавшуюся конфигурацию (с точностью до переименования переменных), либо переменную, либо нуль-арный конструктор. Можно сказать, что построенное таким образом дерево представляет все возможные пути вычисления выражения, находящегося в корне дерева. Ветвления в дереве получаются из разбора возможных значений переменных.

Однако, в получившемся дерева есть избыточность – есть узел (с конфигурацией `app(Cons(u, app(us, ys)), zs)`), через который трасса вы-

числения проходит безусловно. Этот узел можно безопасно удалить и спрямить входящую и выходящую из него дуги. В результате получится упрощенное дерево:



Удаленный нами узел называется в терминологии суперкомпиляции транзитным узлом. Удаление транзитных узлов – главный механизм оптимизации с помощью суперкомпиляции.

В получившемся частичном дереве процессов достаточно информации (в узлах и на дугах), чтобы превратить его в программу. В итоге получаем следующую программу:

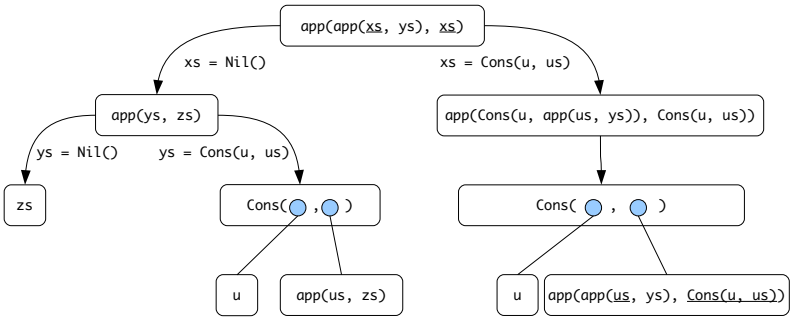
```
a3(Nil(), ys, zs)       = a2(ys, zs);
a3(Cons(u, us), ys, zs) = Cons(u, a3(us, ys, zs));

a2(Nil(), zs)           = zs;
a2(Cons(u, us), z)      = Cons(u, a2(us, z));
```

Вычисление выражения `a3(xs, ys, zs)` в новой программе эффективнее, чем вычисление выражения `app(app(xs, ys), zs)` в первоначальной. При вычисления исходного выражения делалось $O(2|xs| + |ys|)$ шагов вычислений, в то время, как для вычисления нового выражения требуется $O(|xs| + |ys|)$ шагов вычислений, где $|xs|$ – длина списка `xs`

Произведенный процесс преобразований можно рассматривать как состоящий из трех этапов: символическое вычисление, поиск повторений и построение новой программы.

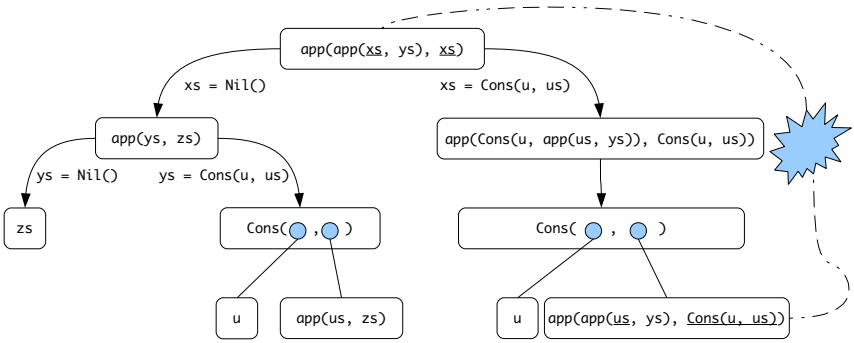
Пример 2

Однако, в большинстве случаев при построении частичного дерева процессов таким образом будут возникать бесконечные ветви, на которых не будет точных повторений конфигураций. Рассмотрим вычисление выражения `app(app(xs, ys), xs)` для программы из первого примера. Повторяем те же самые шаги и получаем следующее дерево:

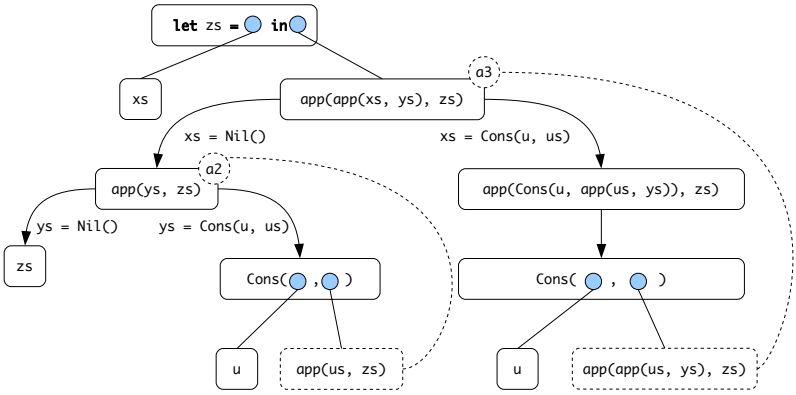


Конфигурация в нижнем правом узле похожа на конфигурацию в верхнем узле, но не совпадает с ней (нижнее выражение не является переименованием верхнего выражения). При продолжении символического вычисления правого нижнего выражения не получится добиться, чтобы конфигурация в листе совпала с конфигурацией в предках, и процесс построения конечного дерева (используя только символическое вычисление и поиск повторений) не завершится. Тот факт, что две конфигурации похожи, но не совпадают является (эвристическим) сигналом[10] того, что дальнейшее построение ветви может не завершится. Обозначим это:

[10]исторически называемого в суперкомпиляции свистком



Причина возникновения бесконечной ветви – повторная переменная `xs` в начальном выражении. Однако, как мы видели, если проигнорировать повторную переменную и считать разные вхождения переменной `xs` в начальной конфигурации разными переменными, процесс построения частичного дерева процессов завершается. Поэтому одним из выходов из сложившейся ситуации является рассмотрение более общей начальной конфигурации – *обобщение*. Будем рассматривать второе вхождение переменной `xs` как переменную `zs`, – для обозначения обобщения используются специальный let-узел. После обобщения и дальнейшего построения дерева получается дерево, аналогичное дереву из первого примера:



При генерации нового выражения и программы из полученного дерева let-выражения могут быть устранены подстановкой соответствующего нового выражения вместо переменной из привязки. В итоге получается

Рис. 1.1 SLL: грамматика

P	::=	$d_1 \ldots d_n$	программа
d	::=	$f(v_1, \ldots, v_n) = e;$	f-функция
	|	$g(p_1, v_1, \ldots, v_n) = e_1;$	g-функция
		$\ldots$	
		$g(p_m, v_1, \ldots, v_n) = e_m;$	
e	::=		$e \in E_{C \cup F \cup V}$, выражение
		v	$v \in V$, переменная
	|	$C(e_1, \ldots, e_n)$	$C \in C$, конструктор
	|	$f(e_1, \ldots, e_n)$	$f \in F$, вызов f-функции
	|	$g(e_1, \ldots, e_n)$	$g \in F$, вызов g-функции
p	::=	$C(v_1, \ldots, v_n)$	pattern

новое выражение `a3(xs, ys, xs)` и новая программа:

```
a3(Nil(), ys, zs)       = a2(ys, zs);
a3(Cons(u, us), ys, zs) = Cons(u, a3(us, ys, zs));

a2(Nil(), zs)           = zs;
a2(Cons(u, us), z)      = Cons(u, a2(us, z));
```

Как и раньше, преобразование проходило в три этапа, но второй этап (поиск повторений) был более сложным – при обнаружении “почти повторения” мы заменяли целое поддерево на новый узел в результате обобщения.

1.2.2 Синтаксис и семантика языка SLL

Рассмотрим теперь суперкомпиляцию для языка SLL более формально. Для этого нам необходимо вначале определить синтаксис и семантику языка.

На Рис. 1.1 показан синтаксис языка SLL, рассматриваемого в работах [103, 108].

SLL – ленивый функциональный язык первого порядка. В языке SLL используется перечислимое множество символов для представления переменных $v \in V$, конструкторов $C \in C$, и имён функций $f \in F$ и $g \in F$.

Рис. 1.2 SLL: подстановка

$v\theta$	$=$	e	если $v := e \in \theta$
	$=$	v	в противном случае
$h(e_1, \ldots, e_n)\theta$	$=$	$h(e_1\theta, \ldots, e_n\theta)$	

Все символы имеют фиксированную арность. SLL-программы обрабатывают данные, представляющие собой конечные деревья, построенные с помощью конструкторов. Выражение языка SLL – либо переменная, либо конструктор, либо вызов функции. Программа состоит из набора определений функций. Причем функции делятся на два класса: f-функции и g-функции. Определение g-функции состоит из одного или нескольких правил, при этом в каждом правиле присутствует образец, с помощью которого анализируется структура первого аргумента. Определение f-функции состоит из одного правила, в котором все аргументы являются переменными. В левой части правила не допускаются повторные переменные. Все переменные, присутствующие в правых частях правил, должны быть определены в левой части.

Как было отмечено раньше, синтаксически функции не отличаются от конструкторов (единственное отличие – то, что имена конструкторов начинаются с заглавной буквы, а функций – с прописной), поэтому далее будем использовать запись $h(e_1, .., e_n)$ для обозначения и конструкторов, и функций.

Определение 1 (SLL-подстановка)**.** Подстановкой будем называть конечный список пар вида $\theta = \{\overline{v_i := e_i}\}$, каждая пара в котором связывает переменную v_i с ее значением e_i. Область определения θ определяется как $domain(\theta) = \{\overline{v_i}\}$. Область значений θ определяется как $range(\theta) = \{\overline{e_i}\}$. Результат применения подстановки θ к выражению e записывается $e\,\theta$.

Определение 2 (Применение SLL-подстановки)**.** Результат применения подстановки θ к выражению e, $e\theta$ вычисляется по правилам на Рис. 1.2.

Любое выражение языка SLL можно представить либо в виде наблюдаемого выражения, либо в виде контекста редукции с помещенным в дыру редексом. Грамматика такой декомпозиции представлена на Рис. 1.3.

Рис. 1.3 SLL: декомпозиция выражений

$$
\begin{array}{lcl}
obs & ::= & C(e_1, \ldots, e_n) \mid v \\
con & ::= & \langle\rangle \mid g(con, \ldots) \\
red & ::= & f(e_1, \ldots, e_n) \mid g(C(e_1, \ldots, e_n), \ldots) \mid g(v, \ldots)
\end{array}
$$

Рис. 1.4 Операционная семантика SLL

$$
\begin{array}{ll}
con\langle f(e_1, \ldots, e_n)\rangle & \mapsto con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle \\
 & \quad \text{при } f(x_1, \ldots, x_n) = e \\
con\langle g(C(e_1, \ldots, e_m), e_{m+1}, \ldots, e_n)\rangle & \mapsto con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle \\
 & \quad \text{при } g(C(v_1, \ldots, v_m), v_{m+1}, \ldots, v_n) = e
\end{array}
$$

Определение 3 (Слабая головная нормальная форма)**.** Выражение e языка SLL находится в слабой головной нормальной форме, если в нем на верхнем уровне находится конструктор (то есть $e = C(e_1, \ldots, e_n)$).

Определение 4 (Шаг редукции)**.** Шаг редукции $\mapsto$ – наименьшее отношение на множестве SLL выражений, удовлетворяющее правилам на Рис. 1.4.

Шаг редукции определяет семантику вычисления. Значением вычисления является выражение в слабой головной нормальной форме (конструктор). В обычном случае вычисляются выражения без переменных (объектные выражения). Обратите внимание, что если на каком-то шаге вычисляемое выражение становится конструктором, то вычисления заканчиваются.

Определение 5 (Равенство SLL-выражений)**.** Два выражения e_1 и e_2 считаются равными, если они различаются совпадают текстуально. Равенство выражений e_1 и e_2 записывается как $e_1 \equiv e_2$.

Определение 6 (Частный случай SLL-выражения)**.** Выражение e_2 называется *частным случаем* выражения e_1, $e_1 \lessdot e_2$, если существует подстановка θ такая, что $e_1\theta \equiv e_2$. Для выражений языка SLL такая подстановка является единственной и обозначается $\theta = e_1 \circledcirc e_2$.

Определение 7 (Переименование SLL-выражения)**.** Выражение e_2 называется *переименованием* выражения e_1, $e_1 \simeq e_2$, если $e_1 \lessdot e_2$, и $e_2 \lessdot e_1$.

Другими словами, e_1 и e_2 различаются только именами свободных переменных.

1.2.3 Обобщение и гомеоморфное вложение SLL-выражений

В данном разделе даются определения, необходимые для формулировки алгоритма построения частичного дерева процессов.

Определение 8 (Строгая декомпозиция SLL-выражения)**.** Let-выражение

$$e_l = let\ \overline{v_i = e_i};\ in\ e_0$$

называется *строгой декомпозицией* выражения e, $e_l \sqsubset e$ если $e_0 \lessdot e$, $e_0 \not\simeq e$ и $e = e_0\{\overline{v_i := e_i}\}$.

Определение 9 (Обобщение SLL-выражений)**.** Обобщением выражений e_1 и e_2, называется тройка $(e_g, \theta_1, \theta_2)$, где e_g – выражение, θ_1 и θ_2 – подстановки, такие, что $e_g\theta_1 \equiv e_1$ и $e_g\theta_2 \equiv e_2$. Множество всех обобщений для выражений e_1 и e_2 обозначается как $e_1 \frown e_2$.

Определение 10 (Тесное обобщение двух SLL-выражений)**.** Обобщение $(e_g, \theta_1, \theta_2)$ называется *тесным* обобщением выражений e_1 и e_2, если для любого обобщения $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$ верно, что $e'_g \lessdot e_g$, т.е. e_g является частным случаем e'_g. Мы предполагаем, что определена операция $\sqcap$ такая, что $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$.

Определение 11 (Несопоставимые выражения)**.** Выражения e_1 и e_2 называются несопоставимыми, $e_1 \leftrightarrow e_2$, если $e_1 \sqcap e_2 = (v, \theta_1, \theta_2)$, то есть обощенное выражение является переменной.

Известным результатом является алгоритм нахождения тесного обобщения выражений первого порядка без связанных переменных [66].

Определение 12 (Итеративный алгоритм обобщения SLL-выражений)**.** Тесное обобщение выражений e' и e'', $e' \sqcap e''$ находится с помощью применений правил общего функтора и общего подвыражения (Рис. 1.5) к начальному тривиальному обобщению $(v, \{v := e'\}, \{v := e''\})$.

Рис. 1.5 SLL: итеративное обобщение выражений e' и e''

Начальное тривиальное обобщение

$(v, \{v := e_1\}, \{v := e_2\})$

Правило общего функтора

$$\begin{pmatrix} e \\ \{v := h(e'_1,\ \ldots,\ e'_n)\} \cup \theta' \\ \{v := h(e''_1,\ \ldots,\ e''_n)\} \cup \theta'' \end{pmatrix} \Rightarrow \begin{pmatrix} e\{v := h(v_1,\ \ldots,\ v_n)\} \\ \{v_1 := e'_1,\ \ldots,\ v_n := e'_n\} \cup \theta' \\ \{v_1 := e''_1,\ \ldots,\ v_n := e''_n\} \cup \theta'' \end{pmatrix}$$

Правило общего подвыражения

$$\begin{pmatrix} e \\ \{v_1 := e',\ v_2 := e'\} \cup \theta' \\ \{v_1 := e'',\ v_2 := e''\} \cup \theta'' \end{pmatrix} \Rightarrow \begin{pmatrix} e\{v_1 := v_2\}) \\ \{v_2 := e'\} \cup \theta' \\ \{v_2 := e''\} \cup \theta'' \end{pmatrix}$$

Рис. 1.6 SLL: рекурсивный алгоритм обобщения

Тесное обобщение

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

Правило общего функтора

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $h(e'_1, \ldots, e''_n) \mathbin{\widetilde{\sqcap}} h(e''_1, \ldots, e''_n) = (h(e_1, \ldots, e_n), \bigcup \theta'_i, \bigcup \theta''_i)$, где
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

Правило общего подвыражения

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$, где
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$,
    если $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$
    в противном случае

Алгоритм 12 является традиционной ([104, 105, 108]) записью алгоритма обобщения в итеративной форме (определяется “малый шаг” обобщения).

Однако, процесс обобщения удобнее (как мы увидим в дальнейшем)

Рис. 1.7 SLL: гомеоморфное вложение

Embedding

$e' \trianglelefteq e''$ if $e' \trianglelefteq_v e''$ or $e' \trianglelefteq_d e''$ or $e' \trianglelefteq_c e''$

Variables

$v' \trianglelefteq_v v''$

Coupling

$h(e'_1, \ldots, e'_n) \trianglelefteq_c h(e''_1, \ldots, e''_n)$ if $\forall i : e'_i \trianglelefteq e''_i$

Diving

$e \trianglelefteq_d h(e'_1, \ldots, e'_n)$ if $\exists i : e \trianglelefteq e'_i$

определить напрямую через рекурсивные функции (семантика "большого шага"):

Определение 13 (Рекурсивный алгоритм обобщения SLL-выражений)**.** $e' \sqcap e'' = s(e' \widetilde{\sqcap} e'')$, где операции $\widetilde{\sqcap}$ и s определены на Рис. 1.6.

Процесс нахождения обобщения происходит в 2 этапа: первый этап соответствует применению правила общего выражения, – нахождение обобщения (вычисление $e_1 \widetilde{\sqcap} e_2$) двух совпадающих по оболочке выражений сводится к задаче нахождения обобщений их частей. Второй этап (упрощение обобщения с помощью операции s) заключается в удалении из результата повторяющихся частей и соответствует правилу общего подвыражения. Правила применяются в порядке их перечисления.

Определение 14 (Гомеоморфное вложение $\trianglelefteq$ SLL-выражений)**.** Простое вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 1.7.

Свойства вложения и обобщения SLL-выражений

Будем рассматривать SLL-выражения из множества $E_{C \cup F \cup V}$, где C – множество имен конструкторов, F – множество имен функций, V – множество имен переменных (см. Рис. 1.1).

Теорема 15 (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* $e_i \trianglelefteq e_j$.

Доказательство можно найти в [26].

Отношение $\trianglelefteq_c$ также является вполне-квазиупорядочением на $E_{C \cup F \cup V}$.

Если два SLL-выражения вложены через сцепление, то их тесное обобщение нетривиально (не является переменной). Использование отношения $\trianglelefteq_c$ в качестве свистка имеет следующие плюсы:

1. Позволяет гарантировать завершаемость суперкомпилятора (в силу Теоремы 15).

2. Позволяет в случае обобщения всегда учитывать историю вычислений.

Нам хотелось сохранить это свойство при определении гомеоморфного вложения для выражение языка HLL.

Выражения со связанными переменными

В работе [104] рассматривается язык SLL, дополненный `case`-выражениями (со связанными переменными), но отношение вложения и алгоритм обобщения приводятся только для случая выражений без связанных переменных, – читателю предлагается самостоятельно расширить приведенные соотношения на случай связанных переменных в `case`-выражениях, поэтому в данном разделе мы не затрагиваем вопросы вложения и обобщения выражений со связанными переменными.

1.2.4 Построение дерева процессов

Рассмотрим сначала, как строится дерево процессов для SLL-программ.

В случае плоского функционального языка, такого как S-Graph [33] или TSG [134], где можно использовать только хвостовую рекурсию, понятие дерева процессов для конфигурации c_0 определяется достаточно просто – метками дерева процессов являются конфигурации и на дугах дерева процессов обозначены сужения конфигураций. Пути в дереве процессов представляют множество трасс вычисления. Любому объектному выражение $e \in c_0$ из начальной конфигурации соответствует некоторый путь в дереве процессов.

Рис. 1.8 SLL: шаг прогонки

$\mathcal{D}[\![v]\!]$	$\Rightarrow [v]$
$\mathcal{D}[\![C(e_1, \ldots, e_n)]\!]$	$\Rightarrow [e_1, \ldots, e_n]$
$\mathcal{D}[\![con\langle f(e_1, \ldots, e_n)\rangle]\!]$	$\Rightarrow [con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]$ $f(x_1, \ldots, x_n) = e \in P$
$\mathcal{D}[\![con\langle g(C(e_1, \ldots, e_m), e_{m+1}, \ldots, e_n)\rangle]\!]$	$\Rightarrow [con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]$ В программе есть предложение: $g(C(v_1, \ldots, v_m), v_{m+1}, \ldots, v_n) = e$
$\mathcal{D}[\![con\langle g(v, e_1, \ldots, e_n)\rangle]\!]$	$\Rightarrow [\ldots, con\langle e'_i\{v := p_i, \ldots, v_n := e_n\}\rangle, \ldots]$ В программе определяется: $\overline{g(p_i, v_1, \ldots, v_n) = e'_i;}$
$\mathcal{R}[\![con\langle g(v, e_1, \ldots, e_n)\rangle]\!]$	$\Rightarrow [\overline{v = p_i;}]$ В программе определяется: $\overline{g(p_i, v_1, \ldots, v_n) = e'_i;}$

Концептуально это означает, что дерева процессов для конфигурации c_0 *достаточно* для вычисления любого выражения $e \in c_0$. То есть, в некотором смысле, дерево процессов заменяет программу.

Для языка SLL дерева процессов для конфигурации c_0 должно быть достаточно для вычисления любого выражения $e \in c_0$ в любом контексте – обычное вычисление выражения завершается, когда на верхнем уровне появляется конструктор, однако при вычислении этого выражения в контексте нам может потребоваться вычислить аргумент конструктора.

Определение 16 (SLL-сужение)**.** SLL-сужением называется пара (v, p), где v – переменная, а p – образец. Для удобства SLL-сужение записывается как $v = p$.

Определение 17 (Дерево процессов для SLL-выражений)**.** Дерево процессов представляет собой ориентированное (возможно – бесконечное) дерево, каждому узлу n которого приписано некоторое выражение. Для некоторых узлов всем исходящим из ним дугам приписаны сужения. Мы будем обозначать выражение, помещенное в узел n, как $n.expr$. Пусть дано SLL-выражение e. Дерево процессов pt для SLL-выражения e определяется следующим образом:

- Выражение e находится в корневом узле pt.

- В листьях pt находятся только переменные и нуль-арные конструкторы.
- Для каждого узла α, имеющего дочерние узлы $\overline{\beta_i}$ выражение в узле и выражения в дочерних узлах связаны как $\mathcal{D}[\![\alpha.expr]\!] = [\overline{\beta_i.expr}]$, где оператор $\mathcal{D}$ (шаг прогонки) определен на Рис. 1.8.
- Если на дугах, исходящих из узла α имеются сужения $\overline{r_i}$, то $[\overline{r_i}] = \mathcal{R}[\![\alpha.expr]\!]$, где оператор $\mathcal{R}$ определен на Рис. 1.8.

Дерево процессов строится однозначным образом и в общем случае является бесконечным.

1.2.5 Построение частичного дерева процессов

Конечная цель суперкомпиляции – превратить дерево процессов в конечный (возможно, циклический) граф, который называется *частичным деревом процессов*:

Определение 18 (Частичное дерево процессов)**.** Частичное дерево процессов – дерево процессов, в котором:

- допускаются специальные обратные дуги, проведенные от *листьев* дерева к потомкам.
- узлы могут быть помечены let-выражениями (Определение 8).

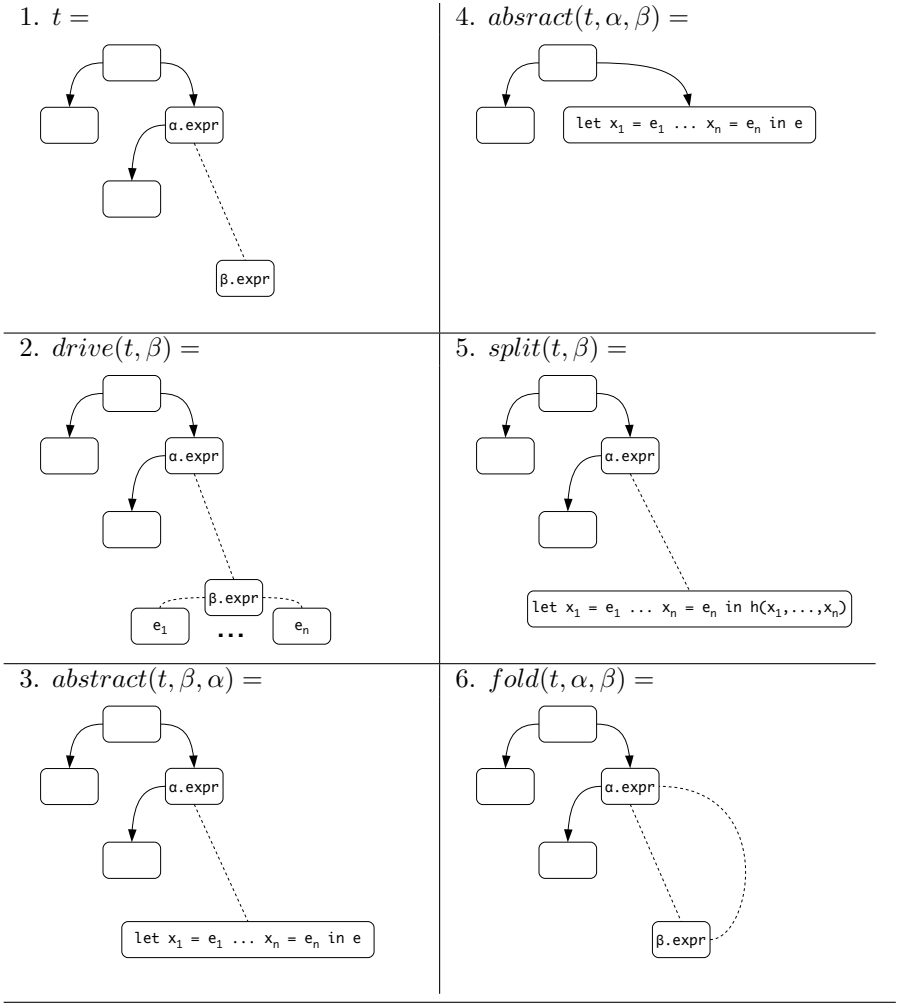
Мы будем обозначать специальные обратные дуги как $\rightrightarrows$ в тексте и штрихованными дугами на диаграммах.

Определение 19 (Обработанный лист)**.** Узел β считается *обработанным*, если выполняется одно из следующих условий:

- Узел уже “зациклен”: из узла выходит двойная дуга $\beta \rightrightarrows \alpha$,
- $\beta.expr$ является нуль-арным конструктором,
- $\beta.expr$ является переменной.

Определение 20 (Тривиальный узел)**.** Узел β считается *тривиальным*, если выполняется одно из следующих условий:

Рис. 1.9 Операции над частичным деревом процессов



- Узел помечен let-выражением.
- Узел помечен конструктором.

Рассмотрим самый простой алгоритм построения частичного дерева процессов для SLL-конфигурации.

Определение 21 (Алгоритм построения частичного дерева процессов)**.**

Рис. 1.10 Алгоритм построения частичного дерева процессов для конфигурации c_0

$t = \boxed{c_0}$
while $incomplete(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $\alpha = find(\beta, \trianglelefteq)$
  if $\alpha \neq \bullet$ *and* $\alpha.expr \simeq \beta.expr$ **then**
    $t = fold(t,\ \alpha,\ \beta)$
  else if $\alpha \neq \bullet$ *and* $\alpha.expr \lessdot \beta.expr$ **then**
    $t = abstract(t, \beta, \alpha)$
  else if $\alpha \neq \bullet$ **then**
    if $\alpha.expr \leftrightarrow \beta.expr$ **then**
      $t = split(t, \beta, \beta.expr)$
    else
      $t = abstract(t, \alpha, \beta)$
    end
  else
    $t = drive(t, \beta)$
  end
end

Частичное дерево процессов строится по алгоритму на Рис. 1.10. В самом начале конструируется дерево $t = \boxed{c_0}$ из одного узла, в который помещается рассматриваемая конфигурация.

В алгоритме используются следующие операции над частичным деревом процессов:

- $incomplete(t)$ – возвращает $True$, если есть хотя бы один необработанный лист дерева t.
- $unprocessed$ – возвращает необработанный лист в дереве t.
- $find(\beta,\ \trianglelefteq)$ – находит среди предков узла β первый узел α такой, что $\alpha.expr \trianglelefteq \beta.expr$.
- $drive(t,\ \beta)$ – делает шаг прогонки выражения, находящегося в узле β: подвешивает к узлу β дочерние узлы и помещает в них выражения $\mathcal{D}[\![\beta.expr]\!]$.

- $abstract(t, \alpha, \beta)$ – обобщает конфигурацию в узле α с учетом конфигурации в узле β: заменяет выражение в α на выражение

$$let\ \overline{v'_i := e'_i}\ in\ e_g,$$

где $\alpha.expr \sqcap \beta.expr = (e_g, \{\overline{v'_i := e'_i}\}, \{\overline{v''_i := e''_i}\})$. Если узел α был корнем некоторого поддерева, то это поддерево уничтожается.

- $fold(t,\ \alpha,\ \beta)$ – "зацикливает" α и β: конструирует специальную обратную дугу – $\beta \rightrightarrows \alpha$.

- $split(t,\ \beta)$ – делает декомпозицию текущей конфигурации: заменяет находящееся в узле β выражение $h(e_1, \ldots, e_n)$ на выражение

$$let\ v_1 = e_1 \ldots v_n = e_n\ in\ h(v_1, \ldots, v_n)$$

Некоторый операции показаны схематично на Рис.1.9.

Теорема 22. *Построение частичного дерева процессов по алгоритму на Рис. 1.10 завершается.*

Доказательство можно найти в [108].

Все операции и шаги в алгоритме строго формализованы и однозначны. Алгоритм преобразования полученного частичного дерева процессов в новое выражение и новую программу можно найти в [103, 100, 62]. Мы не будем подробно останавливаться на этом алгоритме.

Рассмотренный алгоритм – самый простой. В литературе рассматриваются приемы, повышающие глубину преобразований суперкомпилятора. Перечислим следующие:

1. Использование $\trianglelefteq_c$ (см. Рис. 1.7) вместо $\trianglelefteq$ – тогда нет нужды в операции *split*, так как обобщение вложенных через сцепление конфигураций нетривиально.

2. Разбиение всех конфигурации на конечное число классов и поиск вложения только в своем классе.

3. Рассмотрение в некоторых случаях не всех предков, а лишь ближайших (глобальный-локальный контроль).

Рис. 1.11 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	+	+	+	+	+	+

Стоит отметить, что все эти дополнительные приемы не влияют на завершаемость суперкомпилятора.

1.3 Анализ состояния дел в суперкомпиляции с точки зрения трансформационного анализа программ

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 воз-

можность убедиться в корректности реализации суперкомпилятора.

Как было отмечено, использование трансформационного анализа плодотворно, если язык предлагает изобразительные средства, облегчающие написание спецификаций, такие как:

- *Функции высших порядков.* Появляется возможность иметь кванторы по функциям в спецификациях.
- *Работа с бесконечными данными.* Признак ленивого языка – ленивые языки лучше поддаются преобразованиям.

Сравнение суперкомпиляторов по перечисленным критериям приведено в таблице на Рис. 1.11.

К сожалению, на 2007 год не существовало суперкомпилятора, удовлетворяющим даже обязательным требованиям. Поэтому в первую очередь автор поставил перед собой задачу разработать такой суперкомпилятор для языка Haskell.

1.4 Выводы

В главе рассмотрен позитивный суперкомпилятор для функционального языка первого порядка SLL. Все алгоритмы суперкомпилятора представлены полностью и формально. В работе [62] данный суперкомпилятор приведен в виде программы.

Приведен анализ существовавших суперкомпиляторов с точки зрения трансформационного анализа. Из анализа следует, что полноценного суперкомпилятора, пригодного для использования в трансформационном анализе, не было.

С другой стороны, в последнее время появились суперкомпиляторы для языков высших порядков, многие части которых описаны неформально или неполностью, что затрудняет воспроизвести описываемые авторами результаты.

Поэтому автор поставил перед собой цель создать полноценный суперкомпилятор, ориентированный на трансформационный анализ программ на языке высшего порядка (Haskell) и имеющий полное и формальное описание своего устройства.

Глава 2

Язык HLL: синтаксис и семантика

В данной главе формально описывается выбранный объект анализа – язык HLL.

2.1 Формализация языка HLL

Одна из целей данной диссертации – *полное и формальное описание методов и алгоритмов* суперкомпиляции функций высших порядков, используемых в суперкомпиляторе HOSC, вместе *с полным и формальным доказательством их корректности и завершаемости*. Очевидно, что для достижения этой цели необходимо четко и формально определить рассматриваемый язык и описать его семантику.

Язык программирования Haskell обладает достаточно богатым синтаксисом. Однако, из этого богатого синтаксиса выделяется так называемое ядро языка Haskell (Haskell Kernel [110]), которое лежит в основе практически любой реализации языка Haskell [111]. Ядро языка Haskell достаточно высокоуровнево, чтобы быть реальным языком программирования. Программы на ядре Haskell читабельны и могут быть предметом анализа.

В основе языка Haskell лежат алгебраические типы данных, при определении которых явным образом перечисляются конструкторы этих данных (всегда конечный набор конструкторов). Представленные в данной работе методы и алгоритмы суперкомпиляции основаны на конечности числа рассматриваемых конструкторов и распространении позитивной

информации.

Однако, как и в любом практическом языке программировании в языке Haskell есть заранее определенные типы данных – символы, строки, целые числа, действительные числа и т.д. (будем их называть далее для краткости литералами). Количество объектов-литералов в принципе бесконечно. Также в языке Haskell есть заранее определенные стандартные операции над литеральными объектами. *Мы не рассматриваем литеральные объекты и заранее определенные данные над ними.*

Поддержка литеральных объектов и примитивных операциq концептуально проста, но сильно увеличила бы *формальное* описание синтаксиса языка, семантики, алгоритмов суперкомпиляции. То же самое относится и к заранее определенным в языке функциям.

Haskell предусматривает возможность аварийного останова программ: это может случиться явным образом, – инициирована исключительная ситуация, или, например, из-за отсутствия разбора определенного случая (образца) в программе. Мы не рассматриваем такие ситуации. Предполагается, что любая программа завершается *естественным образом*, либо зацикливается. Добавление поддержки аварийных завершений также концептуально просто, но сильно бы затруднило задачу *формального и полного* описания всех частей суперкомпилятора.

Таким образом, мы рассматриваем ядро языка Haskell без:

1. Литеральных объектов (чисел, символов, строк).
2. Заранее заданных функций (в том числе, выброс ошибок).
3. Неполного набора образцов в case-выражениях.

Можно сказать, что рассматривается “чистое” (pure) подмножество языка Haskell. Это позволяет достаточно кратко описать методы и алгоритмы суперкомпиляции.

Мы намеренно добавили в язык HLL специальный вариант let-выражения – letrec, чтобы иметь возможность различать рекурсивные и нерекурсивные локальные определения на *синтаксическом* уровне. Также для простоты работы с языком, в HLL вводятся некоторые ограничения (let-выражение не может быть рекурсивным). Это позволяет достаточно крат-

ко формально описать семантику языка и обойтись без введения лишних понятий при доказательстве корректности суперкомпилятора.

Язык HLL 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.

Таким образом, любая HLL-программа является исполняемой Haskell-программой[1]. А любая программа на чистом ядре Haskell без труда переводится в HLL.

В основе системы типизации языка Haskell лежит полиморфная типизация по Хиндли-Милнеру [22, 44]. Также Haskell поддерживает механизм перегрузки имен с помощью классов, так называемый ad-hoc полиморфизм [129]. Однако ad-hoc полиморфизм легко транслируется в ядро Haskell (с помощью таблиц функций). Поэтому мы будем рассматривать только алгебраические типы данных, определяемые пользователем, и полиморфную типизацию по Хиндли-Милнеру и не будем рассматривать классы языка Haskell и ad-hoc полиморфизм.

Для полного и формального определения семантики и понятия эквивалентных программ нам будет легче вначале рассмотреть бестиповый вариант языка HLL, определить для него перечисленные понятия, а затем сделать их ревизию с учетом типизации.

2.2 Синтаксис языка HLL

Синтаксис бестипового варианта языка HLL. представлен на Рис. 2.1. Будем считать (вплоть до раздела 2.5) язык HLL бестиповым и игнорировать определения типов и вопросы типизации.

Программа на языке HLL состоит из целевого выражения и определений верхнего уровня. Выражение – это локальная переменная, конструктор, глобальная переменная, λ-абстракция, аппликация, `case`-выражение, локальное рекурсивное определение (`letrec`-выражение), локальное определение переменных (`let`-выражение) или выражение в скобках.

[1]После простой механической замены letrec на let, замены свободных переменных в целевой выражении на их значения и “оборачивания” целевого выражения в функцию `main`.

Рис. 2.1 Язык HLL (бестиповый)

$prog ::= e$ **where** $\overline{f_i = e_i;}$	программа
$e ::= v$	переменная
$\mid c\ \overline{e_i}$	конструктор
$\mid f_g$	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
$\mid$ **letrec** $f = e_0$ **in** e_1	локальное определение
$\mid$ **let** $\overline{v_i = e_i;}$ **in** e	let-выражение
$\mid (e)$	выражение в скобках
$p ::= c\ \overline{v_i}$	образец

Выражение e_0 в case-выражении называется селектором, выражения $\overline{e_i}$ – ветвями. Мы будем использовать две записи для обозначения аппликации: $e_1\ e_2$ и $e_0\ \overline{e_i}$. В первом случае e_1 может быть любым выражением. Во втором случае мы требуем, чтобы список аргументов $\overline{e_i}$ был непустым, а выражение e_0 не было аппликацией.

Глобальное определение задает соответствие между *глобальной* переменной и выражением в правой части. Локальные определения *связывают локальную переменную* с выражением в данном контексте. Образец в case-выражении также *связывает локальные переменные* с частями сопоставленного выражения.

Неотъемлемой частью описанных далее алгоритмов является работа с переменными. Поскольку в HLL-выражениях присутствуют связанные переменные, нам необходимо детально формализовать работу со связанными переменными, а также ввести соглашения, уточняющие работу с переменными.

Замечание 23 (Три типа переменных)**.** В языке SLL переменные в выражениях не отличаются друг от друга. В выражениях языка HLL могут встречаться переменные трех типов. Переменная является *связанной* переменной, если (1) она является аргументом объемлющей λ-абстракции, либо (2) определяется в образце объемлющей ветви `case`-выражения или (3) связана через `let`-выражение или `letrec`-выражение. Переменная f_g

Рис. 2.2 HLL: множество связанных переменных выражения

$bv[\![f_g]\!]$	$= \{\}$
$bv[\![v]\!]$	$= \{\}$
$bv[\![c\ \overline{e_i}]\!]$	$= \bigcup bv[\![e_i]\!]$
$bv[\![\lambda v \to e]\!]$	$= bv[\![e]\!] \cup \{v\}$
$bv[\![e_1\ e_2]\!]$	$= bv[\![e_1]\!] \cup bv[\![e_2]\!]$
$bv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= bv[\![e_0]\!] \cup (\bigcup bv[\![e_i]\!]) \cup (\bigcup v_{ik})$
$bv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= bv[\![e]\!] \cup (\bigcup bv[\![e_i]\!]) \cup (\bigcup v_i)$
$bv[\![letrec\ f = e_1\ in\ e_2]\!]$	$= bv[\![e_1]\!] \cup bv[\![e_2]\!] \cup \{f\}$

считается *глобальной*, если в программе есть определение $f_g = e$. Все остальные переменные – *свободные*.

Неформальное разделение переменных на три типа в замечании 23 ведет к неоднозначностям, – приходится разделять понятие переменной и вхождение переменной.

В приведенной ниже простой программе переменная `bottom` в целевом выражении может считаться как глобальной, так и связанной.

```
let bottom = x in bottom
  where
bottom = bottom;
```

Мы введем несколько соглашений, устраняющих неоднозначности в вопросе классификации переменных и не требующих явного рассмотрения областей видимости переменных. Для этого мы вначале формально (и несколько волюнтаристски) определим как вычисляются множества свободных, глобальных и связанных переменных выражения. Определения данных множеств также позволит просто и формально разделить рекурсивные и нерекурсивные локальные определения.

Мы обозначаем переменную с индексом f_g, если в программе есть определение $f_g = e$; и без индекса, v в противном случае.

Определение 24 (Множества связанных, глобальных и свободных переменных выражения)**.** Множества $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ связанных, глобальных и свободных переменных выражения e определяются по правилам, представленных на Рис. 2.2, Рис. 2.3 и Рис. 2.4 соответственно.

Рис. 2.3 HLL: множество глобальных переменных выражения

$gv[\![f_g]\!]$	$= \{f_g\}$
$gv[\![v]\!]$	$= \{\}$
$gv[\![c\ \overline{e_i}]\!]$	$= \bigcup gv[\![e_i]\!]$
$gv[\![\lambda v \to e]\!]$	$= gv[\![e]\!]$
$gv[\![e_1\ e_2]\!]$	$= gv[\![e_1]\!] \cup gv[\![e_2]\!]$
$gv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$
$gv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$
$gv[\![letrec\ f = e_1\ in\ e_2]\!]$	$= gv[\![e_1]\!] \cup gv[\![e_2]\!]$

Рис. 2.4 HLL: множество свободных переменные выражения

$fv[\![f_g]\!]$	$= \{\}$
$fv[\![v]\!]$	$= \{v\}$
$fv[\![c\ \overline{e_i}]\!]$	$= \bigcup fv[\![e_i]\!]$
$fv[\![\lambda v \to e]\!]$	$= fv[\![e]\!] \setminus \{v\}$
$fv[\![e_1\ e_2]\!]$	$= fv[\![e_1]\!] \cup fv[\![e_2]\!]$
$fv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= fv[\![e]\!] \cup (\bigcup fv[\![e_i]\!] \setminus \{\overline{v_{ik}}\})$
$fv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$	$= (fv[\![e]\!] \setminus (\bigcup v_i)) \cup (\bigcup fv[\![e_i]\!])$
$fv[\![letrec\ f = e_1\ in\ e_2]\!]$	$= fv[\![e_1]\!] \cup fv[\![e_2]\!] \setminus \{f\}$

Соглашение 25 (Именование переменных)**.** Чтобы избежать возможного конфликта имен, мы требуем, чтобы для любого выражения e множества $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ попарно не пересекались. Дополнительно, каждая связанная переменная может быть определена в выражении не более одного раза.

Соглашение 25 не только устраняет неоднозначность в классификации переменных, но также накладывает дополнительное ограничение на `let`-выражение: локальная переменная, введенная в `let`-выражении может использоваться только в теле `let`-выражения и не может использоваться в определении другой переменной того же `let`-выражения, – это гарантирует *независимость* локальных переменных `let`-выражения друг от друга.

Глобальные определения программы должны быть замкнутыми, то есть множество свободных переменных правой части глобального определения должно быть пустым. Целевое выражение e в программе *prog* на языке HLL может содержать свободные переменные.

2.3 Подстановка

Определение 26 (HLL-подстановка)**.** Подстановкой будем называть конечный список пар вида $\theta = \{\overline{v_i := e_i}\}$, каждая пара в котором связывает переменную v_i с ее значением e_i. Область определения θ определяется как $domain(\theta) = \{\overline{v_i}\}$. Область значений θ определяется как $range(\theta) = \{\overline{e_i}\}$. Результат применения подстановки θ к выражению e записывается $e\ \theta$.

Язык HLL основан на λ-исчислении. В λ-исчислении подстановка является фундаментальной операцией. Чтобы обеспечить корректность подстановки, выражения рассматриваются с точностью до переименования связанных переменных, то есть с точностью до $\equiv_\alpha$ ([12], 2.1.11). Таким образом, операция подстановки должна быть корректной на классах $\equiv_\alpha$ - эквивалентности ([12], Приложение C). То есть:

$$e \equiv_\alpha e',\ e_i \equiv_\alpha e'_i \Rightarrow e\{\overline{v_i := e_i}\} \equiv_\alpha e'\{\overline{v_i := e'_i}\}$$

Существуют различные способы решения этой задачи:

1. Можно рассматривать “канонические” безымянные выражения, где связанные переменные не имеют имен [25]. В данном подходе целый класс α-конгруэнтных обычных выражений соответствует ровно одному выражению с безымянными переменными. Такой подход хорош для машинных преобразований, но затруднителен для восприятия человеком.

2. Можно при применении операции подстановки заменять связанные переменные на “свежие” переменные по мере необходимости, чтобы избежать “захвата” связанных переменных [21, 109]. Такой подход неудобен в случае, когда подстановка является результатом некоторой операции (например, обобщения), – нужно учитывать, что какая-то составляющая подстановки реально не будет применяться.

3. Можно вообще избежать подстановок, используя механизм наподобие *явных подстановок* [1, 94]. Однако в случае суперкомпиляции применение такого подхода сильно бы усложнило конфигурационный анализ.

Рис. 2.5 HLL: подстановка

$v\theta$	$=$	e	если $v := e \in \theta$
	$=$	v	иначе
$f_g\theta$	$=$	f_g	
$(c\ \overline{e_i})\theta$	$=$	$c\ \overline{(e_i\theta)}$	
$(\lambda v \to e)\theta$	$=$	$\lambda v \to (e\theta)$	
$(e_1\ e_2)\theta$	$=$	$(e_1\theta)\ (e_2\theta)$	
$(case\ e\ of\ \{\overline{p_i\ \to e_i;}\})\theta$	$=$	$case\ (e\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\}$	
$(let\ \overline{v_i = e_i;}\ in\ e)\theta$	$=$	$let\ \overline{v_i = (e_i\theta);}\ in\ (e\theta)$	
$(letrec\ f = e_1\ in\ e_2)\theta$	$=$	$letrec\ f = (e_1\theta)\ in\ (e_2\theta)$	

4. Можно расширить Соглашение 25 на случай применения подстановки – определить понятие корректной (или допустимой) подстановки по отношению к выражению e и рассматривать только корректные подстановки.

В контексте суперкомпиляции (точнее, – в контексте нахождения тесного обобщения двух выражений) наиболее удобен последний вариант.

Определение 27 (Допустимая HLL-подстановка)**.** HLL-подстановка θ допустима по отношению к выражению e, если

1. $bv[\![e]\!] \cap domain(\theta) = \emptyset$
2. $\forall e_i \in range(\theta) : bv[\![e]\!] \cap fv[\![e_i]\!] = \emptyset$

Соглашение 28 (Использование подстановок)**.** В дальнейшем мы рассматриваем только допустимые подстановки.

Соглашение 28 сродни так называемому соглашению Барендрегта ([12], 2.1.13). Использование соглашений 25 и 28 позволяет нам обеспечить корректность простого (“наивного”) применения подстановки:

Определение 29 (Применение HLL-подстановки)**.** Результат применения подстановки θ к выражению e, $e\theta$ вычисляется по правилам на Рис. 2.5.

2.4 Семантика языка HLL

В данном разделе описывается семантика языка HLL с вызовом по имени.

Рис. 2.6 HLL: декомпозиция выражений

obs	::=	$v\ \overline{e_i}$
	|	$c\ \overline{e_i}$
	|	$(\lambda v \to\ e)$
con	::=	$\langle\rangle$
	|	$con\ e$
	|	$case\ con\ of\ \{\overline{p_i \to e_i;}\}$
red	::=	f_g
	|	$(\lambda v \to e_0)\ e_1$
	|	$case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}$
	|	$case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}$
	|	$let\ \overline{v_i = e_i;}\ in\ e$
	|	$letrec\ f = e_1\ in\ e_2$

Определение 30 (Замкнутое HLL-выражение)**.** HLL-выражение называется замкнутым, если множество его свободных переменных пусто, т.е.

$$bv[\![e]\!] = \emptyset$$

Определение 31 (Контекст)**.** Контекст – выражение с дырой [ ] вместо одного из подвыражений. $C[e]$ - выражение, полученное из контекста заменой дыры на выражение e.

Любое выражение языка HLL можно представить либо в виде наблюдаемого выражения, либо в виде контекста редукции с помещенным в дыру редексом. Грамматика такой декомпозиции представлена на Рис. 2.6.

Определение 32 (Оператор неподвижной точки)**.** Оператор неподвижной точки определяется как специальная функция fix:

$$fix = \lambda f \to f(fix\ f);$$

Замечание 33 (Letrec)**.** Мы рассматриваем *letrec*-выражения как макрос (синтаксический сахар), который раскрывается следующим образом:

$$letrec\ f = e\ in\ e' \stackrel{\text{def}}{=} (\lambda f \to e')(fix\ (\lambda f \to e))$$

Определение 34 (Слабая головная нормальная форма)**.** Выражение

Рис. 2.7 Операционная семантика HLL

$c\ \overline{e_i}$	$\mapsto c\ \overline{e_i}$	(E_1)
$\lambda v_0 \to e_0$	$\mapsto \lambda v_0 \to e_0$	(E_2)
$con\langle f_0 \rangle$	$\mapsto con\langle unfold(f_0) \rangle$	(E_3)
$con\langle (\lambda v \to e_0)\ e_1 \rangle$	$\mapsto con\langle e_0\{v\ :=\ e_1\} \rangle$	(E_4)
$con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\} \rangle$	$\mapsto con\langle e_j\{\overline{v_{jk}\ :=\ e'_k}\} \rangle$	(E_5)
$con\langle let\ \overline{v_i = e_i;}\ in\ e \rangle$	$\mapsto con\langle e\{\overline{v_i := e_i}\} \rangle$	(E_6)
$con\langle letrec\ f = e\ in\ e' \rangle$	$\mapsto con\langle (\lambda f \to e')(fix\ (\lambda f \to e)) \rangle$	(E_7)

e языка HLL находится в слабой головной нормальной форме, если в нем на верхнем уровне находится конструктор (то есть $e = c\ \overline{e_i}$) или λ-абстракция (то есть $e = \lambda v \to\ e_1$).

Определение 35 (Шаг редукции)**.** Шаг редукции $\mapsto$ – наименьшее отношение на множестве замкнутых HLL выражений, удовлетворяющее правилам на Рис. 2.7.

Определение 36 (Сходимость)**.** Замкнутое выражение e сходится к слабой головной нормальной форме w, $e \Downarrow w$, если $e \mapsto^* w$.

$e \Downarrow$ обозначает, что существует w такое, что $e \Downarrow w$.

Определение 37 (Аварийное завершение с ошибкой времени выполнения)**.** Вычисление замкнутого выражения e завершается с ошибкой времени выполнения, $e \Uparrow error$, если $e \mapsto^* e'$, e' не является слабой головной формой и к e' не применимо ни одно из правил Рис. 2.7.

Определение 38 (Запуск HLL-программы)**.** Запуск HLL-программы с целевым выражением e состоит в задании такой подстановки θ, что $e\theta = e'$ – замкнутое выражение. Если $e\theta \Downarrow w$, то w является результатом запуска.

Определение 39 (Аппроксимация)**.** Выражение e_1 является операционной аппроксимацией выражения e_2, $e_1 \sqsubseteq e_2$, если в любом контексте C, таком, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, из $C[e_1] \Downarrow$ следует $C[e_2] \Downarrow$ и из $C[e_1] \Uparrow error$ следует $C[e_2] \Uparrow error$.

Определение 40 (Эквивалентность)**.** Выражение e_1 операционно эквивалентно выражению e_2, $e_1 \cong e_2$, если $e_1 \sqsubseteq e_2$ и $e_2 \sqsubseteq e_1$.

Рис. 2.8 Язык HLL (типизированный)

$tDef$ ::= data $tCon = \overline{dCon_i}$;	определение типа
$tCon$ ::= $tn\ \overline{tv_i}$	конструктор типа
$dCon$ ::= $c\ \overline{type_i}$	конструктор данных
$type$ ::= $tv \mid tCon \mid type \to type \mid (type)$	типовое выражение
$prog$ ::= $\overline{tDef_i}\ e$ **where** $\overline{f_i = e_i}$;	программа
e ::= e'	неявно типизированное
$\mid e' :: type$	явно типизированное
e' ::= v	переменная
$\mid c\ \overline{e_i}$	конструктор
$\mid f_g$	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
$\mid$ **letrec** $f = e_0$ **in** e_1	локальное определение
$\mid$ **let** $\overline{v_i = e_i;}$ **in** e	let-выражение
$\mid (e)$	выражение в скобках
p :: = $c\ \overline{v_i}$	образец

2.5 Типизация

Типизированный вариант языка HLL представлен на Рис. 2.8.

Программа на языке HLL состоит из определений типов данных, целевого выражения и определений верхнего уровня.

Левая часть определения типа содержит имя типа (точнее, имя конструктора типа), за которым следует список типовых переменных. Правая часть – декларации конструкторов данных (разделенные вертикальной чертой, то есть $\overline{dCon_i} \Rightarrow dCon_1 \mid \ldots \mid dCon_n$).

Мы требуем, чтобы в case-выражении были перечислены *все* конструкторы соответствующего типа данных – т.е. образцы в case-выражении должны быть *полны и ортогональны*. Также для упрощения последующих записей введем соглашение о порядке образцов в `case`-выражениях:

Соглашение 41 (Порядок образцов в `case`-выражениях)**.** Будем считать, что в `case`-выражениях образцы перечислены в том же порядке,

что и конструкторы в декларации соответствующего типа данных.

Язык HLL типизирован по Хиндли-Милнеру [22, 44, 110].

В HLL (как и в Haskell) программист может явно приписать тип выражению. При типизации по Хиндли-Милнеру это не является обязательным, – если программа корректно типизирована, то можно указать самые общие (principal) типы для всех выражений в программе. В случае наличия явных типов у выражений, происходит дополнительная проверка того, что указанные типы соответствуют выведенным типам. Однако, явное указание типов выражений и функций позволяет огранить контекст, в котором выражение может быть использовано. (Это также, как мы увидим, окажется необходимым для ограничения контекста, в котором может использоваться остаточная программа.)

С учетом типизации необходимо уточнить семантику языка HLL.

Определение 42 (Аварийное завершение с ошибкой типизации)**.** Вычисление замкнутого выражения e моментально завершается с ошибкой типизации, $e \Uparrow typingError$, если e не является корректно типизированным выражением.

В дальнейшем будем рассматривать только корректно типизированные программы. Корректно типизированная программа p может аварийно завершиться с ошибкой типизации, если пользователь указал означивание θ для свободных переменных целевого выражения e программы p, несовместимое по типам (выражение $e\theta$ нельзя корректно типизировать). С другой стороны, типизация гарантирует, что типизированная программа не может аварийно завершится с ошибкой времени выполнения. Таким образом, при рассмотрении эквивалентности выражений в соответствии с определением 40 необходимо принимать во внимание только возможные ошибки типизации выражений $C[e_1]$ и $C[e_2]$ и можно не рассматривать ошибки времени выполнения этих выражений (так как они не могут произойти в силу типизации).

Очевидно, что если *все* свободные переменные выражений e_1 и e_2 имеют одинаковый выведенный (или явно указанный) тип, то в любом контексте C из $C[e_1] \Uparrow typingError$ следует $C[e_2] \Uparrow typingError$ и наоборот.

Как мы увидим далее, в результате суперкомпиляции может получиться программа с более общими *выведенными* типами, нежели типы в исходной программе.

Замечание 43. До раздела 4.5 “Типизация и корректность” мы рассматриваем только неявно типизированные программы (в стиле Карри). И считаем, что мы рассматриваем только такие контексты C для выражения e, что выражение $C[e]$ корректно типизировано. В разделе 4.5 мы уточним вопрос типизации и корректности.

2.6 Алгебра HLL-выражений

Определение 44 (Равенство выражений)**.** Два выражения e_1 и e_2 считаются равными, если они различаются только именами связанных переменных. Равенство выражений e_1 и e_2 записывается как $e_1 \equiv e_2$.

Определение 45 (Частный случай выражения)**.** Выражение e_2 называется *частным случаем* выражения e_1, $e_1 \lessdot e_2$, если существует подстановка θ такая, что $e_1\theta \equiv e_2$. Для выражений языка HLL такая подстановка является единственной и обозначается $\theta = e_1 \circledcirc e_2$.

Определение 46 (Переименование выражения)**.** Выражение e_2 называется *переименованием* выражения e_1, $e_1 \simeq e_2$, если $e_1 \lessdot e_2$, и $e_2 \lessdot e_1$. Другими словами, e_1 и e_2 различаются только именами свободных переменных.

Определение 47 (Строгая декомпозицией)**.** Let-выражение

$$e_l = let\ \overline{v_i = e_i};\ in\ e_0$$

называется *строгой декомпозицией* выражения e, $e_l \sqsubset e$ если $e_0 \lessdot e$, $e_0 \not\simeq e$ и $e = e_0\{\overline{v_i := e_i}\}$.

Определение 48 (Обобщение)**.** Обобщением выражений e_1 и e_2, называется тройка $(e_g, \theta_1, \theta_2)$, где e_g – выражение, θ_1 и θ_2 – подстановки, такие, что $e_g\theta_1 \equiv e_1$ и $e_g\theta_2 \equiv e_2$. Множество всех обобщений для выражений e_1 и e_2 обозначается как $e_1 \frown e_2$.

Определение 49 (Тесное обобщение)**.** Обобщение $(e_g, \theta_1, \theta_2)$ называется тесным обобщением выражений e_1 и e_2, если для любого обобщения $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$ верно, что $e'_g \lessdot e_g$, т.е. e_g является частным случаем e'_g. Мы предполагаем, что определена операция $\sqcap$ такая, что $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$.

2.7 Выводы

В данном разделе формально и полностью описаны синтаксис и семантика языка HLL, близкого ядру языка Haskell.

При описании синтаксиса особое внимание уделено работе с переменными и применению подстановок. Вводятся простые *соглашения* об именах переменных и подстановках, которые устраняют возможные неоднозначности в описании дальнейших методов и алгоритмов.

Язык Haskell использует семантику вычислений "вызов по необходимости"[110]. И если суперкомпиляция используется как инструмент оптимизации, необходимо учитывать семантику "вызова по необходимости", как это делается в [78, 77, 15] и бережно рассматривать let-выражения. Требование сохранения семантики "вызова по необходимости" накладывает определенные ограничения на возможные преобразования программы и в некотором смысле уменьшает глубину преобразований программы. Однако, при семантике вычислений "вызов по имени", конечный результат вычислений будет тот же самый, разница будет лишь во времени вычислений и использовании памяти. Поэтому нам удобнее определить семантику вычислений "вызов по имени", – благодаря этому в дальнейшем мы сможем более агрессивно распространять позитивную информацию.

Операционная эквивалентность λ-выражений определена в [88] и рассматривается более общо в [96] в контексте операционной теории улучшений, используемой для доказательств корректности преобразований программ. Операционная семантика языка HLL определяется с учетом типизации и типизация учитывается при определении эквивалентности программ.

Конструкции языка HLL специально выбраны таким образом, чтобы язык HLL оставался как можно ближе к ядру языка Haskell и одновре-

менно минимизировать количество рассматриваемых в дальнейшем сущностей.

В данной главе заложен фундамент для полного и формального описания рассматриваемых далее алгоритмов и методов.

Глава 3

Структура суперкомпилятора HOSC

Суперкомпиляция является методом, а не алгоритмом. В конкретном суперкомпиляторе помимо идей и методов суперкомпиляции могут использоваться идеи и методы из других областей информатики.

Однако, сами методы суперкомпиляции достаточно абстрактны и их необходимо уточнять применительно к рассматриваемому языку, отталкиваясь от целей, с которыми создается суперкомпилятор. Необходимо уточнить, как представляются конфигурации, как строится частичное дерево процессов, как по частичному дереву процессов строится остаточная программа.

В данной главе описывается *методологическая* основа суперкомпилятора HOSC в виде отношения трансформации (а не конкретного алгоритма) и обосновываются выбранные методы с точки зрения выбранной цели – трансформационного анализа программ.

3.1 Устранение локальных определений

Непосредственно перед суперкомпиляцией исходная программа p преобразуется методом λ-лифтинга [47] в программу p', – в результате в программе p' отсутствуют `let`-выражения. Решено устранять `let`-выражения по следующим причинам:

1. Для выражений с `let`-выражениями понятия переименования, частного случая и обобщения определяются гораздо сложнее, – нужно учитывать возможность перестановки привязок в `let`-выражениях.

2. Традиционно `let`-выражения используются в суперкомпиляции для обозначения результата обобщения. Мы не будем отступать от этой практики. Если не устранять `let`-выражения, то необходимо различать `let`-выражения исходной программы и результаты обобщения, что приводит к излишним усложнения алгоритмов.

3. Поскольку суперкомпилятор HOSC предназначен для анализа программ, устранение `let`-выражений позволяет агрессивнее распространять позитивную информацию, что обеспечивает более глубокое преобразование программы. Поэтому в дальнейшем, если это особо не оговорено, рассматриваются HLL-выражения без конструкций `let`.

При разработке оптимизирующего суперкомпилятора для Haskell ([78, 77, 15]) устранять `let`-выражения таким образом недопустимо, так как при таком подходе остаточная программа может дублировать вычисления выражений, которые вычислялись в исходной программе один раз, и размер кода может сильно вырасти по сравнению с размером кода исходной программы.

3.2 Представление множеств

Построение дерева процессов и построение частичного дерева процессов являются метавычислениями. В метавычислениях рассматривают не только одиночное состояние вычисления, но также *множества* состояний вычислений. Как отмечено в [133], множества должны быть представлены конструктивно – в виде выражений некоторого языка.

Мы будем представлять множество состояний вычисления HLL-выражений в виде HLL-выражения со свободными переменными. Более точно, HLL-выражение со свободными переменными называется конфигурацией, а свободные переменные – конфигурационными переменными.

Соответственно, `let`-выражения представляет декомпозицию конфигураций. Мы допускаем только такие декомпозиции, которые являются строгими декомпозициями (см. Определение 47).

Существует и другие способы задания языка для представления конфигураций, например в работе “Rethinking Supercompilation” [77] конфи-

гурацией является только выражение в *специальной* форме. В работе "Supercompilation by Evaluation" [15] в качестве конфигурации рассматривается состояние абстрактной машины, состоящее из кучи, стека и активного выражения.

3.3 Построение частичного дерева процессов

Определение 50 (Дерево процессов для HLL-конфигурации)**.** Деревом процессов называется ориентированное (возможно бесконечное) дерево, каждому узлу которого приписана HLL-конфигурация. Дуги дерева процессов будем обозначать $\rightarrow$.

В результате метавычислений в общем случае строится бесконечное дерево процессов. Задача суперкомпиляции – превратить его в конечный (возможно, циклический) граф, который называется *частичным деревом процессов*.

Определение 51 (Частичное дерево процессов для HLL-конфигурации)**.** Частичное дерево процессов является расширением понятия дерева процессов и отличается от обычного дерева процессов следующими свойствами:

- В узлах частичного дерева процессов помимо конфигураций могут присутствовать декомпозиции конфигураций.
- В частичном дереве процессов могут быть *циклы*: пусть лист β является потомком узла α и выражение в узле β является переименованием выражения в узле α ($\beta.expr \simeq \alpha.expr$). Тогда можно провести специальную дугу $\beta \rightrightarrows \alpha$ из *повторного* (рекурсивного) узла β в *базовый* (или функциональный) узел α. Из узла может выходить не более одной специальной дуги $\rightrightarrows$.

Определение 52 (Обработанный лист)**.** Лист дерева β называется обработанным, если находящееся в нем выражение $\beta.expr$ является константой, конфигурационной переменной или из него выходит специальная дуга $\rightrightarrows$.

Рис. 3.1 Операции над частичным деревом процессов

$incomplete(t)$	Возвращает $true$, если дерево t не является завершенным.
$trivial(node)$	Возвращает $true$ или $false$ в зависимости от того, является данный узел тривиальным или нет.
$children(t,\ \alpha)$	Возвращает упорядоченный список дочерних узлов узла α дерева t
$addChildren(t,\ \beta,\ es)$	Подвешивает к узлу β дерева t дочерние узлы и помещает в них выражения es. Количество подвешенных узлов совпадает с количеством элементов списка es.
$replace(t,\ \alpha,\ expr)$	Заменяет узел α на новый узел β такой, что $\beta.expr = expr$. Поддерево, имеющее своим корнем α, удаляется.
$ancestors(t,\ \beta)$	Возвращает узел всех предков узла β
$find(ns,\ \beta,\ p)$	Находит первый среди узлов ns узел α, такой, что верен предикат $p(\alpha.expr,\ \beta.expr)$.
$fold(t,\ \alpha,\ \beta)$	"Зацикливает" α и β: $\beta \rightrightarrows \alpha$.
$abstract(t,\ \alpha,\ \beta)$	$= replace(t,\ \alpha,\ let\ \overline{v_i := e_i};\ in\ e_g)$, где $(e_g, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $e_g\theta_1 = e_g\{\overline{v_i := e_i}\} = let\ \overline{v_i = e_i};\ in\ e_g$.
$\lceil \alpha \not\downarrow t \rceil$	Возвращает все повторные узлы α, $\lceil \alpha \not\downarrow t \rceil = [\overline{\beta_i}] : \beta_i \rightrightarrows \alpha$, или $\bullet$, если α не является базовым узлом.
$\lceil \alpha \uparrow\uparrow t \rceil$	Возвращает базовый узел α, $\lceil \alpha \uparrow\uparrow t \rceil = \beta : \alpha \rightrightarrows \beta$, или $\bullet$, если α не является повторным узлом.
$drive(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}[\![\alpha.expr]\!])$ - делает шаг прогонки с распространением позитивной информации.
$drive_0(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}_0[\![\alpha.expr]\!])$ - делает шаг прогонки без распространения позитивной информации.
$drive^*(t, \alpha)$	$= choice\{drive_0(t, \alpha),\ drive(t, \alpha)\}$
$decompose(t,\ \beta)$	$= replace(t,\ \beta,\ e_1)$, где e_1 – некоторая строгая декомпозиция конфигурации $\beta.expr$.
$unprocessedLeaf(t)$	Возвращает самый левый необработанный лист α дерева t, или $\bullet$, если все листья обработаны.

Определение 53 (Завершенное частичное дерево процессов)**.** Частичное дерево процессов является завершенным, если все его листья являются обработанными.

Операции над частичным деревом, которые нам понадобятся в дальнейшем, представлены на Рис. 3.1 в формальном виде. Схематическое представление некоторых операций можно найти на Рис. 1.9. Некоторые операции не используются в данной главе, но будут использоваться дальше – в главе 5.

Рис. 3.2 Шаг прогонки без распространения позитивной информации

$$
\begin{array}{lcll}
\mathcal{D}_0[\![v\ \overline{e_i}]\!] & \Rightarrow & [v,\ \overline{e_i}] & (D_1^0) \\
\mathcal{D}_0[\![c\ \overline{e_i}]\!] & \Rightarrow & [\overline{e_i}] & (D_2^0) \\
\mathcal{D}_0[\![\lambda v_0 \to e_0]\!] & \Rightarrow & [e_0] & (D_3^0) \\
\mathcal{D}_0[\![con\langle f_0\rangle]\!] & \Rightarrow & [con\langle unfold(f_0)\rangle] & (D_4^0) \\
\mathcal{D}_0[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] & \Rightarrow & [con\langle e_0\{v_0 := e_1\}\rangle] & (D_5^0) \\
\mathcal{D}_0[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] & \Rightarrow & \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] & (D_6^0) \\
\mathcal{D}_0[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] & \Rightarrow & \left[v\ \overline{e'_j},\ \overline{con\langle e_i\rangle}\right] & (D_7^0) \\
\mathcal{D}_0[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & \Rightarrow & [e,\ \overline{e_i}] & (D_8^0)
\end{array}
$$

Рис. 3.3 Шаг прогонки с распространением позитивной информации

$$
\begin{array}{lcll}
\mathcal{D}[\![v\ \overline{e_i}]\!] & \Rightarrow & [v,\ \overline{e_i}] & (D_1) \\
\mathcal{D}[\![c\ \overline{e_i}]\!] & \Rightarrow & [\overline{e_i}] & (D_2) \\
\mathcal{D}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & [e_0] & (D_3) \\
\mathcal{D}[\![con\langle f_0\rangle]\!] & \Rightarrow & [con\langle unfold(f_0)\rangle] & (D_4) \\
\mathcal{D}[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] & \Rightarrow & [con\langle e_0\{v_0 := e_1\}\rangle] & (D_5) \\
\mathcal{D}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] & \Rightarrow & \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] & (D_6) \\
\mathcal{D}[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] & \Rightarrow & \left[v\ \overline{e'_j},\ \overline{con\langle e_i\rangle\{v\ \overline{e'_j} := p_i\}}\right] & (D_7) \\
\mathcal{D}[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & \Rightarrow & [e,\ \overline{e_i}] & (D_8)
\end{array}
$$

Остановимся подробно на операциях прогонки – $drive_0$ и $drive$. Операция $drive_0$ делает шаг прогонки без распространения позитивной информации. Оператор $\mathcal{D}_0$ (Рис. 3.2) принимает в качестве аргумента выражение, находящееся в листе, и возвращает ноль или более выражений $e_1, \ldots, e_k$. Результатом выполнения операции $drive_0(t, \beta)$ является подвешивание к листу β k дочерних узлов с выражениями $e_1, \ldots, e_k$. Операция $drive$ отличается от операции $drive_0$ только тем, что для получения выражений в дочерних узлах используется оператор $\mathcal{D}$ (Рис. 3.2).

Рис. 3.4 *HOSC*: построение дерева процессов для конфигурации e_0

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

Рис. 3.5 *HOSC*: построение частичного дерева процессов 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

Операторы $\mathcal{D}_0$ и $\mathcal{D}$ отличаются только седьмым правилом. При использовании оператора $\mathcal{D}$, когда необходимо сделать шаг прогонки для выражения вида $con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$, рассматриваются различные ветви с учетом того, что вычисление может достигнуть i-й ветви case-выражения только в том случае, когда $v\ \overline{e'_j}$ имеет вид p_i, – информация об этом распространяется в тело выбранной ветви. Оператор $\mathcal{D}_0$ не распространяет эту информацию. Особенностью оператора $\mathcal{D}_0$ является то, что при применении седьмого правила все выражения в дочерних узлах буду строго меньше по размеру, чем выражение в родительском узле – это факт будет использован в главе 5, для того, чтобы гарантировать завершаемость суперкомпилятора.

Рассмотрим построение дерева процессов для конфигурации e_0. Соответствующий алгоритм представлен на Рис. 3.4. Вначале создается дерево t с единственным узлом, в который помещается стартовая конфигурация: $t = \boxed{e_0}$. Затем, пока есть хотя бы один необработанный узел, к нему применяются правила прогонки. В большинстве случаев дерево процессов будет бесконечным.

Перейдем к рассмотрению построения частичного дерева процессов

для стартовой конфигурации e_0. Недетерминированный алгоритм построения частичного дерева процессов для выражения e_0 приведен на Рис. 3.5. Оператор *choice* используется как оператор недетерминированного выбора – выполняется любая (допустимая) операция из аргументов оператора *choice*.

Алгоритм построения частичного дерева процесса процессов является расширением алгоритма построения дерева процессов – при построении дерева процессов для необработанного узла осуществляется шаг прогонки с распространением позитивной информации. При построении частичного дерева процессов по алгоритму на Рис. 3.5 разрешается для необработанного узла либо осуществить шаг прогонки с распространением позитивной информации, либо осуществить шаг прогонки без распространения позитивной информации, либо произвести декомпозицию конфигурации, либо, если среди предков есть совпадающая конфигурация (с точностью до переименования конфигурационных переменных), выбрать любую такую конфигурацию и провести специальную дугу $\rightrightarrows$.

Можно рекурсивно перечислить все частичные деревья процессов, которые могут быть построены по данному недетерминированному алгоритму для стартовой конфигурации e_0 – достаточно на каждом шаге рассмотреть все возможные выборы. Обозначим такое множество завершенных частичных деревьев процессов как $T_{HOSC}(e_0)$. Отметим, что в общем случае среди всех возможных трасс выборов построение частичного дерева процессов завершается только на малой части трасс.

3.4 Генерация остаточной программы

Любое частичное дерево процессов $t \in T_{HOSC}(e_0)$, построенное для стартовой конфигурации e_0, является достаточным для вычисления любого выражения $e \in e_0$ (без свободных переменных) в любом контексте. Дерево процессов t также можно преобразовать в остаточную программу. Однако, как и при построении частичного дерева процессов, у нас есть степени свободы – можно разными способами преобразовать частичное дерево процессов в остаточную программу, которая будет эквивалентна исходной.

Рис. 3.6 *HOSC*: генерация остаточной программы

$$
\begin{array}{lll}
\mathcal{C}\ [\![\alpha]\!]_{t,\Sigma} & & \\
\quad \Rightarrow letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\theta'\ in\ f'\overline{v'_i} & \text{если } \lceil \alpha \between t \rceil \neq \bullet & (C_1^*) \\
\qquad \text{где} & & \\
\qquad\quad [\overline{\beta_i}] = \lceil \alpha \between t \rceil,\ \theta_i = \alpha.expr \otimes \beta_i.expr, & & \\
\qquad\quad \overline{v'_i} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v'_i := v_i}\}, & & \\
\qquad\quad \Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ и } \overline{v_i} - \text{новые} & & \\
\quad \Rightarrow f'_{sig}\theta & \text{если } \lceil \alpha \Uparrow t \rceil \neq \bullet & (C_2^*) \\
\qquad \text{где} & & \\
\qquad\quad f'_{sig} = \Sigma(\lceil \alpha \Uparrow t \rceil),\ \theta = \lceil \alpha \Uparrow t \rceil.expr \otimes \alpha.expr & & \\
\quad \Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} & \text{иначе} & (C_3^*)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\} & (C'_1) \\
\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C'_2) \\
\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C'_3) \\
\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C'_4) \\
\mathcal{C}'[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} & \\
\quad \Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of\{\overline{p_i \to \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma'};}\} & (C'_5) \\
\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C'_6) \\
\mathcal{C}'[\![con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C'_7) \\
\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C'_8)
\end{array}
$$

Чтобы уменьшить громоздкость правил, принято следующее соглашение. Если в правой части правила используется γ_i, считается, что: $[\overline{\gamma_i}] = children(t,\ \alpha)$, – в этом случае все дочерние узлы рассматриваются единообразно. Если же в правой части используются γ' и γ'_i, считается, что: $[\gamma',\ \overline{\gamma'_i}] = children(t,\ \alpha)$, – это соглашение используется в случае, когда дочерний узел заведомо один (в правой части правила используется только γ') или же первый дочерний узел требует “особого рассмотрения”.

В данном разделе описывается детерминированный алгоритм преобразования графа конфигураций t в программу на языке HLL.

Алгоритм (представленный на Рис. 3.6) определяется через два взаимно рекурсивных оператора (функции): $\mathcal{C}$ и $\mathcal{C}'$. Остаточная программа для графа конфигураций t определяется как

$$\mathcal{C}\ [\![t.root]\!]_{t,\{\}}$$

Для построения остаточной программы мы обходим частичное дерево сверху вниз (начиная, естественно, с корня). При обходе базового узла,

генерируется определение рекурсивной функции в форме letrec-а. Соответствие между базовым узлом и сигнатурой заносится в Σ. Впоследствии Σ используется для генерации рекурсивного вызова.

Алгоритм обладает следующими особенностями:

1. Остаточная программа является одним самодостаточным выражением без глобальных определений – все рекурсивные функции определяются локально.

2. В остаточной программе есть только рекурсивные функции.

3. При конструировании рекурсивных функций аргументами становятся только *изменившиеся* части конфигураций, что позволяет понизить арность рекурсивных функций в остаточной программе.

Как показывают эксперименты (см. главы 5, 7 и 8), эти особенности положительно сказываются на способности суперкомпилятора $HOSC$ к распознаванию эквивалентности выражений и улучшающих лемм.

3.5 Отношение транформации HOSC

Определение 54 (Отношение трансформации $HOSC$)**.** Исходная программа $prog$ и остаточная программа $prog'$ связаны отношением трансформации $HOSC$ ($prog\ HOSC\ prog'$), если существует частичное дерево процессов $t \in T_{HOSC}(prog)$ такое, что $prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$, где $\mathcal{C}$ определяется по правилам на Рис. 3.6.

Отношение трансформации $HOSC$ задает для данной входной программы p множество остаточных программ. Как было отмечено ранее, для целей анализа может быть полезно иметь несколько остаточных программ, – возможно, что некоторые остаточные программы будут легче поддаваться анализу, чем другие.

В следующей главе будет показано, что если две программы связаны отношением трансформации $HOSC$, то они эквивалентны.

3.6 Выводы

В данной главе описана *методологическая* основа суперкомпилятора HOSC в виде отношения трансформации (а не конкретного алгоритма) и обоснованы выбранные методы с точки зрения выбранной цели – трансформационного анализа программ.

Отношение трансформации определяется как *недетерминированный* алгоритм построения частичного дерева процессов для стартовой конфигурации e_0 и детерминированный алгоритм преобразования частичного дерева процессов в остаточную программу. Недетерминированный алгоритм задает *множество* частичных деревьев процессов $T_{HOSC}(e_0)$. Соответственно, отношение $HOSC$ определяет *множество* остаточных программ для данной входной программы.

Суперкомпиляция как метод преобразования программ допускает различные конкретные реализации. Подавляющее большинство авторов рассматривают суперкомпиляторы как конкретные алгоритмы (иногда допускающие некоторую параметризацию). Отношение трансформации $HOSC$ также допускает множество конкретных алгоритмов, но фиксирует несколько существенных моментов:

1. Непосредственно перед суперкомпиляцией исходная программа преобразуется в программу без глобальных определений методом λ-лифтинга.

2. Множество состояний вычисления представляется в виде HLL-выражения со свободными (конфигурационными) переменными.

3. Определяется шаг прогонки – его можно сделать как с распространением позитивной информации, так и без распространения позитивной информации.

4. Разрешается зацикливать любые конфигурации.

5. Остаточная программа является самодостаточным выражением, где присутствуют только локальные рекурсивные функции. Фиксируется алгоритм генерации остаточной программы из частичного дерева процессов.

Глава 4

Корректность суперкомпилятора HOSC

В данной главе доказывается корректность отношения трансформации $HOSC$. Любые две программы, связанные через отношение трансформации $HOSC$, операционно эквивалентны.

Для доказательства корректности мы пользуемся операционной теорией улучшений Сэндса [95, 97], в основе которой лежит понятие стоимости вычисления и в качестве единицы стоимости используется вызов функции. Говоря неформально, выражение e_1 является *улучшением* выражения e_0, если вычисление e_1 обходится не дороже, чем вычисление выражения e_0. Выражение e_1 является *строгим улучшением* выражения e_0, если e_1 является улучшением e_0 и e_1 и e_0 – эквивалентны.

Дефорестация [74] – техника преобразования программ, являющаяся подмножеством суперкомпиляции. В работе [96] была доказана корректность дефорестации функций высших порядков с помощью теории улучшений (среди прочего, было показано, что результат дефорестации является строгим улучшением исходной программы).

Ключевым (и очень существенным) моментом доказательства корректности дефорестации является то, что при дефорестации программы свертка двух конфигураций c_1 и c_2 допускается только в случае, когда $c_1 = con\langle f \rangle$ (то есть следующим шагом вычислений будет замена вызова функции f на ее определение).

Отношение трансформации HOSC не имеет такого ограничения, – допускается свертка *любых* конфигураций.

Рассмотрим вначале отношение $HOSC_0$, допускающее свертку только

узлов вида $con\langle f\rangle$. $HOSC_0$ отличается от дефорестации функций высших порядков только наличием шага обобщения. Достаточно прямолинейно доказывается утверждение, что любая остаточная программа, удовлетворяющая отношению $HOSC_0$, является строгим улучшением исходной программы, и, следовательно, остаточная програма эквивалентна исходной.

Затем докажем более общие утверждения о корректности отношений трансформации $HOSC_{1/2}$ и $HOSC$.

4.1 Операционная теория улучшений

Определение 55 (Улучшение)**.** Выражение e_2 является улучшением выражения e_1, $e_1 \trianglerighteq e_2$, если в любом контексте C, таком, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, при завершении вычисления выражения $C[e_1]$ за n вызовов функций, вычисление выражения $C[e_2]$ завершается не более чем за n вызовов функций.

Определение 56 (Строгое улучшение)**.** Выражение e_2 является строгим улучшением выражения e_1, $e_1 \trianglerighteq_s e_2$, если $e_1 \trianglerighteq e_2$ и $e_1 \ncong e_2$.

Определение 57 (Эквивалентность стоимости вычислений)**.** Выражения e_1 и e_2 эквивалентны по стоимости вычислений $e_1 \lhd\!\rhd e_2$, если $e_1 \trianglerighteq e_2$ и $e_2 \trianglerighteq e_1$.

Определение 58 (Преобразование определений)**.** Пусть имеется программа с определениями $\{f_i = e_i;\}$ и программа с определениями $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$, где $\overline{e'_i}$ не зависят от $\overline{g_i}$, а $\overline{g_i}$ не зависят соответственно от $\overline{f_i}$. Считается, что $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – преобразование $\{f_i = e_i;\}$.

Теорема 59 (Теорема об улучшении [96])**.** *Пусть* $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – *преобразование* $\{f_i = e_i;\}$ *такое, что* $e_i \trianglerighteq e'_i$*, тогда* $f_i \trianglerighteq g_i$.

Следствие 60. *Пусть* $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – *преобразование* $\{f_i = e_i;\}$ *такое, что* $e_i \trianglerighteq_s e'_i$*, то* $f_i \trianglerighteq_s g_i$.

Теорема 61 (Теорема о локальном улучшении [96])**.** *Если переменные* h *и* $\overline{x_i}$ – *это все свободные переменые выражений* e_0 *и* e_1 *и*

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_0 \trianglerighteq_s letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_1$$

Рис. 4.1 $HOSC_0$: построение частичного дерева процессов

$t = \boxed{e_0}$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive^*(t,\beta), decompose(t,\beta), fold_0(t,\beta)\}$
end

то для всех выражений e справедливо

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e \trianglerighteq_{\sim_s} letrec\ h = \lambda\overline{x_i} \to e_1\ in\ e$$

Утверждение 62 (Леммы об улучшениях [96]). *Справедливы следующие утверждения:*

1. *Если $e \trianglerighteq_{\sim_s} e'$, то $C[e] \trianglerighteq_{\sim_s} C[e']$*
2. $con\langle case\ e\ of\ \{\overline{p_i \to e_i;}\}\rangle \lhd\!\rhd case\ e\ of\ \{\overline{p_i \to con\langle e_i\rangle;}\}$
3. $con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := e\};}\}\rangle \trianglerighteq_{\sim_s} con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := p_i\};}\}\rangle$
4. *Если $e \mapsto e'$, то $C[e] \trianglerighteq_{\sim_s} C[e']$*
5. *Для всех выражений e и подстановок θ таких, что $h \notin domain(\theta)$, если $e_0 \mapsto t$, то*

$$letrec\ h = \lambda\overline{y_i} \to t\ in\ e\{z := e_0\theta\} \lhd\!\rhd letrec\ h = \lambda\overline{y_i} \to t\ in\ e\{z := (h\ \overline{y_i})\theta\}$$

Пусть e и e' – целевые выражения программ $prog$ и $prog'$ соответственно. В дальнейшем используется следующее соглашение: будем писать $prog \trianglerighteq_{\sim_s} prog'$, если $e \trianglerighteq_{\sim_s} e'$ и $e \cong e'$, если $prog \cong prog'$.

Под корректностью отношения трансформации T мы понимаем следующее:

$$prog\ T\ prog' \Rightarrow prog \cong prog'$$

Рис. 4.2 $HOSC_0$: генерация остаточной программы

$\mathcal{C}\ [\![\alpha]\!]_{t,\Sigma}$

$\Rightarrow letrec\ f' = \lambda\overline{v_i} \rightarrow (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\ in\ f'\overline{v_i}$ если $\lceil \alpha \between t \rceil \neq \bullet$ (C_1)

где

$\overline{v_i} = fv(\alpha.expr)$

$\Sigma' = \Sigma \cup (\alpha,\ \{f' := \lambda\overline{v_i} \rightarrow \alpha.expr\})$,

f' – новая

$\Rightarrow f'\overline{v_i}$ если $\lceil \alpha \Uparrow t \rceil \neq \bullet$ (C_2)

где

$f' = domain(\Sigma(\lceil \alpha \Uparrow t \rceil))$, $\overline{v_i} = fv(\alpha.expr)$

$\Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma}$ иначе (C_3)

$\mathcal{C}'[\![let\ \overline{v_i = e_i};\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma}}\}$ (C_1')

$\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}}$ (C_2')

$\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}}$ (C_3')

$\mathcal{C}'[\![\lambda v_0 \rightarrow e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C_4')

$\mathcal{C}'[\![con\langle case\ v\ \overline{e_j'}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle]\!]_{t,\alpha,\Sigma}$

$\Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of \{\overline{p_i \rightarrow \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\}$ (C_5')

$\mathcal{C}'[\![con\langle(\lambda v_0 \rightarrow e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C_6')

$\mathcal{C}'[\![con\langle case\ c\ \overline{e_j'}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C_7')

$\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C_8')

4.2 Корректность отношения трансформации $HOSC_0$

Докажем вначале корректность отношения трансформации $HOSC_0$, позволяющего осуществлять свертку только узлов вида $con\langle f\rangle$.

Это отношение представлено на Рис. 4.1 и Рис. 4.2. При генерации остаточной программы по правилам на Рис. 4.2 осуществляется рекурсивный обход частичного дерева процессов. При этом обходе в таблицу Σ заносятся соответствия базовых узлов и сигнатур новых рекурсивных функций.

Определение 63 (Отношение трансформации $HOSC_0$)**.** Исходная программа $prog$ и остаточная программа $prog'$ связаны отношением трансформации $HOSC_0$ ($prog\ HOSC_0\ prog'$), если среди реализаций частичного дерева процесса для программы $prog$ по недетерминированному алгоритму на Рис. 4.1 существует дерево t такое, что $prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$, где $\mathcal{C}$ определяется по правилам на Рис. 4.2.

При генерации остаточной программы по правилам на Рис. 4.2 *все* свободные переменные базовой конфигурации становятся аргументами новой локальной рекурсивной функции.

Будем считать, что $\Sigma(\bullet) = \{\}$ (пустая сигнатура). Определим подстановку

$$\rho_\Sigma = range(\Sigma)$$

Теорема 64. *Для любого дерева процесса t, построенного по правилам на Рис. 4.1, при генерации остаточной программы по правилам на Рис. 4.2 для любого состояния обхода дерева верно:*

$$\alpha.expr \trianglerighteq_s (\mathcal{C}[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

Доказательство. Доказываем по индукции по структуре выражения $\alpha.expr$. Рассмотрим вначале правило (C_3), – необходимо доказать, что

$$\alpha.expr \trianglerighteq_s (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

- (C'_1) Требуется показать, что

$$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

  По правилу (C'_1):

$$(\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma = (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\})\rho_\Sigma$$

  В силу построения ρ_Σ:

$$(\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\})\rho_\Sigma = (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

  Таким образом, необходимо показать, что:

$$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

  По построению дерева (правило D_8 3.3): $e = \gamma'.expr,\ e_i = \gamma'_i.expr$.

По гипотезе индукции:

$$e \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad e_i \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

Отсюда по лемме 62(1) следует:

$$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

- (C'_2) Требуется показать, что

$$v\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

По гипотезе индукции:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

Отсюда по лемме 62(1) следует требуемое.

- (C'_3) Требуется показать, что

$$c\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

По гипотезе индукции:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

Отсюда по лемме 62(1) следует требуемое.

- (C'_4) Требуется показать, что

$$\lambda v_0 \to e_0 \trianglerighteq_s (\lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma})\rho_\Sigma = \lambda v_0 \to (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)$$

По гипотезе индукции:

$$e_0 \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma$$

Отсюда по лемме 62(1) следует требуемое.

- (C'_5) Требуется показать, что

$$con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle \trianglerighteq_s (case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma} of\{\overline{p_i \to \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma};}\})\rho_\Sigma =$$

$$= case\ (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma) of\{\overline{p_i \to (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma);}\}$$

Здесь требуется рассмотреть два случая – шаг прогонки с распространением позитивной информации и шаг прогонки без распространения позитивной информации. Рассмотрим вначале первый случай. По построению дерева (правило D_7 3.3):

$$\gamma'.expr = v\ \overline{e'_j}, \quad \gamma'_i.expr = con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle$$

По гипотезе индукции:

$$v\ \overline{e'_j} \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

Отсюда по леммам 62(1, 2, 3) следует требуемое. Правило прогонки D^0_7 3.2 рассматривается аналогично.

- $(C'_6), (C'_7), (C'_8)$. В этих случаях

$$\alpha.expr \mapsto \gamma'.expr$$

По гипотезе индукции и лемме 62(4) следует требуемое.

Рассмотрим теперь правило (C_2). Требуется показать, что

$$\alpha.expr \trianglerighteq_s (\mathcal{C}[\![\alpha.expr]\!]_{t,\Sigma})\rho_\Sigma = (f'\overline{v_i})\rho_\Sigma$$

По построению (операция $fold_0$ на Рис. 3.1)

$$(f'\overline{v_i})\rho_\Sigma = (\lambda\overline{v_i} \to \alpha.expr)\overline{v_i}$$

Из

$$(\lambda\overline{v_i} \to \alpha.expr)\overline{v_i} \trianglelefteq \alpha.expr$$

следует требуемое.

Рис. 4.3 Вспомогательная лемма

Лемма 65.

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \lessapprox letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Доказательство. Рассмотрим выражение

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

В соответствии с правилом (C_2) все свободные вхождения f' находятся в подвыражениях вида $f'\overline{v_i'}$, Предположим что таких вхождений n штук – $\overline{f'\overline{v_i}\theta_k}$, где θ_k – просто переименование переменных $\overline{v_i}$. Тогда

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'}) = e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}$$

где $f' \notin fv(e')$. Значит,

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \equiv (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\}$$

$$\lessapprox e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\} \lessapprox e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

Так как $\alpha.expr \mapsto \gamma'.expr$, то по лемме 62(5)

$$e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

$$\lessapprox letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma$$

$$\equiv letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

□

Осталось рассмотреть правило (C_1). По построению надо показать, что

$$\alpha.expr \trianglerighteq_s (letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\ in\ f'\overline{v_i})\rho_\Sigma$$

Что равносильно

$$\alpha.expr \trianglerighteq_s letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

Так как

$$\alpha.expr \mapsto \gamma'.expr$$

то по лемме 62(5)

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \alpha.expr \lessapprox letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Так как $f' \notin fv(\alpha.expr)$, то

$$\alpha.expr \lessapprox letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Таким образом, достаточно показать, что

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i} \gtrsim_s$$

$$\gtrsim_s letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

По теореме 61 достаточно показать, что

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr \gtrsim_s$$

$$\gtrsim_s letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Исходя из того, что $letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr = \gamma'.expr$ и леммы 65, достаточно показать, что

$$\gamma'.expr \gtrsim_s (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

что соответствует гипотезе индукции. □

4.3 Корректность отношения трансформации $HOSC_{1/2}$

Трансформация $HOSC_0$ позволяет делать свертку только конфигураций вида $con\langle f \rangle$. В данном разделе покажем, что отношение трансформации $HOSC_{1/2}$, позволяющее делать свертку *любых* конфигураций, корректно.

Определение 66 (Отношение трансформации $HOSC_{1/2}$)**.** Исходная программа $prog$ и остаточная программа $prog'$ связаны отношением трансформации $HOSC_{1/2}$ ($prog\ HOSC_{1/2}\ prog'$), если среди реализаций частичного дерева процесса для программы $prog$ по недетерминированному алгоритму на Рис. 3.5 существует дерево t такое, что $prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$, где $\mathcal{C}$ определяется по правилам на Рис. 4.2.

Теорема 67 (Корректность $HOSC_{1/2}$)**.** *Для любого дерева процесса t, построенного по правилам на Рис. 4.1, при генерации остаточной программы по правилам на Рис. 3.6 верно: $e \cong \mathcal{SC}_{1/2}[\![e]\!]$*

Доказательство. Для доказательства корректности отношения $HOSC_{1/2}$ применяется следующий прием: рассмотрим в частичном дереве процессов t, допускаемом трансформацией $HOSC_{1/2}$, свернутые конфигурации c_1, c_2, ..., декомпозиция которых не соответствует виду $con\langle f\rangle$. Мы вставим непосредственно сверху узлов с конфигурациями c_1, c_2, ... узлы с конфигурациями c'_1, c'_2, ... вида $con\langle g\rangle$, где g – некоторая новая функция и $c'_i \to c_i$, и перенесем свертку со старых конфигураций на новые конфигурации. Мы анализируем историю появления конфигураций c_1, c_2, ..., и находим подвыражения в исходной программе e_1, e_2, ..., которые при прогонке трансформировались в конфигурации c_1, c_2, ... Заменяем найденные выражения e_i на $g\ \overline{e'_i}$. Следует отметить, что для этого, возможно, потребуется привести части исходной программы к суперкомбинаторному виду [85] с абстракцией максимально свободных выражений (см. пример в разделе 4.3.1).

Повторяем это преобразование, пока в частичном дереве процессов остаются свернутые конфигурации, не допускаемые $HOSC_0$. В итоге получаем частичное дерево процессов t' и новую программу $prog'$. В силу построения, остаточные программы, сгенерированные по деревьям t и t' будут совпадать. Обозначим эти программы $prog''$. Также в силу построения, дерево t' допустимо трансформацией $HOSC_0$. По определению e_i является строгим улучшением $g\ \overline{e'_i}$ (так как $g\ \overline{e'_i} \to e_i$), то есть $prog$ является строгим улучшением $prog'$ (по теореме об улучшениях). Имеем:

$$prog' \trianglerighteq_{\lesssim_s} prog,\ prog'' = \mathcal{SC}_0[\![prog']\!]$$

Отсюда (в силу того, что трансформация $HOSC_0$ – строго улучшающая) следует:

$$prog' \trianglerighteq_{\lesssim_s} prog''$$

По определению строгого улучшения

$$prog \cong prog''$$

Рис. 4.4 Программа *prog*

```
data Stream = S Stream;

case x of {S y1 → (S (id1 y1));} where

id1 = λx → case (id x) of { S y → S (id1 y);};
id  = λx → x;
```

Рис. 4.5 Программа $prog'$

```
data Stream = S Stream;

g x where

id1 = λx → g (id x);
g = λx → case x of {S y → S (id1 y);};
id = λx → x;
```

Но с другой стороны

$$prog'' = \mathcal{SC}_{1/2}[\![prog]\!]$$

Следовательно

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

□

4.3.1 Пример сведения отношения $HOSC_{1/2}$ к отношению $HOSC_0$

Рассмотрим простую программу *prog* на Рис. 4.4. На Рис. 4.7 показано частичное дерево процессов t, построенное для этой программы трансформацией $HOSC_{1/2}$ (для уменьшения размеров дерева, тривиальная β-редукция пропущена). Отношение трансформации $HOSC_{1/2}$ позволяет нам сделать свертку конфигураций с case-выражениями. Результат такой суперкомпиляции является программа $prog''$, показанная на Рис. 4.6.

Используя прием, описанный ранее, строим программу $prog'$. Программа *prog* является улучшением программы $prog'$, показанной на Рис. 4.5. Видно, что при построении $prog'$, нам пришлось абстрагировать подвыражение `id x` внутри определения функции `id1`. На Рис. 4.7 показано

Рис. 4.6 Программа *prog″*

```
data Stream = S Stream;
letrec f = λp → case  p  of {S y →  S (f y);} in f x
```

Рис. 4.7 Частичное дерево процессов для *prog* по $HOSC_{1/2}$

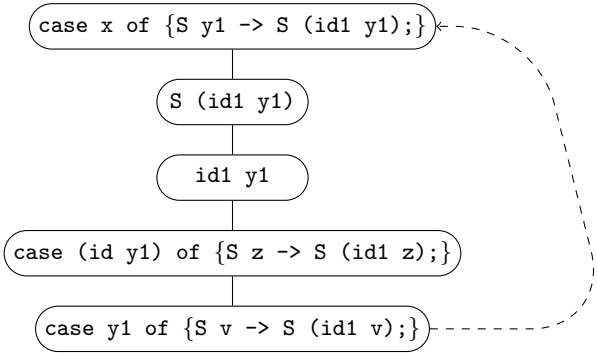
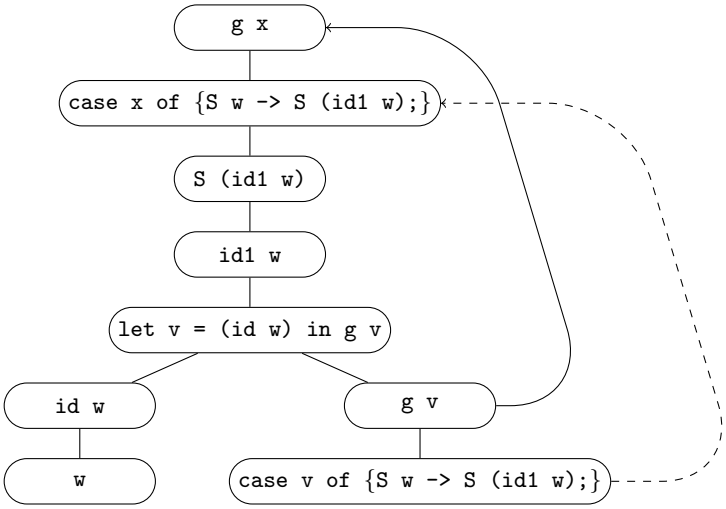


Рис. 4.8 Преобразование дерева $HOSC_{1/2}$ в дерево $HOSC_0$



совмещение дерева t', построенного для *prog′* трансформацией $HOSC_0$, и дерева t. Обратные дуги дерева t обозначены пунктиром, обратные дуги дерева t' обозначены сплошной линией. Очевидно, что остаточные программы для деревьев t и t' совпадают. Это является иллюстрацией того,

Рис. 4.9 `russel (MkU russel)`: исходная программа

```
data Bool = True | False;
data U = MkU (U → Bool);

russel (MkU russel) where

russel = λu → case u of {MkU p → p u;};
```

что $prog'' \cong prog$.

4.4 Корректность отношения трансформации $HOSC$

Теорема 68 (Корректность $HOSC_{1/2}$)**.** *Для любого дерева процесса t, построенного по правилам на Рис. 3.5, при генерации остаточной программы по правилам на Рис. 3.6 верно: $e \cong \mathcal{SC}_{1/2}[\![e]\!]$*

Доказательство. Отношение между $\mathcal{SC}_{1/2}[\![prog]\!]$ и $\mathcal{SC}[\![prog]\!]$ есть ничто иное, как частный случай λ-отбрасывания (λ-dropping [24]).

$$\mathcal{SC}_{1/2}[\![prog]\!] \xrightarrow{\lambda-dropping} \mathcal{SC}[\![prog]\!]$$

В [23, 99] показана корректность λ-отбрасывания. То есть,

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!] \quad \mathcal{SC}[\![prog]\!] \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

Отсюда следует корректность отношения трансформации $HOSC$:

$$prog \cong \mathcal{SC}[\![prog]\!]$$

□

4.5 Типизация и корректность

Если рассматривать только неявно типизированные программы, то отношение трансформации HOSC не является корректным.

Рис. 4.10 `russel (MkU russel)`: остаточная программа

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f)
```

Рис. 4.11 `russel (MkU russel)`: остаточная программа с явной типизацией

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f) :: Bool
```

Выведенные типы для остаточной программы могут быть более общими, нежели для исходной. Таким образом, суперкомпиляция может *расширять* область определения (в смысле типизации) при неявной типизации по Хиндли-Милнеру.

Рассмотрим программу на Рис. 4.9. Выведенный тип ее целевого выражения:

```
(russel (MkU russel)) :: Bool
```

Самый простой вариант суперкомпиляции этой программы приведен на Рис. 4.10. Выведенный тип для остаточного выражения:

```
(letrec f=f in f) :: a
```

Как видно, выведенный тип остаточного выражения является более общим, чем тип исходного выражения, а значит, существуют контексты, в которых второе выражение может находится без ошибок типизации, а первое – нет.

То, что выводимая типизация по Хиндли-Милнеру не сохраняется при β-редукции, является известным фактом ([79]).

Эквивалентность двух выражений подразумевает, что эти два выражения в программах можно безопасно взаимозаменять.

Чтобы достичь безопасной взаимозаменяемости исходной и остаточной программы для типового варианта HLL, необходимо, чтобы типизация остаточной программы совпадала с типизацией исходной программы.

Чтобы исходное выражение e и остаточное выражение e' оставались эквивалентными с учетом ошибок типизации, необходимо, чтобы в любом контексте C либо $C[e]$ и $C[e']$ корректно типизируемы, либо оба выражения $C[e]$ и $C[e']$ содержат ошибку типизации.

Можно достичь этого достаточно простым способом. Приписываем явно выведенные типы остаточному выражению (совпадает с типом исходного выражения) и каждой свободной переменной в остаточном выражении.

Остается тонкий случай, когда в остаточной программе нет некоторых переменных из исходной программы, – для полной корректности достаточно добавить искусственные аппликации λ-абстракций только для того, чтобы передать в остаточную программу информацию об устраненных переменных.

Возвращаясь к примеру суперкомпиляции программы на Рис. 4.9. На Рис. 4.11 показано явное приписывание типов в остаточной программе.

4.6 Выводы

Новизна материала, изложенного в данной главе, заключается в следующем:

- Суперкомпилятор *HOSC* описан в общем виде - в виде отношения трансформации. Показана корректность трансформации. В работах [78, 52, 75] описаны конкретные детерминированные реализации суперкомпиляции.

- *HOSC*, в отличие от суперкомпиляторов [78, 52] осуществляет свертку любых узлов – показана корректность такой свертки. Благодаря сворачиванию любых узлов увеличивается способность суперкомпилятора к нормализации выражений, что полезно для применения суперкомпиляции как средства анализа программ.

- В [53] указано, что преобразование статического аргумента (генерация *letrec*-выражений со свободными переменными, [98]) может быть иногда полезно для оптимизации программ, но не показана корректность такого преобразования. Мы показали, что генерация

letrec-выражений со свободными переменными корректна для отношения трансформации $HOSC$.

- В работах [78, 49] предполагается типизация по Хиндли-Милнеру (в стиле Карри), но не рассматривается вопрос о корректности суперкомпиляции в смысле типизации. Мы показали, что суперкомпиляция может расширять область определения выражения, выводимые типы в остаточном выражении могут быть более общими, нежели в исходном. Через явное приписывание типов в остаточной программе можно сузить область определения до исходной.

- Мы показали корректность отношения трансформации $HOSC$ с помощью теории операционных улучшений, что позволило показать, что трансформация $HOSC_0$ дает на выходе более эффективную программу. В [75] доказательство корректности проведено с помощью бисимуляции, что не позволяет сказать как относятся исходная и остаточная программа в смысле производительности. Мы работали с полной (sound) типизацией по Хиндли-Милнеру. В [75] рассматривается потенциально неполный (unsound) вариант системы F.

Глава 5

Схема суперкомпилятора HOSC

В главе 3 суперкомпилятор HOSC рассматривался в самом общем виде – в виде *отношения трансформации*. В главе 4 была доказана корректность отношения трансформации HOSC.

Однако, суперкомпилятор, использующийся для анализа программ, должен быть полностью и формально описан. В данной главе полностью и формально рассматривается несколько алгоритмов суперкомпиляции для языка HLL, удовлетворяющих отношению трансформации HOSC. Алгоритм суперкомпилятора HOSC обосновывается на ряде примеров на основе сравнения с другими алгоритмами.

5.1 Язык HLL: вложение и обобщение

Определение 69 (Несопоставимые HLL-выражения)**.** Выражения e_1 и e_2 называются несопоставимыми, $e_1 \leftrightarrow e_2$, если $e_1 \sqcap e_2 = (v, \theta_1, \theta_2)$, то есть обобщенное выражение является переменной.

Алгоритм нахождения тесного обобщения для выражений первого порядка хорошо описан в литературе. Один из вариантов такого алгоритма (в рекурсивной форме) представлен на Рис. 1.6. Однако, как было отмечено в главе 1 в работах, описывающих суперкомпиляторы для языков высшего порядка ([78, 50]), вопрос нахождения тесного обобщения выражений со связанными переменными обходится стороной.

Если рассматривать λ-абстракции и `case`-выражения как специальные конструкторы и “наивным” образом расширить алгоритм на Рис. 1.6, то

Рис. 5.1 HLL: алгоритм обобщения

Тесное обобщение

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

Правило общего функтора

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $c\ \overline{e'_i} \mathbin{\widetilde{\sqcap}} c\ \overline{e''_i} = (c\ \overline{e_i},\ \bigcup \theta'_i,\ \bigcup \theta''_i)$, где
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e'_1\ e'_2 \mathbin{\widetilde{\sqcap}} e''_1\ e''_2 = (e_1\ e_2,\ \theta'_1 \mathbin{\widetilde{\sqcap}} \theta'_2,\ \theta''_1 \mathbin{\widetilde{\sqcap}} \theta''_2)$, где
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $\lambda v' \to e' \sqcap \lambda v'' \to e'' = (e_g, \theta', \theta'')$, если θ' и θ'' допустимы для e_g, где
  - $e_g = \lambda v \to e$
  - $(e, \theta', \theta'') = e'\{v' := v\} \mathbin{\widetilde{\sqcap}} e''\{v'' := v\}$
- $case\ e'_0\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \mathbin{\widetilde{\sqcap}} case\ e''_0\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} = (e_g, \bigcup \theta'_i, \bigcup \theta''_i)$, если все θ'_i и θ''_i допустимы e_g, где
  - $e_g = case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$
  - $(e_0, \theta'_0, \theta''_0) = e'_0 \mathbin{\widetilde{\sqcap}} e''_0$
  - $(e_i, \theta'_i, \theta''_i) = e'_i\{\overline{v'_{ik} := v_{ik}}\} \mathbin{\widetilde{\sqcap}} e''_i\{\overline{v''_{ik} := v_{ik}}\}$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

Правило общего подвыражения

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ где
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$
    если $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$
    в противном случае

в обобщении, найденном таким образом, могут появляться некорректные подстановки (см. Определение 27). В результате обобщения должна по-

Рис. 5.2 HLL: простое гомеоморфное вложение

Вложение	
$e' \trianglelefteq e''$	если $e' \trianglelefteq_v e''$, $e' \trianglelefteq_c e''$ или $e' \trianglelefteq_d e''$
Вложение переменных	
$f_g \trianglelefteq_v f_g$	
$v \trianglelefteq_v v'$	
Сцепление (Coupling)	
$c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i}$	если $\forall i : e'_i \trianglelefteq e''_i$
$\lambda v' \to e' \trianglelefteq_c \lambda v'' \to e''$	если $e' \trianglelefteq e''$
$e'_1\ e'_2 \trianglelefteq_c e''_1\ e''_2$	если $\forall i : e'_i \trianglelefteq e''_i$
$case\ e'\ of\ \{c_i\ \overline{v'_{ik}} \to e'_i;\} \trianglelefteq_c case\ e''\ of\ \{c_i\ \overline{v''_{ik}} \to e''_i;\}$	если $e' \trianglelefteq e''$ и $\forall i : e'_i \trianglelefteq e''_i$
Погружение (Diving)	
$e \trianglelefteq_d c\ \overline{e_i}$	если $\exists i : e \trianglelefteq e_i$
$e \trianglelefteq_d \lambda v_0 \to e_0$	если $e \trianglelefteq e_0$
$e \trianglelefteq_d e_1\ e_2$	если $\exists i : e \trianglelefteq e_i$
$e \trianglelefteq_d case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$	если $\exists i : e \trianglelefteq e_i$

лучиться такая тройка $(e_g, \theta_1, \theta_2)$, где θ_1 и θ_2 – допустимые по отношению к e_g подстановки. В свете выбранных нами соглашений о переменных и подстановках, алгоритм нахождения тесного обобщения должен гарантировать корректность подстановок, полученных в результате обобщения.

Обобщение λ-абстракций и `case`-выражений необходимо рассматривать особым образом – если в результате решения подзадач обобщения соответствующих подвыражений возникает некорректная подстановка, это означает, что обобщением рассматриваемых выражений является переменная.

Определение 70 (Рекурсивный алгоритм обобщения HLL-выражений)**.** $e' \sqcap e'' = s(e' \widetilde{\sqcap} e'')$, где операции $\widetilde{\sqcap}$ и s определены на Рис. 5.1.

Алгоритм 70 применим для нахождения тесного обобщения двух *любых выражений* языка HLL и учитывает требования соглашений 25 и 28. Правила применяются в порядке их перечисления. Переменные v и v_{ik} в 3-м, 4-м и 5-м правилах общего подвыражения – новые, ранее не встречавшиеся, переменные. Наиболее интересные детали алгоритма, учитывающие новые обстоятельства, подчеркнуты. Отметим, что алгоритм 70

сформулирован в рекурсивной форме. В литературе по суперкомпиляции алгоритм обобщения традиционно описывается в итеративной форме. Сформулировать алгоритм обобщения выражений со связанными переменными в итеративной форме представляется крайне затруднительно.

В литературе по суперкомпиляции языков высшего порядка используется классическое гомеоморфное вложение, "адаптированное" для выражений со связанными переменными, – λ-абстракции и `case`-выражения рассматриваются как специальные конструкторы и связанные переменные и свободные переменные не различаются.

Определение 71 (Простое гомеоморфное вложение $\trianglelefteq$)**.** Простое вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 5.2.

Однако, в контексте суперкомпиляции, простое гомеоморфное вложение обладает существенным недостатком: несопоставимые выражения могут вкладываться через сцепление: $e_1 \trianglelefteq_c e_2$. В результате нам придется делать декомпозицию одного из выражений без учета истории построения частичного дерева, что противоречит принципу обобщения в суперкопиляции (см. [118]).

Однако, если различать свободные и связанные переменные и потребовать, чтобы связанные переменные могли вкладываться только соответствующие связанные переменные, то можно сформулировать уточненное гомеоморфное вложение, не допускающее вложение через сцепление несопоставимых выражений.

При проверке двух HLL-выражений на уточненное вложение используется таблица соответствия связанных переменных ρ:

$$\rho = \{(v'_1,\ v''_1),\ \ldots,\ (v'_n,\ v''_n)\}$$

Определение 72 (Уточненное гомеоморфное вложение $\trianglelefteq^*$)**.** Уточненное вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 5.3.

Напомним, что для SLL-выражений (раздел 1.2.3) верно следующее: если $e_1 \trianglelefteq_c e_2$, то обобщение $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$ – нетривиально (e_g - не

Рис. 5.3 HLL: уточненное гомеоморфное вложение

Вложение

$e' \trianglelefteq^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq^* e'' \mid_{\{\}}$

$e' \trianglelefteq_c^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_c^* e'' \mid_{\{\}}$

$e' \trianglelefteq_d^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_d^* e'' \mid_{\{\}}$

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

$e_1'\ e_2' \trianglelefteq_c^* e_1''\ \underline{e_2'' \mid_\rho}$ если $\forall i : e_i' \underline{\trianglelefteq^* e_i'' \mid_\rho}$

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

переменная). Это свойство сохраняется для уточненного гомеоморфного вложения $\trianglelefteq_c^*$ HLL-выражений.

5.2 Параметризованный HLL суперкомпилятор

Уточненное вложение $\trianglelefteq^*$ (Рис. 5.3) является достаточно сильным усложнением адаптированного вложения $\trianglelefteq$ (Рис. 5.2). Чтобы обосновать практичность уточненного гомеоморфного вложения применительно к использованию суперкомпилятора для трансформационного анализа, рассмотрим различные варианты суперкомпилятора (как с использованием адаптированного вложения, так и с использованием уточненного вложения) и сравним их на модельной задаче – доказательстве эквивалентности

Рис. 5.4 HLL: расщепление конфигурации

$split(t, \beta, e_1 e_2)$	$=$	$replace(t, \beta, e_s)$, где $e_s = let\ v_1 = e_1;\ v_2 = e_2;\ in\ v_1 v_2$
$split(t, \beta, case\ v\ of\ \{\overline{p_i \to e_i;}\})$	$=$	$addChildren(t, \beta, [v, \overline{e_i}])$
$split(t, \beta, case\ e\ of\ \{\overline{p_i \to e_i;}\})$	$=$	$replace(t, \beta, e_s)$, где $e_s = let\ v = e\ in\ case\ v\ of\ \{\overline{p_i \to e_i;}\}$
$split(t, \beta, e)$	$=$	$drive(t, \beta)$

выражений.

Поскольку при использовании адаптированного вложения $\trianglelefteq$ несопоставимые выражения могут вкладываться через сцепление, нам придется делать декомпозицию текущей конфигурации. В случае языка SLL это делается достаточно просто – текущей конфигурацией является вызов функции, и мы отдельно рассматриваем аргументы функции. Такая декомпозиця обладает важным свойством, что размеры всех компонент получившейся декомпозиции меньше размера обобщаемого выражения, что критически важно для того, чтобы гарантировать завершаемость.

В случае HLL-выражений операция расщепления конфигурации, несопоставимой с вложенной конфигурацией, усложняется из-за наличия `case`-выражений, – необходимо учитывать связанные переменные. Одновременно, желательно, чтобы размер выражений в дочерних узлах был строго меньше размера расщепляемого выражения, – чтобы гарантировать завершаемость суперкомпилятора.

Определение 73 (Операция декомпозиции конфигурации *split*)**.** Операция *split* осуществляется в соответствии с правилами на Рис. 5.4.

Особенность определенной операции *split* состоит в работе с `case`-выражениями. Если селектор `case`-выражения является переменной, то делаем шаг, похожий на прогонку, – рассматриваем ветви, но *не распространяем позитивную информацию* – таким образом, размер выражений в новых дочерних узлах будет строго меньше размера расщепляемого выражения (это важно для завершаемости суперкомпилятора). Если селектор не является переменной, то мы обобщаем селектор.

Рассмотрим алгоритм построения частичного дерева процессов, представленный на Рис. 5.5. Концептуально, этот алгоритм является обобще-

Рис. 5.5 HLL суперкомпилятор

```
t = [c0]
while unprocessedLeaf(t) ≠ • do
    β = unprocessedLeaf(t)
    relAncs = computeRelAncs(β)
    α = find(relAncs, β, whistle)
    if α ≠ • and α.expr ≃ β.expr then
        t = fold(t, α, β)
    else if α ≠ • and α.expr ◁ β.expr then
        t = abstract(t, β, α)
    else if α ≠ • then
        if α.expr ↔ β.expr then
            t = split(t, β, β.expr)
        else
            t = abstract(t, α, β)
        end
    else
        t = drive(t, β)
    end
end
```

Рис. 5.6 Типы выражений

$expClass(let\ \overline{v_i = e_i}\ in\ e)$	=	0
$expClass(v\ \overline{e_i})$	=	0
$expClass(c\ \overline{e_i})$	=	0
$expClass(\lambda v \to\ e)$	=	0
$expClass(con\langle(\lambda v \to e_0)\ e_1\rangle)$	=	1
$expClass(con\langle f_g\rangle)$	=	2
$expClass(con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle)$	=	3
$expClass(con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle)$	=	4

нием алгоритма на Рис. 1.10 и отличается от него двумя деталями:

1. При поиске кандидата на зацикливание рассматриваются не все предки, а только $computeRelAncs(\beta)$.

2. Рассматривается любой предикат *whistle* в качестве свистка.

Прежде чем перейти к рассмотрению конкретных функций *whistle* и *computeRelAncs*, нам необходимо ввести несколько определений.

Определение 74 (Классы HLL-выражений)**.** Все выражения языка HLL разбиваются на 5 классов в соответствии с правилами на Рис. 5.6.

Определение 75 (Тривиальный узел)**.** Узел β называется тривиальным, если находящееся в нем выражение $\beta.expr$ является наблюдаемым или `let`-выражением. ($expClass(\beta.expr) = 0$)

Определение 76 (β-транзитный узел)**.** Узел β называется β-транзитным, если $\beta.expr = con\langle(\lambda v_0 \to e_0)\ e_1\rangle$. ($expClass(\beta.expr) = 1$)

Определение 77 (Глобальный узел)**.** Узел β называется глобальным, если $\beta.expr = con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$. ($expClass(\beta.expr) == 4$)

Определение 78 (Локальный узел)**.** Все узлы за исключением глобальных считаются локальными.

Определение 79 (Кандидат на зацикливание)**.** Кандидатом на зацикливание являются все узлы, за исключением тривиальных и β-транзитных узлов. ($expClass(\beta.expr) > 1$)

Замечание 80 (Обоснование выбора кандидатов)**.** Если включать β-транзитные узлы в число кандидатов на зацикливание, то все рассматриваемые далее суперкомпиляторы показывают неудовлетворительные результаты. Исключение β-транзитных узлов из списка кандидатов является безопасным для завершаемости суперкомпилятора – типизация гарантирует, что в частичном дереве процессов не будет бесконечной ветки, на которой нет ни одного кандидата.

Определение 81 (Релевантные с учетом контроля предки-кандидаты)**.** Пусть β – узел-кандидат. Релевантные с учетом контроля предки-кандидаты узла β определяются следующим образом:

- все глобальные предки $\overline{\alpha_i}$ узла β, являющиеся кандидатами, если β – глобальный узел
- все локальные предки $\overline{\alpha_i}$ узла β, являющиеся кандидатами, такие, что на пути от α_i до β не встречается глобальный узел, если β – локальный узел

5.2.1 Конкретные HLL суперкомпиляторы

Рассмотрим конкретизацию алгоритма – алгоритм SC_{ijk}. Алгоритм SC_{ijk} зависит от трех параметров:

1. i - Какое гомеоморфное вложение использовать: $\trianglelefteq_c^*$ $(+)$ или $\trianglelefteq_c$ $(-)$?
2. j - Разделять ли узлы на глобальные и локальные при поиске релевантных кандидатов [105] $(+)$ или не разделять $(-)$?
3. k - Разделять ли выражения на классы в соответствие в типом редекса $(+)$ или нет $(-)$?

Более формально:

- $i = -$:
  $w'(\alpha, \beta) = \alpha.expr \trianglelefteq_c \beta.expr$.
- $i = +$:
  $w'(\alpha, \beta) = \alpha.expr \trianglelefteq_c^* \beta.expr$.
- $j = -$:
  $computeRelAncs$ – предки-кандидаты.
- $j = +$:
  $computeRelAncs$ – предки-кандидаты с учетом контроля.
- $k = -$:
  $whistle(\alpha, \beta) = w'(\alpha, \beta)$.
- $k = +$:
  $whistle(\alpha, \beta) = w'(\alpha, \beta)$ и $expClass(\alpha.expr) = expClass(\beta.expr)$.

Таким образом, алгоритм SC_{ijk} описывает восемь вариантов суперкомпилятора.

Теорема 82 (Корректность)**.** *Все представленные суперкомпиляторы* $\mathbf{SC_{ijk}}$ *корректны в смысле операционной эквивалентности исходной и остаточной программы.*

Доказательство. Все алгоритмы удовлетворяют отношению суперкомпиляции HOSC. □

Рис. 5.7 Тесты

`length (concat xs)`	$\cong$	`sum (map length xs)`	(1)
`map f (append xs ys)`	$\cong$	`append (map f xs) (map f ys)`	(2)
`filter p (map f xs)`	$\cong$	`map f (filter (compose p f) xs)`	(3)
`map f (concat xs)`	$\cong$	`concat (map (map f) xs)`	(4)
`iterate f (f x)`	$\cong$	`map f (iterate f x)`	(5)
`map (compose f g)`	$\cong$	`compose (map f) (map g)`	(6)
`map f xs`	$\cong$	`join xs (compose return f)`	(7)

Рис. 5.8 Сравнение суперкомпиляторов на тестах

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

5.3 Сравнение суперкомпиляторов

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.

Для сравнения восьми вариантов суперкомпилятора используется модельная задача: доказательство эквивалентности выражений [63]. Рассматриваемые тестовые примеры используют функции высших порядков и оперируют потенциально бесконечными данными. Функции над парами, списками и числами, используемые в тестах, показаны на Рис. 5.9. Сами тесты представлены на Рис. 5.7. Тест считается пройденным суперкомпилятором, если он сумел привести оба выражения к одной и той же синтаксической форме, и не пройденным в противном случае. Результаты эксперимента, представленные на Рис. 5.8, показывают, что наилучшим сочетанием параметров является использование уточненного гомеоморфного вложения, разделения узлов на локальные и глобальные и разделе-

Рис. 5.9 Определения функций для тестов

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;
data Boolean = True | False;
data Pair a b = P a b;

compose = λf g x →  f (g x);
outl = λp → case p of { P a b → a;};
outr = λp → case p of { P a b → b;};
uncurry = λf p → case p of {P x y → f x y;};
curry = λf b c → f (P b c);
cond = λp f g a → case (p a) of {True → f a; False → g a;};
foldn = λc h x → case x of {
  Z → c;
  S x1 → h (foldn c h x1);
};
plus = foldn (λx → x) (λf y → S (f y));
foldr = λc h xs → case xs of {
  Nil → c;
  Cons y ys → h y (foldr c h ys);
};
concat = foldr Nil append;
sum = foldr Z plus;
filter = λp → foldr Nil
  (curry (cond (compose p outl) (uncurry (λx xs → Cons x xs)) outr));
iterate = λf x → Cons x (iterate f (f x));
length = foldr Z (λx y → S y);
join = λm k → foldr Nil (compose append k) m;
return = λx → Cons x Nil;
map = λf → foldr Nil (λx xs → Cons (f x) xs);
append = λxs ys → case xs of {
  Nil → ys;
  Cons x1 xs1 → Cons x1 (append xs1 ys);
};
```

ние выражений на классы в соответствии с типом редекса.

Рис. 5.10 HLL: уточненное гомеоморфное вложение, учитывающее арность аппликации

Вложение

$e' \trianglelefteq^{**} e'' \stackrel{\text{def}}{=} e' \trianglelefteq^{**} e'' \mid_{\{\}}$

$e' \trianglelefteq^{**}_{c} e'' \stackrel{\text{def}}{=} e' \trianglelefteq^{**}_{c} e'' \mid_{\{\}}$

$e' \trianglelefteq^{**}_{d} e'' \stackrel{\text{def}}{=} e' \trianglelefteq^{**}_{d} e'' \mid_{\{\}}$

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 $\exists i : e \trianglelefteq^{**} e_i \mid_{\rho}$

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

5.4 Усиление уточненного вложения с учетом типизации

Вложение $\trianglelefteq^{*}$ рассматривает аппликацию как бинарный конструктор. Однако, вложение $\trianglelefteq^{*}$ можно усилить – рассматривать аппликацию как n-арный конструктор, первый аргумент которого не является аппликацией.

Определение 83 (Уточненное гомеоморфное вложение $\trianglelefteq^{**}$, учитывающее арность аппликации)**.** Вложение HLL-выражений $\trianglelefteq^{**}$ определяется индуктивно в соответствии с правилами на Рис. 5.10.

Как будет показано в следующей главе, вложение $\trianglelefteq^{**}$ является вполне-квазиупорядочением на множестве выражений, возникающих при постро-

ении частичного дерева процессов для программы p, типизированной по Хиндли-Милнеру. Что делает его пригодным для использования в качестве свистка при суперкомпиляции программ на языке Haskell.

Однако, есть широко распространенные расширения языка Haskell, предлагающие более общую систему типизации (например, экзистенциальные типы), для таких расширений вложение $\trianglelefteq^{**}$ уже не является вполне-квазиупорядочением на множестве выражений, возникающих при построении частичного дерева процессов. Однако, вложение $\trianglelefteq^{*}$ остается вполне-квазиупорядочением, что делает его более универсальным.

Будем рассматривать в качестве первого параметра суперкомпилятора $\mathbf{SC_{ijk}}$ (см. Раздел 5.2.1) $i = *$, означающий, что в качестве свистка используется отношение $\trianglelefteq_c^{**}$.

Таким образом, у нас появляются 4 новых варианта суперкомпилятора: $\mathbf{SC}_{*--}$, $\mathbf{SC}_{*-+}$, $\mathbf{SC}_{*+-}$, $\mathbf{SC}_{*++}$. Все эти варианты также являются корректными, так как удовлетворяют отношению трансформации HOSC.

В некоторых случаях использование отношения $\trianglelefteq^{**}$ позволяет избежать преждевременного обобщения.

Далее под суперкомпилятором HOSC понимается суперкомпилятор SC_{*++}.

5.5 Выводы

В данной главе полностью и формально описано внутреннее устройство суперкомпилятора HOSC.

Впервые сформулирован алгоритм нахождения тесного обобщения для любых двух выражений со связанными переменными.

Рассмотрено три варианта гомеоморфного вложения для HLL выражений.

Предложено 12 вариантов суперкомпилятора.

Произведено сравнение 8 вариантов суперкомпилятора, показывающее плодотворность использования уточненного гомемоморфного вложения в суперкомпиляторе, предназначенном для трансформационного анализа программ.

Глава 6

Завершаемость суперкомпилятора HOSC

В предыдущей главе были определены 12 вариантов суперкомпилятора HOSC. Мы покажем завершаемость всех 12 вариантов суперкомпилятора.

Вначале мы докажем теорему о завершаемости суперкомпилятора $\mathbf{SC}_{*--}$. Из нее сразу следуют теоремы о завершаемости суперкомпиляторов $\mathbf{SC}_{*-+}$ и $\mathbf{SC}_{*++}$, откуда следует завершаемость всех остальных суперкомпиляторов.

Мы используем инструментарий для доказательства завершаемости *абстрактных преобразователей программ*, разработанный Морте Сёренсеном [108], поскольку суперкомпилятор HOSC можно рассматривать как абстрактный преобразователь программ (частичных деревьев процессов).

Построение для данной программы частичного дерева процессов суперкомпилятором $\mathbf{SC}_{*--}$ можно неформально описать так [59]: построение начинается с того, что в корень дерева помещается исходное (целевое) выражение. Затем к листьям дерева добавляются дочерние узлы, пока все листья не станут *обработанными*.

Пусть β - необработанный лист строящегося дерева:

1. Если β является тривиальными узлом, "метавычисляем" $\beta.expr$ – делаем один шаг прогонки.

2. Если у β есть предок α такой, что $\alpha.expr \simeq \beta.expr$, то делаем узел α базовым узлом β – проводим специальную дугу из β в α: $\beta \rightrightarrows \alpha$, тем самым лист β становится обработанным.

3. Если у β есть предок α такой, что $\alpha.expr \lessdot \beta.expr$, то обобщаем $\beta.expr$.
4. Если у β есть предок α такой, что $\alpha.expr \trianglelefteq_c^{**} \beta.expr$, то обобщаем $\alpha.expr$.
5. В противном случае "метавычисляем" $\beta.expr$ – делаем шаг прогонки.

Результатом шага прогонки является подвешивание к узлу β дочерних узлов и помещение в них результата метавычисления $\beta.expr$.

Теорема 84 (Завершаемость суперкомпилятора HOSC)**.** *Суперкомпилятор* $\mathbf{SC}_{*--}$ *завершается на любой входной программе.*

Идея доказательства состоит в следующем:

- Шаг 1 может выполниться подряд только конечное число раз, так как при прогонке тривиального узла, выражения в дочерних узлах будут *строго меньше* по размеру.
- Шаг 2 не влияет на завершаемость, так как не изменяет узлы дерева и для конечного дерева может выполниться конечное число раз.
- Таким образом, любая последовательность шагов 1 и 2 конечна. Предположим, что HOSC выполняет шаг 3 или шаг 4, обобщая выражение в узле. Мы покажем, что обобщения не могут продолжаться бесконечно, так как обобщение уменьшает "размер выражения".
- Остается показать, что выполнение шага 5 не может повторяться бесконечно. Это следует из того, что $\trianglelefteq^{**}$ (взятое за основу для $\trianglelefteq_c^{**}$) – вполне-квазиупорядочение (см. ниже). Поэтому любая бесконечная последовательность шагов 5, включает шаг 4.

Наиболее сложной и трудоёмкой задачей будет показать, что отношение $\trianglelefteq^{**}$ является вполне-квазиупорядочением на множестве HLL-выражений, возникающих при построении частичного дерева процессов.

6.1 Абстрактные преобразователи программ

В данном разделе кратко излагается теория абстрактных преобразователей программ, изложенная в [108].

Определение 85 (Квазиупорядочение)**.** Пусть на множестве S задано отношение $\leq$. $(S, \leq)$ называется квазиупорядочением (quasi-order), если $\leq$ транзитивно и рефлексивно. Мы пишем $s < s'$, если $s \leq s'$ и $s' \not\leq s$.

Определение 86 (Вполне фундированное квазиупорядочение)**.** Пусть $(S, \leq)$ – квазиупорядочение. $(S, \leq)$ является вполне фундированным квазиупорядочением (well-founded quasi-order), если не существует бесконечной последовательности $s_0, s_1, \ldots \in S$ такой, что $s_0 > s_1$.

Определение 87 (Вполне-квазиупорядочение)**.** Пусть $(S, \leq)$ – квазиупорядочение. $(S, \leq)$ является вполне-квазиупорядочением (well-quasi-order), если для любой бесконечной последовательности $s_0, s_1, \ldots \in S$ существуют $i < j$ такие, что $s_i \leq s_j$.

Определение 88 (Дерево над множеством выражений)**.** Пусть $\mathcal{L}$ – некоторое множество выражений. Частичное дерево процессов t из [59] является деревом над множеством $\mathcal{L}$ если $\forall \gamma \in t : \gamma.expr \in \mathcal{L}$.

Множество всех деревьев над $\mathcal{L}$ будем обозначать как $T(\mathcal{L})$

Определение 89 (Абстрактный преобразователь программ)**.** Абстрактный преобразователь программ на $\mathcal{L}$ – отображение $M\colon T(\mathcal{L}) \to T(\mathcal{L})$.

Определение 90 (Завершаемость преобразователя программ)**.** (1) Абстрактный преобразователь программ M на $\mathcal{L}$ завершается на $t \in T(\mathcal{L})$, если $M^i(t) = M^{i+1}(t)$ для некоторого i. (2) Преобразователь M на $\mathcal{L}$ завершается, если он завершается на всех деревьях $t \in T(\mathcal{L})$, состоящих из одного узла.

Для удобства будем обозначать t_0 – начальное преобразуемое дерево, t_i – дерево после i шагов работы преобразователя.

Утверждение 91 (Преобразователь Коши)**.** *Пусть $(\mathcal{L}, \leq)$ – вполне фундированное квазиупорядочение. Абстрактный преобразователь программ M на $\mathcal{L}$ является преобразователем Коши, если в ходе его работы:*

$$t_{i+1} = t_i\{\gamma := t'\}$$

для некоторого узла γ и выполняется одно из следующих условий:

- $\gamma \in leaves(t_i)$ *и* $\gamma.expr = t'.root.expr$,
- $\gamma.expr > t'.root.expr$.

Утверждение 92 (Конечный непрерывный предикат)**.** *Пусть* $\{\mathcal{L}_1, \mathcal{L}_2\}$ - *разбиение* $\mathcal{L}$, $(\mathcal{L}_1, \leq_1)$ – *вполне-квазиупорядочение,* $(\mathcal{L}_2, \leq_2)$ – *вполне фундированное квазиупорядочение. Тогда* p*:* $T(\mathcal{L}) \to \mathbb{B}$, *определенное как*

$$p(t) = \begin{cases} 0 & \text{если } \exists \alpha, \beta : \alpha \text{ - предок } \beta, \\ & \alpha.expr, \beta.expr \in \mathcal{L}_1 \text{ и } \alpha.expr \leq_1 \beta.expr \\ 0 & \text{если } \exists \alpha, \beta : \alpha \to \beta,\ \alpha.expr, \beta.expr \in \mathcal{L}_2 \text{ и } \alpha.expr \not> _2 \beta.expr \\ 1 & \text{в противном случае} \end{cases}$$

является конечным непрерывным предикатом.

Определение 93 (Внутренность дерева)**.** Корневой узел дерева t и все его узлы за исключением листьев – внутренность дерева t.

Внутренность t обозначается t^0

Утверждение 94. *Пусть* $p : T(\mathcal{L}) \to \mathbb{B}$ – *конечный непрерывный предикат, тогда* q, *определяемое как* $q(t) = p(t^0)$, – *также непрерывный конечный предикат.*

Теорема 95 (Сёренсен)**.** *Пусть абстрактный преобразователь программ* $M : T(\mathcal{L}) \to T(\mathcal{L})$ *сохраняет предикат* $p : T(\mathcal{L}) \to \mathbb{B}$. *Если*

1. M – *преобразователь Коши и*
2. p – *непрерывный конечный предикат,*

то M *завершается.*

Нам потребуется различать HLL-выражения без let-выражений (присутствующие как в текстах входных программ на языке HLL, так и в узлах деревьев процессов) и let-выражения (которые могут появляться только в узлах дерева процессов).

- $\mathcal{E}$ обозначает множество HLL выражений без let-выражений,
- $\mathcal{L}$ обозначает множество $\mathcal{E}$, дополненное множеством let-выражений.

Любое HLL-выражение $e \in \mathcal{E}$ считается эквивалентным $let\ in\ e$.

Суперкомпилятор $\mathbf{SC}_{*\_\_}$ является абстрактным преобразователем программ на $\mathcal{L}$, где $\mathcal{L}$ – множество всех HLL выражений, расширенное let-выражениями.

6.2 Гомеоморфное вложение $\trianglelefteq^{**}$

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* [116, 118].

Не только в суперкомпиляции, но и в других методах преобразования программ в качестве свистка хорошо себя зарекомендовало отношение гомеоморфного вложения [67, 68]. Свисток, основанный на гомеоморфном вложений, 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 предотвратить появление бесконечных ветвей в частичном дереве процессов. Таким образом, *существенным* свойством свистка является то, что он срабатывает на последовательности выражений, *порождаемых прогонкой*. Говоря формально, свисток основан на *вполне-квазиупорядочении*.

В разделе 1.2.3 показано, что гомеоморфное вложение $\trianglelefteq$ для SLL-выражений (см. Теорему 15) является вполне-квазиупорядочением на $E_{C \cup F \cup V}$ при условии, что множества F и C конечны.

6.2.1 Связанные переменные

В отличие от языка первого порядка, рассматриваемого Сёренсеном, суперкомпилятор HOSC работает с языком HLL (с функциями высшего порядка). В HLL-выражениях присутствуют связанные переменные в в λ-абстракциях и case-выражениях. Концептуально связанные переменные в case-выражениях не отличаются от связанных переменых в λ-

абстракциях. Поэтому в дальнейшем проблемы, относящиеся к связанным переменным, рассматриваются только для λ-абстракций.

λ-абстракцию можно рассматривать как "особый" конструктор при проверке на гомеоморфное вложение. Однако, при упрощенном подходе во время суперкомпиляции могут возникнуть трудности, так как вложение через сцепление не подразумевает наличие нетривиального обобщения. Действительно, если не различать связанные переменные, то следующие выражения "наивно" сцеплены, но нет нетривиального обобщения:

```
λx y → Pair x y
λx y → Pair y x
```

Однако, если различать связанные переменные, данные выражения не вкладываются. Несмотря на синтаксическое сходство двух выражений, соответствующие безымянные функции различаются *по смыслу*. Поэтому в определении расширенного вложения $\trianglelefteq^*$ используется таблица ρ для записи соответствия связанных переменных. Пусть $\trianglelefteq^1$ – вложение $\trianglelefteq$, учитывающее соответствие связанных переменных.

Однако, $\trianglelefteq^1$ еще достаточно грубо сравнивает выражения, поскольку существуют выражения, сцеленные через $\trianglelefteq^1$, для которых отстутствует нетривиальное обобщение. Например, следующие выражения:

```
λx → x
λx → (S x)
```

Поэтому в определениях вложений $\trianglelefteq^*$ и $\trianglelefteq^{**}$ вводится дополнительное требование: выражение, свободные переменные которого уже есть в таблице соответствия ρ, не может быть погружено в другое выражение – условие для погружения одного выражения в другое. Обозначим через $\trianglelefteq^2$ – вложение $\trianglelefteq^*$.

6.2.2 Высший порядок и арность

Теорема Крускала и Хигмана применима для отношения $\trianglelefteq$ SLL-выражений, так как арность функторов (функций и конструкторов) фиксирована, множество функторов в исходной программе конечно, и во время прогонки не могут появиться функторы, отсутствующие в исходной программе.

Однако, в случае языка высшего порядка, функция может быть частично применена, поэтому, потенциально, во время прогонки могут появляться вызовы функций произвольной "арности": `f`, `f x`, `f x y` и т.д. Вдобавок, аргументами функций могут быть значения-функции. Поэтому логично рассматривать имена функций как переменные, а не функторы, и ввести специальный конструктор для представления аппликации. Это можно сделать двумя способами:

1. Ввести только один бинарный конструктор `App`, первый аргумент которого – голова аппликации (которая может быть аппликацией), а второй – аргумент аппликации. Обозначим такую кодировку $\mathcal{A}_2$.
2. Не допускать аппликации быть в голове аппликации – ввести множество *различных* конструкторов `App2`, `App3` ... разной арности. Обозначим такую кодировку $\mathcal{A}_*$.

Выражение `f x y` будет представимо следующими способами:

1. `f x y = App(App(f, x), y)`
2. `f x y = App3(f, x, y)`

В обоих случаях, если выражения e_1 и e_2 вложены через сцепление $\trianglelefteq^2$, то существует их нетривиальное обобщение. Также ясно, что при кодировке $\mathcal{A}_2$ свисток будет свистеть чаще, чем при $\mathcal{A}_*$.

В принципе, в суперкомпиляторе может быть использован любая из кодировок. Однако в некоторых случаях первый подход (отношение $\trianglelefteq^*$) приводит к избыточному обобщению из-за того, что если во время прогонки встречается λ-абстракция, то HOSC начинает преобразовывать тело это абстракции.

Рассмотрим прогонку выражения `iterate (λn →S n)`. После нескольких шагов получится дерево, показанное на Рис. 6.1.

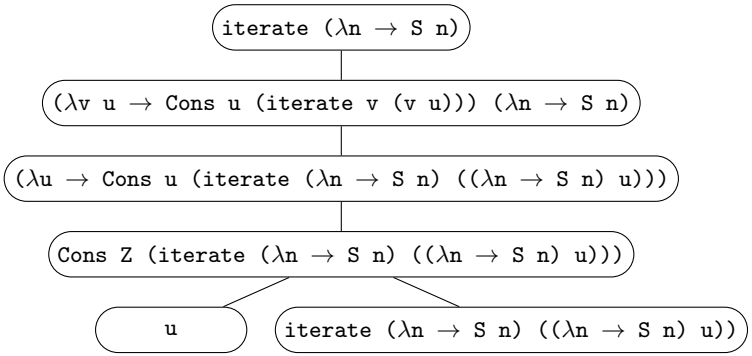
При использовании кодировки $\mathcal{A}_2$ верхнее выражение будет вложено через сцепление в правое нижнее выражение:

`App(iterate, λn→ S n)` $\trianglelefteq_c^*$ `App(App(iterate, λn→ S n), App(λn→ S n, u))`

Верхнее выражение будет обобщено до

```
let f = iterate in f (λn→ S n)
```

Рис. 6.1 Прогонка выражения `iterate (λn →S n)`



и функция `iterate` не будет проспециализирована относительно своего первого аргумента.

Использование кодировки $\mathcal{A}_*$ приведет к специализации вызова функции `iterate`, и в общем случае предоставляет больше возможностей для специализации. Именно поэтому определение вложения в [59] основано (неявно) на использовании кодировки $\mathcal{A}_*$. Будем обозначать через $\trianglelefteq^3$ вложение $\trianglelefteq^{**}$.

6.3 Вполне-квазиупорядочение $\trianglelefteq^{**}$

Цель данного раздела – показать, что отношение $\trianglelefteq^{**}$ (оно же – $\trianglelefteq^3$) является вполне-квазиупорядочением на любой последовательности выражений получаемых при прогонке во время суперкомпиляции. Напомним, что отношение $\trianglelefteq^3$ – это отношение $\trianglelefteq$, с наложенными на него ограничениями. В следующих разделах мы покажем, что несмотря на дополнительные ограничения, $\trianglelefteq^3$ остается вполне-квазиупорядочением последовательности выражений, возникающих в процессе *метавычислений*.

Определение 96 (Выражения, достижимые метавычислениями)**.** Пусть дана программа $prog$. Выражение e достижимо для метавычислений, если оно находится в дереве процессов, удовлетворяющее отношению трансформации HOSC. Множество выражений, достижимых метавычислениями программы $prog$, обозначается $\mathcal{M}(prog)$.

В следующих разделах мы покажем что отношение $\trianglelefteq^3$ является квазиупорядочением на $\mathcal{M}(prog)$. Это достигается заменой представления связанных переменных таким образом, что теорема Крускала будет применима.

Вначале избавимся от связанных переменных в λ-абстракциях и case-выражениях.

6.3.1 Замена case-выражений на конструкторы

Избавимся от связанных переменных в case-выражениях. Это можно сделать, представив case-выражение в виде композиции специального конструктора и λ-абстракцй. Имя конструктора определяется типом селектора, первый аргумент конструктора – селектор, остальные – ветви, представленные λ-абстракциями. Ветви упорядочены в соответствии с порядком перечисления конструкторов в декларации типа.

Например, следующее выражение

```
case x of {Z → Z; S y → S y;}
```

кодируется так:

```
CaseNat(x, Z, λy → S y)
```

Пусть дана программа $prog$. Поскольку $prog$ конечна, количество деклараций типов данных в $prog$ также конечно. Таким образом, количество возможных case-конструкторов тоже конечно. В процессе прогонки не могут возникать новые case-конструкторы, так как case-выражения в $\mathcal{M}(prog)$ либо устраняются в результате шага редукции, либо вводятся при разворачивании определения функции. Таким образом, множество case-конструкторов, присутствующих в $\mathcal{M}(prog)$, конечно.

6.3.2 Замена имен переменных на индексы де Брюина

К сожалению, в результате прогонки может возникнуть (потенциально) неограниченное количество связанных переменных в λ-абстракциях. Однако, от них можно избавиться с помощью индексов де Брюина [25], – натуральных чисел, где число k означает “переменная, связанная k-й охватывающей λ-абстракцией”.

При проверке на гомеоморфное вложение, можно считать индексы де Брюина специальными константами. Будет показано, что, в отличие от имен переменных, множество индексов де Брюина на $\mathcal{M}(prog)$ конечно.

После замены именованных переменных на индексы де Брюина, предыдущее выражение принимает вид:

```
CaseNat(x, Z, λ S 1)
```

Результат кодирования выражения e с помощью case-конструкторов и индексов де Брюина обозначается $\mathcal{E}_1[\![e]\!]$, результат кодирования программы $prog$ – $\mathcal{E}_1[\![prog]\!]$

Лемма 97 ($\mathcal{E}_1$, $\trianglelefteq$, $\trianglelefteq^1$)**.** $e_1 \trianglelefteq^1 e_2 \Leftrightarrow \mathcal{E}_1[\![e_1]\!] \trianglelefteq \mathcal{E}_1[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам. □

Лемма 98. *Индексы де Брюина ограничены на* $\mathcal{M}(prog)$.

Доказательство. Индексы де Брюина, встречающиеся в $\mathcal{E}_1[\![prog]\!]$ (а следовательно, и в закодированном целевом выражении), ограничены в силу конечности $\mathcal{E}_1[\![prog]\!]$. Индекс в $\mathcal{M}_{\mathcal{E}_1}(prog)$ ограничен наименьшей верхней гранью индексов в $\mathcal{E}_1[\![prog]\!]$, поскольку, если выражение e' возникло на шаге прогонки выражения e, то индексы в $\mathcal{E}_1[\![e']\!]$ не могут быть больше индексов в $\mathcal{E}_1[\![e']\!]$ и $\mathcal{E}_1[\![prog]\!]$. Действительно, рассмотрим правила прогонки (Рис. 3.3, Рис. 3.2).

- D_1, D_2, D_8 - шаг прогонки не затрагивает индексы де Брюина.
- D_3. Индекс де Брюина, соответствующий примененной λ-абстракции пропадает (становится свободной переменной), остальные индексы не изменяются
- D_4 - индексы в контексте сохраняются. Индексы в подставленном определении функции ограничены в силу конечности программы.
- D_5 - В силу *нормального порядка* β-редукции, осуществляемой при прогонке, после шага редукции индекс де Брюина, соответствующий v_0, исчезнет, все остальные индексы останутся неизменными.
- D_6, D_7, D_7^0 – аналогично D_5.

При любом варианте декомпозиции индексы де Брюина в let-выражении не превышают максимальный индекс де Брюина в исходном выражении.

□

Лемма 99. $(\mathcal{M}(prog), \trianglelefteq^1)$ – *вполне-квазиупорядочение.*

Доказательство. Число конструкторов в $\mathcal{M}_{\mathcal{E}_1}(prog)$ ограничено в силу леммы 98. Следовательно, $(\mathcal{M}_{\mathcal{E}_1}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. Из леммы 97 следует, что $(\mathcal{M}(prog), \trianglelefteq^1)$ – вполне-квазиупорядочение. □

Важно отметить, что при шагах прогонки по правилам D_5, D_6, D_7 максимальный индекс не увеличивается из-за *нормального порядка* β-редукции. Однако, при полной β-редукции (которую HOSC не делает), максимальный индекс может увеличиваться:

`(λx → (λy z → y) (λv → x)) w ⇒ (λx z v → x) w`

При кодировке с индексами де Брюина:

`( λ → ( λ λ → 2) ( λ → 2)) w ⇒ ( λ λ λ → 3) w`

6.3.3 Расширенные индексы де Брюина

Пересмотрим кодировку $\mathcal{E}_1$ так, чтобы она учитывала дополнительное требование, накладываемое $\trianglelefteq^2$: выражение, свободные переменные которого уже зарегистрированы в таблице ρ не может быть погружено в другое выражение.

Это требование может быть учтено с помощью добавления к каждому индексу де Брюина подиндекса k, такого, что k есть число конструкторов (включая λ-абстракций) между переменной и соответствующей связывающей конструкцией.

Индекс де Брюина отражает только структуру λ-абстракций, в то время, как подиндекс учитывает структуру λ-абстракций и структуру всех остальных конструкторов.

Обозначим через $\mathcal{E}_2$ кодировку $\mathcal{E}_1$, учитывающую подиндексы. Приведем 2 примера:

e	$\mathcal{E}_2[\![e]\!]$
λ`x` $\rightarrow$ `x`	$\lambda \rightarrow 1_1$
λ`x` $\rightarrow$ `S x`	$\lambda \rightarrow$ `S` 1_2

Равенство двух расширенных индексов естественным образом определяется как равенство индексов и подиндексов. При проверке на гомеоморфное вложение будем считать различные расширенные индексы де Брюина различными конструкторами.

Лемма 100 ($\mathcal{E}_2$, $\trianglelefteq$, $\trianglelefteq^2$)**.** $e_1 \trianglelefteq^2 e_2 \Leftrightarrow \mathcal{E}_2[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам (аналогично 97) □

Лемма 101. *Подиндексы де Брюина ограничены на* $\mathcal{M}_{\mathcal{E}_2}(prog)$.

Доказательство. Аналогично доказательству леммы 98. □

Лемма 102. $(\mathcal{M}(prog), \trianglelefteq^2)$ – *вполне-квазиупорядочение.*

Доказательство. Количество конструкторов в $\mathcal{M}_{\mathcal{E}_2}(prog)$ ограничено в силу лемм 98 и 101. Значит, $(\mathcal{M}_{\mathcal{E}_2}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. Из леммы 100 следует, что $(\mathcal{M}(prog), \trianglelefteq^2)$ – вполне-квазиупорядочение. □

6.3.4 Проблема арности

В кодировке $\mathcal{E}_2$ аппликации были представлены с помощью бинарных кострукторов (представление $\mathcal{A}_2$ из пункта 6.2.2). Рассмотрим кодировку $\mathcal{E}_3$, в которой, в отличие от $\mathcal{E}_2$, аппликации представлены набором конструкторов различной арности (кодировка $\mathcal{A}_*$ из пункта 6.2.2).

Типизация по Хиндли-Милнеру находит *главные* типы (principal types) выражений в программе. Главный тип может содержать типовые переменные, которые в разных контекстах могут инстанциироваться различными *конкретными* типами. Рассмотрим простую функцию *id*:

```
id :: a → a;
id = λx → x;
```

Функция id имеет тип $id :: \forall a.a \to a$. Поскольку типовая переменная a под квантором инстанциируется в зависимости от контекста, мы ничего не можем сказать об арности символа id. В действительности арность символа id в конкретном контексте может быть насколько угодно большой. Например, id можно применить к сколько угодно аргументам, каждый из которых тоже является id!

```
id
id id
id id id
...
id id id id ...
```

Это можно сделать в силу того, что функция id - полиморфная.

То есть, если тип некоторого выражения e_0 (более узко - переменной) является полиморфным, то в принципе можно сконструировать *корректно типизированное* выражение $e = e_0\ e_1 \ldots e_n$ с любой арностью n.

Однако, если тип t_0 выражения e_0 не содержит типовых переменных (мономорфный), то легко показать, что возможны только такие *корректно типизированные* выражения $e = e_0\ e_1 \ldots e_n$, где арность n ограничена.

Если рассмотреть idA, конкретизацию функции id для конкретного типа A, то арность символа idA не может превышать 1. Поскольку $idA\ e :: A$, и не является функцией.

Следующие примеры показывают, почему типизации по Хиндли-Милнеру не допускает появление выражений неограниченной арности:

```
f = λx →  f f x;
h = λx →  h x h;
f1 = λx y →  f1 (x y) y;
f2 = λx y →  f2  x (y x);
```

Приведенные функции неограниченно увеличивают число аргументов аппликации, но не являются корректно типизированными.

Определение 103 (Арность аппликации)**.** Арность аппликации $a[\![e]\!]$ вы-

Рис. 6.2 Грамматика типов HLL

t	$::=$	τ	(типовая переменная)
	$\mid$	$t \to t$	(стрелка)
	$\mid$	$TyCon\ \overline{t_i}$	(типовый конструктор)

ражения определяется следующим образом:

$$\begin{aligned}
a[\![v]\!] &= 0 \\
a[\![f]\!] &= 0 \\
a[\![\lambda v_0 \to e_0]\!] &= 0 \\
a[\![c\ \overline{e_i}]\!] &= 0 \\
a[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] &= 0 \\
a[\![e_0\ e_1\ \ldots\ e_n]\!] &= n
\end{aligned}$$

Лемма 104 ($\mathcal{E}_3$, $\trianglelefteq$, $\trianglelefteq^3$)**.** $e_1 \trianglelefteq^3 e_2 \Leftrightarrow \mathcal{E}_3[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам (аналогично доказательству лемм 97 и 100). □

Язык HLL типизован по Хиндли-Милнеру [22]. Синтаксис типов показан на Рис. 6.2.

Определение 105 (Мономорфный тип)**.** Тип называется мономорфным, если в нем отсутствуют типовые переменные.

Определение 106 (Арность типа)**.** Арность типа $A(t)$ определяется по индукции следующим образом:

$$\begin{aligned}
A[\![\alpha]\!] &= 0 \\
A[\![t_1 \to t_2]\!] &= 1 + A(t_2) \\
A[\![TyCon\ \overline{t_i}]\!] &= 0
\end{aligned}$$

$\mathcal{M}_{\mathcal{E}_3}(prog)$ обозначает $\{\mathcal{E}_3[\![e]\!] \mid e \in \mathcal{M}(prog)\}$.

Нам нужно показать, что теорема Крускала выполняется на $\mathcal{M}_{\mathcal{E}_3}(prog)$. Если мы покажем, что арность типов выражений $\mathcal{M}_{\mathcal{E}_3}(prog)$ ограничена, из этого будет следовать ограниченность арности выражений $\mathcal{M}_{\mathcal{E}_3}(prog)$.

То есть, все аппликации в $\mathcal{M}_{\mathcal{E}_3}(prog)$ кодируются с помощью конечного числа конструкторов. Следовательно, теорема Крускала применима.

К сожалению, арность полиморфных типов сложно оценивать, так как типовые переменные полиморфного типа могу инстанциироваться функциональными типами, увеличивая таким образом арность типа. С другой стороны, арность мономорфных типов задана раз и навсегда, поскольку в них отсутствуют типовые переменные.

Мы покажем, что для мономорфно типизированной программы арность выражений, возникающих во время прогонки, ограничена в силу ограниченности арности соответствующих типов.

Тонкость, однако, заключается в том, что алгоритм суперкомпиляции, реализованный в HOSC, учитывает, что входная программа корректно типизирована, но явным образом не рассматривает конкретные типы.

Таким образом, можно воспользоваться следующим приемом. Пусть дана программа $prog$ с полиморфными типами, мы можем заменить ее на мономорфно типизированную программу $prog_m$, такую, что HOSC строит одинаковые деревья процессов для $prog$ и $prog_m$. Значит, если арности аппликаций на $\mathcal{M}_{\mathcal{E}_3}(prog_m)$ ограничены, то они ограничены и на $\mathcal{M}_{\mathcal{E}_3}(prog)$, поскольку $\mathcal{M}_{\mathcal{E}_3}(prog_m) = \mathcal{M}_{\mathcal{E}_3}(prog)$.

В следующем разделе мы опишем процедуру *мономорфизации* исходной программы $prog$.

Мономорфизация

При типизации по Хиндли-Милнеру мы строим граф зависимостей вызовов функций [86, 110]. В результате получаем направленный граф, состоящий из строго связанных компонент, отсортированных в обратном топологическом порядке. Внутри каждой компоненты (рекурсивные) функции, принадлежащей данной компоненте, являются *мономорфными* в том смысле, что каждое вхождение имени функции, определяемой в данной компоненте, имеет один и тот же тип (возможно, содержащий типовые переменные). Функции из компоненты SCC_i не зависят от функций из компонент SCC_j при $i < j$. Введем специальную связанную компоненту SCC_0, соответствующую целевому выражению.

Мономорфизация строит из программы $prog$ новую программу $prog_m$,

операционно эквивалентную исходной. Вначале $prog_m$ совпадает с $prog$. Предполагаем, что каждому подвыражению и функции в $prog_m$ *явно приписан его тип.*

Пусть A - некоторый зафиксированный *базовый* тип, например:

```
data A = A;
```

Мономорфизация шаг за шагом строит новую программу, уменьшая на каждом шаге граф зависимомостей строго связанных компонент.

1. Если в типе в целевом выражении или его подвыражениях встречается типовая переменная, то заменяем ее на базовый тип A, превращая таким образом целевое выражение в мономорфное. После этого компонента SCC_0 становится мономорфной.
2. Выбираем компоненту с SCC_i с минимальным $i > 0$. Если такой нет – мономорфизация завершена.
3. Если в SCC_0 нет ссылок на символы, определенных в SCC_i – удаляем SCC_i из графа зависимостей и переходим к шагу 2.
   (a) Встраиваем конкретизацию компоненты SCC_i в компоненту SCC_0: берем *одно* вхождение какой-либо функции f из SCC_i. Вхождение f в SCC_0 имеет мономорфный тип в силу того, что SCC_0 уже мономорфизована. Заменяем это вхождение на f_i, где f_i – новое имя. Копируем определения $f \ldots g$ из SCC_i в SCC_0 с переименованиями $f_i \ldots g_i$. Если в компоненте SCC_i есть имена функций, типы которых содержат типовые переменные, которые не зависят от контекста, то конкретизируем эти типовые переменные базовым типом A. После этого шага компонента SCC_0 останется мономорфной: вхождение f в SCC_0 было мономорфным, значит (в контексте данного вхождения) мы однозначно приписываем мономорфные типы всем (под)выражениям в скопированной компоненте.

   Если в расширенной компоненте SCC_0 остались вхождения символов из SCC_i, – повторяем шаг 3(a), иначе – переходим к шагу 2.

Лемма 107. *Мономорфизация программы завершается за конечное число шагов.*

Доказательство. Количество вершин (строго связанных компонент) и дуг (зависимостей вызовов) в изначальном графе конечно. Шаг 1 выполняется один раз и не изменяет количество вершин и дуг в графе. Шаг 2 также не меняет количество вершин и дуг. На шаге 3 удаляется вершина графа – уменьшается количество вершин. Для SCC_i шаг 3(а) выполняется пока есть вхождение символов из SCC_i в SCC_0. Но число таких вхождений конечно и каждый шага 3(а) устраняет одно из них. □

Другими словами, мономорфизация программы завершается, так как в графе зависимостей нет циклов, выходящих за пределы одной компоненты, и в пределах одной компоненты каждое вхождение символа, определяемого в данной компоненте имеет один и тот же тип.

Лемма 108. *Частичное дерева процессов, построенное для мономорфизованной программы, совпадает с частичным деревом процессов, построенным для исходной программы, с точностью до индексов, возникших в результате мономорфизации.*

Доказательство. Определения мономорфизованных функций $f_1, \ldots, f_i$ совпадают с исходным определением f (с точностью до индексов). С другой стороны, алгоритм прогонки HOSC не зависит от конкретных имен функций. Поэтому если стереть индексы в дереве процессов, порожденном прогонкой мономорфизованной программы $prog_m$, то получится дерево, порожденном прогонкой исходной программы. □

Конечность арности

Лемма 109. *Арность аппликации любого подвыражения, возникающего при построении частичного дерева процессов для программы prog, ограничена максимальной арностью типа подвыражения в мономорфизованной программе $prog_m$.*

Доказательство. Рассмотрим дерево частичное дерево процессов для программы $prog_m$. Выражение в любом узле дерева является мономорфным. Поэтому достаточно показать, что ограничена арность типа любого подвыражения, порождаемого прогонкой. Это верно для выражения в корне дерева, являющимся целевым выражением программы $prog_m$. Покажем,

что каждый шаг прогонки сохраняет ограниченность арности аппликаций.

- D_1, D_2, D_3, D_8 - при переходе к подвыражениям максимальная арность типа подвыражения не может увеличиться.
- D_4 - максимальная арность после завершения шага не больше максимальной арности до шага и максимальной арности подвыражений в определении f_0.
- D_5, D_6, D_7, D_7^0 - максимальная арность типа подвыражения не увеличивается, так как $type(v_0) = type(e_0)$, $type(v_{ik}) = type(e'_k)$, $type(v\ \overline{e'_j}) = type(p_i)$ соответственно.

При декомпозиции максимальная арность типа подвыражения не может увеличиться. Поскольку арность аппликации во время построения частичного дерева процессов для программы $prog_m$ ограничена, то по лемме 108 она ограничена и при построении частичного дерева процессов для программы $prog$. □

Из леммы следует, что в кодировке $\mathcal{M}_{\mathcal{E}_3}(prog)$ количество конструкторов $App_2, \ldots, App_n$ ограничено.

Следствие 110. *Множество конструкторов $App_2, \ldots, App_n$, появляющихся в $\mathcal{M}_{\mathcal{E}_3}(prog)$, конечно.*

Теорема 111. $(\mathcal{M}(prog), \trianglelefteq^3)$ – *вполне-квазиупорядочение.*

Доказательство. Множество конструкторов в $\mathcal{M}_{\mathcal{E}_3}(prog)$ ограничено в силу лемм 98, 101 и 109. Значит, $(\mathcal{M}_{\mathcal{E}_3}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. По лемме 104 $(\mathcal{M}(prog), \trianglelefteq^3)$ – вполне-квазиупорядочение. □

Следствие 112. *Так как $\trianglelefteq^3$ совпадает с $\trianglelefteq^{**}$, мы показали, что отношение $\trianglelefteq^{**}$ – вполне-квазиупорядочение на $\mathcal{M}(prog)$.*

Следствие 113. *Отношение $\trianglelefteq_c^{**}$ – вполне-квазиупорядочение на $\mathcal{M}(prog)$.*

Доказательство. Следует из теоремы 111 и леммы Хигмана [26]. □

Рис. 6.3 Примеры кодирования

e	$\mathcal{E}_3[\![e]\!]$
`map f`	`App2(map, f)`
`map (compose f g)`	`App2(map, App3(compose, f, g))`
`λx → Cons x Nil`	`λ → Cons(1₂, Nil)`
`case x of {Z → x; S b → f (g b);}`	`CaseNat(x, x, λ→ App2(f, App2(g, 1₃)))`

6.3.5 Кодировка $\mathcal{E}_3$

В разделе 4.5 в [59] приведены примеры вкладывающихся и не вкладывающихся по $\trianglelefteq_c^{**}$ выражений. Некоторые выражения из этих примеров в кодировке $\mathcal{E}_3$ приведены на Рис. 6.3.

Близкая к $\mathcal{E}_3$ кодировка использовалась в [74] для решении проблемы конфликта имен при дефорестации.

Если кодировать выражения через $\mathcal{E}_3$, то нет нужды в использовании таблицы соответствия ρ при проверке на расширенное вложение $\trianglelefteq_c^{**}$.

6.4 Завершаемость суперкомпилятора $\mathbf{SC}_{*--}$

Все теоремы и доказательства этого раздела повторяют (с незначительными модификациями) материал из [108].

Идея – показать, что выполняются условия теоремы 95.

Мы не будем явно рассматривать операцию $fold$, так реально она используется только для удобства записи алгоритма построения остаточной программы, – просто будем считать, что узел β является обработанным в то числе и если выполняется условие для совершения операции $fold$.

Лемма 114. *Суперкомпилятор* $\mathbf{SC}_{*--}$ *- преобразователь Коши.*

Доказательство. Определим отношение $\succeq$ на множестве $\mathcal{L}$ как: $let\ x'_1 = e'_1, \ldots, x'_m = e'_m\ in\ e' \succeq let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \Leftrightarrow m = 0\ \&\ n \geq 0$. Просто показать, что $\succeq$ – вполне фундированное квазиупорядочение. По-

кажем, что для на любом шаге построения частичного дерева процессов

$$t_{i+1} = t_i\{\gamma := t'\}$$

для некоторого узла γ и выполняется одно из следующих условий:

- $\gamma \in leaves(t_i)$ и $\gamma.expr = t'.root.expr$ (прогонка),
- $\gamma.expr \succ t'.root.expr$ (обобщение).

Достаточно проанализировать возможные операции на каждом шаге:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, где $\beta \in leaves(t_i)$ и $t' = \beta.expr \to e_1, \ldots, e_n$. Значит:
$$\beta.expr = t'.root.expr$$

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to \}$, где $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\simeq \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. Тогда $\alpha.expr \equiv e$ и $\beta.expr \equiv \alpha.expr\theta$, но $\alpha.expr \not\simeq \beta.expr$, то есть $n > 0$. Таким образом:
$$\beta.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, где $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c^{**})$, $\alpha.expr\ /\!\lessdot \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Тогда $\alpha.expr \not\equiv e$, но $\alpha.expr = e\theta_1$, то есть $n > 0$. Таким образом:
$$\alpha.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

□

Лемма 115. *Суперкомпилятор* $\mathbf{SC}_{*--}$ *при построении частичного дерева процессов сохраняет конечный непрерывный предикат.*

Доказательство. Определим $l : \mathcal{L} \to \mathcal{E}$:

$$l(let\ \overline{v_i = e_i};\ in\ e_g) = e_g\{\overline{v_i := e_i}\}$$

Определим $\sqsupseteq$ на множестве $\mathcal{L}$:

$$e_1 \sqsupseteq e_2 \Leftrightarrow (\mathcal{S}[\![l(e_1)]\!] > \mathcal{S}[\![l(e_2)]\!]) \vee (\mathcal{S}[\![l(e_1)]\!] = \mathcal{S}[\![l(e_2)]\!] \wedge l(e_1) \succcurlyeq l(e_2))$$

Так как $\preccurlyeq$ – вполне фундированное квазиупорядочение, то $\sqsubseteq$ – вполне фундированное квазиупорядочение. Рассмотрим предикат $q : T(l) \to \mathbb{B}$, определенный как

$$q(t) = p(t^0)$$

где t^0 - внутренность t, а предикат $p : T(l) \to \mathbb{B}$:

$$p(t) = \begin{cases} 0 & \text{если } \exists \alpha, \beta : \alpha = ancestor(t,\ \beta, \trianglelefteq_c^{**}) \text{ и } \alpha, \beta \text{ - нетривиальные} \\ 0 & \text{если } \exists \alpha, \beta : \alpha \to \beta,\ \alpha, \beta \text{ - тривиальные и } \alpha.expr \not\sqsupset \beta.expr \\ 1 & \text{в противном случае} \end{cases}$$

Множества тривиальных и нетривиальных выражений представляют разбиение множества $\mathcal{M}(prog)$. $\trianglelefteq_c^{**}$ – вполне-квазиупорядочение на нетривиальных выражениях $\mathcal{M}(prog)$ и $\sqsubseteq$ – вполне фундированное квазиупорядочение на тривиальных выражениях $\mathcal{M}(prog)$. Следовательно (см. утверждение 94), p и q – конечные непрерывные предикаты. Осталось показать, что $\mathbf{SC}_{*--}$ сохраняет предикат q.

Пусть дано дерево t и узел β, будем считать β *корректным* в дереве t, если выполняются оба следующих условия: (i) β – нетривиальный узел, β не является листом дерева $t \Rightarrow ancestor(t, \beta, \trianglelefteq_c^{**}) = \bullet$
(ii) $\exists \alpha : \alpha \to \beta$ и α – тривиальный узел $\Rightarrow \alpha.expr \sqsupset \beta.expr$

Дерево считается корректным, если все его узлы – корректные.

Легко видеть, что $q(t) = 1$, если t – корректное. Достаточно показать, что на каждом шаге i t_i - корректное. Доказываем по индукции по i.

Для $i = 0$, (i)-(ii) выполняются.

Для $i > 0$, предполагаем, что t_i - корректное. Для t_{i+1} рассмотрим различные операции этого шага:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, где $\beta \in leaves(t_i)$ и $t' = \beta.expr \to e_1, \ldots, e_n$ и $e_1, \ldots, e_n = \mathcal{D}[\![\beta.expr]\!]$. Необходимо показать, что β и $children(t_{i+1}, \beta)$ – корректные в t_{i+1}.

   Рассмотрим β: (1) если β – нетривиальный узел, алгоритм гаранти-

рует, что для β выполнено условие (i), условие (ii) верно по гипотезе индукции, (2) если β – нетривиальный узел, то для β условие (i) выполнено тривиальным образом, и условие (ii) верно по гипотезе.
Рассмотрим $children(t_{i+1}, \beta)$: (1) если β – нетривиальный узел, условия (i) и (ii) выполнены тривиальным образом для $children(t_{i+1}, \beta)$, (2) если β – тривиальный узел, то условие (i) выполнено тривиальным образом, а условие (ii) выполняется, т. к. $\forall i : \beta.expr \sqsupset e_i$, – здесь очень важно, что при прогонке тривиального выражения, размер выражения уменьшается (см. 6.5).

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \rightarrow\}$, где $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\simeq \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. Необходимо показать, что β – корректен в t_{i+1}.
Условие (i) выполняется тривиальным образом. Из гипотезы индукции и факта, что $l(\beta_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\beta_{i+1}.expr)$, следует выполнение условия (ii).
3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \rightarrow\}$, где $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c^{**})$, $\alpha.expr\ /\!\lessdot \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Необходимо показать, что α – корректен в t_{i+1}.
Условие (i) выполняется тривиальным образом. Из гипотезы индукции и факта, что $l(\alpha_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\alpha_{i+1}.expr)$, следует выполнение условия (ii).

□

Следствие 116. *Построение дерева частичных процессов суперкомпилятором* $\mathbf{SC}_{*--}$ *завершается.*

Доказательство. Из лемм 114, 115 следует, что выполняются условия теоремы 95. □

6.5 Пересмотр обработки ситуации зацикливания

В случае суперкомпилятора для языка первого порядка [104, 107] нетривиальными узлами считаются только узлы вида $con\langle f\rangle$ и $con\langle case\ v\ of\{\ldots\}\rangle$.

Рис. 6.4 Оператор неподвижной точки с глобальным определением

```
data D = F (D → D);

λf → apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
where

apply = λx → case x of { F f → f; };
```

Рис. 6.5 Оператор неподвижной точки без глобального определения

```
data D = F (D → D);

λf → (λy → case y of { F g → g; })
    (F (λx → f ((λy → case y of { F g → g; }) x x)))
       (F (λx → f ((λy → case y of { F g → g; }) x x)))
```

И действительно, в случае языка первого порядка, в частичном дереве процессов не может возникнуть бесконечная последовательность идущих подряд тривиальных узлов. Однако, если такое же разбиение узлов на тривиальные и нетривиальные использовать для суперкомпилятора языка HLL, то на некоторых примерах он может не завершаться.

Пусть SC_0 - суперкомпилятор $\mathbf{SC}_{*\_\_}$, в котором нетривиальными узлами являются только узлы с выражениями вида $con\langle case\ v\ of\{\ldots\}\rangle$ и $con\langle f\rangle$.

На Рис. 6.4 показано определение оператора неподвижной точки без использования явной рекурсии. Суперкомпиляция данного выражения завершается суперкомпилятором SC_0, так как при построении дерева процессов встречаются нетривиальные выражения вида

```
apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
```

Однако, если заменить вызов функции `apply` на ее определение, то SC_0 не завершится! Причина заключается в том, что прогонка измененной программы (Рис. 6.5) будет порождать только тривиальные выражения, которые не проверяются на вложение.

Суперкомпилятор $\mathbf{SC}_{*\_\_}$ завершается на этой программе.

6.6 Завершаемость остальных суперкомпиляторов

Пусть рассмотрим отношение $\trianglelefteq_\bullet$ - отношение, выполняющееся для выражений e_1 и e_2, если $e_1 \trianglelefteq^{**} e_2$ и $expClass(e_1) = expClass(e_2)$.

Лемма 117 (Вполне-квазиупорядочение $\trianglelefteq_\bullet$)**.** $(\mathcal{M}(prog), \trianglelefteq_\bullet)$ – *вполне-квазиупорядочение.*

Доказательство. От противного. Предположим, что существует последовательность из выражений $\mathcal{M}(prog)$, на которой $\trianglelefteq_\bullet$ не является вполне-квазиупорядочением. Поскольку типов редексов конечное число, то найдется подпоследовательность из выражений с одинаковым типом редекса. На ней $\trianglelefteq_\bullet$ будет вполне-квазиупорядочением. Противоречие. □

Следствие 118 (Завершаемость суперкомпилятора $\mathbf{SC}_{*-+}$)**.** *Из леммы 117 следует, что суперкомпилятор* $\mathbf{SC}_{*-+}$ *завершается на любой входной программе.*

Теорема 119 (Завершаемость суперкомпилятора $\mathbf{SC}_{*++}$)**.** *Суперкомпилятор* $\mathbf{SC}_{*++}$ *завершается на любой входной программе.*

Доказательство. От противного. Предположим, что суперкомпилятор $\mathbf{SC}_{*++}$ зацикливается на некоторой программе p. Тогда в строящемся дереве процессов есть бесконечный путь. На этом пути не может быть бесконечной подпоследовательности глобальных узлов, т.к. такая подпоследовательность не допускается отношением $\trianglelefteq_\bullet$ с учетом глобальных и локальных узлов, следовательно, на этом пути есть бесконечная цепочка из подряд идущих локальных узлов. Но это также невозможно в силу использования отношения $\trianglelefteq_\bullet$ с учетом глобальных и локальных узлов. Противоречие. □

Утверждение 120 (Завершаемость суперкомпилятора $\mathbf{SC_{ijk}}$)**.** *Если суперкомпилятор* $\mathbf{SC}_{*++}$ *завершается на входной программе* p*, то суперкомпилятор* $\mathbf{SC_{ijk}}$ *с любыми рассмотренными параметрами также завершается на программе* p*.*

6.7 Выводы

Таким образом, в данной главе была показана завершаемость всех суперкомпиляторов $\mathbf{SC_{ijk}}$.

Первое полное описание суперкомпилятора с доказательством завершаемости можно найти в магистерской работе Сёренсена [103]. Суперкомпилятор Сёренсена работает с ленивым функциональным языком первого порядка. Затем Сёренсеном был разработан инструментарий для доказательство завершаемости преобразователей программ, работающих с деревьями [108].

Митчелл и Рансиман описали суперкомпилятор Supero для подмножества функционального языка с ленивой семантикой Haskell [78]. Доказательство завершаемости Supero не опубликовано.

Джонссон и Нордландер описали суперкомпилятор для функционального языка с функциями высших порядков с семантикой вызовов по значению [52] и показали его завершаемость [51].

Данная работа отличается от [78, 52] в следующем:

- При проверке на гомеоморфное вложение HOSC различает связанные переменные (вкладываются только соответствующие связанные переменные). Поэтому свисток HOSC срабатывает реже, чем свисток в [78, 52], и происходит меньше переобобщений.
- HOSC проверяет только вложение через сцепление – это гарантирует наличие нетривиального тесного обобщение. HOSC однозначно строит обобщение. В [78, 52] свисток срабатывает и при погружении – в данном случае может отсутствовать наиболее тесное обобщение и возникает неоднозначность при построении обобщения.
- В [78, 52] глобальные функции имеют фиксированную арность и каждый вызов глобальной функции должна быть насыщенным по аргументам. Поэтому в [78, 52] иногда в программу приходится искусственно вставлять λ-абстракции, чтобы избежать частичного применения глобальной функции. HOSC не имеет такого ограничения и допускает любые (корректно типизированные) частичные аппликации.

Главной новизной данной главы является доказательство того, что

уточненное гомеоморфное вложение, различающее связанные переменные, все равно остается вполне-квазиупорядочением на множестве выражений возникающих во время суперкомпиляции (а не на всем множестве выражении). Однако, этого достаточно для завершаемости суперкомпилятора. Вдобавок, использование более утонченного вложения позволяет в некоторых случая избежать переобобщения.

Глава 7

Распознавание эквивалентности выражений

Программирование в функциональном стиле позволяет разработчикам писать модульные, поддерживаемые и элегантные программы. Однако, за эти преимущества приходиться платить скоростью работы функциональных программ. Использование промежуточных структур данных, функций высшего порядка, ленивого порядка вычислений и композиций функций может существенно увеличить накладные расходы на исполнение такой программы. Первоначально суперкомпиляция разрабатывалась как средство оптимизации программ, написанных на функциональном языке Рефал [113, 116].

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

Более того, если некоторое знание можно формально представить в виде программы, суперкомпиляцию можно использовать для анализа закодированного знания, а также для того, чтобы выявить и явным образом выразить нетривиальные, скрытые факты (свойства) закодированного знания.

Таким образом, в анализе программ суперкомпиляция может играть роль (по крайней мере, потенциально) рентгена в радиографии.

7.1 Моделирование знаний в виде программ

Предположим, что мы хотим закодировать некоторое знание в виде программы, чтобы подвергнуть ее анализу с помощью суперкомпиляции. Какой язык программирования можно считать пригодным для данных целей? Можно утверждать, что:

1. Семантика языка должна быть чётко определена.
2. Желательно, чтобы суперкомпилятору было относительно легко обрабатывать программы на этом языке, особенно если требуется строгое сохранение семантики программы.
3. Язык должен быть удобен для представления знаний в виде программ. В частности, поддержка бесконечных структур данных удобна для для представления бесконечной последовательности событий и тому подобного.
4. Желательно иметь возможность работать с функциями как с данными. Это полезно для формулировки и доказательства утверждений "высшего порядка" – с функциями, стоящими под кванторами.

Первый аргумент уместен, поскольку мы заинтересованы в анализе программ, что достаточно затруднительно для языка с неясной семантикой. Таким образом, функциональный язык программирования является подходящим вариантом в силу прозрачной и простой семантики.

Второе требование легче выполнимо в случае функционального языка с ленивой, а не строгой семантикой вычислений, поскольку многие методы преобразования программ (включая суперкомпиляцию и дефорестацию) основаны на моделировании ленивого вычисления. Если применять такие методы для языков со строгой семантикой вычислений, могут быть нарушены свойства завершаемости. Этого, в принципе, можно избежать, если накладывать некоторые ограничения на входную программу. Например, суперкомпиляция сохраняет свойства завершаемости, если входная программа всегда завершается (см. абсолютное функциональное программирование [126]). Другой подход заключается в ограничении преобразований, осуществляемых во время суперкомпиляции, что требует некоторого дополнительного анализа [49]. Однако для применения суперкомпиляции к анализу программ самым простым решением будет предположить ле-

нивость входного языка. Кроме того, для ленивого языка естественным образом выполняется третье требование.

Четвертое требование основано на том, что практически в любом языке программирования аргументы функции можно понимать как находящиеся под квантором всеобщности. Таким образом, определение функции можно интерпретировать как “для любых аргументов x, y, ...” Если мы работаем с языком первого порядка, то мы можем обобщенно рассматривать данные первого порядка. Однако, если мы работаем с языком высшего порядка, мы можем также абстрагировать функции. Функции могут представлять правила, преобразования, стратегии и т.п.

Помимо прочего, бывают случаи, когда результаты суперкомпиляции сложно представить, используя язык первого порядка [93].

Таким образом, можно считать, что язык HLL хорошо подходит для моделирования знаний в виде программ.

7.2 Доказательство свойств программ методами суперкомпиляции

Как было показано Турчиным [116, 125], некоторые свойства программ могут быть доказаны с помощью преобразования программ. Например, пусть дана функция f (в виде программы), и мы хотим доказать, что для любого аргумента x результат, возвращаемый функцией f, удовлетворяет некоторому свойству p. Тогда мы можем закодировать свойство p в виде программы и попытаться “упростить” выражение $p(f(x))$ с помощью суперкомпилятора. Если структура преобразованного суперкомпилятором выражения настолько проста, что несложно видеть, что вычисление выражения всегда завершается и результатом не может быть `False`, мы можем заключить, что результатом вычисления исходного выражения $p(f(x))$ всегда является `True`. Следовательно, результат вычисления $f(x)$ всегда удовлетворяет свойству p.

Предположим, что написана библиотека на языке HLL, фрагмент которой показан на Рис. 7.1. Функция `append` конкатенирует два списка. Функция `containsA` определяет, содержится ли в списке элемент `A`.

Допустим, мы хотим проверить свойство этих функций: при конкате-

Рис. 7.1 Фрагмент первой версии библиотеки

```
data List a = Nil | Cons a (List a);
data Enum = A | B;
data B = True | False;

append = λxs ys →
    case xs of {
        Nil → ys;
        Cons z zs → Cons z (append zs ys);
    };

containsA = λxs →
    case xs of {
        Nil → False;
        Cons x1 xs1 → case x1 of{
            A → True;
            B → containsA xs1;
        };
    };
```

нации трех списков, один из которых состоит из одного элемента `A` получается список `l`, для которого `containsA l` возвращает `True`.

При тестировании проверяется выполнение этого свойства на конкретных данных:

```
containsA (app Nil (app (Cons A Nil) Nil))
containsA (app (Cons B Nil) (app (Cons A Nil) (Cons B Nil)))
...
```

Гарантировать, что это свойство выполняется для любого набора списков, при помощи тестирования не представляется возможным, так как таких наборов бесконечно много. Необходимо показать, что вычисление выражения

```
containsA (app x (app (Cons A Nil) z))
```

может завершиться только с результатом `True` для любых списков `x` и `z`.

Но суперкомпиляция при построении частичного дерева процессов рассматривает все возможные множества данных. Результат суперкомпиляции выражения `containsA (app xs (app (Cons A Nil) zs))` приведен на Рис. 7.2.

Рис. 7.2 `containsA (append xs (append (Cons A Nil) zs))`: результат суперкомпиляции

```
data List a = Nil | Cons a (List a);
data Enum = A | B;
data B = True | False;

letrec f = λu1 →
    case u1 of {
        Nil  → True;
        Cons t w → case t of {
            A  → True;
            B  → f w;
        };
    }
in f xs
```

Видно, что единственным возможным результатом вычисления может быть только `True`. На некоторых списках, допустим, когда список `xs` является бесконечным и состоит только из элементов `B`, вычисление не завершается, но результатом вычисления не может быть `False`.

Предположим, что при дальнейшей разработке библиотеки, как это обычно бывает, частные случаи были обобщены – функция `containsA` была обобщена до функции `contains`, которая принимает предикат и список и проверяет, существует ли в списке элемент, удовлетворяющий предикату `p` – см. Рис. 7.2.

Теперь свойство кодируется как:

```
contains (equals A) (append xs (append (Cons A Nil) zs))
```

Однако, можно вместо конкретного элемента `A` рассматривать произвольный элемент `y` и рассмотреть более общее свойство:

```
contains (equals y) (append xs (append (Cons y Nil) zs))
```

Результат суперкомпиляции этого выражения приведен на Рис. 7.4. С помощью простого анализа показывается, что результатом выполнения остаточной программы может быть только `True`, а значит `contains (equals y) (append xs (append (Cons y Nil) zs))` может завершиться только с результатом `True`.

Рис. 7.3 Фрагмент второй версии библиотеки

```
data List a = Nil | Cons a (List a);
data Enum = A | B;
data B = True | False;

append = λxs ys →
    case xs of {
        Nil → ys;
        Cons z zs → Cons z (append zs ys);
    };

contains = λp xs →
    case xs of {
        Nil → False;
        Cons x1 xs1 → or (p x1) (contains p xs1);
    };

equals = λx y → case x of {
    A → case y of {A → True; B → False;};
    B → case y of {A → False; B → True;};
};

or = λx y → case x of {True → True; False → y;};
```

Рис. 7.4 `contains (equals y) (append xs (append (Cons y Nil) zs))`: результат суперкомпиляции

```
data List a = Nil  | Cons a (List a);
data Enum  = A  | B ;
data B  = True  | False ;

letrec f = λv9 w9 →
    case v9  of {
        Nil → case w9 of { A → True; B → True; };
        Cons s4 p5 →
            case  w9  of {
                A → case s4 of { A → True; B  → f p5 A; };
                B → case s4 of { A → f p5 B; B  → True; };
          };
    }
in f xs y
```

Недавно Немытых и Лисица показали плодотворность такого подхода [71] к тестированию программ, – верификация ряда кэш-когерентных протоколов была осуществлена с помощью суперкомпилятора SCP4.

7.3 Доказательство эквивалентности выражений

7.3.1 Доказательство эквивалентности выражений, основанное на равенстве

Как было показано Турчиным [113], доказательство эквивалентности двух выражений `t1` и `t2` можно свести к доказательству свойства одного 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`.

Рассмотрим программу на Рис. 7.5, в которой определена функция `plus` принимающая в качестве параметров два натуральных числа (в унарной системе) и возвращающая их сумму. Мы хотим показать, что

```
equals (plus (S x) y) (plus x (S y))
```

или, в более “математической” записи, что

$$(x + 1) + y = x + (y + 1)$$

Результат суперкомпиляции этой программы показан на Рис. 7.6. Несложно видеть, что в результате вычисления преобразованной программы никогда не может получиться `False`. Однако, в рассуждениях такого рода остаются непроясненные моменты.

Пусть вычисление выражения `equals t1 t2` не может завершиться с результатом `False`. Значит ли это, что `t1` и `t2` действительно “эквивалентны”?

Естественно, ответ зависит от того, как мы определяем понятие “эквивалентность”. Доказательство эквивалентности выражений, основанное на равенстве предполагает следующее:

1. Существует заранее определенная (встроенная) функция `equals`, или,

Рис. 7.5 $(x+1)+y=x+(y+1)$: исходная программа

```
data Nat = Z | S Nat;
data Boolean = True | False;

equals (plus (S x) y) (plus x (S y)) where

plus = λx y →
    case x of {
        Z → y;
        S x1 → S (plus x1 y);
    };

equals = λx y →
    case x of {
        Z →
            case y of {
                Z → True;
                S y1 → False;
            };
        S x1 →
            case y of {
                Z → False;
                S y1 → equals x1 y1;
            };
    };
```

по крайней мере, `equals` можно определить для результатов вычисления `t1` и `t2`.

2. Все рассматриваемые структуры данных – конечны.
3. Вычисление `t1` и `t2` всегда завершается.

Предположение 1 обычно верно в случае аппликативных функциональных языков первого порядка (как Рефал [113, 71]). Однако, в случае языков высшего порядка возникают некоторые проблемы, так как результаты вычисления `t1` и `t2` могут оказаться функциональными значениями, которые невозможно проверить на равенство.

Предположение 2 не выполняется автоматически в случае языков с ленивой семантикой (даже первого порядка).

Предположение 3 может быть неверным во многих интересных случаях. К примеру, если `t1` и `t2` оперируют бесконечными структурами

Рис. 7.6 $(x+1)+y = x+(y+1)$: преобразованная программа

```
data Nat = Z | S Nat;
data Boolean = True | False;

letrec f = λp2 r2 →
    case p2 of {
        Z → letrec g = λs2 →
            case s2 of {
                Z → True;
                S w → g w;
            } in g r2;
        S p1 → f p1 r2;
    }
in f x y
```

данных и никогда не завершаются, но тем не менее, эквивалентны (т.е. имеют один и тот же смысл в соответствии с семантикой языка).

7.3.2 Доказательство эквивалентности выражений, основанное на нормализации

Чтобы избавиться от рассмотрения предикатов равенства нам нужно другое, более общее определение эквивалентности выражений, чем в случае языка первого порядка. Напомним, что мы понимаем эквивалентность двух выражений в операционном смысле [87]:

> Грубо говоря, два выражения M и M' некоторого языка программирования эквивалентны, если любые вхождения выражений M и M' во всей программе могут быть взаимно заменены друг на друга, не затрагивая результат вычисления.

В частности, процитированное определение предполагает, что две программы эквивалентны самым тривиальным образом, если они текстуально совпадают друг с другом.

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 [56]).

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 [10, 11, 19, 27].

Заметим, что 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 [73] и была применена для доказательства эквивалентности программ на функциональном языке первого порядка, при условии, что программы оперируют только конечными данными и гарантированно завершаются.

> Мы утверждаем, 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.

Рассмотрим программу на Рис. 7.7, содержащую определения нескольких хорошо известных функций, работающих со списками. В результате суперкомпиляции выражения `map (compose f g) xs` получается программа, показанная на Рис. 7.8. С другой стороны, результате суперкомпиляции выражения `(compose (map f) (map g)) xs` получается та же самая остаточная программа (с точностью до переименования связанных переменных)! Следовательно, мы доказали, что верно следующее:

```
map (compose f g) xs = (compose (map f)(map g)) xs
```

для всех `f`, `g`, и `xs`, допустимых системой типизации языка HLL. Отметим, что данное утверждение верно для любых списков `xs`, включая

Рис. 7.7 Функции над списками

```
data List a = Nil | Cons a (List a);
data Boolean = True | False;
data Pair a b = P a b;

compose = λf g x → f (g x);
unit = λx → Cons x Nil;
rep = λxs → append xs;
abs = λf → f Nil;
iterate = λf x → Cons x (iterate f (f x));
fp = λp1 p2 → case p1 of {
    P a1 a2 → case p2 of { P b1 b2 → P (a1 b1) (a2 b2); };
};
map = λf xs → case xs of {
    Nil → Nil;
    Cons x1 xs1 → Cons (f x1) (map f xs1);
};
join = xs → case xs of
    Nil → Nil;
    Cons x1 xs1 → append x1 (join xs1);
};
append = λxs ys → case xs of {
    Nil → ys;
    Cons x1 xs1 → Cons x1 (append xs1 ys);
};
idList = λxs → case xs of {
    Nil → Nil;
    Cons x1 xs1 → Cons x1 (idList xs1);
};
filter = λp xs → case xs of {
    Nil → Nil;
    Cons x xs1 → case p x of {
        True → Cons x (filter p xs1);
        False → filter p xs1;
    };};
zip = λp → case p of { P xs ys → case xs of {
        Nil → Nil;
        Cons x1 xs1 → case ys of {
            Nil → Nil;
            Cons y1 ys1 → Cons (P x1 y1) (zip (P xs1 ys1));};
    };
};
```

Рис. 7.8 Результат суперкомпиляции *map* (*compose f g*) *xs*

```
data List a = Nil | Cons a (List a)
letrec h = λys →
    case ys of {
        Nil → Nil;
        Cons y1 ys1 → Cons (f (g y1)) (h ys1);
    }
in h xs
```

бесконечные списки и $\bot$, чьи элементы могут быть данными первого порядка, функциями или $\bot$. Также отметим, что функции `f` и `g` могут не завершаться.

> Таким образом, подход, основанный на нормализации, позволяет доказывать утверждения, которые даже невозможно сформулировать в терминах подхода, основанного на равенстве.

В специализаторе HOSC реализован распознаватель эквивалентных выражений, в основе которого лежит подход, основанный на нормализации выражений. Ниже приведены примеры эквивалентных выражений, распознаваемых суперкомпилятором $\mathbf{SC}_{*++}$:

```
compose (map f) unit = compose unit f
compose (map f) join = compose join (map (map f))
append (map f xs) (map f ys) = map f (append xs ys)
append (append xs ys) zs = append xs (append ys zs)
filter p (map f xs) = map f (filter (compose p f) xs)
iterate f (f x) = map f (iterate f x)
map (compose f g) xs = (compose (map f)(map g)) xs
rep (append xs ys) zs = (compose (rep xs) (rep ys)) zs
(compose abs rep) xs = idList xs
map (fp (P f g)) (zip (P x y)) = zip (fp (P (map f) (map g)) (P x y))
append r (Cons p ps) =
  case (append r (Cons p Nil)) of {
    Nil  → ps;
    Cons v vs → Cons v (append vs ps);
  }
```

Отметим, что некоторые из приведенных пар эквивалентных выражений являются так называемыми “бесплатными” теоремами [128, 45].

На сайте проекта можно найти подборку примеров распознавания эквивалентных выражений. В качестве тестов для проверки способности суперкомпилятора HOSC распознавать эквивалентнтость выражений рассматриваются примеры из первой главы книги “Алгебра программирования” Ричарда Бёрда [14], где эквивалентность выражений доказывается ручными преобразованиями. Суперкомпилятор HOSC распознает эквивалентость всех 25 примеров в полностью автоматическом режиме. Стоит отметить, что если использовать в качестве свистка не предложенное автором уточненное вложение $\trianglelefteq^{**}$, а адаптацию классического вложения $\trianglelefteq$, то суперкомпилятор HOSC распознает лишь 6 эквивалентностей из 25. Это показывает плодотворность использования уточненного вложения $\trianglelefteq^{**}$.

Пусть дана программа на Рис. 7.5, ассоциативность сложения может быть доказана с помощью суперкомпиляции обоих частей уравнения:

```
plus (plus x y) z = plus x (plus y z)
```

Можно ожидать, что коммутативность сложения

```
plus x y = plus y x
```

также может быть доказана описанным методом. Однако, это не так, просто потому, что это предположение неверно. Язык HLL – язык с ленивым порядком вычислений, поэтому

`plus (S Z) ⊥ (S ⊥),`

но

`plus ⊥ (S Z) = ⊥`

7.3.3 Генерация множеств эквивалентных выражений

Поскольку множество всех выражений (использующих функции и конструкторы из данной программы) рекурсивно перечислимо, используя изложенный метод можно сконструировать генератор, производящий множества эквивалентных выражений. Простая процедура может быть такой.

Рис. 7.9 Монада `Maybe`

```
data Maybe a = Just a | Nothing;

compose = λf g x →  f (g x);
return = λx →  Just x;
fmap = λf m →  case m of {
    Nothing →  Nothing;
    Just x →  Just (f x);
};
join = λm k →  case m of {
    Nothing →  Nothing;
    Just x →  k x;
};
id = λm →  case m of {
    Nothing →  Nothing;
    Just x →  Just x;
};
gid = λx →  x;
bind = λm k →  join m (λx →  k);
mzero = Nothing;
```

Вначале генерируем потенциально бесконечную последовательность выражений, упорядоченную по размеру выражения. Далее, эта последовательность фильтруется, – остаются только корректно типизированные выражения. Затем оставшиеся выражения нормализуются суперкомпиляцией и разбиваются на классы эквивалентности предложенным методом.

Очевидно, что описанная выше процедура “неполна”, так как проблема эквивалентности программ алгоритмически неразрешима в общем случае. Таким образом, для любого суперкомпилятора есть эквивалентности, которые он не доказывает. Однако, важная деталь описанной выше процедуры заключается в том, что она способна автоматически *распознавать* эквивалентность выражений, а не только доказывать ее.

7.4 Проверка корректности реализаций монад

В языке Haskell существуют требования к согласованности некоторых операций, которые определяются программистом при реализации монады. Монада – это некоторый алгебраический тип данных и набор функций

Рис. 7.10 Монада `List`

```
data List a = Nil | Cons a (List a);

compose = λf g x →  f (g x);
foldr = λf z xs →
    case xs of {
        Nil → z;
        Cons y ys → f y (foldr f z ys);
    };
append = λxs ys →
    case xs of {
        Nil → ys;
        Cons x1 xs1 → Cons x1 (append xs1 ys);
    };
map = λf xs →
  case xs of {
    Nil → Nil;
    Cons x1 xs1 → Cons (f x1) (map f xs1);
  };
fmap = map;
join = λm k → foldr (compose append k) Nil m;
return = λx → Cons x Nil;
id = λx → case x of {
    Nil → Nil;
    Cons y ys → Cons y (id ys);
};
gid = λx → x;
mzero = Nil;
bind = λm k → join m (λx → k);
```

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 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. $\forall m.\ join\ m\ return \cong id\ m$
4. $\forall f, g, xs.\ fmap\ (compose\ f\ g)\ xs \cong compose\ (fmap\ f)\ (fmap\ g)\ xs$
5. $\forall f, xs.\ fmap\ gid\ xs \cong id\ xs$
6. $\forall f, xs.\ fmap\ f\ xs \cong join\ xs\ (compose\ return\ f)$

Рис. 7.11 Монада State

```
data Pair a b = P a b;

compose = λf g x →  f (g x);
gid = λx → x;
idP = λp → case p of {
    P a b → P a b;
};
fmap = λf m s → case m s of {
    P a s1 → P (f a) s1;
};
id = λf x → idP (f x);
return = λa s → P a s;
join = λm k s → case m s of {
    P a s1 → k a s1;
};
bind = λm k → join m (λx → k);
mzero = λs → undefined;
undefined = undefined;
```

7. $\forall f.\ join\ mzero\ f \cong mzero$
8. $\forall v.\ bind\ v\ mzero \cong mzero$

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

Однако, эта задача сводится к распознаванию эквивалентности выражений. На рисунках 7.9, 7.10, 7.11 приведены реализации монад Maybe, List и State соответственно[1].

Мы использовали суперкомпилятор $\mathbf{SC}_{*++}$ для проверки корректности реализаций монад. Результаты распознавания эквивалентностей среди монадических законов применительно к реализациям монад показаны на Рис. 7.12. В 22 двух случаях из 24 обе части соотношения были приведены через нормализацию суперкомпиляцией к одному и тому же синтаксическому виду.

Для монад Maybe и List остаточные программы, полученных при проверке 8-го соотношения, не совпали.

[1] В реальной библиотека языка Haskell реализации имеют несколько иной вид из-за использования типовых классов, в языке HLL мы не рассматриваем типовые классы, поэтому на рисунках приведены адаптации реализаций монад для языка HLL.

Рис. 7.12 Проверка корректности монад

	Maybe	List	State
(1)	+	+	+
(2)	+	+	+
(3)	+	+	+
(4)	+	+	+
(5)	+	+	+
(6)	+	+	+
(7)	+	+	+
(8)	-	-	+

Рассмотрим вначале монаду Maybe. При суперкомпиляции выражения `bind v mzero` получается:

```
case  v  of {
    Nothing  →  Nothing;
    Just s →  Nothing;
}
```

В то время как суперкомпиляция выражения `mzero` дает:

```
Nothing
```

Простой анализ показывает, что соотношение выполняется только для строгих `v` (вычисление `v` завершается). Достаточно просто находится пример, когда соотношение не выполняется. Например при `v` равном

```
letrec x = x in x
```

вычисление первого выражения зацикливается, в то время как результатом вычисления второго выражения всегда является `Nothing`.

Рассмотрим теперь монаду List. При суперкомпиляции выражения `bind v mzero` получается:

```
letrec f= λx →  case x of {
    Nil →  Nil;
    Cons y z →  (f z);
}
in f v
```

В то время как суперкомпиляция выражения `mzero` дает:

```
Nil
```

Аналогично показывается, что соотношение выполняется только для строгих `v` (вычисление `v` завершается и представляют конечный список). Достаточно просто находится пример, когда соотношение не выполняется. Например при `v` равном

```
letrec x = Cons Nil x in x
```

вычисление первого выражения зацикливается, в то время как результатом вычисления второго выражения всегда является `Nil`.

То, что 8-е соотношение выполняется только для строгих значений является известным фактом, который выявляется трансформационным анализом.

Стоит отметить, что даже в случае, когда остаточные программы при проверке эквивалентности выражений через нормализацию суперкомпиляцией не совпадают, анализ остаточных программ может дать полезную информацию.

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

7.5 Выводы

Мы показали, что можно доказывать эквивалентность выражений с помощью суперкомпиляции без использования предиката равенства, что делает такой метод применимым для языков высшего порядка с ленивой семантикой вычислений.

Глава 8

Метод многоуровневой суперкомпиляции

Концепция *метасистемного перехода* была описана В.Ф. Турчиным в его книге 1977 года “Феномен науки” [124]. В контексте информатики, Турчин формулировал (несколько упрощенно) главную идею метасистемного перехода следующим образом [125]:

> Расмотрим некоторую систему S произвольного типа. Пусть существует способ создавать ее копии, возможно с вариациями. Пусть эти системы объединены в новую систему S', для которой системы типа S являются подсистемами, и которая обладает некоторым механизмом наблюдения, контроля, модификации и воспроизведения подсистем S. Будем называть S' *метасистемой* по отношению к S и создание S' – *метасистемным переходом*. В результате последовательных метасистемных переходов возникает многоуровневая иерархия управления, демонстрирующая сложные формы поведения.

Проекции Футамуры [29] могут служить хорошим примером метасистемного перехода. Пусть p – программа, i – интерпретатор и s – специализатор программ. Тогда $s(i, p)$ можно рассматривать как результат компиляции программы p (“первая проекция”), $s(s, i)$ можно рассматривать как компилятор (“вторая проекция”) и $s(s, s)$ можно рассматривать как генератор компиляторов (“третья проекция”). (Ершов ссылается на вторую проекцию Футамуры как на “теорему Турчина о двойной прогонке” [28].)

Во второй проекции для вычисления $s(s,i)$ необходимо иметь две копии специализатора s, причем вторая копия "наблюдает и управляет" первой. В третьей проекции 3 копии s: третья управляет второй, которая управляет первой. Более того, как показано Робертом Глюком [30], можно рассматривать и четвертую проекцию, соответствующую следующему метасистемному переходу.

Проекции Футамуры – не единственный способ применить идею метасистемного перехода, комбинируя преобразователей программ. В данной главе рассматривается другой метод построения многоуровневой иерархии управления, используя *суперкомпиляторы* в качестве элементарных строительных блоков.

Известно, что классическая суперкомпиляция неудовлетворительно преобразует нелинейные (т.е. содержащие повторные переменные) выражения и функции с накапливающими параметрами. Мы покажем, как глубина преобразования суперкомпилятора может быть увеличена, если создать несколько копий классического суперкомпилятора, управляющих друг другом. Такой тип суперкомпиляции, основанный на комбинации нескольких метауровней будем называть многоуровневой суперкомпиляцией.

Метод, рассматриваемый в данной главе [64], является (концептуально) простым и модульным и основан на применении *улучшающих лемм* [96, 97], которые автоматически генерируются суперкомпилятором нижнего уровня для суперкомпилятора верхнего уровня.

8.1 Многоуровневая суперкомпиляция

Для краткости выкладок будем под суперкомпилятором HOSC понимать суперкомпилятор $\mathbf{SC}_{*-+}$ [1]. Схема HOSC в форме псевдокода показана на Рис. 8.1.

Рис. 8.1 Алгоритм базового суперкомпилятора

```
def scp(tree)
    b = unprocessed_leaf(tree)
    if b == null
        return makeProgram(tree)
    if trivial(b)
        return scp(drive(b, tree))
    a = ancestor(tree, b, renaming)
    if a != null
        return scp(fold(tree, a, b))
    a = ancestor(tree, b, instance)
    if a != null
        return scp(abstract(tree, b, a))
    a = ancestor(tree, b, whistle)
    if a == null
        return scp(drive(b, tree))
    return scp(abstract(tree, a, b))
```

Рис. 8.2 `even (double x Z)`: исходная программа

```
data Bool = True | False;
data Nat = Z | S Nat;

even (double x Z) where

even = λx →
    case x of {
        Z → True;
        S x1 → odd x1;
    };

odd = λx →
    case x of {
        Z → False;
        S x1 → even x1;
    };

double = λx y →
    case x of {
        Z → y;
        S x1 → double x1 (S (S y));
    };
```

Рис. 8.3 `even (double x Z)`: прогонка

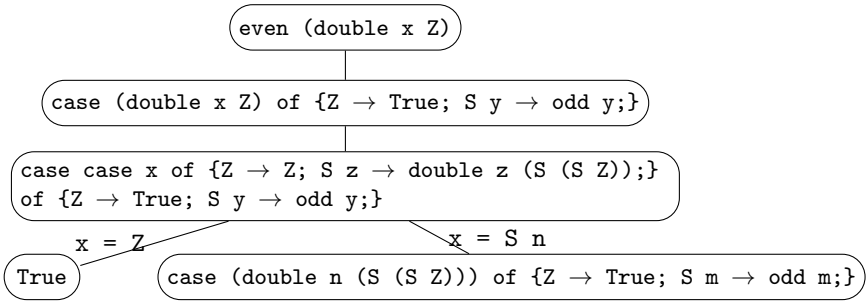


Рис. 8.4 The result of "zero-level" supercompilation

```
letrec
    f = λw2 p2 →
        case w2 of {
            Z →
                letrec g = λr2 →
                    case  r2  of {
                        S r →
                            case r of {
                                Z → False;
                                S z2 → g z2;
                            };
                        Z → True;
                    }
                in g p2;
            S z → f z (S (S p2));
        }
    in f x Z
```

8.1.1 Накапливающий параметр: базовая суперкомпиляция

Попробуем применить суперкомпилятор HOSC к программе, приведенной на Рис. 8.2. После первых шагов прогонки получается (незавершенное) дерево процессов, показанное на Рис. 8.3.

В этот момент срабатывает свисток, – в конфигурацию b вкладывается встреченная ранее конфигурация a, но конфигурация b не является подконфигурацией a:

[1] Использование суперкомпилятора $\mathbf{SC}_{*++}$ применительно к примерам данной главы дает те же самые результаты, но деревья процессов получаются намного объемнее.

Рис. 8.5 Результат применения леммы

```
letrec f = λt →
    case t of {
        Z → True;
        S s → f s;
    }
in f x
```

```
case double x Z of {Z  → True; S y → odd y);}
    ⊴_c^** case double n (S (S Z)) of {Z  → True; S m → odd m;}
```

Суперкомпилятору HOSC приходится удалить целое поддерево, расположенное ниже a и обобщить a, – заменить a на конфигурацию a', накрывающую a и b. В результате дальнейшей суперкомпиляции получается остаточная программа, приведенная на Рис. 8.4.

Результат суперкомпиляции корректен, но является неудовлетворительным. В целевом выражении `even (double x Z)` внутренняя функция `double` удваивает свой аргумент, а внешняя функция `even` проверяет, является ли результат четным числом. Таким образом, в результате вычисления целевого выражения не может получиться `False`. Это трудно увидеть, глядя на остаточную программу, поскольку в остаточной программе встречается `False`.

8.1.2 Накапливающий параметр: применение леммы

Как было показано Бёрсталлом и Дарлингтоном [17], возможности системы преобразования программ могут быть расширены, если ей при преобразованиях использовать леммы или алгебраические законы (такие, как ассоциативность и коммутативность сложения и умножения). Применительно к суперкомпиляции это сводится к замене конфигурации в дереве процессов на эквивалентную конфигурацию.

Вернемся к дереву процессов на Рис. 8.3. Результат суперкомпиляции получился неудовлетворительным, потому что суперкомпилятору HOSC пришлось обобщить конфигурацию в узле, что в итоге привело к потере точности преобразований. Но мы можем избежать обобщения, если применим попробуем использовать лемму об эквивалентности следующих

выражений:

```
case double n (S (S Z)) of {Z  →  True; S m →  odd m;} ≅
    ≅ even (double n Z)
```

Заменяем левую часть на правую. Конфигурация `even (double n Z)` теперь является переименованием уже рассмотренной ранее конфигурации, и суперкомпилятор может зациклить соответствующие узлы. В результате такой замены получается программа, показанная на Рис. 8.5. Отметим, что в этот раз в остаточной программе отсутствует конструкция `False`, то есть суперкомпилятору удалось показать, что в результате вычисления программы не может получиться `False`.

Таким образом, есть основания полагать, что полезно использовать такие леммы во время суперкомпиляции, но остаются вопросы: (1) как доказывать леммы и (2) как находить леммы.

Как было показано в предыдущей главе, доказывать леммы об эквивалентности выражений можно с помощью суперкомпиляции.

8.1.3 Соединение суперкомпиляторов, переход к многоуровневой суперкомпиляции

Как мы показали, 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 [124, 125], как следствие получается идея многоуровневой суперкомпиляции: построим башню суперкомпиляторов, – суперкомпиляторы верхнего уровня управляют суперкомпиляторами, расположенными ниже, чтобы получить (и затем применить) полезные леммы.

Это можно сделать, незначительно изменив схему суперкомпилятора: добавим новый параметр `n` – уровень суперкомпиляции – к функции `scp(tree)`, приведенной на Рис. 8.1. Измененный алгоритм суперкомпиляции приведен на Рис. 8.6.

Отметим, что при `n = 0` алгоритм вырождается в алгоритм “классической” суперкомпиляции. Но в случаях, когда `n > 0` активный суперкомпи-

Рис. 8.6 Алгоритм многоуровневой суперкомпиляции

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return makeProgram(tree)
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  if n > 0
    e = findEqExpr(b.expr, n)
    if e != null
      return scp(replace(tree, b, e), n)
  return scp(abstract(tree, a, b))

def findEqExpr(e1, n)
  e = scp(e1, n-1)
  cands = candidates(e1, n)
  for cand <- cands
    if equivalent(scp(cand, n-1), e)
      return cand
  return null

def candidates(e1, n)
  ...




  ...
```

лятор вызывает функцию `findEqExpr(b.expr,n)` и передает ей в качестве аргументов выражение в текущем узле `b` и активный уровень суперкомпиляции. Эта функция пытается выдать как результат выражение, эквива-

лентное выражению `b.expr`, генерируя множество выражений-кандидатов и выбирая из этого множества выражения, которые распознаны как эквивалентные выражению `b.expr`. Причем распознавание эквивалентных выражений осуществляется через вызов суперкомпилятора на один уровень ниже: `n - 1`.

8.1.4 Генерация множества остаточных программ

Алгоритм на Рис. 8.6 предполагает единственность результата суперкомпиляции. Однако, в процессе суперкомпиляции возникают ситуации, когда у суперкомпилятора есть возможность выбора из нескольких вариантов. Таким образом, если в таких ситуация мы не будем выбирать один-единственный вариант развития событий, а будем рассматривать несколько (или даже все множество) возможных вариантов, то для данной исходной программы можно получить несколько (эквивалентных) остаточных программ.

Это можно использовать для расширения возможностей распознавания эквивалентных выражений, основанного на нормализации суперкомпиляцией. Пусть требуется проверить эквивалентность двух выражений e_1 и e_2. Вместо генерации и сравнения только двух остаточных выражений, мы можем подвергнуть e_1 и e_2 суперкомпиляции, получить два множества остаточных выражений и попытаться найти выражение, принадлежащее обоим множествам.

Для осуществления этой идее нам необходим вариант суперкомпилятора, выдающий множество остаточных программ (см. Рис. 8.7). Этот вариант суперкомпилятора, вместо того, чтобы выбрать некоторое приемлемое выражение из множества кандидатов, рассматривает все подходящие выражения. Отметим, что при любом n множество остаточных программ включает результат базового суперкомпилятора ($n = 0$).

8.1.5 Несколько открытых вопросов

На Рис. 8.6 и Рис. 8.7 в схематичной форме представлена идея многоуровневой суперкомпиляции. Однако остается несколько неразрешенных вопросов:

Рис. 8.7 Алгоритм многоуровневой суперкомпиляции: несколько остаточных программ

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return [makeProgram(tree)]
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  progs = scp(abstract(tree, a, b))
  if n > 0
    for e <- findEqExpr(b.expr, n)
      progs ++= scp(replace(tree, b, e), n)
  return progs

def findEqExpr(e1, n)
  es = scp(e1, n-1)
  cands = candidates(e1, n)
  exps = []
  for cand <- cands
    if not_disjoint(scp(cand, n-1), es)
      exps = exps + cand
  return exps

def candidates(e1, n)
  ...



  ...
```

Корректность На первый взляд кажется, что замена выражения на эквивалентное ему выражения является естественным и безопасным по построению действием. И все же, как было показано Дэвидом Сэндсом

[96, 97], неограниченное использование эквивалентных выражений может привести к неправильным преобразованиям, не сохраняющим смысл программ.

Генерация выражений-кандидатов Суперкомпиляция может быть использована для распознавания эквивалентных выражений, но не позволяет простым образом найти достойных кандидатов для проверки на эквивалентность.

8.2 Корректность = эквивалентность + улучшение

Напомним, что мы используем $\mathcal{SC}[\![e]\!]$ для обозначения выражения, полученного в результате суперкомпиляции выражения e "классическим" "базовым" суперкомпилятором, запись $e \equiv e'$ означает совпадение выражения e с выражением e' (с точностью до переименования связанных переменных).

Далее предполагаем, что суперкомпилятор сохраняет операционную эквивалентность, то есть $e' = \mathcal{SC}[\![e]\!]$ предполагает $e' \cong \mathcal{SC}[\![e]\!]$ (что верно для суперкомпилятора HOSC)

Замена выражения e на эквивалентное ему выражение e' и последующая свертка конфигураций может привести к некорректной остаточной программе (некоторые примеры можно найти в [97]).

Как было показано Дэвидом Сэндсом [97], замена выражения e_1 на эквивалентное ему выражение e_2 не нарушает корректность преобразования, если выполняются следующие условия: $e_1 \cong e_2$ и $e_1 \trianglerighteq e_2$. То есть e_2 является строгим улучшением выражения e_1. Нам для удобства будем называть упорядоченную пару таких выражений улучшающей леммой.

Определение 121 (Улучшающая лемма)**.** Упорядоченная пара (e_1, e_2) является улучшающей леммой, если $e_1 \cong e_2$ и $e_1 \trianglerighteq e_2$.

8.2.1 Распознавание улучшающих лемм

Пусть с помощью суперкомпиляции доказана эквивалентность выражений e_1 и e_2. Этого недостаточно, чтобы одно из них было улучшением другого.

Суперкомпиляция следующих двух выражений (в контексте программы на Рис 8.10) показывает операционную эквивалентность:

```
or (even n) (odd n)
    ≅ case (even n) of {True → True; False → odd (S (S n));}
```

Однако, ни одно из них не является улучшением другого. Действительно, при `n = Z` вычисления выражений завершаются за 2 вызова и 1 вызов функций соответственно. Но при `n = S Z` вычисления завершаются за 5 и 6 вызовов функций соответственно. Следовательно, эту лемму небезопасно использовать при преобразовании программ.

К счастью, проверка того, что $e_1 \trianglerighteq e_2$ верно для выражений e_1 и e_2, таких, что $e_1 \cong e_2$, может быть осуществлена с помощью суперкомпиляции. Это можно сделать без специальных усилий следующим способом.

Чтобы проверить эквивалентность выражений e_1 и e_2, мы преобразуем методом суперкомпиляции эти выражения в эквивалентные им выражения e_1' и e_2'. Проверка эквивалентности завершается успешно, если e_1' и e_2' совпадают с точностью по переименования связанных переменных. Можно сказать, что в e_1' и e_2' не содержится информации, достаточной для того, чтобы сделать вывод о том, является ли выражение e_2 улучшением выражения e_1. Однако, в частичных деревьях процессов содержится больше информации, чем в преобразованных выражениях.

А именно: рассмотрим частичные деревья процессов и пометим звездочкой (*) дуги, соответствующие раскрытию вызова функции. Например, при суперкомпиляции рассмотренных выражений получаются деревья, приведенные на Рис. 8.12 и Рис. 8.14 (поддеревья для `odd x` совпадают и опущены для краткости). Помеченные дуги содержат некоторую информацию, которая может быть использована для проверки того, является ли одно из исходных выражений улучшением другого. Однако, эта информация недоступна извне (из остаточных выражений). Но в случае суперкомпилятора HOSC, можно добавить эту дополнительную информацию в остаточные выражения, слегка модифицировав алгоритм преобразования частичного дерева процессов в остаточное выражение.

Измененный алгоритм преобразует помеченные звездочками дуги в специальные аннотации (комментарии) в остаточных выражениях. При обходе помеченной дуги, остаточное выражение, полученное из (един-

Рис. 8.8 Распознавание улучшения, основанное на аннотировании остаточных выражений

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

ственного) дочернего узла аннотируется звездочкой (*). Таким образом, информация о раскрытии вызовов функций переносится в остаточное выражение, – суперкомпилятор более низкого уровня используется как "черный ящик".

Например, на Рис. 8.13 и Рис. 8.16 приведены проаннотированные программы, полученные из деревьев на Рис. 8.12 и Рис. 8.14.

Теперь, пусть $\rhd^*_\sim$ обозначает отношение на аннотированных выражениях: $e \rhd^*_\sim e'$, если (1) e и e' различаются только аннотациями и (2) после стирания некоторых аннотаций в выражении e получается выражение e'. Более формально это отношение определено на Рис. 8.8, где ${}^n\phi$ означает конструкцию языка (подвыражение) ϕ, помеченную n звездочками. (Стоит отметить, что отношение $\rhd^*_\sim$ можно рассматривать как частный случай отношения гомеоморфного вложения).

Теорема 122 (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* $e_1 \trianglerighteq_s e_2$.

Доказательство. Поскольку $e'_1 \equiv e'_2$, то e'_1 совпадает с e'_2 (с точностью до аннотаций и переименования связанных переменных). Следовательно, если мы поместим эти выражения в контекст C и попытаемся вычислить $C[e'_1]$ и $C[e'_2]$ (не обращая внимания на аннотации), мы получим одни и те же трассы вычислений, различающиеся только в количестве встреченных во время вычисления звездочек. Поскольку $e'_1 \rhd^*_\sim e'_2$, после любого числа шагов редукции, количество звездочек, встретившиеся при вычислении выражения $C[e'_2]$ не может превышать число звездочек, встреченных при вычислении выражения $C[e'_1]$, и эти звездочки соотносятся вызовам функций в исходных выражениях e_1 и e_2. Таким образом, если вычислять $C[e'_1]$ и $C[e'_2]$ и учитывать встретившиеся звездочки, мы можем посчитать количество вызовов функций, осуществленных при вы-

числении $C[e_1]$ и $C[e_2]$. Тогда, количество количество вызовов функций, осуществленных при вычислении $C[e_2]$ не больше, чем при вычислении $C[e_1]$. Следовательно, $e_1 \trianglerighteq e_2$. □

Таким образом, анализируя проаннотированный результат суперкомпиляции, мы можем распознавать улучшающие леммы (среди исходных выражений). Рассмотрим, к примеру, проаннотированные результаты суперкомпиляции выражений

```
or (even n) (odd n)
```

и

```
case (even n) of {True → True; False → odd (S (S n));},
```

показанные на Рис. 8.16 и Рис. 8.16 соответственно. Посколько проаннотированные остаточные выражения не соотносятся через $\trianglerighteq^*_{\sim}$, мы не можем заключить, что они соотносятся через $\trianglerighteq_{\sim_s}$.

8.3 Модельный двухуровневый суперкомпилятор

Хотя основная идея многоуровневой суперкомпиляции концептуально проста, остается ряд вопросов, требующих решения при практической реализации.

- Какой суперкомпилятор использовать в качестве первого приближения при реализации многоуровневого суперкомпилятора?
- Как обеспечить корректность суперкомпилятора?
- Как порождать полезные улучшающие леммы?
- Как обеспечить завершаемость многоуровневого суперкомпилятора?

Чтобы продемонстрировать потенциальные возможности многоуровневой суперкомпиляции, мы реализовали простой двухуровневый суперкомпилятор TLSC[2], незначительно изменив суперкомпилятор HOSC. Мы взяли HOSC в качестве первого приближения, так как (1) HOSC сохраняет семантику программ (включая их завершаемость) [61, 60], (2) способен доказывать леммы, в которых функции и бесконечные структуры данных находятся под квантором всеобщности и (3) выдает на выходе программу

[2]TLSC = Two-Level SuperCompiler

Рис. 8.9 Размер выражения

$\mathcal{S}[\![v]\!]$	$= 1$
$\mathcal{S}[\![c\ \overline{e_i}]\!]$	$= 1 + \sum_i \mathcal{S}[\![e_i]\!]$
$\mathcal{S}[\![\lambda v \to e]\!]$	$= 1 + \mathcal{S}[\![e]\!]$
$\mathcal{S}[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$	$= 1 + \mathcal{S}[\![e_0]\!] + \sum_i \mathcal{S}[\![e_i]\!]$
$\mathcal{S}[\![e_1\ e_2]\!]$	$= \mathcal{S}[\![e_1]\!] + \mathcal{S}[\![e_2]\!]$

в виде одного выражения, что делает возможным свести проверку эквивалентности программ к проверке эквивалентности выражений.

Корректность преобразований обеспечивается тем, что TLSC использует только улучшающие леммы. Отметим, что распознавание улучшающих лемм основано на суперкомпиляции “в первом приближении”, поэтому в TLSC реализована только двухуровневая (а не многоуровневая) иерархия суперкомпиляторов, состоящая из “верхнего” и “нижнего” суперкомпиляторов.

Наименее проработанными моментами остаются порождение полезных улучшающих лемм.

Сейчас порождение выражений-кандидатов реализовано довольно простым и грубым способом. Когда верхний суперкомпилятор обнаруживает конфигурацию e, в которую вкладывается ранее встреченная конфигурация, он порождает и рассматривает в качестве кандидатов все такие выражения e', чей размер (определенный на Рис. 8.9) меньше, чем размер выражения e. Затем для проверки, является ли пара выражений (e, e') улучшающей леммой, вызывается нижний суперкомпилятор, и если улучшающая лемма распознана, поиск лемм завершается.

Обеспечение завершаемости верхнего суперкомпилятора является само по себе интересной задачей, требующей дальнейших исследований. Завершаемость нижнего суперкомпилятора достигается с помощью проверки на гомеоморфное вложение и обобщения [104, 108, 59]. Однако, главная идея многоуровневой суперкомпиляции заключается в избежании обобщений. Когда обнаружено гомеоморфное вложение конфигураций, применение леммы дает возможность избежать обобщения и продолжать построение дерева процессов. И это может стать потенциальным источником незавершаемости.

Рис. 8.10 `or (even m) (odd m)`: исходная программ

```
data Bool = True | False;
data Nat = Z | S Nat;

or (even m) (odd m) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

or = λx y → case x of { True → True; False → y;};
```

Однако, с практической точки зрения, незавершаемости можно избежать, наложив некоторые ограничения на применения лемм. Простое решение состоит в следующем.

Пусть проверка на вложение обнаружила что конфигурация в верхнем узле a вложена в конфигурацию в нижнем узле b. После применения улучшающей леммы узел b заменяется на b' и суперкомпиляция продолжается без обобщения. Но факт применения леммы заносится (как дополнительная информация) в узел a. И в следующий раз, когда конфигурация в a вкладывается в другую конфигурацию, конфигурация в a будет обобщена, как и случае суперкомпилятора первого приближения.

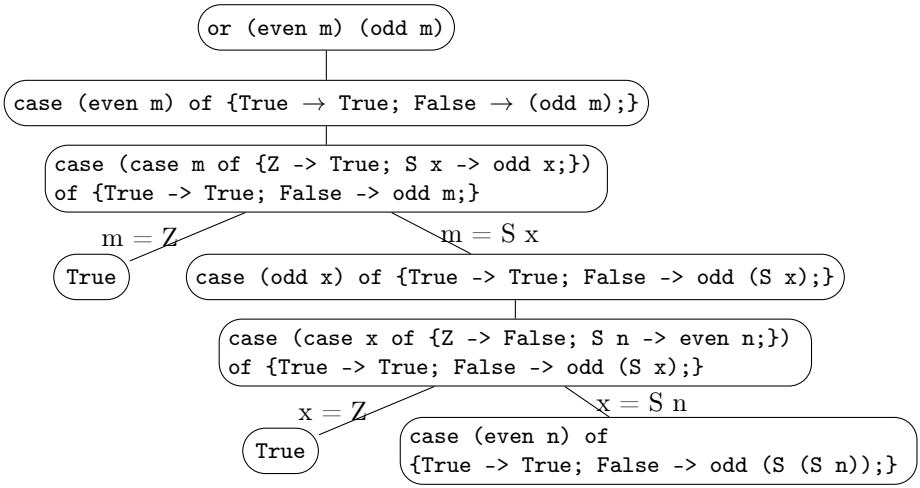
До некоторой степени это похоже на идею “перекрестного опыления” (“cross-fertilization”), используемой Бойером и Муром [16] в их доказывателе теорем. Они утверждают, что гипотеза индукции должна быть использована только в первый раз, а во второй раз необходимо делать обобщение.

8.4 Примеры

8.4.1 Суперкомпиляция нелинейного выражения

Расмотрим суперкомпиляцию программы, приведенной на Рис. 8.10. Заранее известно, что `False` никогда не может быть результатом вычисления выражения `or (even m) (odd m)`, поскольку натуральное число m либо четное, либо нечетное. Но этот факт неочевиден их текста исходной программы. После нескольких шагов прогонки получается дерево, приве-

Рис. 8.11 `or (even m) (odd m)`: обнаружение вложения



денное на Рис. 8.11. На этом шаге обнаруживается вложение:

```
case (even m) of {True → True; False → (odd m);}
    ⊴c** case (even n) of {True → True; False → (odd (S (S n)));}
```

Но второе выражение не является частным случаем первого, поэтому нельзя осуществить свертку конфигураций. Таким образом, суперкомпилятор первого приближения делает обобщение, заменяя первое выражение на `let`-выражение:

```
let x = m in case (even m) of {True → True; False → (odd x);}
```

Затем части `let`-выражения преобразуются отдельно друг от друга, таким образом игнорируется факт, что x и m – одно и то же. В результате такой потери информации об обобщенной конфигурации получается остаточная программа (Рис. 8.13), содержащая `False`, несмотря на то, что это выражение никогда не может быть результатом вычисления этой программы.

Однако, двухуровневый суперкомпилятор TLSC пытается найти и применить улучшающую лемму. Первое найденное эквивалентное выражение размера 5:

Рис. 8.12 `or (even m) (odd m)`: после обобщения

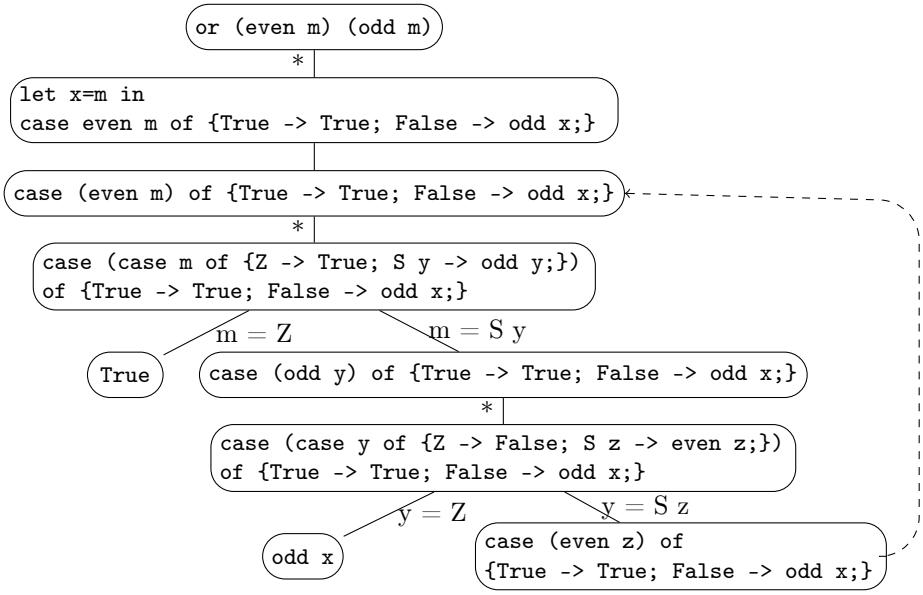
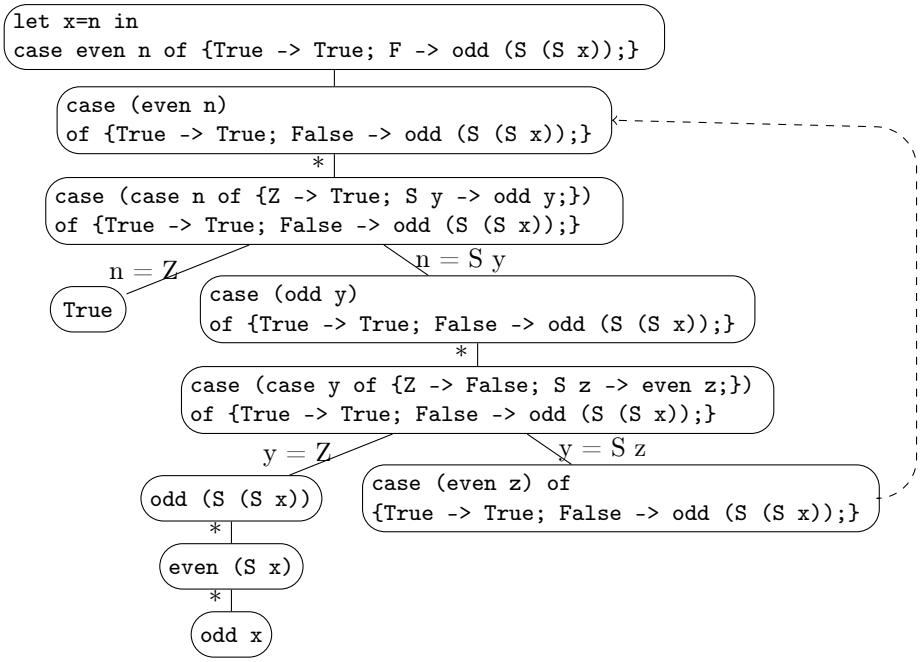


Рис. 8.13 `or (even m) (odd m)`: результат суперкомпиляции “в первом приближении”

```
*(letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          letrec g = *(λw→
            case w  of {
              Z  → False;
              S t → *(case t of {Z  → True; S z → g z;});})
           in g m;
        S x → f x;
      });
  })
in f m)
```

Рис. 8.14 `case even n of {True → True; False → odd (S (S n))}` – аннотированное частичное дерево процессов




```
case (even n) of {True → True; False → (odd (S (S n)));}
    ≅ or (even n) (odd n)
```

Однако, данная лемма не является улучшающей леммой, и двухуровневый суперкомпилятор после сравнения аннотированных остаточных выражений, полученных нижним суперкомпилятором (Рис. 8.16 и Рис. 8.13, соответственно), отвергает ее.

Оказывается, что существует две улучшающих леммы с выражениями размера 6:

```
case (even n) of {True → True; False → odd (S (S n));}
    ⊵ case (even n) of {True → True; False → odd n;}

case (even n) of {True → True; False → odd (S (S n));}
    ⊵ case (odd n) of {True  → odd n; False → True;}
```

Рис. 8.15 `case even n of {True → True; False → odd n}` – аннотированное частичное дерево процессов

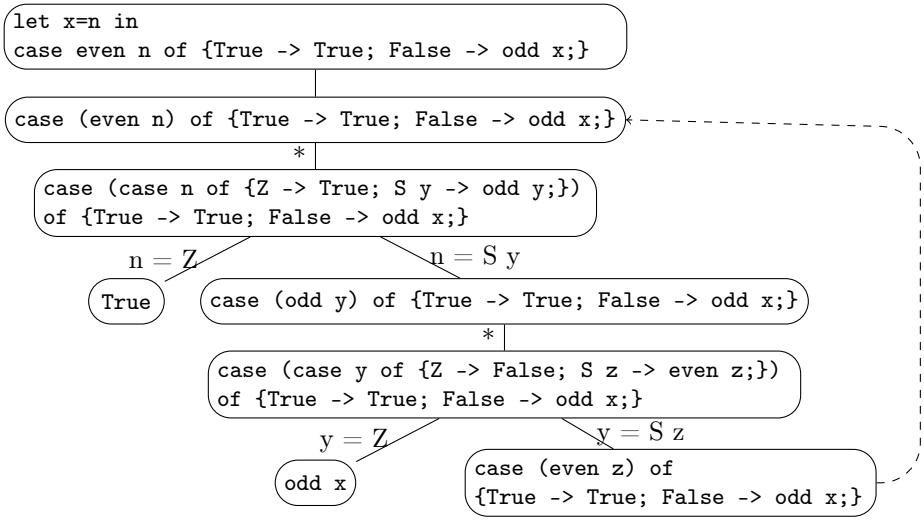


Рис. 8.16 `case even n of {True → True; False → odd (S (S n))}` – аннотированная остаточная программа

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          **(letrec g = *(λw→
            case w  of {
              Z  → False;
              S t → *(case t of {Z  → True; S z →  g z;});})
           in g n);
        S x →  f x;
      };)
  })
in f n
```

Рис. 8.17 `case even n of {True → True; False → odd n}` – аннотированная остаточная программа

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          (letrec g = *(λw→
            case w  of {
              Z  → False;
              S t → *(case t of {Z  → True; S z → g z;});})*
           in g n);
        S x → f x;
      };)
  })
in f n
```

Рис. 8.18 `or (even m) (odd m)`– результат двухуровневой суперкомпиляции

```
letrec f=λw→
  case w of {
    Z → True;
    S x → case x  of { Z  → True; S z → f z;};
  }
in f m
```

Двухуровневый суперкомпилятор распознает и использует первую из этих лемм, избегая таким образом обобщения. В итоге получается программа, показанная на Рис. 8.18. Теперь `False` не присутствует в остаточной программе.

8.4.2 Накапливающий параметр

Пересмотрим рассмотренную ранее программу на Рис. 8.2. При суперкомпиляции обнаруживается вложение следующих выражений:

```
case double x Z of {Z  → True; S y → odd y);}
    ⊴_c^**  case double n (S (S Z)) of {Z  → True; S m → odd m;}
```

Рис. 8.19 `even (double x Z)` – результат многоуровневой суперкомпиляции

```
case  x  of {
  Z → True;
  S y1 →
    letrec f=λt2→
      case  t2  of {Z  → True; S u2 → f u2;}
    in f y1;
}
```

Существуют две улучшающие леммы (минимального размера):

```
case double n (S (S Z)) of {Z  → True; S m →  odd m;}
       ⊵ case double n (S Z) of {Z  → True; S m →  even m;}
case double n (S (S Z)) of {Z  → True; S m →  odd m;}
       ⊵ case double n (S Z) of {Z  → False; S m →  even m;}
```

Двухуровневый суперкомпилятор TLSC распознает и использует первую из этих лемм. Затем, после нескольких шагов прогонки снова обнаруживается вложение:

```
case double n (S Z) of {Z  → True; S m →  even m;}
    ⊴c** case double p (S (S (S Z))) of {Z  → True; S m →  even m;}
```

Существуют две улучшающие леммы (минимального размера):

```
case double p (S (S (S Z))) of {Z  → True; S m →  even m;}
        ⊵ case double p (S Z) of {Z  → True; S m →  even m;}
case double p (S (S (S Z))) of {Z  → True; S m →  even m;}
        ⊵ case double p (S Z) of {Z  → False; S m →  even m;}
```

Применение первой леммы позволяет осуществить свертку конфигураций без обобщения, в результате двухуровневый суперкомпилятор TLSC выдает программу, приведенную на Рис. 8.19.

8.4.3 Улучшение асимптотики программ

В качестве демонстрации того, что многоуровневая суперкомпиляции способна улучшать асимптотику программ, рассмотрим преобразование наивного определение функции `reverse` в (более эффективную) функцию с накапливающим параметром.

Рис. 8.20 `append (reverse xs) ys`: исходная программа

```
data List a = Nil | Cons (List a)

append (reverse xs) ys where

append = λxs ys →
    case xs of {
        Nil → ys;
        Cons x1 xs1 → Cons x1 (append xs1 ys);
    };

reverse = λzs →
    case zs of {
        Nil → Nil;
        Cons z1 zs1 → append (reverse zs1) (Cons z1 Nil);
    };
```

Рассмотрим суперкомпиляцию программы, приведенную на Рис. 8.20.

После шага прогонки, мы получаем конфигурацию:

```
case reverse xs of
    Nil → ys                                    (1)
    Cons x3 x4 → Cons x3 (append x4 ys)
```

После еще одного шага прогонки:

```
case
    case xs of
        Nil  → Nil
        Cons x5 x6 → append (reverse x6) (Cons x5 Nil)   (2)
of
    Nil → ys
    Cons x3 x4 → Cons x3 (append x4 ys)
```

Теперь мы рассматриваем два случая: `xs` = `Nil` и `xs` = `Cons x5 x6`. В случае, когда `xs` = `Nil`, конфигурация сводится к `ys`, и во втором случае, преобразуется к:

```
case append (reverse x6) (Cons x5 Nil) of
    Nil → ys                                    (3)
    Cons x3 x4 → Cons x3 (append x4 ys)
```

Выражение (1) вложено в выражение (3), но (3) не является частным случаем (1). Поэтому суперкомпилятор HOSC обобщает (1). Однако

Рис. 8.21 `append (reverse xs) ys` – результат двухуровневой суперкомпиляции

```
data List a = Nil | Cons (List a)

letrec reverse1 = λxs1.λys1.
    case xs1 of
        Nil → ys1
        Cons x2 xs2 → reverse1 xs2 (Cons x2 ys1)
in
  reverse1 xs ys
```

двухуровневый суперкомпилятор TLSC может найти следующую улучшающую лемму:

```
append r (Cons p ps) =
case (append r (Cons p Nil)) of
    Nil  → ps
    Cons v vs → Cons v (append vs ps)
```
(4)

Применяя подстановку $\{$`r = reverse x6`, `p = x5`, `ps = ys`$\}$ к приведенной лемме, мы заменяем выражение (3) на улучшение

```
append (reverse x6) (Cons x5 ys)
```

Новая конфигурация является частным случаем начальной конфигурации `append (reverse xs) ys`. Происходит свертка конфигураций и в результате двухуровневой суперкомпиляции получается остаточная программа, приведенная на Рис. 8.21.

Следовательно, было показано, что

```
append (reverse xs) ys = reverse1 xs ys.
```

откуда следует:

```
reverse xs = append (reverse xs) Nil = reverse1 xs Nil.
```

Время вычисления исходной функции `reverse` было пропорционально квадрату длины списка, в то время как время вычисления преобразованной функции линейно по дли не списка. Таким образом, многоуровневая суперкомпиляция может улучшать асимптотику программ.

8.5 Выводы

Главная идея многоуровневой суперкомпиляции основана на принципе *метасистемного перехода* [116, 125].

Другой подход к расширению возможностей суперкомпиляции, основанный на принципе метасистемного перехода – *дистилляция* [39, 43, 41].

Во многих случаях дистилляция и многоуровневая суперкомпиляции показывают схожие результаты, но очевидным преимуществом многоуровневой суперкомпиляции является концептуальная простота и модульность: ее можно реализовать с помощью небольшой модификации "классического" суперкомпилятора, добавив (концептуально) тривиальный генератор лемм и обеспечив взаимодействие нескольких копий суперкомпилятора. Поскольку генератор лемм использует суперкомпилятор как "черный ящик", структура генератора лемм не зависит от деталей процесса суперкомпиляции.

Описанная реализация многоуровневой суперкомпиляции преследовала цель продемонстрировать возможности многоуровневой суперкомпиляции и может быть улучшена различными способами.

Во-первых, нынешний алгоритм многоуровневой суперкомпиляции (см. Рис. 8.6) применяет лемму к нижней конфигурации в целом. Но леммы могут применяться более тонко:

- Улучшающая лемма (или ее частный случай) может быть применена к подвыражению.
- Улучшающая лемма (или ее частный случай) может быть применена к верхней конфигурации (или ее составляющей).

Во-вторых, поиск лемм реализован примитивным способом: структура вкладывающихся конфигурации не учитывается. Однако, существуют методы, разработанные в области механического доказывания теорем по индукции (как сопоставление разностей [13], рипплинг [90] и критика расхождения [130]), которые могут быть использованы в более проработанном генераторе лемм.

Многоуровневый суперкомпилятор не зависит от незначительных деталей реализации суперкомпилятора "первого приближения", лежащего в его основе. Тем не менее, некоторые свойства суперкомпилятора имеют

значение. Прежде всего, распознавание улучшающих лемм [96, 97] полагается на то, что суперкомпилятор сохраняет свойства завершаемости программ. Не все суперкомпиляторы обладают таким свойством. Например, суперкомпилятор SCP4 [71], работающий с программами на языке Рефал, может расширять область область определения программ, поэтому доказательство эквивалентности выражений, основанное на нормализации суперкомпиляцией, применимо только для завершающихся выражений, оперирующих конечными данными [73].

При суперкомпиляции свойства завершаемости легче сохранить для языка с ленивой семантикой вычисления, нежели со строгой. Тем не менее, Джонссон добился успеха [49] в разработке сохраняющего семантику суперкомпилятора для языка высшего порядка со строгой семантикой вычислений. Таким образом, многоуровневая суперкомпиляция в принципе применима к языкам высшего порядка со строгой семантикой вычислений.

Поскольку любая остаточная программа, порожденная суперкомпилятором HOSC, является самодостаточным выражением, распознавание эквивалентности может сводится к тривиальному сравнению выражений. В случае суперкомпилятора наподобие Supero [78, 76], остаточная программа имеет менее тривиальную структуру, соответственно, сравнение остаточных программ становится более сложным, нежели в случае суперкомпилятора HOSC.

В принципе, многоуровневая суперкомпиляции может быть реализована на базе суперкомпилятора для императивного объектно-ориентированного языка, как например Jscp (суперкомпилятор для Java [58]), но остается ряд технических проблем для исследования.

Заключение

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 частно-

сти произведена проверка корректности реализации монад для ряда классов из стандартной библиотеки языка Haskell.

Литература

[1] *Abadi M., Cardelli L., Curien P.L.* Explicit substitutions // *Journal of functional programming.* — 1991. — Vol. 1, no. 4. — Pp. 375–416.

[2] *Abramov S.M.* Metacomputation and program testing // Proceedings of 1st International Workshop on Automated and Algorithmic Debugging. — 1993.

[3] *Abramov S.M., Glück R.* Semantics modifiers: an approach to non-standard semantics of programming languages // Third Fuji International Symposium on Functional and Logic Programming / Citeseer. — 1998.

[4] *Abramov S.M., Glück Robert.* Combining Semantics with Non-standard Interpreter Hierarchies // FST TCS 2000: Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science. — Vol. 1974 of *LNCS.* — Springer-Verlag, 2000. — Pp. 201–213.

[5] *Abramov S.M., Glück R.* The Universal Resolving Algorithm: Inverse Computation in a Functional Language // MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction. — Vol. 1837 of *LNCS.* — Springer-Verlag, 2000. — Pp. 187–212.

[6] *Abramov S.M., Glück R.* From standard to non-standard semantics by semantics modifiers // *International Journal of Foundations of Computer Science.* — 2001. — Vol. 12, no. 2. — Pp. 171–211.

[7] *Abramov S.M., Glück R.* Inverse Computation and the Universal Resolving Algorithm // *Wuhan University Journal of Natural Sciences.* — 2001. — Vol. 6, no. 1. — Pp. 31–45.

[8] *Abramov S.M., Glück R.* Principles of inverse computation and the universal resolving algorithm // The essence of computation. — Vol. 2566 of *LNCS*. — Springer, 2002. — Pp. 269–295.

[9] *Abramov S.M., Glück R., Klimov Y.* Faster Answers and Improved Termination in Inverse Computation of Non-Flat Languages. — 2003.

[10] *Albert E., Vidal G.* The narrowing-driven approach to functional logic program specialization // *New Generation Computing*. — 2002. — Vol. 20, no. 1. — Pp. 3–26.

[11] *Alpuente M., Falaschi M., Vidal G.* Partial evaluation of functional logic programs // *ACM Transactions on Programming Languages and Systems (TOPLAS)*. — 1998. — Vol. 20, no. 4. — Pp. 768–844.

[12] *Barendregt H.P.* The lambda calculus: its syntax and semantics. — North-Holland, 1984.

[13] *Basin D. A., Walsh T.* Difference Matching // CADE-11: Proceedings of the 11th International Conference on Automated Deduction. — Springer-Verlag, 1992. — Pp. 295–309.

[14] *Bird R., de Moor O.* Algebra of programming. — Prentice-Hall, Inc., 1997.

[15] *Bolingbroke M., Peyton Jones S.L.* Supercompilation by Evaluation // Haskell 2010 Symposium. — 2010.

[16] *Boyer R.S., Moore J.S.* Proving Theorems about LISP Functions // *Journal of the ACM (JACM)*. — 1975. — Vol. 22, no. 1. — Pp. 129–144.

[17] *Burstall R.M., Darlington J.* A Transformation System for Developing Recursive Programs // *Journal of the ACM (JACM)*. — 1977. — Vol. 24, no. 1. — Pp. 44–67.

[18] *Clarke E., Grumberg O., D. Peled.* Model Checking. — MIT Press, 1999.

[19] *Cockett R.* Deforestation, program transformation, and cut-elimination // *Electronic Notes in Theoretical Computer Science*. — 2001. — Vol. 44, no. 1. — Pp. 88–127.

[20] *Cousot P., Cousot R.* Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) // Proceedings of

the 1994 International Conference on Computer Languages, ICCL. — Vol. 94. — 1994. — Pp. 95–112.

[21] *Curry H.B., Feys R., Craig W.* Combinatory logic. — North-Holland, 1958. — Vol. 1.

[22] *Damas L., Milner R.* Principal type-schemes for functional programs // Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages / ACM New York, NY, USA. — 1982. — Pp. 207–212.

[23] *Danvy O.* An Extensional Characterization of Lambda-Lifting and Lambda-Dropping // FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming. — Vol. 1722 of *LNCS*. — Springer-Verlag, 1999. — Pp. 241–250.

[24] *Danvy O., Schultz U.P.* Lambda-dropping: transforming recursive equations into programs with block structure // *Theor. Comput. Sci.* — 2000. — Vol. 248, no. 1-2. — Pp. 243–287.

[25] *De Bruijn N.G.* Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem // *Indagationes Mathematicae.* — 1972. — Vol. 34. — Pp. 381–392.

[26] *Dershowitz N.* Termination of rewriting // *J. Symb. Comput.* — 1987. — Vol. 3, no. 1-2. — Pp. 69–116.

[27] *Dybjer P., Filinski A.* Normalization and Partial Evaluation // Applied Semantics. — Vol. 2395 of *Lecture Notes in Computer Science*. — Springer, 2002. — Pp. 137–192.

[28] *Ershov A.P.* On the essence of compilation // Formal Description of Programming Concepts / Ed. by E. Neuhold. — North-Holland, 1978. — Pp. 391–420.

[29] *Futamura Y.* Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler // *Systems, Computers, Controls.* — 1971. — Vol. 2, no. 5. — Pp. 45–50.

[30] *Glück R.* Is there a fourth Futamura projection? // PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation. — ACM, 2009. — Pp. 51–60.

[31] *Glück R., Jørgensen J.* Generating Transformers for Deforestation and Supercompilation // Static Analysis. — Vol. 864 of *LNCS*. — 1994.

[32] *Glück R., Jørgensen J.* Generating optimizing specializers // IEEE International Conference on Computer Languages. — IEEE Computer Society Press, 1994.

[33] *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. — London, UK: Springer-Verlag, 1993. — Pp. 112–123.

[34] *Glück R., Sørensen M.H.* Partial Deduction and Driving are Equivalent // PLIPL'94. — Vol. 844 of *LNCS*. — Springer-Verlag London, UK, 1994. — Pp. 165–181.

[35] *Glück R., Sørensen M.H.* A Roadmap to Metacomputation by Supercompilation // Selected Papers From the International Seminar on Partial Evaluation. — Vol. 1110 of *LNCS*. — 1996. — Pp. 137–160.

[36] *Glück R., Turchin V.F.* Experiments with a self-applicable supercompiler: Tech. Rep. 4: City University of New York, 1989.

[37] *Glück R., Turchin V.F.* Application of metasystem transition to function inversion and transformation // ISSAC '90: Proceedings of the international symposium on Symbolic and algebraic computation. — ACM, 1990. — Pp. 286–287.

[38] *Hamill Paul.* Unit test frameworks. — O'Reilly, 2004.

[39] *Hamilton G.W.* Distillation: extracting the essence of programs // Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation / ACM Press New York, NY, USA. — 2007. — Pp. 61–70.

[40] *Hamilton G.W.* Distilling Programs for Verification // *Electron. Notes Theor. Comput. Sci.* — 2007. — Vol. 190, no. 4. — Pp. 17–32.

[41] *Hamilton G.W.* Extracting the Essence of Distillation // Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, Psi 2009, Novosibirsk, Russia, June 15-19, 2009, Revised Papers. — Vol. 5947 of *LNCS*. — 2010.

[42] *Hamilton G.W.* A Graph-Based Definition of Distillation // Second International Workshop on Metacomputation in Russia. — 2010.

[43] *Hamilton G.W., Kabir M.H.* Constructing Programs From Metasystem Transition Proofs // Proceedings of the First International Workshop on Metacomputation in Russia. — 2008.

[44] *Hindley J.R.* The principal type-scheme of an object in combinatory logic // *Transactions of the American Mathematical Society.* — 1969. — Vol. 146. — Pp. 29–60.

[45] *Holst C.K., Hughes J.* Towards binding-time improvement for free // Functional Programming / Springer Verlag. — 1990. — P. 83.

[46] *Hudak P., Jones M.P.* Tech. Rep.: : Yale University, Department of Computer Science, 1994.

[47] *Johnsson T.* Lambda lifting: Transforming programs to recursive equations // Functional programming languages and computer architecture. — Vol. 201 of *LNCS*. — 1985. — Pp. 190–203.

[48] *Jones N.D.* The Essence of Program Transformation by Partial Evaluation and Driving // Logic, language and computation. — Vol. 792 of *LNCS*. — Springer-Verlag, 1994. — Pp. 62–79.

[49] *Jonsson P.A.* — Positive supercompilation for a higher-order call-by-value language. — Master's thesis, Luleå University of Technology, 2008.

[50] *Jonsson P.A., Nordlander J.* Positive Supercompilation for a higher order call-by-value language // IFL 2007. — 2007. — Pp. 441–456.

[51] *Jonsson P.A., Nordlander J.* Positive Supercompilation for a Higher Order Call-By-Value Language. Extended Proofs: Tech. Rep. 17: Luleå University of Technology, 2008.

[52] *Jonsson P.A., Nordlander J.* Positive Supercompilation for a higher order call-by-value language // *ACM SIGPLAN Notices.* — 2009. — Vol. 44, no. 1. — Pp. 277–288.

[53] *Jonsson P.A., Nordlander J.* Supercompiling Overloaded Functions. — submitted to ICFP 2009.

[54] *Jonsson P.A., Nordlander J.* Strengthening Supercompilation For Call-By-Value Languages // Second International Workshop on Metacomputation in Russia. — 2010.

[55] *King J.C.* Symbolic execution and program testing // *Commun. ACM.* — 1976. — Vol. 19, no. 7. — Pp. 385–394.

[56] *Klimov A.V.* A Program Specialization Relation Based on Supercompilation and its Properties // First International Workshop on Metacomputation in Russia. — 2008. — Pp. 54–77.

[57] *Klimov A.V.* An approach to Supercompilation for Object-Oriented Languages: the Java Supercompiler Case Study // First International Workshop on Metacomputation in Russia. — 2008.

[58] *Klimov A.V.* A Java Supercompiler and its Application to Verification of Cache-Coherence Protocols // Perspectives of Systems Informatics. — Vol. 5947 of *LNCS*. — 2010. — Pp. 185–192.

[59] *Klyuchnikov I.* Supercompiler HOSC 1.0: under the hood: Preprint 63. — Moscow: Keldysh Institute of Applied Mathematics, 2009.

[60] *Klyuchnikov I.* Supercompiler HOSC 1.1: proof of termination: Preprint 21. — Moscow: Keldysh Institute of Applied Mathematics, 2010.

[61] *Klyuchnikov I.* Supercompiler HOSC: proof of correctness: Preprint 31. — Moscow: Keldysh Institute of Applied Mathematics, 2010.

[62] *Klyuchnikov I., Romanenko S.* SPSC: a Simple Supercompiler in Scala // PU'09 (International Workshop on Program Understanding). — 2009.

[63] *Klyuchnikov I., Romanenko S.* Proving the Equivalence of Higher-Order Terms by Means of Supercompilation // Perspectives of Systems Informatics. — Vol. 5947 of *LNCS*. — 2010. — Pp. 193–205.

[64] *Klyuchnikov I., Romanenko S.* Towards Higher-Level Supercompilation // Second International Workshop on Metacomputation in Russia. — 2010.

[65] *Krustev D.* A Simple Supercompiler Formally Verified in Coq // Second International Workshop on Metacomputation in Russia. — 2010.

[66] *Lassez J.-L., Maher M.J., Marriott K.* Unification Revisited // Proceedings of the Workshop on Foundations of Logic and Functional Programming. — Vol. 306 of *LNCS*. — Springer-Verlag, 1988. — Pp. 67–113.

[67] *Leuschel M.* On the power of homeomorphic embedding for online termination // Static Analysis. — Vol. 1503 of *Lecture Notes in Computer Science*. — Springer, 1998. — Pp. 230–245.

[68] *Leuschel M.* Homeomorphic Embedding for Online Termination of Symbolic Methods // The essence of computation. — Vol. 2566 of *Lecture Notes in Computer Science*. — Springer, 2002. — Pp. 379–403.

[69] *Lisitsa A., Nemytykh A.* Towards Verification via Supercompilation // COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference. — IEEE Computer Society, 2005. — Pp. 9–10.

[70] *Lisitsa A., Nemytykh A.* A Note on Specialization of Interpreters // CSR '07: Proceedings of the 2nd international symposium on Computer Science in Russia. — Vol. 4649 of *LNCS*. — 2007. — Pp. 237–248.

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

[72] *Lisitsa A., Nemytykh A.* Reachability Analysis in Verification via Supercompilation // *International Journal of Foundations of Computer Science*. — 2008. — Vol. 19, no. 4. — Pp. 953–969.

[73] *Lisitsa A., Webster M.* Supercompilation for Equivalence Testing in Metamorphic Computer Viruses Detection // Proceedings of the First International Workshop on Metacomputation in Russia. — Ailamazyan University of Pereslavl, 2008. — Pp. 113–118.

[74] *Marlow S., Wadler P.* Deforestation for Higher-Order Functions // Proceedings of the 1992 Glasgow Workshop on Functional Programming. — Springer, 1992. — Pp. 154–165.

[75] *Mendel-Gleason G.E., Hamilton G.W.* Equivalence in Supercompilation and Normalisation By Evaluation // Second International Workshop on Metacomputation in Russia. — 2010.

[76] *Mitchell N.* Transformation and Analysis of Functional Programs: Ph.D. thesis / University of York. — 2008.

[77] *Mitchell N.* Rethinking Supercompilation // ICFP 2010. — 2010.

[78] *Mitchell N., Runciman C.* A Supercompiler for Core Haskell // Implementation and Application of Functional Languages. — Vol. 5083 of *LNCS*. — Springer-Verlag, 2008. — Pp. 147–164.

[79] *Monsuez B.* Polymorphic Typing for Call-by-Name Semantics // Proceedings of the International Conference on Formal Methods in Programming and Their Applications. — Springer-Verlag, 1993. — Pp. 156–169.

[80] *Nemytykh A.P.* Supercompiler SCP4: Use of quasi-distributive laws in program transformation // *Wuhan University journal of natural sciences*. — 2001. — Vol. 6, no. 1. — Pp. 375–382.

[81] *Nemytykh A.P.* A note on elimination of simplest recursions // ASIA-PEPM '02: Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation. — ACM, 2002. — Pp. 138–146.

[82] *Nemytykh A.P.* The Supercompiler SCP4: General Structure // PSI 2003. — Vol. 2890 of *LNCS*. — Springer, 2003. — Pp. 162–170.

[83] *Nemytykh A., Pinchuk V.* Program Transformation with Metasystem Transitions: Experiments with a Supercompiler // *LNCS*. — 1996. — Pp. 249–260.

[84] *Nemytykh A.P., Pinchuk V.A., Turchin V.F.* A Self-Applicable Supercompiler // Selected Papers from the Internaltional Seminar on Partial Evaluation. — Vol. 1110 of *LNCS*. — 1996. — Pp. 322–337.

[85] *Peyton Jones S.L.* An introduction to fully-lazy supercombinators // Combinators and Functional Programming Languages. — Vol. 242 of *LNCS*. — Springer, 1986. — Pp. 175–206.

[86] *Peyton Jones S.L.* The Implementation of Functional Programming Languages. — Prentice-Hall, Inc., 1987.

[87] *Pitts A.M.* Operationally-based theories of program equivalence // *Semantics and Logics of Computation*. — 1997. — Pp. 241–298.

[88] *Plotkin G.D.* Call-by-name, call-by-value and the λ-calculus // *Theoretical computer science*. — 1975. — Vol. 1, no. 2. — Pp. 125–159.

[89] *Reich J.S., Naylor M., Runciman C.* Supercompilation and the Reduceron // Proceedings of the Second International Workshop on Metacomputation in Russia. — 2010.

[90] Rippling: a heuristic for guiding inductive proofs / A. Bundy, A. Stevens, F. van Harmelen et al. // *Artif. Intell.* — 1993. — Vol. 62, no. 2. — Pp. 185–253.

[91] *Romanenko A. Y.* The generation of inverse functions in Refal // Partial evaluation and mixed computation: proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, Gammel Avernæs, Denmark, 18-24 October, 1987 / North Holland. — 1988. — P. 427.

[92] *Romanenko A. Y.* Inversion and metacomputation // PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. — ACM, 1991. — Pp. 12–22.

[93] *Romanenko S.A.* Higher-Order Functions as a Substitute for Partial Evaluation // Proceedings of the First International Workshop on Metacomputation in Russia. — Ailamazyan University of Pereslavl, 2008.

[94] *Rose K.H.* Explicit Substitution – Tutorial & Survey: Tech. Rep. LS-96-3: BRICS, 1996.

[95] *Sands D.* Operational theories of improvement in functional languages // Proceedings of the Fourth Glasgow Workshop on Functional Programming. — 1991.

[96] *Sands D.* Proving the correctness of recursion-based automatic program transformations // *Theoretical Computer Science.* — 1996. — Vol. 167, no. 1-2. — Pp. 193–233.

[97] *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.

[98] *Santos A.* Compilation by Transformation in Non-Strict Functional Languages: Ph.D. thesis / Glasgow University, Department of Computing Science. — 1995.

[99] *Schultz U.P.* — Implicit and explicit aspects of scope and block structure. — Master's thesis, DAIMI, Department of Computer Science,

University of Aarhus, 1997.

[100] *Secher J.P.* — Perfect Supercompilation. — Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[101] *Secher J.P.* Driving-Based program transformation in theory and practice: Ph.D. thesis / Department of Computer Science, Copenhagen University. — 2002.

[102] *Secher J.P., Sørensen M.H.* On Perfect Supercompilation // PSI '99. — Vol. 1755 of *LNCS*. — Springer-Verlag, 2000. — Pp. 113–127.

[103] *Sørensen M.H.* — Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. — Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.

[104] *Sørensen M.H., Glück R.* An algorithm of generalization in positive supercompilation // Logic Programming: The 1995 International Symposium / Ed. by J. W. Lloyd. — 1995. — Pp. 465–479.

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

[106] *Sørensen M.H., Glück R., Jones N.D.* Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC // Programming Languages and Systems. — Vol. 788 of *LNCS*. — 1994.

[107] *Sørensen M.H., Glück R., Jones N.D.* A Positive Supercompiler. // *Journal of Functional Programming*. — 1996. — Vol. 6, no. 6. — Pp. 811–838.

[108] *Sørensen M. H.* Convergence of Program Transformers in the Metric Space of Trees // Mathematics of Program Construction. — Vol. 1422 of *LNCS*. — 1998. — Pp. 315–337.

[109] *Stoughton A.* Substitution revisited // *Theor. Comput. Sci.* — 1988. — Vol. 59, no. 3. — Pp. 317–325.

[110] *The GHC Team.* Haskell 2010 Language Report. — `http://haskell.org/definition/haskell2010.pdf`. — 2010.

[111] *Tolmach A., Chevalier T., The GHC Team.* An External Representation for the GHC Core Language. — `http://www.haskell.org/ghc/docs/6.12.2/core.pdf`. — 2010.

[112] *Turchin V.F.* A supercompiler system based on the language REFAL // *SIGPLAN Not.* — 1979. — Vol. 14, no. 2. — Pp. 46–54.

[113] *Turchin V.F.* The Language Refal: The Theory of Compilation and Metasystem Analysis. — Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[114] *Turchin V.F.* Semantic definitions in REFAL and the automatic production of compilers // Semantics-Directed Compiler Generation, Proceedings of a Workshop. — Vol. 94 of *LNCS*. — Springer-Verlag, 1980. — Pp. 441–474.

[115] *Turchin V.F.* Program transformation by supercompilation // Programs as Data Objects. — Vol. 217 of *LNCS*. — Springer, 1986. — Pp. 257–281.

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

[117] *Turchin V.F.* A constructive interpretation of the full set theory // *Journal of Symbolic Logic*. — 1987. — Vol. 52, no. 1. — Pp. 172–201.

[118] *Turchin V.F.* The algorithm of generalization in the supercompiler // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop. — 1988.

[119] *Turchin V.F.* Program transformation with metasystem transitions // *Journal of Functional Programming*. — 1993. — Vol. 3, no. 03. — Pp. 283–313.

[120] *Turchin V.F.* On Generalization of Lists and Strings in Supercompilation: Tech. Rep. TR 95-002: City University of New York, 1996.

[121] *Turchin V.F.* Supercompilation: Techniques and Results // Perspectives of System Informatics / Springer. — Vol. 1181 of *LNCS*. — 1996.

[122] *Turchin V.F., Nemytykh A.P.* Metavariables: Their implementation and use in Program Transformation: Tech. Rep. TR 95-012: City College of the City University of New York, 1995.

[123] *Turchin V.F., Nirenberg R.M., Turchin D.V.* Experiments with a supercompiler // LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming. — ACM, 1982. — Pp. 47–55.

[124] *Turchin V. F.* The phenomen of science. A cybernetic approach to human evolution. — Columbia University Press, 1977.

[125] *Turchin V. F.* Metacomputation: Metasystem Transitions plus Supercompilation // Partial Evaluation. — Vol. 1110 of *Lecture Notes in Computer Science*. — Springer, 1996. — Pp. 481–509.

[126] *Turner D.A.* Total Functional Programming // *Journal of Universal Computer Science*. — 2004. — Vol. 10, no. 7. — Pp. 751–768.

[127] Verification of Parameterized Systems Using Logic Program Transformations / A. Roychoudhury, K.N. Kumar, C. R. Ramakrishnan et al. // TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems. — Springer-Verlag, 2000. — Pp. 172–187.

[128] *Wadler P.* Theorems for free! // FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture. — ACM New York, NY, USA, 1989. — Pp. 347–359.

[129] *Wadler P., Blott S.* How to make ad-hoc polymorphism less ad hoc // POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1989. — Pp. 60–76.

[130] *Walsh T.* A divergence critic for inductive proof // *J. Artif. Int. Res.* — 1996. — Vol. 4, no. 1. — Pp. 209–235.

[131] *Xu D.N., Peyton Jones S.L., Claessen K.* Static contract checking for Haskell // *SIGPLAN Not.* — 2009. — Vol. 44, no. 1. — Pp. 41–52.

[132] *Абрамов С.М.* Метавычисления и логическое программирование // *Программирование*. — 1991. — № 3.

[133] *Абрамов С.М.* Метавычисления и их применение. — Наука-Физматлит, 1995.

[134] *Абрамов С.М., Пармёнова Л.В.* Метавычисления и их применение. Суперкомпиляция. — Институт программных систем РАН, 2006.

[135] *Ключников И.Г., Романенко С.А.* SPSC: Суперкомпилятор на языке Scala // *Программные продукты и системы*. — 2009. — № 2.

[136] *Турчин В.Ф.* Метаязык для формального описания алгоритмических языков // Цифровая вычислительная техника и программирование. — 1966.

[137] *Турчин В.Ф.* Алгоритмический язык рекурсивных функций (РЕФАЛ): Препринт 4: ИПМ, 1968.

[138] *Турчин В.Ф.* Метаалгоритмический язык // *Кибернетика.* — 1968. — № 4.

[139] *Турчин В.Ф.* Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ // Теория языков и методы построения систем программирования. — Киев-Алушта: 1972.

[140] *Турчин В.Ф.* Эквивалентные преобразования программ на РЕФАЛе: Труды ЦНИПИАСС 6: ЦНИПИАСС, 1974.

[141] *Флоренцев С.Н., Олюнин В.Ю., Турчин В.Ф.* РЕФАЛ-интерпретатор // Тезисы 1-й Всесоюзной конференции по программированию. — Киев: 1968.