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

Препринты ИПМ • Препринт № 209 за 2018 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Романенко С.А.

Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления

Рекомендуемая форма библиографической ссылки: Романенко С.А. Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления // Препринты ИПМ им. М.В.Келдыша. 2018. № 209. 32 с. doi:10.20948/prepr-2018-209
URL: http://library.keldysh.ru/preprint.asp?id=2018-209

Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М. В. Келдыша
Российской академии наук

С. А. Романенко

Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления

Москва — 2018

Романенко С. А.

Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления

Рассматриваются следующие вопросы, связанные с суперкомпиляцией: (1) использование отношения гомеоморфного вложения для обеспечения завершаемости процесса суперкомпиляции, (2) особенности суперкомпиляции для языков с передачей параметров по имени и с передачей параметров по значению и (3) преимущества и недостатки суперкомпиляции по сравнению с частичными вычислениями.

Ключевые слова: суперкомпиляция, анализ программ, оптимизация программ, специализация программ, преобразование программ, метавычисления.

Sergei Anatolievich Romanenko

Supercompilation: homeomorphic embedding, call-by-name, partial evaluation

There are considered a number of issues related to supercompilation: (1) the use of the homeomorphic embedding relation for ensuring termination of supercompilation, (2) the peculiarities of supercompilation for the languages with call-by-name and call-by-value parameter passing, and (3) the advantages and drawbacks of supercompilation and partial evaluation.

Key words: supercompilation, program analysis, program optimization, program specialization, program transformation, metacomputation.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 16-01-00813-а.

1 Введение

В препринте [19] были рассмотрены основные принципы и базовые понятия «суперкомпиляции», представляющей собой метод анализа и преобразования программ, основанный на выполнении программы не для конкретных входных данных, а в «общем» виде (для произвольных входных данных, представленных в «символической» форме).

Цель суперкомпиляции — построение конечного «графа конфигураций». «Конфигурации», находящиеся в узлах графа, изображают множества возможных состояний вычислительного процесса, а дуги графа — возможные переходы между множествами состояний. Основные операции, с помощью которых строится граф конфигураций, — это *прогонка*, *обобщение* и *зацикливание*.

Прогонка — это порождение новых конфигураций из уже имеющихся. Неограниченное применение прогонки, как правило, приводит к построению *бесконечного* дерева конфигураций [19]. Однако целью суперкомпиляции является получение *конечного* графа конфигураций, и эта цель может быть достигнута с помощью двух других операций: обобщения и зацикливания. Обобщение — это замена одной конфигурации на другую, «более абстрактную», представляющую больше состояний, чем представляла исходная конфигурация. А зацикливание — это сведéние конфигурации к эквивалентной конфигурации (уже встречавшейся ранее).

Важно отметить, что обобщение конфигураций является самой сложной частью суперкомпиляции, поскольку именно в процессе обобщения возникает возможность принимать различные решения, от которых зависит конечный результат суперкомпиляции.

С одной стороны, при каждом обобщении происходит потеря информации, поскольку расширяется множество состояний, описываемых конфигурацией, и снижается точность анализа вычислительного процесса (ради которого, собственно говоря, и выполняется суперкомпиляция). И, в худшем случае, может получиться программа, совпадающая с исходной (с точностью до имен переменных и функций). Другими словами, суперкомпиляция может вырождаться в тождественное преобразование (польза от которого — весьма сомнительна).

С другой стороны, если выполнять обобщение недостаточно энергично, это может привести к тому, что процесс суперкомпиляции никогда не завершится, поскольку будут бесконечно порождаться всё новые и новые конфигурации.

Таким образом, необходимо использовать какие-то стратегии, которые, с одной стороны, гарантировали бы завершаемость процесса суперкомпиляции, но, с другой стороны, приводили бы к получению нетривиальных результатов суперкомпиляции. Этот вопрос, в силу его сложности, подробно в препринте [19] не рассматривался, и ему посящен Раздел 2 данного препринта.

Кроме того, в Разделе 3 рассматриваются особенности суперкомпиляции

для языков с передачей параметров по имени и с передачей параметров по значению, а в Разделе 4 — преимущества и недостатки суперкомпиляции по сравнению с частичными вычислениями.

2 Методы обеспечения конечности графа конфигураций

При «прогонке» (символическом/обобщённом исполнении программы) в общем случае получается бесконечное дерево конфигураций. Задача суперкомпиляции — превратить это дерево в конечный граф.

В этом разделе мы рассмотрим методы, с помощью которых удается свернуть бесконечное дерево в конечный граф конфигураций.

2.1 Приведение частного к общему

В препринте «Суперкомпиляция: основные принципы и базовые понятия» [19] было рассмотрено такое простейшее «правило зацикливания». Допустим, в дереве встретился такой узел, который содержит конфигурацию X_2, и при этом среди предков этого узла в дереве есть конфигурация X_1, такая, что X_1 и X_2 совпадают (с точностью до переименования переменных). Понятно, что при таких условиях нет смысла строить поддерево для X_2, поскольку оно будет выглядеть точно так же, как и поддерево, подвешенное к X_1.

Поэтому выполнялась следующая операция. К графу добавлялась обратная стрелка из X_2 в X_1, а поддерево, подвешенное к X_2, удалялось из графа. Точнее, оно не удалялось, а вообще не строилось!

Вот так и получался конечный граф. Правда, только для некоторых «особенно хороших» исходных программ.

Поэтому в препринте «Суперкомпиляция: основные принципы и базовые понятия» [19] рассматривалось и более сложное «правило зацикливания». Допустим, что X_2 и X_1 — действительно разные (X_1 нельзя превратить в X_2 простым переименованием переменных), но X_2 является «частным случаем» X_1. В том смысле, что, применив к переменным в X_1 некоторую подстановку, можно превратить X_1 в X_2. Тогда можно преобразовать X_2 таким образом, чтобы она совпала с X_1 (с точностью до переименования переменных). Это делается так: выдумываем новые, уникальные имена переменных v_1, ..., v_k и X_2 приводится к виду

$$\texttt{let } v_1 \texttt{ = } e_1\texttt{, ..., } v_k \texttt{ = } e_k \texttt{ in } e_0,$$

где e_0 уже совпадает с X_1 (с точностью до переименования переменных). На следующем шаге let-выражение разбирается на части e_0, e_1,..., e_k, каждая из которых обрабатывается отдельно. При этом уже можно добавить к графу обратную стрелку из e_0 в X_1.

Переход от X_2 к let-выражению, содержащему e_0, называется «обобщением», поскольку X_2 является «частным случаем» e_0.

Таким образом, благодаря обобщению конфигураций, расширился класс программ, для которых получался конечных граф конфигураций.

И стал понятен способ борьбы с разрастанием дерева. Если встретилась конфигурация, которая является частным случаем уже рассмотренной конфигурации — сводим более специфическую конфигурацию к более общей. Возникает вопрос: достаточно ли этого правила, чтобы получить конечный граф конфигураций для любой исходной программы? Ответ на это — отрицательный... (Видимо, Судьба так распорядилась, чтобы авторам суперкомпиляторов жизнь мёдом не казалась.)

2.2 Частное сводится к общему не всегда

Рассмотрим следующий простой (но коварный) пример программы. Определим функцию умножения `mult` через уже нам известную функцию сложения `add`.

```
add(Z, y) = y;
add(S(x), y) = S(add(x, y));
mult(Z, y) = Z;
mult(S(x), y) = add(mult(x, y), y);
```

При этом, натуральные числа 0, 1, 2, 3 представлены в виде `Z`, `S(Z)`, `S(S(Z))`, `S(S(S(Z)))`, ..., а определения функций основаны на следующих свойствах натуральных чисел:

$$0 + y = y,$$
$$(x + 1) + y = (x + y) + 1,$$
$$0 * y = 0,$$
$$(x + 1) * y = x * y + y.$$

Возьмём начальную конфигурацию `mult(a, b)` и попробуем построить из неё граф конфигураций. Получается дерево, показанное на Рис. 1.

Видно, что если двигаться по правой ветви этого дерева, то получается такая (бесконечная) последовательность конфигураций:

```
mult(a, b)
add(mult(a1, b), b)
add(add(mult(a2, b), b), b)
add(add(add(mult(a3, b), b), b), b)
...
```

При этом ни одна из конфигураций *не является* «частным случаем» какой-либо из предыдущих конфигураций! Так получается из-за того, что с каждым шагом снаружи добавляется новый вызов функции `add`.

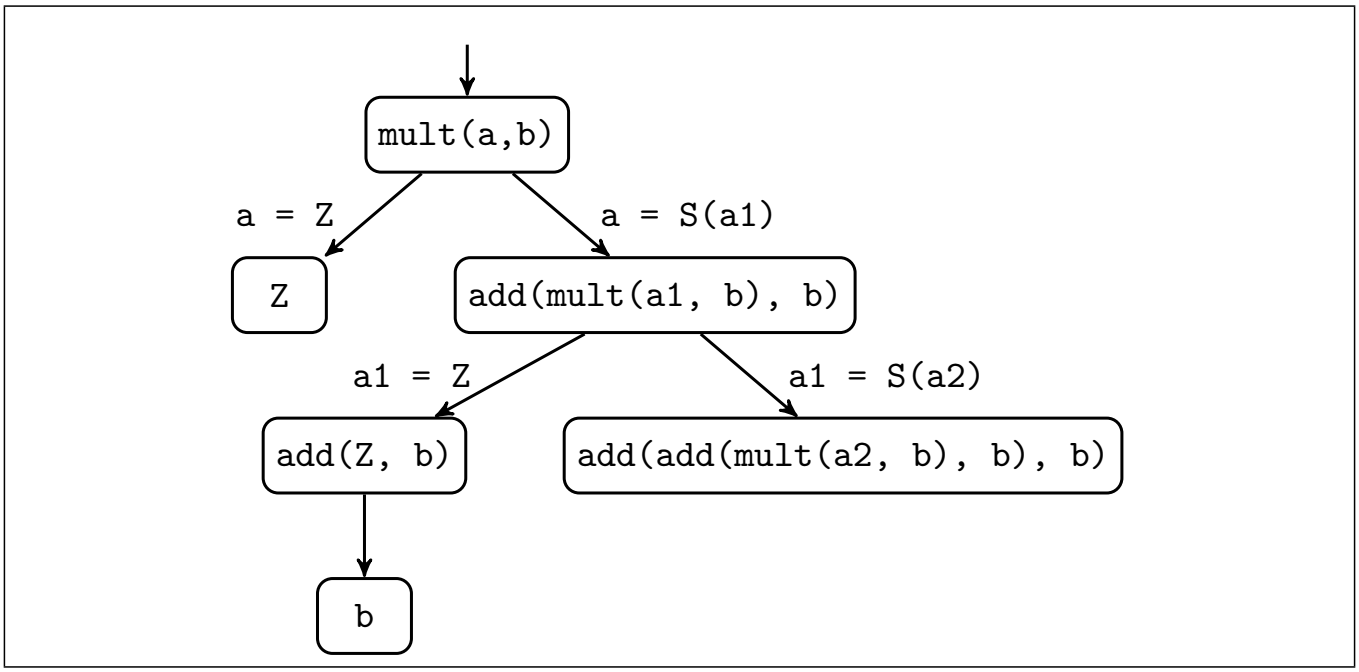

Рис. 1: Результат прогонки для `mult(a, b)`.

Что делать? Здравый смысл подсказывает, что у всех этих конфигураций всё же есть общая часть: подвыражение вида `mult(`a_k`, b)`. В частности, можно свести `add(mult(a1, b), b)` к `mult(a, b)` с помощью «обобщения», использовав конструкцию let:

```
let v1 = mult(a1, b), v2 = b in add(v1, v2)
```

после чего уже можно добавить к графу обратную стрелку от `mult(a1, b)` к `mult(a, b)`. В результате получается (незаконченный) граф конфигураций, показанный на Рис. 2.

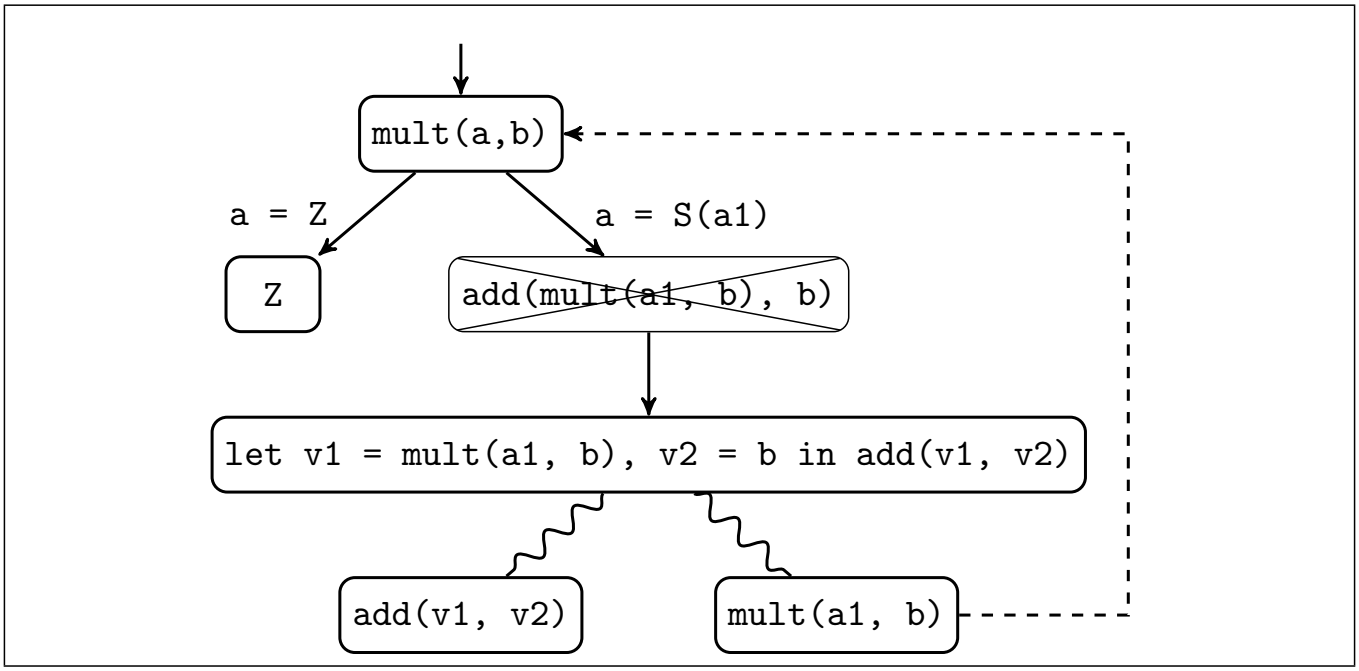

Рис. 2: Результат прогонки для `mult(a, b)` после декомпозиции.

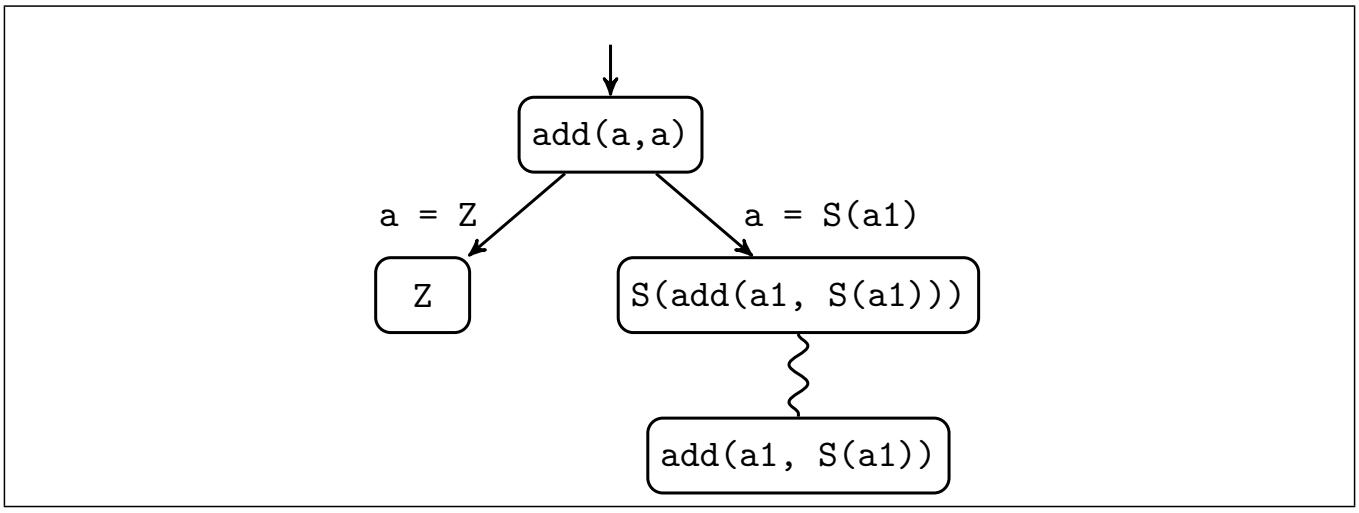

Рис. 3: Результат прогонки для `add(a, a)`.

Суперкомпилятор SPSC [14, 19] действительно разделывается с этой программой именно так! В этом можно убедиться, пропустив через SPSC задание `mult_a_b`. Правда, результат суперкомпиляции получается какой-то неинтересный: остаточная программа, по существу, совпадает с исходной... Ну так ведь и программа такова, что непонятно, что можно было бы сделать с ней интересного? Вот если бы начальная конфигурация была не `mult(a, b)`, а какого-то более специального вида... Но об этом — позже.

2.3 Обобщение «верхней» конфигурации

В предыдущем разделе нам удалось построить конечный граф конфигураций, используя то, что часть одной из конфигураций оказалась частным случаем уже ранее встретившейся конфигурации. Однако этот способ зацикливания срабатывает не всегда.

Рассмотрим снова фунцию сложения `add` и попробуем просуперкомпилировать начальную конфигурацию `add(a, a)`. Когда мы суперкомпилировали `add(a, b)`, всё получалось хорошо. Но в случае `add(a, a)` различие заключается в том, что переменная `a` повторяется 2 раза! И это создает информационную связь между двумя частями конфигурации.

В результате прогонки получается дерево, показанное на Рис. 3.

Видно, что порождается такая (бесконечная) последовательность конфигураций:

```
add(a, a)
add(a1, S(a1))
add(a2, S(S(a2)))
add(a3, S(S(S(a3))))
...
```

Видно, что разбор случаев для первого аргумента функции `add` приводит к «распуханию» её второго аргумента. И при этом ни одна из получающихся

конфигураций не является частным случаем предыдущей. Попробуем, например, найти подстановку, такую, чтобы `add(a, a)` совпало с `add(a1, S(a1))`. Для этого надо «решить уравнение»

```
add(a, a) = add(a1, S(a1)),
```

подобрав для `a` такую подстановку, чтобы левая часть совпала с правой. Но такого значения для `a` подобрать невозможно! Факт очевидный, хотя формальное его обоснование – длинное и нудное... (Что в очередной раз подтверждает, что очевидные вещи обосновывать гораздо труднее, чем неочевидные...)

Чтобы получился конечный граф, нужно сделать обобщение (с помощью всё той же конструкции let). В данном случае, беда произошла из-за того, что переменная `a` входит в `add(a, a)` два раза. Так сделаем же из одной переменной две разные:

```
let v1 = a, v2 = a in add(v1, v2)
```

в результате чего конфигурация `add(a, a)` превращается в `add(v1, v2)`, при суперкомпиляции которой получается конечное дерево конфигураций. Какое именно — рассматривается в препринте «Суперкомпиляция: основные принципы и базовые понятия» [19].

Сразу следует отметить одну тонкость. До сих пор мы делали обобщение таким способом: сравнивали «верхнюю» конфигурацию X_1 с «нижней» конфигурацией X_2 и «обобщали» X_2 таким образом, чтобы она совпала с X_1 (с точностью до имён переменных). Например, сравнивали `add(a, b)` и `add(a1, S(b))` и обобщали `add(a1, S(b))` до `add(v1, v2)`. При этом «прогресс» при построении графа конфигураций был непрерывным и «поступательным»: к уже существующим узлам пристраивались дополнительные узлы и стрелки.

Но при сравнении `add(a, a)` и `add(a1, S(a1))` мы столкнулись с принципиально другим случаем! `add(a1, S(a1))` невозможно обобщить до `add(a, a)`! Конечно, можно преобразовать `add(a1, S(a1))` следующим образом:

```
let v1 = a1, v2 = S(a1) in add(v1, v2)
```

При этом, получается, что `add(a1, S(a1))` — это частный случай `add(v1, v2)`. Но беда в том, что `add(v1, v2)` не является частным случаем `add(a, a)` (чего нам хотелось бы добиться). Совсем наоборот! `add(a, a)` — это частный случай `add(v1, v2)`, поскольку существует такая подстановка `{v1 := a, v2 := a}`, что

```
add(v1, v2) {v1:=a, v2:= a} = add(a, a)
```

Вот так-то! Абстрактно говоря, сначала встретилась конфигурация X_1, а потом — конфигурация X_2. Попробовали обобщить X_2 до X_1, но выяснилось, что это невозможно. Единственно, чего можно добиться, так это найти такую конфигурацию Y, которая является обобщением как по отношению к X_1, так и по отношению к X_2. В данном случае:

```
add(v1, v2) {v1:=a, v2:= a} = add(a, a)
add(v1, v2) {v1:=a1, v2:=S(b)} = add(a1, S(b))
```

Как должен действовать суперкомпилятор в подобных случаях? Наверное, возможны разные варианты... Но, наверное, самое простое решение проблемы таково.

Как мы помним, граф конфигураций — это дерево плюс обратные стрелки. X_2 находится в поддереве, подвешенном под X_1. Пусть Y — обобщение по отношению к X_1 и X_2. Тогда суперкомпилятор уничтожает всё поддерево, подвешенное к X_1 (в том числе — и X_2), а узел X_1 заменяет на узел вида

`let` v_1 `=` E_1`, ...,` v_k `=` E_k `in` Y

После чего суперкомпиляция продолжается. Суперкомпилятор SPSC [14] именно так и поступает, в чем можно убедиться, пропустив задание `add_a_a`.

> Таким образом, вместо поступательного построения дерева получается некоторая последовательность попыток: строим поддерево, «разочаровываемся» в нём, уничтожаем его, обобщаем конфигурацию, из которой оно выросло, строим поддерево заново, и т.д.

Возникает вопрос: а не будет ли этот процесс продолжаться бесконечно? К счастью, как показал Сёренсен [16], в случае языка SLL, с которым работает суперкомпилятор SPSC [14], этот процесс рано или поздно завершается. Дело в том, что в случае SLL для любой конфигурации существует только конечное число обобщений (с точностью до переименования переменных). Таким оборазом, «верхняя» конфигурация может подвергнуться обобщению только конечное число раз.

2.4 Что такое «гомеоморфное вложение»?

Итак, если мы решили, что две конфигурации X_1 и X_2 — «похожи», то можно либо обобщить X_2 до X_1, либо, на худой конец, построить конфигурацию Y, являющуюся обобщением по отношению как к X_1, так и к X_2. Вопрос только в том, как формализовать само понятие «похожи»? Можно было бы рассматривать некоторое симметричное отношение «похожести» для конфигураций, но нам нужно не это. Суперкомпиляция — это имитация «обычных» вычислений. Тем самым, процесс суперкомпиляции имеет некоторое направление! Поэтому при сравнении конфигураций нужно учитывать, какая из конфигураций появилась раньше, а какая — позже.

А само сравнение конфигураций делается для того, чтобы процесс построения дерева конфигураций (с обратными стрелками, превращающими его в граф) завершался за конечное время.

Вот, допустим, под конфигурацией X_1 в дереве висит конфигурация X_2. Нам нужно решить, нужно ли обобщать X_2 (или даже X_2 вместе с X_1), или

же дела идут хорошо, и можно продолжать процесс, приделывая, с помощью прогонки, к X_2 новое поддерево.

Здесь возникает конфликт интересов. С одной стороны, суперкомпиляция делается для того, чтобы проанализировать поведение программы «в общем виде». Каждая конфигурация описывает некоторое множество возможных состояний вычислительного процесса. Если конфигураций в дереве конфигураций много, это означает, что суперкомпилятор разбивает множество состояний на много частей, рассматривает много частных случаев. Стало быть, анализ процесса вычислений — утончённый и глубокий. А если конфигураций мало — это означает, что суперкомпилятор работает «топорно», и ничего интересного с помощью суперкомпиляции не получается (например, на выходе суперкомпилятора просто вываливается исходная программа).

А обобщение приводит к уменьшению количества рассматриваемых конфигураций. Обобщили — значит свалили два разных множества состояний в одну кучу (и часто даже ещё и что-то лишнее в эту кучу втащили). Результат суперкомпиляции становится менее утончённым и менее интересным.

Таким образом, если не обобщать — дерево конфигураций может получиться бесконечным. А если слишком ретиво обобщать — дерево получится конечным, но неинтересным.

Итак, разумным представляется такой подход: сравниваем X_1 и X_2. Если X_2 на X_1 «не похожа», то не обобщаем X_2 и X_1. А если X_2 «похожа» на X_1, то надо изучить ситуацию подробнее. Если X_2 «меньше по размерам», чем X_1, то разумно X_2 не обобщать, поскольку количество конфигураций, которые «меньше», чем X_1 — конечно, и мы можем себе позволить их все перебрать, не рискуя тем, что дерево конфигураций получится бесконечным. (Разумеется, это верно, если понятие «меньше» определено надлежащим образом.) А вот если X_1 и X_2 похожи, и при этом X_2 — «больше», чем X_1, то ситуация — опасная, нужно обощать! Введем для ситуации, когда X_1 и X_2 «похожи», но при этом X_2 «больше», чем X_1, такое обозначение:

$$X_1 \trianglelefteq X_2$$

Как определить такое отношение, разные умные люди думали. Но первыми придумали Хигман [3] и Кру́скал [8].

Хигман и Крускал догадались, что определить отношение $\trianglelefteq$ можно гениально простым способом! А именно, пусть считается, что $X_1 \trianglelefteq X_2$, если X_2 можно превратить в X_1 стерев в X_2 некоторые его части. Сразу убивается два зайца:

- Если X_2 можно превратить в X_1, кое-что стирая в X_2, то кто же усомнится, что X_1 и X_2 — «похожи»?
- Если X_2 можно превратить в X_1, кое-что стирая в X_2, то это и означает, что X_2 — «толще» или «больше», чем X_1.

Хигман применил эту идею к линейным строкам, а Крускал — обобщил для деревьев (термов, составленных из n-арных конструкторов). А отношение

$\trianglelefteq$ научно назвали «отношением гомеоморфного вложения», так что $X_1 \trianglelefteq X_2$ читается так: «X_1 гомеоморфно вложено в X_2». Ну да, можно считать, что X_1 полностью присутствует в X_2, но в «разбавленном» виде.

Наглядно это можно представить себе так. Допустим, у нас есть 50 граммов кефира, и мы доливаем в него 150 граммов кваса. Тогда можно считать, что в получившейся смеси 50 граммов кефира «вложены» в 200 граммов смеси. При этом исходные 50 граммов кефира можно получить обратно, отпив из стакана 150 граммов кваса.

В случае выражений, с которыми работает суперкомпилятор, эти выражения содержат не только конструкторы, но ещё и вызовы функций и переменные. Ну, вызовы функций ничего принципиально нового не добавляют (поскольку имена конструкторов и имена функций не совпадают). Так что с вызовами функций можно обращаться так же, как с конструкторами. Но что делать с переменными? Самый простой подход — свалить все переменные в одну кучу и считать, что всем переменным как бы соответствует один нульарный конструктор. Есть и более тонкие способы обойтись с переменными, о которых можно прочитать, например, в статье Лёйшеля [9].

Рассмотрим, например, `add(a, b)` и `add(a1, S(b))`. Верно ли, что

`add(a, b)` $\trianglelefteq$ `add(a1, S(b))` ?

Верно! Стираем в `add(a1, S(b))` конструктор `S` и получаем `add(a1, b)`. А это выражение совпадает с `add(a, b)` с точностью до имён переменных.

Рассмотрим `mult(a, b)` и `add(mult(a1, b), b)`. Верно ли, что

`mult(a, b)` $\trianglelefteq$ `add(mult(a1, b), b)` ?

Верно! Стираем в `add(mult(a1, b), b)` вызов функции add с её вторым аргументом и получаем `mult(a1, b)`, что совпадает с `mult(a, b)` с точностью до имён переменных.

Рассмотрим `add(a, a)` и `add(a1, S(a1))`. Верно ли, что

`add(a, a)` $\trianglelefteq$ `add(a1, S(a1))` ?

Верно! Стираем в `add(a1, S(a1))` конструктор `S` и получаем `add(a1, a1)`, что совпадает с `add(a, a)` с точностью до имён переменных.

Можно дать и формальное определение гомеоморфного вложения с помощью индуктивного определения в виде трёх правил:

1. $x \trianglelefteq y$ для любых переменных x и y.
2. $X \trianglelefteq f(Y_1, ..., Y_k)$, если f — имя конструктора или функции и для некоторого i выполнено $X \trianglelefteq Y_i$.
3. $f(X_1, ..., X_k) \trianglelefteq f(Y_1, ..., Y_k)$, если f — имя конструктора или функции и для всех i выполнено $X_i \trianglelefteq Y_i$.

Правило 2 называется правилом «ныряния» (левое выражение «ныряет» в один из аргументов правого), а привило 3 — правилом «сочетания» (аргументы конструктора/функции с двух сторон «сочетаются браком» друг с другом).

2.5 Использование гомеоморфного вложения при суперкомпиляции

Интересно, что отношение гомеоморфного вложения нашло первое применение не в суперкомпиляции, а в системах переписывания термов [2].

Однако (спустя некоторое время) Сёренсен и Глюк обнаружили, что отношение $\trianglelefteq$ можно использовать и для организации обобщения и зацикливания в процессе суперкомпиляции [17].

Интересно, что отношение $\trianglelefteq$ хорошо работает также в случаях, когда у суперкомпилятора появляется возможность выполнить какие-то вычисления над известными частями данных во время суперкомпиляции. Например, вспомним функцию `addAcc`, которая выполняет сложение с помощью накапливающего параметра (аккумулятора):

```
addAcc(Z, y) = y;
addAcc(S(x), y) = addAcc(x, S(y));
```

Попробуем просуперкомпилировать выражение `addAcc(S(S(a)), b)` с помощью суперкомпилятора SPSC [14], пропустив через него задание `addAcc_SSa_b`.

В процессе построения дерева процесса на одной из ветвей получается такая последовательность конфигураций:

1. `addAcc(S(S(a)), b)`
2. `addAcc(S(a), S(b))`
3. `addAcc(a, S(S(b)))`
4. `addAcc(a1, S(S(S(b))))`

Видно, что второй аргумент всё время угрожающе раздувается. Но суперкомпилятор не пугается! Второй аргумент раздувается, но зато первый аргумент уменьшается. Поэтому конфигурация 1 не вложена в конфигурации 2, 3 и 4, а конфигурация 2 — в конфигурации 3 и 4. А вот конфигурация 3 вложена в конфигурацию 4! И при этом 4 является частным случаем 3. Поэтому 4 обобщается до 3 и в дереве конфигураций создаётся обратная стрелка на конфигурацию 3.

Возникает такой вопрос: а не может ли получиться так, что в процессе прогонки у нас всё время будут получаться разные конфигурации, которые не будут вкладываться друг в друга? Оказывается, это не так! Для любой бесконечной последовательности конфигураций X_1, X_2, X_3, ... обязательно найдутся такие i и j, что $i < j$ и

$$X_i \trianglelefteq X_j$$

Соответствующую теорему доказал Кру́скал [8, 15]. Для случая, когда выражения составлены только из конструкторов. При этом принципиальным

является то, что все конструкторы принадлежат конечному множеству. Однако в суперкомпиляторе SPSC [14], например, выражения могут содержать ещё и вызовы функций и переменные. Тем не менее, теорема всё равно применима, поскольку вызовы функций, с синтаксической точки зрения — то же самое, что и конструкторы (ибо смысл термов для гомеоморфного вложения безразличен). При этом в конфигурациях могут появляться только те конструкторы и имена функций, которые присутствуют в исходной программе. Некоторую проблему представляют переменные, ибо переменные-то могут появляться в конфигурациях в неограниченном количестве: суперкомпилятор их генерирует по мере надобности. Как быть? Простейшее решение — свалить все переменные в одну кучу и не различать их между собой. Т.е. считать, что

$$x \trianglelefteq y$$

для любых переменных x и y. Вот и получается, что с точки зрения $\trianglelefteq$ переменная как бы одна-единственная, и её можно воспринимать как нуль-арный конструктор.

Тут следует упомянуть, что Турчиным был предложен и другой [18] подход к организации обобщения и зацикливания в суперкомпиляторе, не основанный на гомеоморфном вложении конфигураций.

Но это — тема для отдельного разговора...

Вот мы и ознакомились с «азами» суперкомпиляции: прогонкой (с построением бесконечного дерева), обобщением и зацикливанием (с помощью гомеоморфного вложения) и построением остаточной программы из конечного графа конфигураций.

Можно посмотреть, как всё это реализовано в суперкомпиляторе SPSC [14], написанном на языке Idris [1] (программирование на котором весьма похоже на программирование на языке Haskell, но при этом система статической типизации — более продвинутая и позволяет выражать формулировки и доказательства теорем).

Хотя SPSC [14] и невелик по размерам, он — полностью работоспособен и в нём присутствуют все «принципиально важные» элементы суперкомпилятора.

2.6 «Глобальное» и «локальное» зацикливание

В препринте «Суперкомпиляция: основные принципы и базовые понятия» [19] рассматривалось такое задание на суперкомпиляцию:

```
add(add(a, b), c)
where

add(Z, y) = y;
add(S(x), y) = S(add(x, y));
```

При этом после выполнения шага прогонки

```
add(add(a, b), c) ⟶{a=S(a1)}⟶ add(S(add(a1, b)), c)
```

получалась конфигурация `add(S(add(a1, b)), c)`. Затем выполнялся ещё один шаг прогонки, и получалась конфигурация `S(add(add(a1, b), c))`. Далее, в результате извлечения аргумента конструктора, возникала конфигурация `add(add(a1, b), c))`, которая совпадала с исходной конфигурацией (с точностью до переименования переменных), что и позволяло выполнить зацикливание.

Теперь предположим, что суперкомпилятор использует гомеоморфное вложение для распознавания ситуаций, когда требуется выполнять обобщение, и сравним две конфигурации:

```
add(add(a, b), c)
add(S(add(a1, b)), c)
```

Оказывается, что первая конфигурация гомеоморфно вложена во вторую (для наглядности, вложенные части выделены подчеркиваниями). Поэтому суперкомпилятор, вроде бы, должен был в этой ситуации выполнить обобщение верхней конфигурации до `let d = add(a, b) in add(d, c)`. Но в данной ситуации было бы неразумно так поступать, поскольку обобщение конфигурации приводит к потере информации, что снижает глубину и точность анализа вычислительного процесса. И суперкомпилятор SPSC [14], действительно, воздерживается от обобщения и выполняет шаг прогонки! В этом можно убедиться, пропустив через него задание `add_add_a_b_c`.

Дело в том, что в SPSC реализована идея разделения *локального* и *глобального* управления зацикливанием.

Суть идеи сводится к следующему. Согласно теореме Хигмана-Кру́скала [3, 8], *для любой* бесконечной последовательности конфигураций рано или поздно возникает ситуация, когда некоторая конфигурация оказывается вложена в одну из последующих конфигураций. Следовательно, если из исходной последовательности выбрать любую бесконечную подпоследовательность, то ситуация вложения возникнет и для неё (ибо теорема приложима *к любой* бесконечной последовательности).

Отсюда возникает следующая идея: а что если мы разобъём множество всех возможных конфигураций на непересекающиеся подмножества и будем сравнивать только конфигурации, принадлежащие к одной и той же категории? Именно эта идея и реализована в суперкомиляторе SPSC [14]. А именно, SPSC разбивает все конфигурации на три категории: «тривиальные», «глобальные» и «локальные».

- T – тривиальные
  - Переменная: v

  - Обобщение: `let` v_1 `=` e_1`, ...,` v_k `=` e_k `in` e_0
  - Конструктор: `C(...)`
- G – глобальные (редукция с сужением)
  - `g2(g1(v,...),...)`
- L – локальные (редукция без сужения)
  - `g2(g1(C(...),...),...)`
  - `g2(g1(f(...),...),...)`
  - `f(...)`

Тривиальные конфигурации обладают тем свойством, что любая последовательность, состоящая только из тривиальных конфигураций, – конечна. Поэтому тривиальные конфигурации можно вообще игнорировать при сравнении конфигураций. Локальные конфигурации соответствуют шагам прогонки, которые можно выполнить, не используя информацию о значениях переменных, а глобальные конфигурации соответствуют шагам прогонки, при выполнении которых требуется анализ значений переменных (путем разбора случаев).

Разделение «локального» и «глобального» управления зацикливанием заключается в том, что локальные конфигурации сравниваются только с локальными, а глобальные — с глобальными. При этом используется и ещё одна дополнительная идея: глобальное и локальное управление можно упорядочить «лексикографически», разбив последовательность конфигураций на подпоследовательности локальных конфигураций, разделенные глобальными конфигурациями. И после этого – сравнивать только локальные конфигурации, принадлежащие к одной подпоследовательности.

Почему лексикографическое упорядочение гарантирует возникновение вложения? Да потому, что если последовательность содержит *бесконечное* количество глобальных конфигураций, то гарантировано, что вложение возникнет между ними. А если количество глобальных конфигураций конечно, то после последней глобальной конфигурации идет *бесконечное* количество локальных конфигураций. И гарантировано, что вложение произойдет между локальными конфигурациями.

Теперь, если мы сравним конфигурации `add(add(a, b), c)` и `add(S(add(a1, b)), c)`, то мы увидим, что первая из них является глобальной, а вторая — локальной. Стало быть, SPSC их вообще не сравнивает и далает шаг прогонки для второй из них.

Первоначально идея разделения локального и глобального контроля была предложена в области частичной дедукции (специализации Пролог-программ) [10, 9], а затем эта идея была приспособлена для суперкомпиляции функциональных программ [16].

3 Какой язык удобнее суперкомпилировать: «строгий» или «ленивый»?

Как объяснялось в препринте «Суперкомпиляция: основные принципы и базовые понятия» [19], метавычисления (разновидностью которых является суперкомпиляция) представляют собой некоторое обобщение обычных вычислений. Поэтому разным моделям обычных вычислений соответствуют и разные разновидности метавычислений.

Логика получается такая. Выбираем язык, программы на котором собираемся преобразовывать. Выбираем какой-то вариант семантики этого языка. При этом понятно, что удобнее всего исходить из операционной семантики, ибо метавычисления — это ведь тоже «вычисления», и операционная семантика к ним находится ближе всего.

А из разных вариантов операционной семантики ближе всего к суперкомпиляции находится семантика, основанная на последовательном преобразовании выражений («упрощении» или «редукции» этих выражений).

При этом в случае семантики, основанной на редукции, большинство языков разделяется на две группы:

- «Строгие» языки. При вычислении вызова функции функция не начинает работать до тех пор, пока не вычислятся все её аргументы. Такой тип вычислений известен ещё как «вычисление изнутри наружу» или «передача параметров по значению» (call-by-value).

- «Ленивые» языки. При вычислении вызова функции её аргументы вычисляются ровно настолько, насколько они нужны для вычисления вызова самой функции. Такой тип вычислений известен ещё как «вычисление снаружи внутрь» или «передача параметров по имени» (call-by-name). Точнее, между «ленивостью» и «вызовом по имени» есть тонкое различие, но в данный момент оно для нас несущественно.

С каким языком удобнее иметь дело в суперкомпиляторе: «строгим» или «ленивым»?

Предположим, что входной язык суперкомпилятора — «строгий». Рассмотрим следующее задание:

```
b(a(u))
where

a(Stop) = Stop;
a(A(x)) = B(a(x));

b(Stop) = Stop;
b(B(x)) = C(b(x));
```

Раз язык «строгий», то прежде чем вызывать функцию, нужно полностью вычислить её аргументы. Допустим, что для `u` задано значение `A(A(Stop))`. Тогда вычисление вызова `b(a(A(A(Stop)))` происходит следующим образом.

Мы видим, что внутри аргумента функции `b` находится вызов функции `a`. Значит, функцию `b` пока не вызываем и вычисляем вызов функции `a`. Получается:

```
b(a(A(A(Stop))) ⟶ b(B(a(A(Stop)))) ⟶
  b(B(B(a(Stop)))) ⟶ b(B(B(Stop)))
```

Ну вот, теперь аргумент функции `b` принял вид `B(B(Stop))`. Вызовов функций в нём больше нет. Поэтому начинает «работать» вызов функции `b`:

```
b(B(B(Stop))) ⟶ C(b(B(Stop))) ⟶
    C(C(b(Stop))) ⟶ C(C(Stop))
```

Теперь обрабатываемое выражение больше не содержит вызовов функций, и окончательным результатом вычислений считается `C(C(Stop))`.

Теперь, допустим, нам захотелось изучить процесс вычисления «в общем виде», когда аргумент неизвестен. Т.е. «вычислить» `b(a(u))`, где значение `u` — неизвестно. Если мы хотим действовать «честно», т.е. так, чтобы метавычисление происходило «точно так же», как и обычное вычисление, мы должны и в суперкомпиляторе вычислять выражения «изнутри наружу».

Берём начальную конфигурацию `b(a(u))` и начинаем делать прогонку. Надо рассмотреть два случая: `u = Stop` и `u = A(u1)`, где `u1` — свежая переменная. Случай `u = Stop` — не очень интересный, поэтому сосредоточимся только на одной ветви в дереве конфигураций. Получается:

```
b(a(u)) ⟶{u=A(u1)}⟶ b(B(a(u1))) ⟶{u1=A(u2)}⟶
    b(B(B(a(u2))))
```

Видно, что выражение раздувается и раздувается: всё время появляются новые экземпляры конструктора `B`, и продолжать так можно бесконечно. С этим нужно как-то бороться! Ведь нужно получить конечный граф конфигураций! И способ борьбы известен: обобщение и зацикливание.

Надо сказать, что «свисток», основанный на отношении гомеоморфного вложения, в данном случае срабатывает вполне адекватно. Сравним выражения

```
b(a(u))
b(B(a(u1)))
```

Если присмотреться, то видно, что верхнее выражение вложено в нижнее: если из нижнего выражения вымарать конструктор `B`, то получается выражение `b(a(u1))`, которое совпадает с `b(a(u))` с точностью до имён переменных.

При этом, `b(B(a(u1)))` не является частным случаем `b(a(u))`, поскольку на что переменную `u` ни заменяй, выражение `b(B(a(u1)))` получить невозможно. Проклятый конструктор `B` мешает!

Поэтому суперкомпилятор сравнивает `b(a(u))` и `b(B(a(u1)))` и находит их максимальную общую часть `b(v)`, т.е. такое выражение, что `b(a(u))` и `b(B(a(u1)))` являются его частными случаями. Заменяем `v` на `a(u)` — получаем первое выражение. Заменяем `v` на `B(a(u1))` — получаем второе выражение.

После этого суперкомпилятор уничтожает всё поддерево, которое «выросло» из `b(a(u))`, и «обобщает» выражение `b(a(u))`, заменяя его на `let v=a(u) in b(v)`. В результате получается граф конфигураций, показанный на Рис. 4, которому соответствует остаточная программа, по сути, совпадающая с исходной программой. Другими словами, суперкомпиляция ничего интересного не даёт.

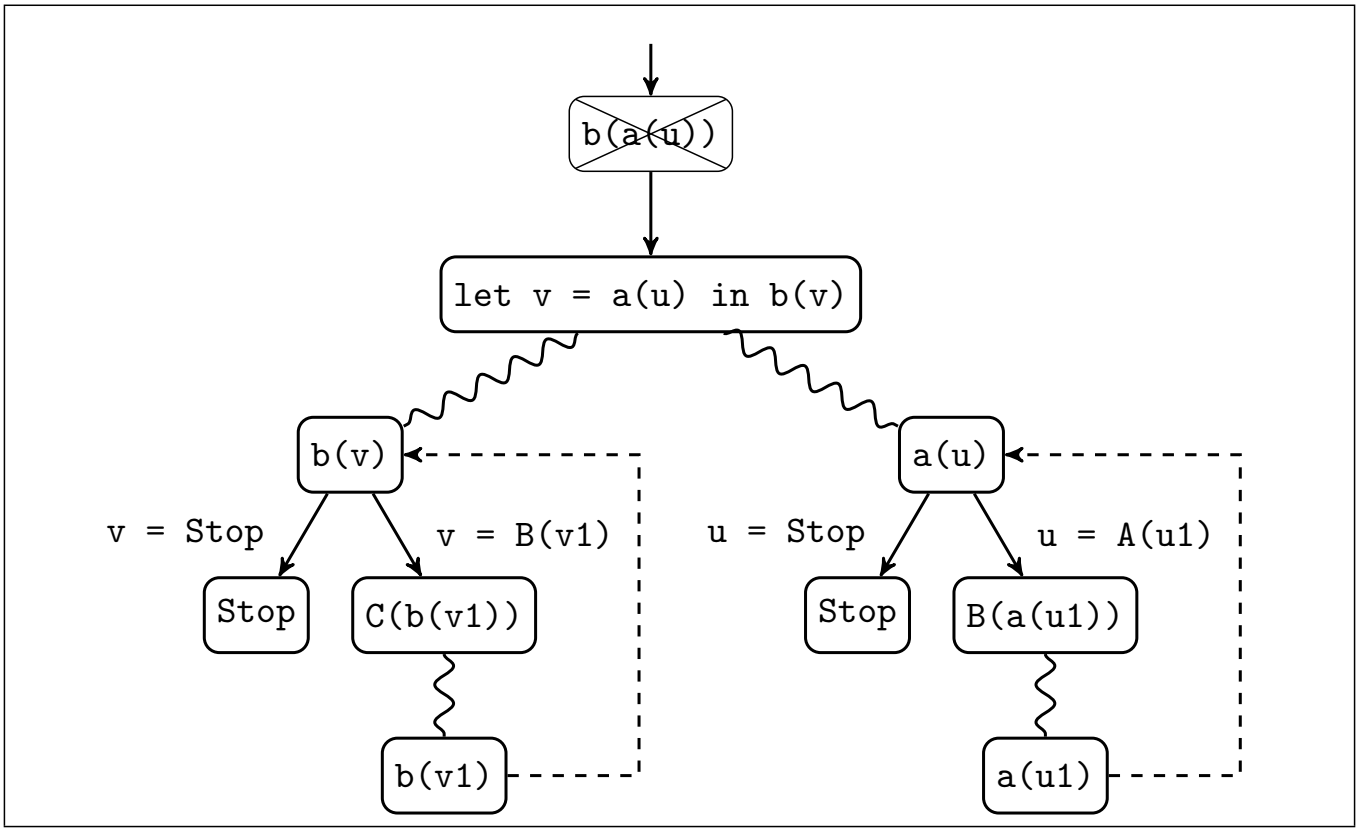

Рис. 4: Суперкомпиляция `b(a(u))` при прогонке изнутри наружу.

Но тут возникает такая «хулиганская» идея: а зачем во время метавычислений строго следовать тому порядку, который установлен для обычных вычислений? Вот, допустим, у нас в процессе метавычислений получилось выражение `b(B(a(u1)))`. Т.е. внутренняя функция `a` поработала-поработала и вытолкнула готовую часть своего результата наружу (в виде конструктора `B`). А почему бы сразу не воспользоваться этой информацией? А то функция `b` торчит снаружи, ничего не делает и скучает. А можно ведь и не дожидаться, пока внутренний вызов функции вычислит результат до конца, а сразу начинать обрабатывать ту часть результата, которая «вылезла» наружу. Поэтому

попробуем воспользоваться правилом

```
b(B(x)) = C(b(x));
```

из определения функции `b`, и выполним такое преобразование:

```
b(B(a(u1))) ⟶ C(b(a(u1)))
```

Теперь вынимаем из конструктора его аргумент `b(a(u1))` и видим, что это выражение совпадает с выражением `b(a(u1))`, которое уже встречалось раньше. Значит, можно «зациклить», добавив к графу обратную ссылку. Получается граф конфигураций, показанный на Рис. 5.

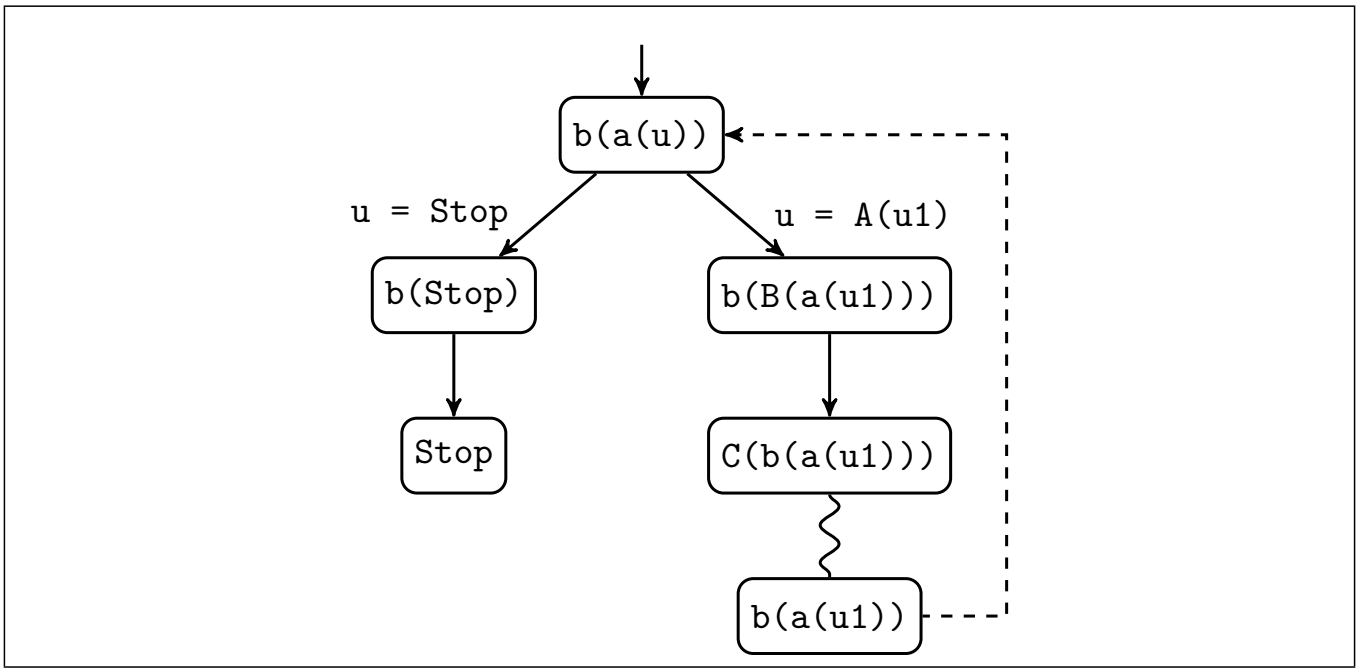

Рис. 5: Суперкомпиляция `b(a(u))` при прогонке снаружи внутрь.

Это уже кое-что! Если из этого графа построить остаточное задание, получается

```
b1(u)
where

b1(Stop) = Stop;
b1(A(x)) = C(b1(x));
```

Видно, что это задание существенно отличается от исходного! Исходная программа реализовывала двухпроходный алгоритм: во время первого прохода все `A` заменяются на `B`, а во время второго прохода все `B` заменяются на `C`. А после суперкомпиляции получается программа, которая делает только один проход по исходным данным, сразу же заменяя `A` на `C`.

Кстати, именно такой результат выдаёт суперкомпилятор SPSC [14], в чем можно убедиться, выполнив задание `compose`.

Итак, получается, что при суперкомпиляции выгодно следовать такой стратегии: даже если при обычном вычислении выражения вычисляются «изнутри наружу», при суперкомпиляции выгодно стараться вычислять выражения «снаружи внутрь». Как только какой-то вызов функции «видит», что появилось достаточно информации для выполнения шага вычислений, можно сразу же этот шаг и выполнить. И тогда суперкомпилятор начинает делать разные интересные вещи, вроде превращения многопроходных алгоритмов в однопроходные.

Однако же сразу возникают и разные нехорошие сомнения и подозрения. Если во время метавычислений придерживаться той же логики, на которой основаны обычные вычисления, то естественно надеяться на то, что суперкомпилятор сгенерирует остаточную программу, эквивалентную исходной. (Хотя, вообще говоря, кто ж его знает? Вопрос тонкий, и, по-хорошему, для каждого конкретного суперкомпилятора необходимо какое-то доказательство того, что он «всё делает правильно».)

Но если обычные вычисления следуют одной логике, а метавычисления — совсем другой, то не получится ли из-за этого какой-нибудь гадости? Например, рассмотрим задание:

```
erase(omega(u))
where

omega(x) = omega(x);
erase(x) = Stop;
```

и попытаемся его исполнить, задав для `u`, например, значение `Nil`.

Если обычное вычисление работает по принципу «изнутри наружу», то процесс исполнения задания зацикливается:

```
erase(omega(Nil)) ⟶ erase(omega(Nil)) ⟶
    erase(omega(Nil)) ⟶ ...
```

Если суперкомпилятор тоже работает по принципу «изнутри наружу», то он изготовит остаточную программу, эквивалентную исходной. А что будет, если суперкомпилятор начнёт выполнять метавычисления «снаружи внутрь»? Тогда во время вычисления получается такая последовательность выражений:

```
erase(omega(u)) ⟶ Stop
```

Ведь «равнодушная» функция `erase` не использует какую-либо информацию о своём аргументе и никак его не использует. А если аргумент не нужен, так зачем же его вычислять?

Кстати, суперкомпилятор SPSC [14] в данном случае порождает остаточное задание

```
Stop
where
```

С точки зрения семантики языка SLL, с которым имеет дело SPSC, остаточное задание полностью эквивалентно исходному заданию `erase_omega`, ибо обычные вычисления в языке являются ленивыми.

Но что получается, если семантика входного языка является «строгой», а не «ленивой»? Тогда выходит, что суперкомпилятор генерирует остаточное задание, которое, мягко говоря (и грубо выражаясь), не совсем эквивалентно исходному. А именно, исполнение исходного задания ***для любых*** исходных данных ***зацикливалось***, а в результате суперкомпиляции получилось задание, исполнение которого ***для любых*** исходных данных ***завершается*** и выдаёт `Stop`.

Тем не менее, многие суперкомпиляторы работают именно так. Например, суперкомпилятор SCP4 [11, 25] обрабатывает программы на языке Рефал (Refal). Рефал — это «строгий» язык, т.е. вызовы функций в нём выполняются «изнутри наружу». Но метавычисления (прогонку) SCP4 выполняет «снаружи внутрь». И остаточная программа не всегда эквивалентна исходной.

Однако дела обстоят не так уж и плохо. Можно доказать, что если суперкомпилятор обрабатывает программу на «строгом» языке, используя стратегию «снаружи внутрь», то остаточная программа всё же эквивалентна исходной на области определения исходной программы. Другими словами, если для некоторых входных данных `X` исходная программа не зацикливается и выдаёт некий результат `R`, то и остаточная программа для исходных данных `X` не зацикливается и выдаёт тот же результат `R`. Но если для некоторых входных данных `X` исходная программа зацикливается (или аварийно завершается), то остаточная программа может сделать всё что угодно: либо тоже зациклиться, либо «упасть» (аварийно завершиться), либо выдать какой-нибудь бред.

Считать ли такое поведение суперкомпилятора «хорошим» или «плохим» - зависит от того, каким способом и для чего мы собираемся использовать суперкомпилятор. Упрощённо говоря, суперкомпиляция может использоваться для двух совершенно разных целей:

- Оптимизации программ (повышение скорости работы и уменьшение размера программ).
- Анализа формальных систем, представленных в виде программ (через выявление и доказательство свойств программ).

Если суперкомпилятор предполагается использовать для ***оптимизации*** программ, то

- Нет возможности выбирать или изменять входной язык суперкомпилятора. Есть некий язык, и требуется программы на этом языке оптимизировать.

- Можно не сохранять поведение исходной программы для тех случаев, когда исходная программа «падает» или зацикливается (если предполагать, что перед запуском остаточной программы есть возможность проверить допустимость данных, подаваемых на вход).

Если суперкомпилятор предполагается использовать для ***анализа*** формальных систем, то ситуация меняется.

- Нет необходимости работать с входным языком, который навязан извне («по историческим причинам» или потому, что «много людей на нём программируют»). Вместо этого, можно использовать входной язык, который (1) обладает большой изобразительной силой и (2) для которого легко и приятно делать суперкомпилятор.

- Желательно, чтобы суперкомпилятор строго сохранял семантику программ.

С первым пунктом всё более или менее понятно. А пункт второй попробую пояснить с помощью конкретного примера.

Допустим, у нас возникло желание доказать эквивалентность двух выражений `A` и `B`. Как это сделать? Можно применить такой «ломовой» способ. Пусть `sc(A)` и `sc(B)` — результат суперкомпиляции выражений `A` и `B` соответственно. И вот, мы сравниваем `sc(A)` и `sc(B)` и видим, что они совпадают (с точностью до «тривиальных различий» вроде переименования переменных). Отсюда строим цепочку заключений:

- `A` «эквивалентно» `sc(A)`,
- `sc(A)` «то же самое, что и» `sc(B)`,
- `sc(B)` «эквивалентно» `B`.

Стало быть, `A` эквивалентно `B`!

Подробнее об этом способе доказательства эквивалентности можно почитать в статье Ключникова и Романенко [7].

Этот способ доказательства — ясный и простой. Но он основан на допущении, что при суперкомпиляции остаточная программа *строго эквивалентна исходной*. А если она «эквивалентна, но не совсем», вся вышеприведённая цепочка рассуждений рассыпается в пыль.

Как же можно преодолеть противоречие между логикой, по которой работают обычные вычисления, и логикой, по которой работает суперкомпилятор? Да очень просто: взять и устранить это различие! Пусть и обычные вычисления выполняются по принципу «снаружи внутрь». (Так и сделано в случае суперкомпиляторов SPSC [6, 24, 14] и HOSC [20, 21, 23, 22].) Зачем преодолевать противоречие, если можно сделать так, чтобы оно просто-напросто исчезло? Как говорили остряки конца 18 века: «Лучшее средство от перхоти — гильотина!».

Однако же такой подход легко осуществим только в том случае, если мы хотим использовать суперкомпиялтор не для оптимизации, а для анализа формальных систем. Если же суперкомпилятор предназначен для оптимизации, то приходится работать со входным языком данным «от Бога», семантику которого «подкрутить» уже невозможно.

Например, суперкомпилятор SCP4 [11, 25] предназначен для обработки программ на Рефале, а в Рефале вызовы функций вычисляются изнутри наружу. Например, рассмотренный выше пример композиции функций `erase` и `omega` на Рефале записывается следующим образом:

```
Omega { e.X = <Omega e.X>; }
Erase { e.X = Stop; }
F { e.U = <Erase <Omega Nil>>; }
```

При обычном исполнении Рефал-программы вызов функции `F` зацикливается. Но если к этой программе применить суперкомпилятор SCP4, то получается остаточная программа

```
F { e.U = Stop; }
```

не эквивалентная исходной.

Хорошо это или плохо? Если использовать SCP4 как средство оптимизации программ, то расширение областей определения функций вполне приемлемо. Если же рассматривать SCP4 как инструмент для анализа программ, то удобнее использовать суперкомпилятор, который сохраняет семантику программ.

Например, нам хочется исследовать свойства завершаемости программы. Суперкомпилируем исходную программу и получаем остаточную программу, для которой можем легко показать, что она всегда завершается. Какой вывод мы можем на основании этого сделать по поводу исходной программы? Если суперкомпилятор сохраняет семантику программы, сразу же делаем заключение, что это верно и в отношении исходной программы. А если суперкомпилятор на обладает этим свойством? Тогда мы о завершаемости исходной программы не узнаём НИЧЕГО.

Последний вопрос такой: если мы хотим, чтобы суперкомпилятор строго сохранял семантику программ, должен ли его входной язык быть обязательно «ленивым»? Ответ отрицательный: входной язык не обязательно должен быть «ленивым»! Если постараться, то можно обойтись и без этого. Например, суперкомпилятор, описанный в [4, 5], работает с входным языком, в котором параметры функций передаются по значению, но, тем не менее, сохраняет семантику программ.

Однако, чтобы обеспечить сохранение семантики в случае «строгого» языка, суперкомпилятор должен предпринимать какие-то дополнительные усилия: нужны дополнительные анализы, дополнительные проверки, рассмотрение разных особых случаев. От этого суперкомпилятор усложняется. А если суперкомпилятор используется как средство анализа или верификации, то

«встаёт ребром» вопрос о корректности самого суперкомпилятора. Нужно доказывать теорему, что «всё чисто», что метавычисления правильно имитируют обычные вычисления. Чем «толще» и запутаннее сам суперкомпилятор, тем «толще» и запутаннее будет доказательство его корректности. И тем выше вероятность, что само доказательство корректности будет содержать ошибки...

4 Что лучше: частичные вычисления или суперкомпиляция?

До сих пор, рассматривая суперкомпиляцию, мы всё время её расхваливали. А теперь, для разнообразия, вдруг захотелось её поругать. А для этого нужно сравнить её с чем-то похожим и сказать, что она хуже этого похожего.

А ближе всего к суперкомпиляции находятся «частичные вычисления» (partial evaluation).

Но, поскольку хочется быть хоть и суровым, но справедливым, мы — для начала — частичные вычисления поругаем и только после этого их похвалим!

Начнём с примера, когда суперкомпилятор показывает себя молодцом, а частичный вычислитель, в такой же ситуации, начинает заниматься явными глупостями.

Пусть целые числа представлены в виде `Z`, `S(Z)`, `S(S(Z))`, ... Рассмотрим программу

```
f(u) = g(u, Z);
g(Z, y) = y;
g(S(x), y) = g(x, S(y));
```

Функция `f` является «тождественной» в том смысле, что если на вход подать число `u`, то `f (u)` выдаст `u`.

Если подвергнуть `g(u, Z)` прогонке, то (на одной из ветвей дерева) получается бесконечная последовательность термов:

$$\texttt{g(u, Z)} \longrightarrow \texttt{g(u1, S(Z))} \longrightarrow \texttt{g(u2, S(S(Z)))} \longrightarrow \ldots$$

Если в суперкомпилятор вделан «свисток», основанный на гомеоморфном вложении, то суперкомпилятор замечает, что `g(u, Z)` гомеоморфно вложено в `g(u1, S(Z))`. Или, говоря по-простому, если в `g(u1, S(Z))` подтереть конструктор `S`, то получится g(u1, Z), что (с точностью до имён переменных) совпадает с `g(u, Z)`.

Поэтому суперкомпилятор делает обобщение и зацикливание. Можно убедиться в этом на примере суперкомпилятора, пропустив через суперкомпилятор SPSC [14].

А в случае частичного вычисления получается другая картина. В частичных вычислителях основная идея — вычислять все константные выражения,

какие можно, т.е. «распространять константы». Но не локально — а глобально, ради этого раскрывая in-line вызовы функций или генерируя специализированные версии функций для разных значений аргументов. Но, в отличие от суперкомпиляции, конфигурации из вложенных вызовов функций при этом не рассматриваются.

Если частичный вычислитель — типа «offline», то он сначала размечает программу, чтобы узнать, какие подвыражения и параметры функций будут **заведомо** известны. Для этого символически выполняются операции над данными, а данные делятся на две категории `S` и `D`. `S` — известные данные, а `D` — неизвестные.

Основная идея «проста как лапоть»:

```
S + S = S
S + D = D
D + S = D
D + D = D
```

В случае простого функционального языка нужно решить, какие из параметров функций будут «статическими» (`S`), а какие — динамическими (`D`). А тип всех остальных конструкций однозначно определяется на основе типов аргументов.

В случае нашего примера есть две функции `f` и `g`. Для `f` возможны две разметки: `f(S)` и f(D). А для `g` — 4 разметки: `g(S, S)`, `g(S, D)`, `g(D, S)`, `g(D,D)`.

Какую-то корректную разметку для программы найти можно всегда: просто везде выставляем `D` — и разметка корректна. Но хорошая разметка — это такая, в которой почаще встречается `S`. Поэтому делается так: в качестве значений всех параметров функций везде выставляем `S` (и эта разметка может быть некорректной). За исключением входных параметров программы, для некоторых из которых может быть выставлено `D`. А потом итеративно начинаем распространять `D` по программе, от входа — и по всей программе.

Когда процесс стабилизируется (достигается «неподвижная точка»), получается корректная разметка: `D` «испортили» уже всё, что смогли.

Применяем этот принцип к нашему примеру.

Для начала перепишем программу в такой форме, которая является более «естественной» с точки зрения большинства частичных вычислителей (и заодно перейдём к «нормальным» целым числам, которые для частичного вычислителя приятнее):

```
f(u) = g(u, z);
g(x, y) = if x == 0 then y else g(x-1, y+1);
```

Для примера будем считать, что нам неизвестно, что будет на входе программы, и задаём начальную разметку: f(D), g(S,S).

```
  D      D  S
f(u) = g(u, z);
  S  S       S           S        S    S
g(x, y) = if x == 0 then y else g(x-1, y+1);
```

Видно, что в одном месте g вызывается с первым параметром, имеющим пометку D. Меняем разметку параметров g с g(S,S) на g(D,S). Получается

```
  D      D  S
f(u) = g(u, z);
  D  S       D           S        D    S
g(x, y) = if x == 0 then y else g(x-1, y+1);
```

И на этом разметка стабилизируется! Везде S получается только из S. Поэтому — разметка «корректна».

Теперь надо принять решение, раскрывать ли вызовы функции g или генерировать её специализированные версии? Не раскрывать — это более осторожная тактика, применим её. Тогда частичный вычислитель действует так. У f разметка не содержит S: f(D). Поэтому в остаточной программе генерируется только одна версия функции f. А разметка функции g содержит S: g(D,S). Это означает, что в процессе метавычислений второй аргумент g будет известен, и что для разных констант k будут сгенерированы специализированные версии g, такие, что g_k(x) = g(x, k). Принцип простой: если где-то получился вызов вида g(x,k), заменяем его на g_k(x) и генерируем определение функции g_k. При этом, с технической точки зрения, можно откладывать переименование вызовов g(x, k) в g_k(x), делая его не сразу, а посредством отдельного прохода по программе. Т.е. просто считается, что значения S-параметров — это «часть имени функции».

А начинается процесс с генерации определения для входной функции f.

Получается так (после вычисления всех подвыражений, содержащих только S-переменные, значение которых известно):

```
f(u) = g(u, 0);
g(x, 0) = if x == 0 then 0 else g(x-1, 1);
g(x, 1) = if x == 0 then 1 else g(x-1, 2);
g(x, 2) = if x == 0 then 2 else g(x-1, 3);
g(x, 3) = if x == 0 then 3 else g(x-1, 4);
...
```

Или, после переименования функций,

```
f(u) = g0(u, 0);
g0(x) = if x == 0 then 0 else g1(x-1);
g1(x) = if x == 0 then 1 else g2(x-1);
g2(x) = if x == 0 then 2 else g3(x-1);
g3(x) = if x == 0 then 3 else g4(x-1);
...
```

Генерируется бесконечная программа, которая для каждого `k` содержит определение функции вида

$$\texttt{g}_k\texttt{(x) = if x == 0 then k else g}_{k+1}\texttt{(x-1);}$$

Эта проблема характерна практически для всех частичных вычислителей (особенно — для самоприменимых). Но стоит не так остро для «специализации в процессе исполнения» (run-time specialization), поскольку можно не генерировать специализированные версии функций «про запас», а по мере надобности. А понадобиться могут не все варианты.

Из сказанного может сложиться ложное впечатление, что частичные вычислители, по сравнению с суперкомпиляторами, — это что-то ущербное. Зачем вообще заниматься частичными вычислениями, если суперкомпиляция «круче»? Но это — не совсем так.

Класс программ, с которыми успешно «справляются» частичные вычислители, — некоторое подмножество, по сравнению с классом программ, с которыми можно делать что-то интересное с помощью суперкомпиляции.

Однако, если частичный вычислитель с какими-то программами всё же справляется, он делает это гораздо лучше, чем суперкомпилятор!

Разница здесь примерно такая же, как между лошадью и автомобилем. Если нужно ехать по шоссе, то автомобиль — быстрее лошади, и грузоподъёмность у него больше. Но если требуется проехать, например, по горной тропе...

Особенно сильно преимущества частичных вычислений перед суперкомпиляцией проявляются в случае, если генерация остаточной программы идёт в два этапа:

- Строится разметка программы, классифицирующая все переменные и операции на «статические» (которые можно будет выполнить) и на «динамические» (которые «выпадут в осадок», т.е. останутся в остаточной программе).

- Выполняется генерация остаточной программы в соответствии с разметкой. При этом всё, что помечено как «статическое», быстро и «бездумно» вычисляется.

Первое преимущество такого подхода очевидно: **генерация остаточной программы идёт быстрее**.

Второе преимущество является, может быть, ещё более важным. Это — **предсказуемость и стабильность результатов**. Ибо уже после стадии разметки становится понятно, какие именно части исходной программы исчезнут, а какие — «выпадут» в остаточную программу, и какую именно структуру будет иметь остаточная программа.

А слабости суперкомпиляции являются «оборотной стороной» её силы. Суперкомпилятор «слишком умён», а значит — менее предсказуем! Результаты его применения в конкретной ситуации могут быть весьма неожиданными... Кстати, одно из определений «искусственного интеллекта» так и звучит: если результаты работы программы предсказуемы, значит, «интеллекта» в ней нет (она просто что-то «вычисляет» — и всё), а если результат предсказать невозможно — значит, «интеллекта» в ней «навалом»...

Понятно, что если метавычисления используются для анализа программ (т.е. выявления и доказательства их свойств), то суперкомпилятор — это «самое что надо». Для того и анализируем, чтобы узнать то, что заранее неочевидно. А если метавычисления используются для специализации программ (или, говоря по-простому, оптимизации программы под конкретные условия её выполнения), то желательна предсказуемость.

Есть и ещё одна интересная ситуация, когда проявляется преимущество (двухэтапных) частичных вычислений перед суперкомпиляцией: когда мы хотим **применить специализатор программ к самому себе**! Например, если у нас есть специализатор `spec` и интерпретатор `int`, то можно преобразовать интерпретатор в компилятор, вычислив `spec(spec,int)`. Это — «вторая проекция Футамуры» (которая была независимо открыта и Турчиным, хотя и немного позже).

Чем интересно выражение `spec(spec,int)`? Да тем, что специализатору предлагается «заглянуть в свою собственную душу» и разобраться в своём собственном поведении по отношению к самому себе. Здесь возникает некоторое противоречие: мудрецу-то легко разобраться в душе простака, но для простака душа мудреца — потёмки. Вот и получается, что если специализатор слишком простодушен, то самоприменять его бесполезно, поскольку сам в себе он разобраться не в состоянии. А если он слишком умён — то он тоже не способен разобраться в себе, но уже из-за своей собственной сложности.

Таким образом, для успешности самоприменения нужно «проползти в игольное ушко»: специализатор должен быть не слишком умён, но и не слишком глуп. И оказалось, что «интеллекта» в самый раз как раз у двухфазных частичных вычислителей (делающих предварительную разметку программы).

Подробнее об одном из таких частичных вычислителей, Unmix [1], можно почитать в работах [26, 12, 13].

[1] `https://github.com/annenkov/unmix`

5 Заключение

На основе вышеизложенного мы приходим к следующим выводам.

1. Алгоритм обобщения конфигураций, основанный на отношении гомеоморфного вложения, позволяет получать нетривиальные результаты суперкомпиляции, обеспечивая при этом её завершаемость.

2. В случае языков с передачей параметров по имени (а не по значению) упрощаются как алгоритм суперкомпиляции, так и обоснование его корректности.

3. Суперкомпиляция и частичные вычисления являются родственными методами, имеющими пересекающиеся, но разные области применения.

Список литературы

[1] Brady E. Idris — A Language with Dependent Types. — 2018.

[2] Dershowitz N., Jouannaud J.-P. Rewrite Systems // Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics / Ed. by J. van Leeuwen. — 1990. — P. 243–320.

[3] Higman G. Ordering by divisibility in abstract algebras // Proceedings of the London Mathematical Society. — 1952. — Vol. 3, no. 2. — P. 326–336.

[4] Jonsson P. A., Nordlander J. Positive Supercompilation for a Higher Order Call-by-value Language // Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '09. — New York, NY, USA : ACM, 2009. — P. 277–288.

[5] Jonsson P. A., Nordlander J. Positive Supercompilation for a Higher-Order Call-By-Value Language // Logical Methods in Computer Science. — 2010. — Aug. — Vol. Volume 6, Issue 3.

[6] Klyuchnikov I. G., Romanenko S. A. SPSC: a Simple Supercompiler in Scala // International Workshop on Program Understanding (PU 2009). — Novosibirsk : A. P. Ershov Institute of Informatics Systems, 2009.

[7] Klyuchnikov I. G., Romanenko S. A. Proving the Equivalence of Higher-Order Terms by Means of Supercompilation // Perspectives of Systems Informatics / Ed. by Amir Pnueli, Irina Virbitskaite, Andrei Voronkov. — Vol. 5947. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. — P. 193–205.

[8] Kruskal J. B. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture // Transactions of the American Mathematical Society. — 1960. — no. 95. — P. 210—-225.

[9] Leuschel M. Homeomorphic Embedding for Online Termination of Symbolic Methods // The Essence of Computation: Complexity, Analysis, Transformation / Ed. by Torben Æ. Mogensen, David A. Schmidt, I. Hal Sudborough. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 379–403. — ISBN 978-3-540-36377-4.

[10] Leuschel M., Martens B. Global control for partial deduction through characteristic atoms and global trees // Partial Evaluation / Ed. by Olivier Danvy, Robert Glück, Peter Thiemann. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. — P. 263–283.

[11] Nemytykh A. P., Pinchuk V. A. Program transformation with metasystem transitions: Experiments with a supercompiler // Perspectives of System Informatics / Ed. by Dines Bjørner, Manfred Broy, Igor V. Pottosin. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. — P. 249–260.

[12] Romanenko S. A. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprizingly Natural and Understandable Structure // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop / Ed. by Dines Bjørner, Andrei P. Ershov, Neil D. Jones. — Amsterdam : North-Holland, 1988. — P. 445–463.

[13] Romanenko S. A. Arity raiser and its use in program specialization // ESOP '90 / Ed. by Neil D. Jones. — Vol. 432. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1990. — P. 341–360.

[14] Romanenko S. A. A Small Positive Supercompiler in Idris. — 2018. — URL: `https://github.com/sergei-romanenko/spsc-idris` (accessed: 25.05.2018).

[15] Sørensen M. H. B. Convergence of Program Transformers in the Metric Space of Trees // Mathematics of Program Construction / Ed. by Johan Jeuring. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. — P. 315–337.

[16] Sørensen M. H. B. Convergence of Program Transformers in the Metric Space of Trees // Science of Computer Programming. — 2000. — Vol. 37, no. 1. — P. 163–205.

[17] Sørensen M. H. B., Glück R. An algorithm of generalization in positive supercompilation // Logic Programming: Proceedings of the 1995 International Symposium / Ed. by John W. Lloyd. — United States : MIT Press, 1995. — P. 465–479.

[18] Turchin V. F. The algorithm of generalization in the supercompiler // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop / Ed. by Dines Bjørner, Andrei P. Ershov, Neil D. Jones. — Amsterdam : North-Holland, 1988. — P. 531–549.

[19] Климов А. В., Романенко С. А. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М. В. Келдыша. — 2018. — № 111. — 36 с. —
URL: http://library.keldysh.ru/preprint.asp?id=2018-111.

[20] Ключников И. Г. Суперкомпилятор HOSC 1.0: внутренняя структура // Препринты ИПМ им. М. В. Келдыша. — 2009. — № 63. — 28 с. —
URL: http://library.keldysh.ru/preprint.asp?id=2009-63.

[21] Ключников И. Г. Суперкомпилятор HOSC 1.1: доказательство завершаемости // Препринты ИПМ им. М. В. Келдыша. — 2010. — № 21. — 27 с. —
URL: http://library.keldysh.ru/preprint.asp?id=2010-21.

[22] Ключников И. Г. Суперкомпилятор HOSC 1.5: гомеоморфное вложение и обобщение для выражений высшего порядка // Препринты ИПМ им. М. В. Келдыша. — 2010. — № 62. — 23 с. —
URL: http://library.keldysh.ru/preprint.asp?id=2010-62.

[23] Ключников И. Г. Суперкомпилятор HOSC: доказательство корректности // Препринты ИПМ им. М. В. Келдыша. — 2010. — № 31. — 28 с. —
URL: http://library.keldysh.ru/preprint.asp?id=2010-31.

[24] Ключников И. Г., Романенко С. А. SPSC: Суперкомпилятор на языке Scala // Программные продукты и системы. — 2009. — № 2. — С. 74–80.

[25] Немытых А. П. Суперкомпилятор SCP4: Общая структура. — М. : Эдиториал УРСС, 2007. — 152 с.

[26] Романенко С. А. Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру. Препринт № 26. — М. : ИПМ им.М.В.Келдыша АН СССР, 1987. — 35 с.

Содержание