ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 77 за 2011 г.

Ключников И.Г., Романенко С.А.

MRSC: инструментарий для создания многорезультатных суперкомпиляторов

Рекомендуемая форма библиографической ссылки: Ключников И.Г., Романенко С.А. MRSC: инструментарий для создания многорезультатных суперкомпиляторов // Препринты ИПМ им. М.В.Келдыша. 2011. № 77. null с. URL: http://library.keldysh.ru/preprint.asp?id=2011-77

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

И.Г. Ключников, С.А. Романенко

MRSC: инструментарий для создания многорезультатных суперкомпиляторов

Москва
2011

Ilya G. Klyuchnikov, Sergei A. Romanenko. MRSC: a toolkit for building multi-result supercompilers

The paper explains the principles of multi-result supercompilation. We introduce a formalism for representing supercompilation algorithms as rewriting rules for graphs of configurations. Some low-level technical details related to the implementation of multi-result supercompilation in MRSC are discussed. In particular, we consider the advantages of using spaghetti stacks for representing graphs of configurations.

Supported by Russian Foundation for Basic Research project No. 09-01-00834-a.

Илья Ключников, Сергей Романенко. MRSC: инструментарий для создания многорезультатных суперкомпиляторов

В работе рассматриваются принципы многорезультатной суперкомпиляции. Вводится формализм для представления алгоритмов суперкомпиляции в виде правил переписываний графа конфигураций. Разбираются технические детали поддержки многорезультатной суперкомпиляции в инструментарии MRSC. Особое внимание уделяется представлению графов конфигураций в виде спагетти-стеков.

Работа выполнена при поддержке проекта РФФИ № 09-01-00834-а.

Содержание

1 Введение

1.1 Нарастающее многообразие в области суперкомпиляции

Суперкомпиляция – это метод преобразования программ, первоначально разработанный В.Ф. Турчиным для языка программирования Рефал (функциональный язык первого порядка с вызовом по имени) [36], и первые суперкомпиляторы разрабатывались и реализовывались для языка Рефал [34, 38, 25].

Это создавало впечатление, что суперкомпиляция – это весьма специфический метод анализа и преобразования программ, предназначенный для языка Рефал (и близких к нему языков).

Дальнейшее развитие суперкомпиляции привело к более абстрактной переформулировке суперкомпиляции, что позволило выяснить, какие части первоначальной формулировки были обусловлены спецификой языка Рефал, а какие – применимы и к другим языкам программирования [28, 32, 6]. В частности, была показана применимость суперкомпиляции для нефункциональных (императивных и объектно-ориентированных) языков программирования [9].

В результате было осознано различие между "суперкомпиляцией" и "суперкомпилятором". Суперкомпиляция – это общий *метод*, который применим к широкому классу языков программирования, а суперкомпилятор – это конкретный *преобразователь* программ, основанный на принципах суперкомпиляции. А при переходе от общей идеи суперкомпиляции к разработке конкретного суперкомпилятора требуется принимать целый ряд решений. А именно, мы должны сделать следующее.

- Выбрать *объектный язык программирования*, программы на котором будет обрабатывать суперкомпилятор. (Заметим, что выходной язык суперкомпилятора может и не совпадать с его входным языком, и в этом случае требуется выбрать также и выходной язык.)

- Для входного языка – выбрать некоторый вариант *операционной семантики*. Это необходимо вследствие того, что прогонка является формой "исполнения" программы в случае, когда обрабатываемые данные известны лишь частично, и вырождается в обычное исполнение программы, если данные полностью известны. Поэтому, прогонка является обобщением по отношению к конкретной форме операционной семантики.

- Разработать (или выбрать) *язык конфигураций* (для описания множеств состояний вычислительного процесса). Реализовать операции над конфигурациями (такие, как распознавание эквивалентности и вложенности множеств, изображаемых конфигурациями).

- Разработать алгоритм *прогонки* (обобщающий процесс вычислений над конкретными состояниями для вычислений над конфигурациями), в основе этого алгоритма прогонки лежит выбранный ранее вариант операционной семантики.

- Разработать (или выбрать) алгоритм, распознающий "опасные" (могущие оказаться бесконечными) ветви в дереве конфигураций, получающиеся в результате прогонки. В области суперкомпиляции такие алгоритмы традиционно называют *свистками*.

- Разработать (или выбрать) алгоритм *обобщения*, заменяющий конфигурации на более общие конфигурации (охватывающие больше состояний).

- Разработать алгоритм построения выходной (остаточной) программы на основе конечного графа конфигураций.

В последнее время, наряду с "классической" суперкомпиляцией начали появляться новые разновидности суперкомпиляции: такие как *дистилляция* [4, 5], *двухуровневая* суперкомпиляция [20, 16] и *многорезультатная* суперкомпиляция [21, 10]. Таким образом, в то время как на начальных этапах развития суперкомпиляции целью работы было создание конкретного *суперкомпилятора*, в настоящее время предметом исследования стали *методы построения* (многочисленных) суперкомпиляторов.

Этот очевидный всплеск разнообразия среди форм суперкомпиляции (как с точки зрения обрабатываемых языков, так и алгоритмов суперкомпиляции) можно рассматривать как проявление общего "закона разрастания предпоследнего уровня при метасистемном переходе" [33].

Также следует отметить, что хотя изначально суперкомпиляция рассматривалась В.Ф. Турчиным и как средство оптимизации, и как средство анализа программ [35], работы в области суперкомпиляции долгое время были, главным образом, посвящены применениям суперкомпиляции для оптимизации программ. В последнее время, однако, наблюдается возвращение интереса к применению суперкомпиляции в качестве средства выявления и доказательства свойств программ [22, 19, 11].

Таким образом, мы, вполне возможно, присутствуем при рождении таких направлений как языково-ориентированная (language-specific) и проблемно-ориентированная (domain-specific) суперкомпиляция. При этом подробности реализации общей идеи суперкомпиляции зависят как от обрабатываемого языка, так и от предполагаемой области применения суперкомпилятора.

В связи с этим меняется и сама парадигма исследований в области суперкомпиляции. До недавнего времени цель работы состояла в том, чтобы найти "наилучшую" комбинацию нескольких компонент, дающую "самый лучший" суперкомпилятор. Теперь же стало очевидно, что "наилучшего" суперкомпилятора просто-напросто не существует: что хорошо для одного языка или области применения, может не подходить для других языков и областей применения.

Возникает новая задача: систематическое исследование *разнообразных форм и методов* суперкомпиляции, а также их применимости *в различных областях*.

В качестве примера можно привести исследование, в котором изучалось влияние различных деталей алгоритма суперкомпиляции на его способность доказывать эквивалентности выражений высшего порядка [14]. При этом сравнивалось поведение 64-х вариантов суперкомпилятора.

1.2 Зачем нужен MRSC?

Очевидно, что для проведения подобных экспериментов возникает потребность в быстром изготовлении большого количества суперкомпиляторов или, по меньшей мере, большого количества модификаций имеющегося суперкомпилятора. Прежние, так сказать, "штучные" и "кустарные" методы изготовления суперкомпиляторов (при которых на изготовление одного суперкомпилятора могло уходить несколько лет), становятся в новых условиях неадекватными.

> Цель проекта MRSC – дать набор "строительных блоков" или, другими словами, "конструктор", дающий возможность быстро разрабатывать и реализовывать экспериментальные версии суперкомпиляторов.

1.3 Многорезультатность – основа MRSC

Термин "многорезультатная суперкомпиляция" (см. раздел 3) подразумевает, что для заданной входной программы суперкомпилятор может выдать не единственную программу, а некоторый (непустой) набор остаточных программ.

Инструментарий MRSC мог бы предоставлять для многорезультатной и однорезультатной суперкомпиляции две отдельные реализации. Однако, как выяснилось в процессе работы над MRSC, более простое решение получается, если реализовывать однорезультатную суперкомпиляцию через многорезультатную: берем самую первую остаточную программу, а все остальные – отбрасываем. Если остаточные программы генерируются и выдаются постепенно, по запросу (а не все сразу), такой подход является вполне приемлемым с точки зрения эффективности.

Имеются следующие аргументы в пользу того, чтобы в MRSC в качестве "основного" случая рассматривать именно многорезультатную суперкомпиляцию.

- Традиционная суперкомпиляция может рассматриваться в качестве частного случая многорезультатной суперкомпиляции. При этом появляется возможность описать различные виды суперкомпиляции единообразным способом. Например – с помощью набора правил переписывания для графов конфигураций (см. разделы 2 и 3).

- Как мы увидим позже, в случае многорезультатной суперкомпиляции можно, в значительной степени, ослабить зависимость различных частей суперкомпилятора друг от друга. В первую очередь – "развязать" свисток и алгоритм обобщения. Благодаря этому появляется возможность для сравнительного исследования "мощности" различных свистков (при фиксированном множестве возможных обобщений). В случае

же традиционной суперкомпиляции, изменения в свистке требуют внесения изменений и в алгоритм обобщения, в результате чего становится невозможным отделить друг от друга эффекты, возникающие из-за изменений в различных частях суперкомпилятора. Таким образом, многорезультатная суперкомпиляция дает дополнительные возможности для проведения сравнительных исследований в области суперкомпиляции.

- Наконец, уже в ходе разработки выяснилось, что "равнение" на многорезультатность позволяет сделать саму конструкцию более модульной, гибкой и декларативной.

Именно поэтому в данной работе делается акцент на многорезультатной суперкомпиляции. Соответственно, MRSC расшифровывается как Multi-Result SuperCompilation (toolkit).

1.4 Структура MRSC

У большинства существующих суперкомпиляторов имеются некоторые общие части, которые не зависят ни от обрабатываемого языка, ни от области применения суперкомпилятора. Например, общая структура графа конфигураций и реализация некоторых операций над ним не зависят от языка конфигураций. Или, например, различные виды свистков и алгоритмов обобщения могут быть сформулированы в абстрактном виде, без использования информации о деталях языка конфигураций.

Одна из целей MRSC – предоставить изготовителю суперкомпилятора некоторые из этих структур данных и операций над ними в виде готовых строительных блоков, позволяющих быстро разрабатывать и реализовывать прототипы суперкомпиляторов. С другой стороны, у клиента должна быть возможность создавать дополнительные компоненты и модифицировать логику работы имеющихся компонент.

Чтобы удовлетворить этим требованиям, в качестве языка реализации MRSC был выбран язык Scala [26] (хотя интересно было бы попробовать реализовать аналогичный инструментарий и в рамках других языков программирования).

Компоненты для конструирования суперкомпиляторов оформлены как трейты, из которых и собирается класс, реализующий основную часть суперкомпилятора – построитель графов конфигураций.

Код проекта MRSC находится в открытом доступе по адресу `https://github.com/ilya-klyuchnikov/mrsc`. Данная работа описывает MRSC версии 1.0.

1.5 Что описано в этом препринте

Ограничения на объем препринта не позволяют нам изложить в одном препринте о MRSC все, что нам хотелось бы: данный препринт открывает *серию*

работ, посвященных MRSC и многорезультатной суперкомпиляции.

Следует отметить, что при разработке инструментария, ориентированного на многорезультатность, выяснилось, что требуется более подробно рассмотреть (и даже в некоторых случаях пересмотреть) принципы конструирования самого нижнего уровня суперкомпилятора – представления графов конфигураций и реализации элементарных операций над ними. Это повлияло на дизайн MRSC, касающийся компонент нижнего уровня.

В данном препринте детально рассматривается только ядро MRSC и его "теоретические основы". А именно:

- Вводится формальное описание нескольких видов суперкомпиляции в форме правил переписывания графов конфигураций: традиционной (однорезультатной, детерминированной), недетерминированной (в виде отношения суперкомпиляции) и многорезультатной.

- Формулируются достаточные условия обеспечивающие конечность любого множества завершенных графов конфигураций, получающегося в результате многорезультатной суперкомпиляции.

- Подробно рассматривается внутреннее представление графов конфигураций в виде спагетти-стеков.

- Описывается метод порождения множеств завершенных графов конфигураций, каждое из которых является конечным (хотя, может быть, и очень большим).

Другие части MRSC, будут рассмотрены в следующих публикациях.

1.6 Структура препринта

Работа структурирована следующим образом:

- В разделе 2 вводится формализм для описания суперкомпиляции в виде правил переписывания графов конфигураций. Дается два набора правил переписывания: для традиционного алгоритма суперкомпиляции (соответствующего детерминированной суперкомпиляции) и для отношения суперкомпиляции (соответствующего недетерминированной суперкомпиляции). Сравнение этих двух наборов правил позволяет ясно увидеть ключевые различия между этими двумя видами суперкомпиляции. В случае традиционной суперкомпиляции правила переписывания обеспечивают генерацию единственного завершенного графа конфигураций, а в случае отношения суперкомпиляции можно построить (в общем случае) бесконечное множество завершенных графов конфигураций.

- В разделе 3 дается набор правил переписывания для случая многорезультатной суперкомпиляции. Эти правила обеспечивают порождение *конечных* множеств завершенных графов конфигураций. Сравнение этого набора правил с предыдущими наборами правил показывает, что многорезультатную суперкомпиляцию можно рассматривать как результат "скрещивания" между детерминированной (традиционной) суперкомпиляцией и недетерминированной суперкомпиляцией (описанной в виде отношения суперкомпиляции).

- В разделе 4 описывается ядро MRSC. Нижний уровень MRSC реализует несколько операций низкого уровня над графами конфигураций. MRSC предоставляет две структуры данных, обеспечивающих два представления для графов конфигураций: `TGraph` (на основе деревьев) и `SGraph` (на основе спагетти-стеков). Объясняется, почему структура данных `SGraph`, в случае многорезультатной суперкомпиляции, является более удобной в процессе построения графов конфигураций. MRSC предоставляет очень простой набор из 5-ти "шагов переписывания" соответствующих элементарным преобразованиям над графами конфигураций, а также абстракцию `GraphRewriteRules` для описания логики многорезультатной суперкомпиляции в виде правил переписывания. Компонента `GraphGenerator`, получив набор правил переписывания, инкрементно (в ответ на запросы) порождает все завершенные графы конфигураций.

- В разделе 5 описываются близкие работы и направления для дальнейших исследований.

Мы предполагаем, что читатель знаком с основами суперкомпиляции – прогонкой, свистком, обобщением, генерацией остаточной программы (работа [32] может служить в качестве хорошего введения в суперкомпиляцию).

2 Схемы традиционной суперкомпиляции

В настоящее время наибольшей популярностью пользуются два подхода к методам описания и реализации суперкомпиляторов.

В соответствии с первым подходом, суперкомпиляция выполняется путем построения графа конфигураций, который затем "резидуализируется" – превращается в "остаточную" программу, которая и является окончательным результатом работы суперкомпилятора [34, 28, 32, 18, 13, 5]. Этот подход соответствует первоначальному описанию суперкомпиляции, данному В.Ф. Турчиным [36].

В соответствии со вторым подходом [24, 23, 2, 7], суперкомпилятор – это преобразователь программ, который порождает выходные программы "напрямую", не создавая промежуточные структуры данных (графы конфигураций)[1]. Такой "прямолинейный" подход особенно хорошо работает, если су-

[1]По крайней мере, не делает это в явном виде

перкомпилятор пишется на ленивом языке (например, на языке Haskell) и при этом должен удовлетворять жестким требованиям по скорости его работы. Недостатком такого подхода, однако, является то, что компоненты суперкомпилятора оказываются слишком тесно связаны друг с другом. В случае же MRSC компоненты должны быть "отвязаны" друг от друга настолько, насколько это возможно.

Поэтому наше описание процесса суперкомпиляции и дизайн MRSC основаны на первом подходе (т.е. явном построении графов конфигураций[2]).

В дальнейших разделах даются абстрактные спецификации трех разновидностей суперкомпиляции: традиционной (детерминированной, одно-результатной), недетерминированной (в виде отношения суперкомпиляции) и многорезультатной.

Эти спецификации описывают процесс построения графов конфигураций абстрагируясь от особенностей конкретного языка программирования, будучи параметризованы по отношению к некоторому набору "элементарных" операций: прогонке, свертке, перестройке и свистку (подобно тому, как это сделано в работах Сёренсена [30, 31, 29]).

2.1 Правила переписывания графов конфигураций

В дальнейшем мы будем считать, что основной задачей суперкомпилятора является построение *завершенного* графа конфигураций, соответствующего программе p и стартовой конфигурации c. Работа начинается так: создается граф конфигураций, состоящий из одного (корневого) узла, в который помещается конфигурация c.

Затем происходит постепенное преобразование графа конфигураций путем применения правил переписывания графов, записанных в следующем виде:

$$\frac{precondition}{g \to g'}$$

Пусть g обозначает текущий граф, а g' – граф, который получается в результате одного шага переписывания. Шаг переписывания $g \to g'$ записывается снизу под горизонтальной чертой. Над горизонтальной чертой записывается условие $precondition$, которое должно быть выполнено для того, чтобы правило было применимо. Будем считать, что есть некоторый предикат $complete(g)$, определяющий, является ли данный граф конфигураций g завершенным.

Правила переписывания не являются взаимоисключающими. Это означает, что к некоторому графу может быть применимо несколько правил, одно правило или же ноль правил. Таким образом, в принципе, начальный граф конфигураций может быть переписан в любое число завершенных графов – от нуля до бесконечности.

[2]Вопрос о том, можно ли создать инструментарий подобный MRSC на основе "прямолинейного" подхода, весьма интересен, и требует дополнительного исследования.

Оказывается, что каждая из разновидностей суперкомпиляции (традиционная, недетерминированная и многорезультатная) может быть описана с помощью набора из трех правил переписывания: *Fold*, *Drive*, *Rebuild*. Легко видеть, что наборы правил, соответствующие разным видам суперкомпиляции, будучи весьма схожи, различаются в некоторых важных деталях, что в наглядной форме показывает различия между тремя видами суперкомпиляции.

2.2 Элементарные операции над графами

На Рис. 1 описан набор операций, с помощью которого можно описать работу суперкомпилятора "в общем виде". Конкретные определения этих операций могут быть различны для различных суперкомпиляторов. (Хорошим примером может служить описание "внутренностей" суперкомпилятора HOSC [13].) Эти операции естественным образом делятся на две группы: операции, которые преобразуют графы конфигураций (*fold*, *addChildren*, *rebuild*) и операции, которые "инспектируют" граф конфигураций (*foldable*, *dangerous*, *rebuilding*, *driveStep*, *rebuildings*).

В данный момент, при рассмотрении суперкомпиляции на высоком уровне абстракции, подробности устройства "инспектирующих" операций для нас несущественны: достаточно знать каковы типы результатов этих операций и как эти результаты используются. Также следует отметить, что для некоторых операций мы используем имена, которые отличаются от имен использующихся в статье Сёренсена: *addChildren* вместо *drive* и *rebuild* вместо *abstract* [30, 31, 29].

Имена операций *rebuild*, *rebuilding* and *rebuildings* заслуживают особого комментария. К сожалению, в области суперкомпиляции, термин *обобщение* является "перегруженным" (т.е. используется для обозначения нескольких понятий). Это можно проиллюстрировать следующими двумя цитатами.

Из работы Сёренсена [31]:

> Обратите внимание, что теперь мы используем термин "обобщение" в двух различных смыслах: для обозначения некоторых операций, выполняемых над деревьями во время суперкомпиляции, и для обозначения вышеупомянутых операций над выражениями. Эти два смысла взаимосвязаны: обобщение в первом смысле использует обобщение во втором смысле.

Из работы Турчина [37]:

> Редукция узла N_1 к узлу N_2 заключается в присвоении таких значений для $var(N_2)$ в терминах $var(N_1)$, что, после их подстановки, конфигурация в N_2 становится идентична конфигурации в N_1. Узел N_2 может быть либо (1) обобщением узла N_1 [...]. Переход по редукционной дуге не соответствует каким-либо шагам машины:

Операции, преобразующие граф конфигураций	
$fold(g,\ \beta,\ \alpha) : Graph$	Зацикливание: создание "обратной дуги" от текущего узла β к узлу α в графе конфигураций.
$addChildren(g,\ \beta,\ cs) : Graph$	Добавление новых узлов: для каждой конфигурации из списка cs создается узел, который становится потомком текущего узла β.
$rebuild(g,\ \beta,\ c') : Graph$	Перестройка графа: конфигурации в текущем узле β заменяется на конфигурацию c'.
$rollback(g,\ \alpha,\ c') : Graph$	Другой тип перестройки графа: конфигурации в узле α (не текущем) заменяется на конфигурацию c', а весь подграф, корнем которого является α, удаляется.
Операции, инспектирующие граф конфигураций	
$foldable(g,\ \beta,\ \alpha) : Bool$	Предикат, распознающий возможность свертки узла β к узлу α.
$dangerous(g,\ \beta) : Bool$	Свисток. Предикат, распознающий "потенциально опасную ситуацию" (ветвь в графе конфигураций, которая может оказаться бесконечной).
$complete(g) : Bool$	Предикат, распознающий завершенность графа конфигураций g.
$rebuilding(g,\ c) : C$	Перестройка конфигурации c, находящейся в текущем узле β, с учетом всего графа.
$driveStep(c) : List[C]$	Шаг прогонки. Выдает список конфигураций, получающихся из текущей конфигурации c.
$rebuildings(c) : List[C]$	Множество перестроек возможных для конфигурации c.

Рис. 1: Операции над графом конфигураций

> *состояние вычислительной машины остается в точности тем же самым; изменяется только представление этого состояния.*

С одной стороны, говорится, что c' является обобщением конфигурацииc, если $c \subset c'$ (что означает, что множество состояний, представляемое конфигурацией c' содержит множество, представляемое конфигурацией c). С другой стороны, предположим, что имеются следующие три конфигурации (выражения языка SLL [18]):

$f(Nil, g(y))$ (c_1)
$f(x, g(y))$ (c_2)
$let\ x = Nil\ in\ f(x, g(y))$ (c_3)

Здесь $c_1 \subset c_2$, т.е. конфигурация c_2 является обобщением конфигурации c_1. Заметим, что по конфигурации c_2 нельзя восстановить изначальную конфигурацию c_1, ибо информации для этого в c_2 недостаточно. Теперь предположим, что c_1 и c_2 находятся в графе конфигураций, и c_1 является текущим узлом. Тогда мы не можем выполнить обобщение, просто заменив c_1 на c_2! Вместо этого, c_1 сначала заменяется на конфигурацию c_3 (содержащую c_2 в качестве подвыражения). По этой причине иногда именно c_3, а не c_2 называют обобщением конфигурации c_1.

Такая перегруженность термина может приводить к путанице, и, чтобы её избежать, мы будем использовать более технический термин *перестройка* (который достаточно популярен в фольклоре суперкомпиляции), придав ему следующий смысл.

Перестройка конфигурации – это изменение представления конфигурации без изменения её смысла (в соответствии с приведенной выше цитатой из Турчина). При этом по перестроенной конфигурации можно однозначно восстановить исходную конфигурацию. Например, конфигурация c_3 является перестройкой конфигурации c_1. Языки, на которых записываются конфигурации, как правило, таковы, что любую конфигурацию можно перестроить только конечным числом способов.

Перестройка графа конфигураций снизу – это замена конфигурации c в текущем (активном) узле на конфигурацию c'.

Перестройка графа конфигураций сверху – это удаление из графа всех потомков узла α, после которого конфигурация c в α заменяется на конфигурацию c'.

В дальнейшем мы будем считать, что $rebuilding(g,\ c) \in rebuildings(c)$.

2.3 Схема традиционного алгоритма суперкомпиляции

Схема традиционной суперкомпиляции описывается SC-правилами, показанными на Рис. 2а. Детерминированность процесса определяется тем, что на каждом шаге применимо ровно одно правило (и при этом – единственным способом).

Эти правила можно истолковать как спецификацию следующего пошагового алгоритма:

- Пока граф конфигураций не является завершенным:
  - Если текущий узел свертываем к одному из узлов графа конфигураций, то сделать соответствующую свертку ($Fold$),
  - иначе, если текущее состояние графа распознано как опасное (сработал свисток), детерминированным образом найти перестройку текущей конфигурации (зависящую как от самой конфигурации,

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad dangerous(g, \beta) \quad c' = rebuilding(g, c)}{g \to rebuild(g, \beta, c')}$$

(a) SC: Детерминированная (традиционная) суперкомпиляция

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

(b) NDSC: Недетерминированная суперкомпиляция (отношение трансформации)

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

(c) MRSC: Многорезультатная суперкомпиляция

Обозначения и сокращения:
- g – текущий граф конфигураций
- β – активный лист графа конфигураций
- c – конфигурация, находящаяся в текущем узле β

Рис. 2: Схемы суперкомпиляции

так и от графа) и выполнить нижнюю перестройку графа конфигураций (*Rebuild*),

– в противном случае сделать шаг прогонки (*Drive*).

2.4 Схема отношения трансформации

Недетерминированная суперкомпиляция (отношение трансформации) не использует свисток, и, если не применимо правило $Fold$, может выполнять любые возможные перестройки.

Схема недетерминированной суперкомпиляции описывается NDSC-правилами, показанными на Рис. 2b. Технически, имеется два отличия от традиционной (детерминированной) суперкомпиляции:

1. Если активная конфигурация не является свертываемой к одной из предыдущих, то разрешается выполнять и перестройку, и шаг прогонки.

2. Перестройка текущей конфигурации является недетерминированной операцией, поскольку может быть использована любая конфигурация из множества $rebuildings(c)$).

Поскольку мы предполагаем, что $rebuilding(g, c) \in rebuildings(c)$, получается, что для любого заданного набора операций над графами отношение суперкомпиляции содержит в себе традиционную суперкомпиляцию. В том смысле, что если детерминированная суперкомпиляция порождает некоторую остаточную программу для заданной входной программы, то и недетерминированный суперкомпилятор тоже способен породить эту остаточную программу.

Для фиксированной входной программы, отношение суперкомпиляции определяет (в общем случае) бесконечное множество графов конфигураций (как завершенных, так и незавершенных).

Описания процесса суперкомпиляции в виде отношений трансформации удобно использовать для доказательства корректности алгоритмов суперкомпиляции или для формулировки некоторых “абстрактных” свойств суперкомпиляции [12, 15, 27].

3 Многорезультатная суперкомпиляция

В сущности, многорезультатная суперкомпиляция представляет собой результат “скрещивания” детерминированной (традиционной) суперкомпиляции с недетерминированной суперкомпиляцией (определенной через отношение трансформации).

3.1 Схема многорезультатной суперкомпиляции

Схема многорезультатной суперкомпиляции описывается MRSC-правилами, показанными на Рис. 2c.

Видно, что MRSC-правила представляют собой некоторую комбинацию SC-правил и NDSC-правил. Правило *Drive* заимствовано из SC-правил, а правило *Rebuild* – из NDSC-правил.

Заметим, что в случае SC-правил, свисток и перестройка тесно взаимосвязаны: когда срабатывает свисток, обязательно нужно выполнять перестройку, а если свисток молчит, то обязательно нужно выполнять прогонку, а перестройка – запрещена.

Однако, в случае MRSC-правил это не так, ибо перестройку разрешается выполнять даже тогда, когда свисток молчит. Тонкость при этом состоит в том, что может возникать ситуация, когда правило $Fold$ – неприменимо, правило $Drive$ – тоже неприменимо из-за того, что сработал свисток, а перестройка невыполнима из-за того, что множество $rebuildings(c)$ – пустое. Таким образом, получается, что процесс суперкомпиляции зашел в тупик, и с графом конфигураций приходится поступить по принципу "выкинуть и забыть".

Итак, в итоге получается следующее. Применяя SC-правила мы получаем единственный завершенный граф, применяя NDSC-правила – бесконечное (в общем случае) множество завершенных графов, а применяя MRSC-правила – конечное множество завершенных графов.

Теорема 1 (Конечность множества завершенных графов)**.** *Пусть*

1. *любая бесконечная ветвь в графе конфигураций распознаваема предикатом dangerous,*
2. *для любой конфигурации c множество перестроек $rebuildings(c)$ конечно,*
3. *число последовательных перестроек не может быть бесконечным (т.е. любая цепочка вида $c_1, c_2, c_3, \ldots$, где $c_{k+1} \in rebuildings(c_k)$ – конечна).*

Тогда при применении MRSC-правил получается конечное множество завершенных графов конфигураций.

Доказательство. Схлопываем последовательные перестройки в одну. Далее все следует из леммы Кёнига [8] (с применением рассуждений, аналогичных рассуждениям Сёренсена в его доказательстве завершаемости традиционной суперкомпиляции [29]). □

Аналогичным образом доказывается и то, что, применяя MRSC-правила, мы получаем конечное множество тупиковых графов конфигураций (к которым не применимо ни одно правило).

Может показаться, что третье условие в формулировке теоремы – лишнее. Однако, это не так. Предположим, что имеется суперкомпилятор для некоторого языка, такой, что (1) в его входном языке значениями переменных могут быть натуральные числ, и (2) в конфигурациях могут накладываться ограничения на значения переменных, имеющие вид $x < N$, где x – переменная, а N – натуральное число.

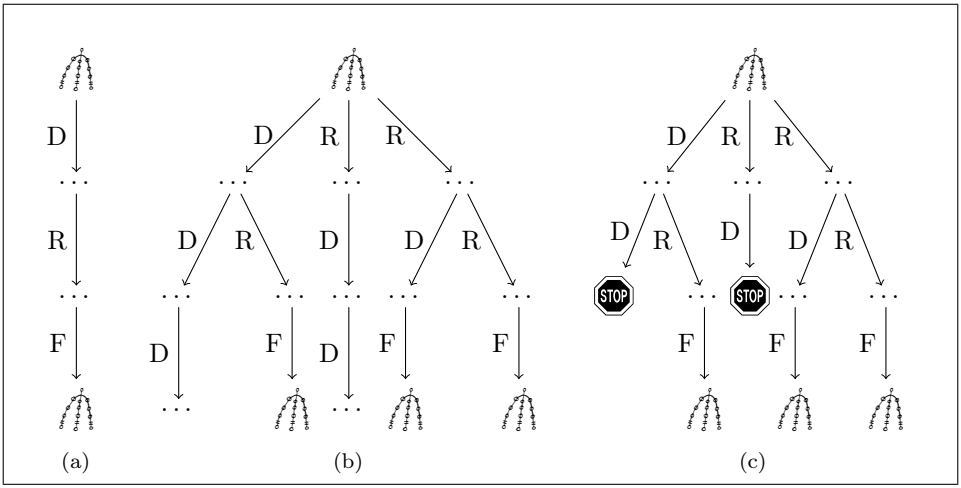


Рис. 3: Деревья графов. (a) Детерминированный алгоритм, (b) Отношение трансформации, (c) Многорезультатная суперкомпиляция.

Допустим, что конечность множества $rebuildings(c)$ обеспечена с помощью следующего правила: если все числовые константы в c не превышают N, то все числа, которые появляются в любой конфигурации $c' \in rebuildings(c)$ не превышают $N+1$. Тогда количество возможных перестроек конфигурации будет конечным, но количество последовательных перестроек может быть бесконечным. Например:

$$f(x)|_{\{x<5\}} \to let\ y = x|_{\{x<5\}}\ in\ f(y)|_{\{y<6\}} \to let\ z = y|_{\{y<6\}}\ in\ f(z)|_{\{z<7\}} \to \dots$$

Если $f(x)|_{\{x<5\}}$ – начальная конфигурация, то может быть порождено бесконечное множество завершенных графов конфигураций[3].

3.2 Дерево графов

Предположим, что нам задана некоторая начальная конфигурация. Тогда правила, показанные на Рис. 2, описывают процесс суперкомпиляции как последовательность шагов переписывания. Будем называть последовательность переписываний “успешной”, если она приводит к завершенному графу конфигураций, и “неуспешной” – если она заводит в тупик (т.е. приводит к графу, к которому нельзя применить ни одно из правил).

Отметим, что при этом (1) SC-правила определяют единственную, успешную и конечную последовательность шагов переписывания, (2) NDSC-правила определяют бесконечное дерево шагов переписывания, в котором

[3]Впрочем, в принципе может оказаться, что это бесконечное множество графов конфигураций порождает лишь конечное число различных остаточных программ.

имеются конечные успешные ветви, конечные неуспешные ветви и бесконечные ветви, а (3) MRSC-правила определяют бесконечное дерево, в котором имеются только конечные ветви, которые могут быть как успешными, так и неуспешными (см Рис. 3).

Таким образом, Теорема 1 может быть переформулирована так: дерево шагов переписывания, определяемое многорезультатной суперкомпиляцией, конечно.

3.3 Разделение свистка и обобщения

Теперь мы более внимательно займемся различиями между детерминированной (традиционной) суперкомпиляцией и многорезультатной суперкомпиляцией.

Сравнивая SC-правила и MRSC-правила, показанные на Рис. 2, мы видим, что все различия между SC- и MRSC-правилами сосредоточены в правиле *Rebuild*. В случае SC-правил, прогонка и обобщение (перестройка) взаимно исключают друг друга, а решение о том, что применять, прогонку или обобщение, принимается свистком. В случае же MRSC-правил, конфигурацию разрешается перестраивать даже если свисток безмолвствует. Таким образом, в MRSC-правилах свисток и алгоритм обобщения никак не связаны друг с другом: свисток не должен заботиться о том, чтобы конфигурация, которую он объявил "опасной", была обязательно "перестраиваемой.

То, что (по сравнению с традиционной суперкомпиляцией) многорезультатная суперкомпиляция позволяет ослабить зависимость частей суперкомпилятора друг от друга, имеет особое значение при выполнении исследовательской работы в области суперкомпиляции. Поскольку свисток больше не должен заботиться о проблемах, связанных с обобщениями/перестройками, становится легче изучать свойства разнообразных, необычных свистков. С другой стороны, алгоритм обобщения больше не обязан пытаться угадывать "наилучший" вариант обобщения: от него требуется только сгенерировать некоторое конечное множество возможных обобщений $rebuildings(c)$.

С практической точки зрения, даже если множество $rebuildings(c)$ заведомо конечно, его размер может быть весьма значительным, что, в свою очередь, может приводить к генерации огромного количества остаточных программ. Однако, это вполне приемлемо, если мы занимаемся исследованием вопроса о том, может ли какой-то свисток давать хорошие результаты *в принципе*? Если ответ – положительный, мы можем переходить к решению следующей задачи: как уменьшить размер множества $rebuildings(c)$, включая в него только "разумные" обобщения.

3.4 Многорезультатная суперкомпиляция как разрастание предпоследнего уровня

Идея многорезультатной суперкомпиляции проста. То, что она, до недавнего времени, не была сформулирована в явном виде, объясняется двумя причинами.

Во-первых, основным применением суперкомпиляции считалась оптимизация. Поэтому предполагалось, что суперкомпилятор должен выдавать единственный вариант остаточной программы (тот, который "лучше всех"). Однако, при применении суперкомпиляции для анализа программ, оказывается, что объяснить суперкомпилятору, какую из остаточных программ следует считать "наилучшей" – не так-то просто. Отсюда и возникает идея: пусть суперкомпилятор выдает несколько остаточных программ.

Во-вторых, возможности многорезультатной суперкомпиляции раскрываются в полной мере только когда она применяется в сочетании с суперкомпиляцией высшего уровня (в особенности, в комбинации с двухуровневой суперкомпиляцией). В то время как в случае традиционной (одноуровневой) суперкомпиляции переход от однорезультатной суперкомпиляции к многорезультатной приводит к некоторым количественным улучшениям результатов, сочетание двухуровневой суперкомпиляции с многорезультатной суперкомпиляцией приводит к получению качественно новых результатов [21].

4 Ядро MRSC

Рассмотрим теперь, какие технические проблемы возникают при разработке многорезультатного суперкомпилятора, и как эти проблемы решены в инструментарии MRSC.

Самой сложной из задач, решаемых суперкомпилятором, является построение графов конфигураций. Суперкомпилятор строит граф конфигураций сверху вниз, начиная со стартовой конфигурации. В случае традиционной однорезультатной суперкомпиляции, когда строится ровно один граф конфигураций, детали его внутреннего представления особой роли не играют. Можно использовать изменяемые структуры данных, и, в императивном стиле, постепенно модифицировать имеющийся экземпляр графа [37]. А можно использовать неизменяемые структуры данных: тогда, если суперкомпилятор написан на "строгом" языке (с вызовом параметров по значению), на каждом шаге работы будет порождаться новая структура данных [18]. Если же суперкомпилятор написан на "ленивом" языке, графы конфигураций могут конструироваться "лениво" [17].

В любом случае, насколько можно видеть из литературы, те суперкомпиляторы, которые строят графы конфигураций в явном виде, обычно используют представление графов на основе деревьев (растущих от корня к листьям). Такое представление удобно при генерации остаточных программ,

```
type TPath = List[Int]
type SPath = List[Int]

case class TNode[C, D](
  conf: C, outs: List[TEdge[C, D]],
  base: Option[TPath], tPath: TPath)

case class TEdge[C, D](
  node: TNode[C, D], driveInfo: D)

case class TGraph[C, D](
  root: TNode[C, D], leaves: List[TNode[C, D]])

case class SNode[C, D](
  conf: C, in: SEdge[C, D],
  base: Option[SPath], sPath: SPath)

case class SEdge[C, D](
  node: SNode[C, D], driveInfo: D)

case class SGraph[C, D](
  incompleteLeaves: List[SNode[C, D]],
  completeLeaves: List[SNode[C, D]],
  completeNodes: List[SNode[C, D]]) {

  val isComplete = incompleteLeaves.isEmpty
  val current = if (isComplete) null else incompleteLeaves.head
}
```

Рис. 4: Графы

поскольку, традиционно, остаточная программа строится в результате обхода графа конфигураций сверху вниз[4]).

Однако, как мы увидим в следующем разделе, в случае многорезультатной суперкомпиляции представление графа конфигураций на основе дерева – крайне неудобно. Поэтому в MRSC используется другое представление графов конфигураций, на основе *спагетти-стеков* [1].

4.1 Две структуры данных для представления графа конфигураций

Для работы с графами в MRSC используется два представления: T-представление (на основе деревьев, T = tree) и S-представление (на основе “спагетти-

[4]Было бы крайне интересно найти способ генерации остаточной программы путем обхода графа снизу вверх.

стеков", S = stack) [1]. Реализация этих представлений на языке Scala показана на Рис. 4.

T-представление используется при преобразовании графа в остаточную программу. S-представление используется при пошаговом построении графов конфигураций. Когда построение графа закончено, он может использоваться непосредственно (в S-представлении), либо может быть преобразован из S-представления в T-представление (для последующей резидуализации).

Графы в T-представлении являются объектами класса `TGraph[C, D]`, содержащими информацию следующих видов:

1. `C` (configuration) – конфигурации, находящиеся в узлах графа.

2. `D` (driving info) – информация на дугах графа, которая описывает, как одна конфигурация "эволюционирует"в другую (например, транзитный шаг редукции, разбор вариантов, декомпозиция). Эта информация полезна при генерации остаточной программы.

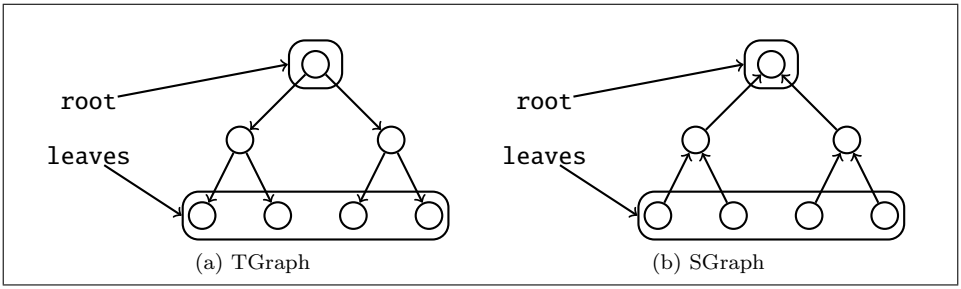
Каждый узел T-графа представлен объектом класса `TNode[C, D]`, в котором содержится некоторая конфигурация и список исходящих из него дуг. Также в узле хранится путь до этого узла от корня графа, что сильно облегчает манипуляции с графом. Путь к узлу, в частности, может использоваться в качестве уникального идентификатора этого узла внутри графа. Информация о зацикливании (свертке) представлена в виде (опционального) пути к базовому узлу. Таким образом, `TGraph` фактически является деревом, в котором к некоторым листьям добавлена информация о зацикливаниях (свертках).

Дуги T-графа представлены объектами класса `TEdge[C, D]`. Дуги являются ориентированными, и каждая дуга хранит информацию только о том узле, в который она направлена.

`TGraph[C, D]` хранит информацию о своём корневом узле ("входной точке"), а также информацию о листьях дерева (которую удобно иметь при резидуализации, т.е. генерации остаточной программы).

Как было сказано выше, T-представление удобно для обхода графов конфигураций сверху вниз. Однако, если требуется что-то добавить к T-графу двумя разными способами, приходится копировать какие-то части графа. А в случае многорезультатной суперкомпиляции как раз приходится делать "расходящиеся в стороны" добавления к графу чуть ли не на каждом шагу. Таким образом, оказывается, что работать с T-графами во время многорезультатной суперкомпиляции – неудобно. Более удобное представление мы получаем перевернув T-граф "вверх тормашками", в результате чего получается S-граф, представленный объектом класса `SGraph[C,D]`.

Объекты класса `SGraph[C,D]` являются "двойственными" по отношению к объектам класса `TGraph[C,D]`. Схематически эти две структуры данных показаны на Рис. 5.

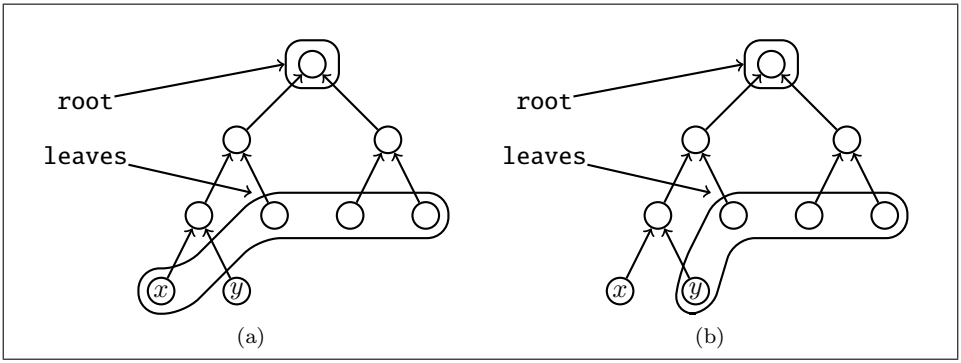


(a) TGraph (b) SGraph

Рис. 5: Структуры данных MRSC

Обе структуры данных являются неизменяемыми. При этом именно S-представление позволяет реализовать достраивание графов в функциональном стиле. Рассмотрим этот аспект более подробно.

Допустим, что к графу, показанному на Рис. 5, можно применить два разных шага переписывания: либо подвесить к левому листу узел с конфигурацией x, либо подвесить к левому листу узел с конфигурацией y. В случае S-графа, для создания двух новых графов достаточно создать два новых узла и повторно использовать узлы, уже имеющиеся в исходном графе! Такое совместное использование узлов схематически показано на Рис. 6.



(a) (b)

Рис. 6: Совместное использование узлов в S-представлении

Также следует отметить, что и для реализации свистков S-представление удобнее, чем T-представление, поскольку большинство свистков находит и просматривает узлы дерева, двигаясь снизу вверх, начиная от текущего узла.

Несмотря на эти преимущества S-представления, многие суперкомпиляторы (в силу сложившихся традиций?) при построении графов конфигураций всё же используют T-представление.

Таким образом, представлением строящегося графа конфигураций является объект класса `SGraph[C, D]`, в котором `current` указывает на текущий узел, `incompleteLeaves` – список ещё не обработанных листьев, а

`completeLeaves` – список обработанных листьев. Также имеется дополнительный список `completeNodes`, который содержит все обработанные узлы графа.

4.2 Базис операций над графами в S-представлении

Одна из основных целей MRSC – позволить программисту сосредоточиться на описании *логики* (многорезультатного) суперкомпилятора, освободив его от забот, связанных с реализацией рутинных операций. По сути дела, "нижний этаж" суперкомпилятора – это набор правил переписывания для графов конфигураций. MRSC позволяет описывать эти правила полудекларативным образом.

MRSC предоставляет базис из пяти "шагов", обозначающих элементарные операции переписывания S-графов. Этот базис схематично показан на Рис. 7.

Каждый из "шагов" преобразования изображается в виде значения (неизменяемого объекта языка Scala) типа `GraphStep[C, D]`, и может быть применен к *текущему* графу конфигураций типа `SGraph[C, D]` (Рис. 8):

1. `CompleteCurrentNodeStep` – превращает текущий узел в обработанный лист графа. Применяется при прогонке.
2. `FoldStep` – делает свертку.
3. `AddChildNodesStep` – добавляет дочерние узлы к текущему узлу. Применяется при прогонке.
4. `RebuildStep` – выполняет перестройку снизу (заменяя конфигурацию в текущем узле).
5. `RollbackStep` – выполняет перестройку сверху (удаляя поддерево, растущее из указанного узла).

Любой граф конфигураций, который может возникнуть в процессе суперкомпиляции, может быть получен в результате применения некоторой последовательности вышеупомянутых элементарных шагов. Эти шаги исполняются с помощью интерпретатора, предоставляемого MRSC в качестве части генератора графов (см. ниже). Суперкомпиляторы, реализованные с помощью MRSC, никогда не преобразуют графы конфигураций напрямую: вместо этого они генерируют шаги, которые интерпретируются внутри генератора графов. Это, в определенной степени, обеспечивает корректность преобразований, выполняемых над графами конфигураций.

Отметим, что использование S-представления графов позволяет реализовать операцию отката (rollback) в элегантном функциональном стиле (см. исходный код MRSC)[5].

[5] В работе [3] откаты реализованы через механизм исключений.

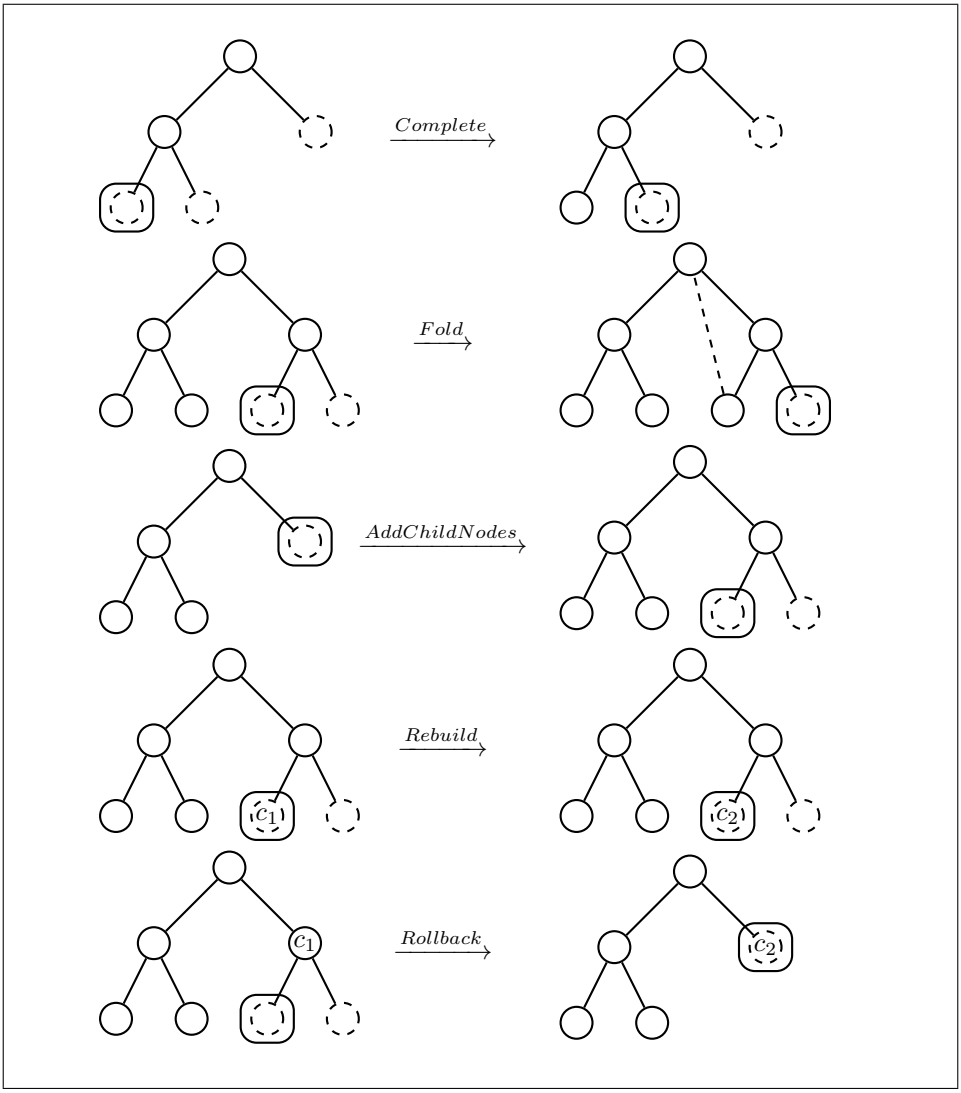


Рис. 7: Базис операций с графами конфигураций

Удобство того, что шаги переписывания представляются значениями первого порядка, заключается в том, что их легко сериализовывать и сохранять для последующего использования. Затем они могут быть поданы на вход другого программного средства, такого как, например, “валидатор” последовательностей шагов построения. Получив на входе начальный граф (из одного узла) и последовательность шагов переписывания, валидатор может проверять, могла ли эта последовательность шагов быть порождена каким-то су-

```
sealed trait GraphRewriteStep[C, D]

case class CompleteCurrentNodeStep[C, D]
  extends GraphRewriteStep[C, D]

case class AddChildNodesStep[C, D](ns: List[(C, D)])
  extends GraphRewriteStep[C, D]

case class FoldStep[C, D](to: SPath)
  extends GraphRewriteStep[C, D]

case class RebuildStep[C, D](c: C)
  extends GraphRewriteStep[C, D]

case class RollbackStep[C, D](to: SPath, c: C)
  extends GraphRewriteStep[C, D]
```

Рис. 8: Шаги переписывания для S-графов

```
trait GraphRewriteRules[C, D] {
  type N = SNode[C, D]
  type G = SGraph[C, D]
  type S = GraphRewriteStep[C, D]
  def steps(g: G): List[S]
}

case class GraphGenerator[C, D]
  (rules: GraphRewriteRules[C, D], conf: C)
  extends Iterator[SGraph[C, D]] { ... }
```

Рис. 9: “Строительный раствор” предоставляемый MRSC для сборки суперкомпиляторов

перкомпилятором (или удовлетворяет ли она отношению суперкомпиляции).

4.3 Порождение графов конфигураций

Технически, суперкомпилятор, созданный с помощью MRSC, основывается на двух компонентах, показанных на Рис. 9: `GraphRewriteRules` и `GraphGenerator`.

Трейт `GraphRewriteRules` описывает логику многорезультатного суперкомпилятора в форме, близкой к той, что представлена на Рис. 2c. Этот трейт содержит объявление метода `steps`. И каждый суперкомпилятор должен дать своё определение этого метода. Другими словами, трейт

`GraphRewriteRules` определяет только интерфейс, через который происходит работа с правилами.

Что же касается класса `GraphGenerator`, то он, напротив, является не полуфабрикатом, а компонентой, полностью готовой для использования: он является составной частью каждого суперкомпилятора, построенного на основе MRSC.

Для заданной начальной конфигурации `conf` и правил переписывания `rules`, `GraphGenerator` генерирует *все* завершенные графы конфигураций, порождаемые этими правилами. Если логика, реализуемая правилами `rules` соответствует логике традиционного однорезультатного суперкомпилятора, результатом работы генератора графов (естественно) является единственный граф.

В общем случае, может получаться громадное количество графов. Поэтому, для того, чтобы удерживать расход памяти в каких-то разумных пределах, генератор графов реализован в виде итератора, порождающего графы постепенно, по запросу.

Внутри генератор графов устроен нехитро (см. исходные тексты). А именно, в каждый момент времени имеется некоторое множество незавершенных S-графов и очередь завершенных графов. Если клиент запрашивает следующий граф, а очередь не пуста, выдается первый граф из этой очереди. В противном случае, извлекается некоторый граф `g` из множества незавершенных графов и вызывается метод `steps(g)`, выдающий множество шагов построения графов (может быть – пустое). Затем каждый из этих шагов применяется к `g`, чтобы получить множество новых графов. Некоторые из новых графов являются завершенными, а некоторые – незавершенными. Завершенные графы помещаются в конец очереди завершенных графов, а незавершенные – добавляются к множеству незавершенных графов.

(Могут быть реализованы и другие стратегии, порождающие завершенные графы в другом порядке. Текущая реализация соответствует обходу “дерева графов” в глубину.)

Что должен делать потребитель с графами, порождаемыми с помощью `GraphGenerator`? В случае традиционной суперкомпиляции, можно преобразовывать графы из S-представления в T-представление, а затем из T-представления – в остаточные программы. Однако, возможны и другие варианты. Например, потребитель графов может фильтровать завершенные графы, чтобы найти среди них те, которые удовлетворяют некоторым критериям. А в некоторых случаях (при использовании суперкомпиляции для анализа программ), интерес могут представлять не конкретные графы, обладающих определенными свойствами, а сам факт их существования.

Заметим, однако, что интерфейс, через который происходит работа с правилами переписывания графов (показанный на Рис. 9), – весьма абстрактен, и не зависит от входного языка суперкомпилятора. Благодаря этому, алгоритм, реализованный в генераторе графов, тоже не зависит от входного языка суперкомпилятора.

5 Заключение

В данном препринте описана только внутренняя структура и самый нижний, "технический" этаж инструментария MRSC. В следующих препринтах мы рассмотрим конкретные примеры, показывающие, как можно использовать MRSC для быстрого прототипирования суперкомпиляторов, а также для реализации предметно-ориентированных суперкомпиляторов.

Проблема разработки инструментария для обобщенного описания и реализации суперкомпиляторов была впервые рассмотрена в работе [37], в которой был описан предметно-ориентированный язык SCPL для описания преобразований, выполняемых над графами конфигураций. К сожалению, в дальнейшем эта работа не получила должного развития.

В некотором смысле, ядро MRSC близко по духу SCPL, однако, имеются и некоторые существенные различия.

Во-первых, MRSC ориентирован на многорезультатную суперкомпиляцию, по отношению к которой традиционная однорезультатная суперкомпиляция является частным случаем. Основная идея многорезультатной суперкомпиляции – (потенциальная) множественность результатов. Ее логическим продолжением является мысль о множественности как (многорезультатных) суперкомпиляторов, так и возможных целей их применения.

Во-вторых, MRSC разработан и реализован в функциональном стиле: основные структуры данных (S-графы) являются неизменяемыми, что дает возможность генерировать тысячи графов, но при этом удерживать потребление памяти в пределах разумного. В принципе, это также позволяет разработать распараллеленную версию MRSC, в которой различные версии графа конфигураций могут обрабатываться одновременно.

Конечно, первая версия инструментария MRSC ещё очень далека от идеала. Однако, мы надеемся, что опыт, полученный в результате использования MRSC для реализации языково- и предметно-ориентированных суперкомпиляторов, послужит основой для дальнейшего совершенствования MRSC.

Благодарности

Авторы выражает признательность участникам Рефал-семинаров, проводимых в ИПМ им. М.В. Келдыша, за ценные замечания и плодотворные обсуждения этой работы, а также Наташе и Лене за их любовь и терпение.

Список литературы

[1] D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16:591–603, October 1973.

[2] M. Bolingbroke and S. L. Peyton Jones. Supercompilation by evaluation. In *Haskell 2010 Symposium*, 2010.

[3] M. Bolingbroke and S. L. Peyton Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation. In *ICFP 2011*, 2011.

[4] G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.

[5] G. W. Hamilton. A graph-based definition of distillation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[6] N. D. Jones. The essence of program transformation by partial evaluation and driving. In *PSI '99*, volume 1755 of *LNCS*, pages 62–79. Springer-Verlag, 2000.

[7] P. Jonsson and J. Nordlander. Taming code explosion in supercompilation. In *PEPM'11*, 2011.

[8] S. Kleene. *Mathematical logic*. Dover books on mathematics. Dover Publications, 2002.

[9] A. Klimov. An approach to supercompilation for object-oriented languages: the java supercompiler case study. In *First International Workshop on Metacomputation in Russia*, 2008.

[10] A. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In *International Workshop on Program Understanding (PU 2011)*, 2011.

[11] A. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *PSI 11*, 2011.

[12] A. V. Klimov. A program specialization relation based on supercompilation and its properties. In *First International Workshop on Metacomputation in Russia*, pages 54–77, 2008.

[13] I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009. URL: http://library.keldysh.ru/preprint.asp?id=2009-63.

[14] I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, 2010. URL: http://library.keldysh.ru/preprint.asp?id=2010-62.

[15] I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010. URL: http://library.keldysh.ru/preprint.asp?id=2010-31.

[16] I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, 2010. URL: http://library.keldysh.ru/preprint.asp?id=2010-81.

[17] I. Klyuchnikov. The ideas and methods of supercompilation. *Practice of Functional Programming*, (7), 2011.

[18] I. Klyuchnikov and S. Romanenko. SPSC: a simple supercompiler in scala. In *PU'09 (International Workshop on Program Understanding)*, 2009.

[19] I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.

[20] I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[21] I. Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In *PSI 2011*, 2011.

[22] A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the scp4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

[23] N. Mitchell. Rethinking supercompilation. In *ICFP 2010*, 2010.

[24] N. Mitchell and C. Runciman. A supercompiler for Core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes In Computer Science*, pages 147–164, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 249–260, 1996.

[26] M. Odersky et al. *Programming in Scala.* Artima, 2nd edition, 2010.

[27] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.

[28] M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.

[29] M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 315–337, 1998.

[30] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Logic Programming: The 1995 International Symposium*, pages 465–479, 1995.

[31] M. H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation. Practice and Theory*, volume 1706 of *LNCS*, pages 246–270, 1998.

[32] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[33] V. F. Turchin. *The phenomenon of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.

[34] V. F. Turchin. A supercompiler system based on the language REFAL. *SIGPLAN Not.*, 14(2):46–54, 1979.

[35] V. F. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis.* Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[36] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[37] V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

[38] V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 47–55, New York, NY, USA, 1982. ACM.