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

Препринты ИПМ • Препринт № 111 за 2018 г.

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

Климов А.В., Романенко С.А.

Суперкомпиляция: основные принципы и базовые понятия

Рекомендуемая форма библиографической ссылки: Климов А.В., Романенко С.А. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М.В.Келдыша. 2018. № 111. 36 с. doi:10.20948/prepr-2018-111
URL: http://library.keldysh.ru/preprint.asp?id=2018-111

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

Анд. В. Климов, С. А. Романенко

Суперкомпиляция: основные принципы и базовые понятия

Москва — 2018

Климов Анд. В., Романенко С. А.

Суперкомпиляция: основные принципы и базовые понятия

Дается введение в метод анализа и преобразования программ, называемый суперкомпиляцией. На примерах объясняются основные принципы и базовые понятия суперкомпиляции: прогонка, обобщение и зацикливание. Кратко описана ранняя история суперкомпиляции. Также приведено руководство по использованию простого суперкомпилятора SPSC, доступного через Интернет вместе с коллекцией примеров и открытыми исходными кодами.

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

Andrei Valentinovich Klimov, Sergei Anatolievich Romanenko

Supercompilation: main principles and basic concepts

An introduction to supercompilation, a program analysis and transformation technique, is given. Main principles and basic concepts of supercompilation are explained by examples: driving, generalization and looping back. The early history of supercompilation is briefly described. The use of a simple supercompiler SPSC, provided in the public domain via the Internet, along with a set of examples and open source codes, is also explained.

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 1960-70-е годы зародились почти все основные используемые сейчас методы анализа и преобразований программ. Среди их авторов был и Валентин Федорович Турчин (1931–2010), предложивший метод, названный им «суперкомпиляцией». Этимологии этого термина коснемся в разделе 3. А раннюю историю работ по суперкомпиляции обсудим в разделе 7.

Прошло 40 лет. Доведение суперкомпиляции до практики оказалось не столь легким делом, как сначала ожидал В. Ф. Турчин. Как теперь понятно, одной из сдерживающих причин была слабость компьютеров тех времен. Но с начала 2000-х годов мощности компьютеров стало хватать для проведения самых разнообразных экспериментов по манипулированию программами, и в мире наблюдается взрыв работ в этой области, включая суперкомпиляцию. Теперь всё зависит от разработчиков и распространения идей.

Однако достижения в области суперкомпиляции оказались разбросанными по множеству работ, причем многие из них добавляют к методам очередное улучшение, что непонятно неспециалистам, не знающим предыдущего состояние дел. Работы В. Ф. Турчина тоже читаются с трудом: как первооткрывателю, ему далеко не всегда удавалось сразу определить, чтó в методах, которые он нащупывал, главное, а что — технические детали реализации или особенности данного языка программирования (в его случае — Рефала).

Наблюдая в настоящее время всё поле работ последователей В. Ф. Турчина, можно заметить, что сформировались два направления:

1. Суперкомпиляторы развитых языков, используемых в каких-либо практических областях: в первую очередь, это исходные работы применительно к Рефалу, разработанному самим В. Ф. Турчиным как раз для такого класса задач, и их продолжение А. П. Немытых [32]. За ними последовали экспериментальные суперкомпиляторы для языков Java [7, 8] и Haskell [1, 6, 10].

2. Суперкомпиляторы модельных языков, умышленно и тщательно упрощенные для выделения сущности метода суперкомпиляции «в чистом виде», для проведения исследований по развитию метода, экспериментов и преподавания.

Для изучения и распространения идей особенно важно второе направления. Данная работа — наш посильный вклад в него.

Линия работ по простым суперкомпиляторам началась в начале 1990-х годов со статьи [4]. Вслед за ней пошел всплеск работ в Копенгагенском университете, из которых особенно «влиятельными» оказались дипломная работа Мортена Сёренсена [11] по основам суперкомпиляции и статьи [14, 12]. В России в те же годы существенным вкладом в разработку основ суперкомпиляции стала докторская диссертация С. М. Абрамова, опубликованная

книгой [23]. Ее современное расширенное издание [24, 25] стало основой курса лекций по метавычислениям, запись которых доступна в Интернете[1,2].

В последнее десятилетие линия «простых» суперкомпиляторов стала давать не столь уж «простые» плоды: на них стали разрабатываться и опробываться новые идеи, которых в предыдущих суперкомпиляторах не было. Это суперкомпиляция языка с функциями высшего порядка (использующими новое гомеоморфное вложение) И. Г. Ключникова [29], синтез суперкомпиляции с так называемым «насыщением равенствами» С. А. Гречаником [27] и многие другие работы как в России, так и за рубежом, представленные, в частности, на 5 международных семинарах по метавычислениям имени В. Ф. Турчина в г. Переславле-Залесском[3,4,5,6,7].

Тем не менее, учебных пособий, обзорных работ, изложений методов суперкомпиляции на простых языках и примерах катастрофически не хватает. Есть несколько вводных статей, излагающих основу методов [13, 28, 30], но они останавливаются до рассмотрения основных проблем, без решения которых суперкомпилятор не может стать практическим инструментом.

Данная публикация открывает серию работ, которая, по замыслу авторов, должна заполнить этот пробел. Здесь на примерах вводятся базовые понятия и алгоритмы: прогонка, зацикливание, обобщение. Этим она недалеко уходит от указанных вводных статей. Но в следующих работах рассмотрим методы, которые почти не излагались в аналогичном вводном стиле.

Описанные методы и примеры не будут чисто «бумажными». Соответствующий суперкомпилятор SPCP реализован, причем на нескольких языках и в нескольких версиях, и доступен в Интернете. Приложение А содержит инструкцию по его использованию.

2 Цель и составные части суперкомпиляции

Начнем со взгляда на суперкомпиляцию «с высоты птичьего полета». А детали обсудим в последующих разделах.

Суперкомпиляция — это метод анализа и преобразования программ, предложенный В.Ф. Турчиным [15, 18], в основе которго лежит выполнение следующих действий.

- Делается попытка «выполнить» программу не для конкретных входных данных, а «символически» в «общем» виде, то есть для произвольных входных данных. Ну или для всех входных данных, удовлетворяющих

[1] `https://www.intuit.ru/studies/courses/1067/221/info`
[2] `https://www.intuit.ru/studies/courses/3478/720/info`
[3] `http://meta2008.pereslavl.ru/`
[4] `http://meta2010.pereslavl.ru/`
[5] `http://meta2012.pereslavl.ru/`
[6] `http://meta2014.pereslavl.ru/`
[7] `http://meta2016.pereslavl.ru/`

каким-то ограничениям. Для этого строится «дерево конфигураций» (= «дерево процессов»). В узлах дерева находятся «конфигурации», которые описывают множества состояний вычислительного процесса. Понятно, что эти множества должны быть описаны на каком-то языке, и могут быть не вполне точными («прихватывать» что-то лишнее). А стрелки, связывающие узлы дерева, соответствуют каким-то действиям и проверкам, происходящим при исполнении программы.

- Если исходная программа содержит циклы и/или рекурсию, то дерево конфигураций получается, вообще говоря, бесконечное. Это нехорошо. Что дальше делать с бесконечным деревом? Поэтому в процессе суперкомпиляции делается попытка свернуть бесконечное дерево в конечный «граф конфигураций». Для этого конфигурации сравниваются между собой. Если появляются две похожие конфигурации, делается попытка «свести» ту конфигурацию, что находится в дереве ниже, к той, что появилась выше. В результате в дереве появляется «обратная» стрелка, и дерево превращается в граф с циклами.
- Построенный конечный граф конфигураций превращается в «остаточную» программу. Название «остаточная» связано с тем, что не все части исходной программы «выпадают в осадок». Некоторые из них могут просто исчезать в результате суперкомпиляции (например, если они недостижимы при заданных ограничениях на входные данные).

3 Смысл терминов «суперкомпиляция» и «суперкомпилятор»

Под «суперкомпилятором» обычно понимают некую систему анализа и преобразования программ, основанную на суперкомпиляции. Другими словами, не бывает «суперкомпиляторов», не использующих суперкомпиляцию. Но авторы суперкомпиляторов имеют полное право использовать в своих сооружениях не только суперкомпиляцию как таковую, но и любые дополнительные методы анализа, преобразования и оптимизации программ. (И обычно так и поступают.)

Сам термин «суперкомпиляция», может быть, и не очень хорош в силу своей двусмысленности. «Супер» может означать «крутой и могучий» («супермен» = «сверхчеловек»), а может означать «тот, кто находится сверху и присматривает» («супервизор» = «надсмотрщик»). Когда придумывался термин «суперкомпилятор», имелась в виду не «могучесть», а «присмотр»...

А теперь постараемся спуститься с сияющих абстрактных вершин к чему-то более конкретному и попытаемся разобраться, как работает суперкомпиляция, на конкретных примерах.

4 Пример суперкомпиляции: сложение чисел

Допустим, что программы, с которыми мы будем иметь дело, написаны на простом функциональном языке SLL. Что это за язык, будет понятно из примеров, но, при желании, описание языка можно посмотреть в Приложении А.

4.1 Сложение чисел Пеано

Как известно, если в нашем распоряжении есть ноль и есть операция прибавления единицы, функцию сложения целых неотрицательных чисел можно определить с помощью двух правил:

```
add(0, y) = y;
add(x + 1, y) = add(x, y) + 1;
```

Однако в «учебном» языке SLL все обрабатываемые данные являются деревьями, построенными с помощью «конструкторов», и числа не являются заранее предопределенным понятием. Поэтому будем считать, что неотрицательные целые числа являются «числами Пеано», которые представлены следующим образом. Ноль — как `Z` (где `Z` — это конструктор без аргументов), а число, следующее за числом `n`, — как `S(n)` (где `S` — это унарный конструктор). Таким образом, числа

0, 1, 2, 3, ...

представляются как

```
Z, S(Z), S(S(Z)), S(S(S(Z)))...
```

Теперь программа сложения приобретает вид:

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

В таком виде программа уже ничего не знает о свойствах чисел: она просто механически перетасовывает `S` и `Z`.

Попробуем вычислить, например, 1+2. С точки зрения нашей программы, это означает, что нужно преобразовывать выражение

```
add(S(Z), S(S(Z)))
```

до тех пор, пока из него не исчезнут все вызовы функций. А сами преобразования должны выполняться путём применения двух правил, из которых состоит определение функции `add`. Такие преобразования часто называют «редукцией», а сами правила — «правилами редукции». В данном случае получаем такую последовательность преобразований:

```
add(S(Z), S(S(Z))) ⟶ S(add(Z, S(S(Z)))) ⟶ S(S(S(Z)))
```

При этом какое правило когда применять, на каждом шаге определяется однозначно.

4.2 Символические вычисления (редукция)

То, что было до сих пор, — это «обычное» исполнение программы. А теперь мы переходим к «метавычислениям» (= «символическим вычислениям», «прогонке»).

А что будет, если мы попробуем отследить вычисления «в общем виде»? Например, попытавшись «вычислить» `add(S(S(Z)), b)`, где `b` — это некая переменная, значение которой неизвестно? А почему бы и нет? Получается:

```
add(S(S(Z)), b) ⟶ S(add(S(Z), b)) ⟶
    S(S(add(Z, b))) ⟶ S(S(b))
```

Выяснилось, что *для любого* `b`, результатом вычисления `add(S(S(Z)), b)` является `b` к которому прибавлено 2. Блестящий результат!

4.3 Вложенные вызовы функций

Теперь попробуем вычислить «в общем виде» что-нибудь более интересное. Например, $(a + b) + c$. Или, в терминах нашего языка, изучим процесс вычисления выражения `add(add(a, b), c)`.

Здесь мы впервые сталкиваемся с вложенными вызовами функций, и возникает вопрос: в каком порядке вычислять вызовы функций? Ответ зависит от того, с каким языком мы имеем дело: «строгим» или «ленивым». В «строгом языке» разрешается раскрывать вызов функции, только если все её аргументы полностью вычислены. А в «ленивом» языке можно раскрывать вызов функции, как только в аргументах появляется достаточно информации, чтобы стало ясно, какое из правил применимо (не дожидаясь, пока все аргументы полностью вычислятся). Это объяснение весьма приблизительно, но пока что нам хватит и такого.

Итак, будем считать, что язык, с которым мы имеем дело, — «ленивый». И рассмотрим, как будет вычисляться $(1 + 0) + 0$:

```
add(add(S(Z), Z), Z) ⟶ add(S(add(Z, Z)), Z) ⟶
    S(add(add(Z, Z), Z)) ⟶ S(add(Z, Z)) ⟶ S(Z)
```

Как и следовало ожидать, в результате получилось 1. Но в самом процессе вычислений есть что-то не вполне удовлетворительное. А именно, при вычислении выражения вида `add(add(a, b), c)` получается так, что `a` обрабатывается 2 раза. Каждый раз, когда внутренний вызов `add` видит конструктор `S`, он снимает его с `a` и выталкивает наружу. Наружный `add` сразу же замечает это, и проталкивает `S` дальше, на верхний уровень. Получается,

что каждый S обрабатывается два раза. А почему бы его не выталкивать на верхний уровень сразу?

4.4 Прогонка (редукция + разбор случаев)

Теперь попробуем «вычислить» выражение `add(add(a, b), c)` (содержащее переменные!).

Пытаемся раскрыть наружный `add` — не получается, поскольку мешает внутренний `add`. Делать нечего, нужно «подпихнуть» внутренний `add`, чтобы он выдал наружу какую-то информацию. Пробуем раскрыть внутренний `add`. Но и это не можем сделать, поскольку его первый аргумент — переменная `a` (значение которой нам неизвестно). Ну что же, тогда применим ломовой приём, отлично известный ещё из школьного курса математики: рассуждение методом «разбора случаев». Рассмотрим 3 взаимоисключающих случая:

- `a` имеет вид `Z`;
- `a` имеет вид `S(a1)`, где `a1` — это переменная, изображающая нечто, о чём мы пока ничего не знаем;
- `a` — это не `Z` и не может быть представлено в виде `S(`*нечто*`)`.

Третий случай нам не интересен, поскольку в программе для него не предусмотрено никакого правила. И попытка раскрыть вызов `add` с таким аргументом приводит к аварийному завершению программы.

Если `a` имеет вид `Z`, то `add(add(a, b), c)` превращается в `add(add(Z, b), c)`, и мы можем раскрыть внутренний вызов `add`, получив `add(b, c)`. Более кратко это можно записать так:

$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=Z}} \texttt{add(add(Z, b), c)} \longrightarrow \texttt{add(b, c)}$$

Если `a` имеет вид `S(a1)`, мы тоже можем раскрыть внутренний вызов `add`:

$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=S(a1)}} \texttt{add(add(S(a1), b), c)} \longrightarrow \texttt{add(S(add(a1, b)), c)}$$

Возникает интересный вопрос: а как узнать, какие именно подстановки следует применять к выражению? Ответ очень простой: после того, как мы выбрали вызов функции, который хотим раскрыть, следует изучить определение этой функции. Если определение функции содержит десять правил, то получим и десять подстановок. Ведь мы выбираем подстановки не как попало, а так, чтобы каждая подстановка дала возможность применить одно из правил. (Таков ответ, правильный по существу, но приблизительный в деталях, в которые мы пока не будем углубляться.)

Итак, мы применяем подстановку к выражению, чтобы сразу же сделать применимым какое-нибудь правило, и сразу же это самое правило и применяем. Подстановка однозначно определяет, какое правило оказывается применимо, поэтому, чтобы уменьшить количество писанины, в дальнейшем мы будем записывать такое преобразование сокращённо, в виде одного шага:

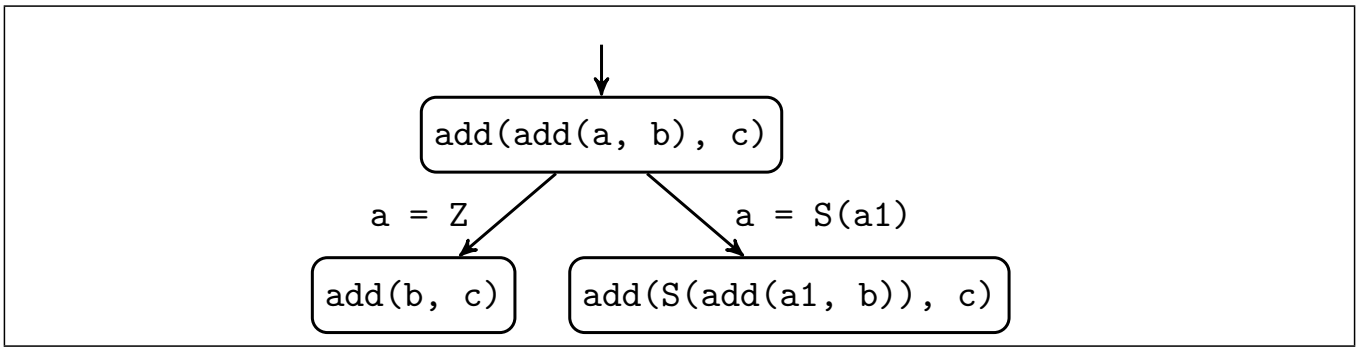

Рис. 1: Граф после прогонки `add(add(a, b), c)`.

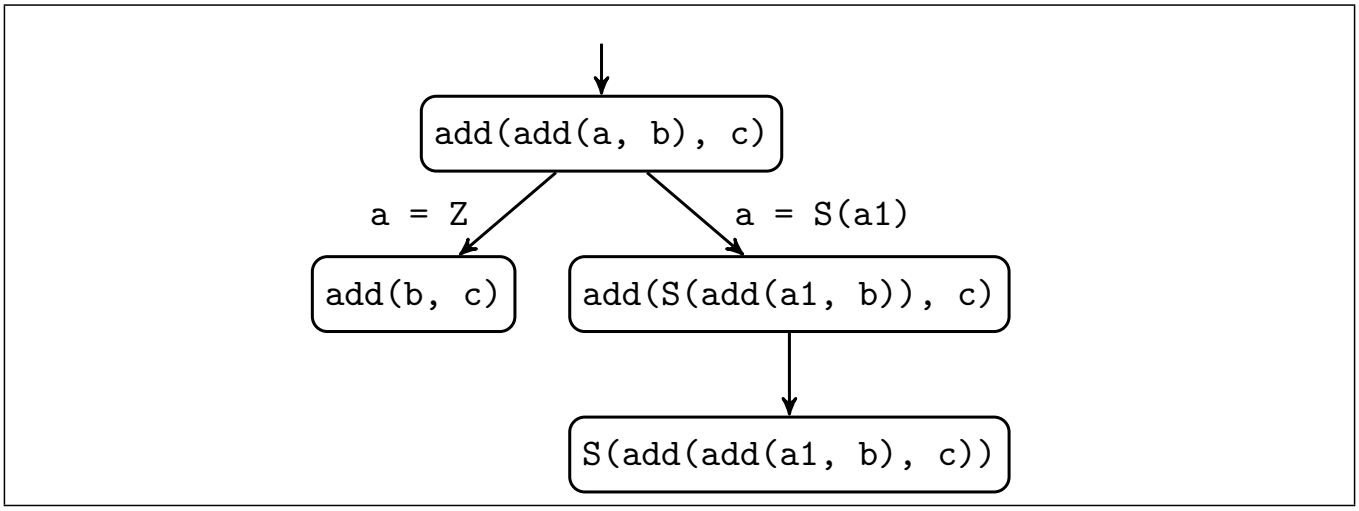

Рис. 2: Граф после прогонки `add(S(add(a1, b)), c)`.

$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=Z}} \texttt{add(b, c)}$$
$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=S(a1)}} \texttt{add(S(add(a1, b)), c)}$$

Ещё более наглядно это можно изобразить в виде графа, показанного на Рис. 1.

Теперь мы видим, что раскрытие внутреннего вызова `add` привело к тому, что вверх «всплыла» некоторая полезная информация в виде конструктора `S`. И теперь имеется достаточно информации, чтобы можно было выполнить раскрытие наружного вызова `add`. При этом применимо только одно правило, то есть разбор случаев не требуется.

$$\texttt{add(S(add(a1, b)), c)} \longrightarrow \texttt{S(add(add(a1, b), c))}$$

В результате получаем граф, показанный на Рис. 2.

4.5 Декомпозиция вызовов конструкторов

Теперь мы видим, что конструктор `S` «всплыл» на самый верх, и ничего интересного с ним происходить уже не будет. А нам следует сосредоточиться

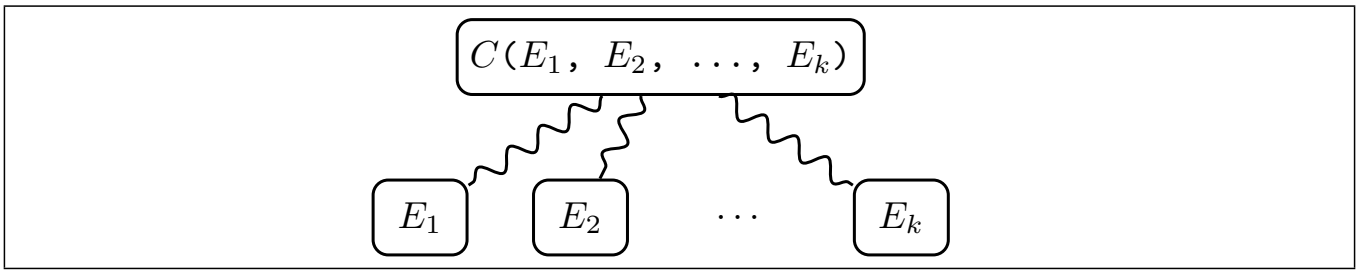

Рис. 3: Извлечение аргументов конструктора.

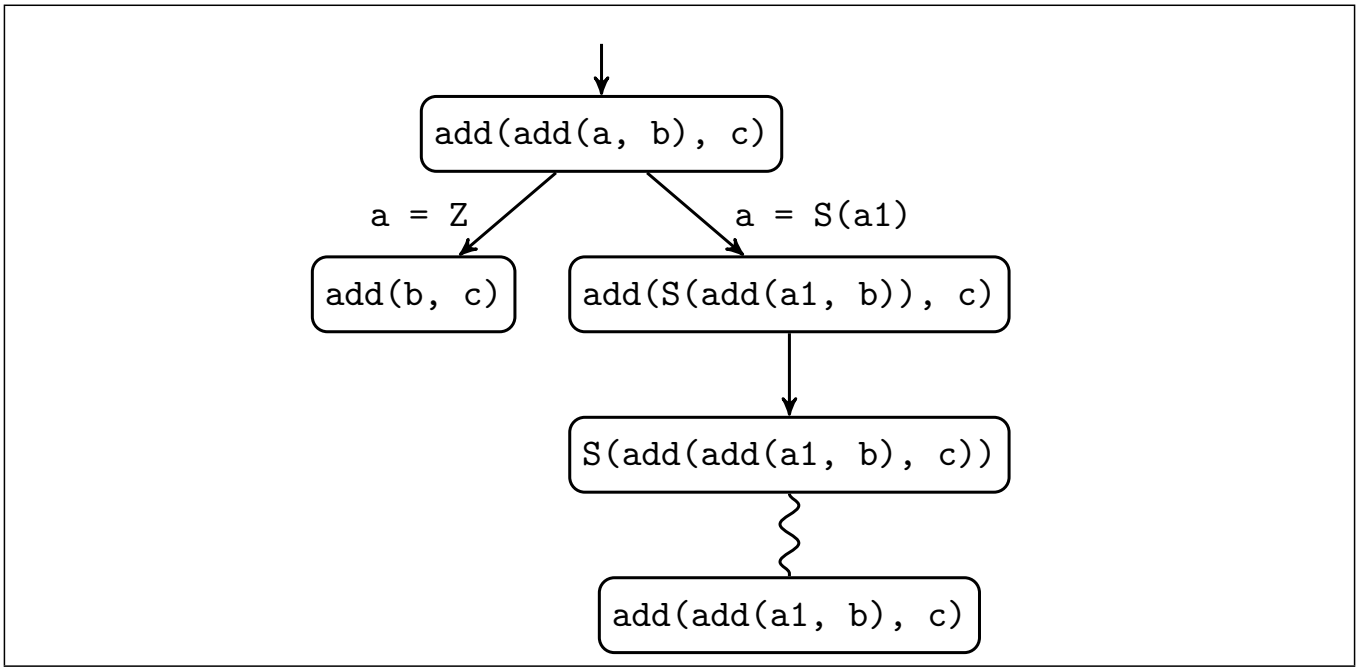

Рис. 4: Граф после декомпозиции `S(add(add(a1, b), c))`.

на анализе его аргумента `add(add(a1, b), c)`. Поэтому мы можем выполнить *декомпозицию* выражения, вытащив аргумент из конструктора, и затем работать только с этим аргументом. Будем записывать это так:

$$\texttt{S(add(add(a1, b), c))} \rightsquigarrow \texttt{add(add(a1, b), c)}$$

или, в общем случае, если некий конструктор C имеет k аргументов, так:

$$C(E_1,\ E_2,\ \ldots,\ E_k) \rightsquigarrow E_1,\ E_2,\ \ldots,\ E_k$$

А при изображении этой операции в виде графа, будем поступать так: под узлом, содержащим $C(E_1,\ E_2,\ \ldots,\ E_k)$, будем подвешивать узлы E_1, E_2, $\ldots$, E_k, как показано на Рис. 3.

Здесь, для наглядности, мы используем волнистые линии. (Однако можно было бы использовать и обычные стрелки, поскольку смысл стрелок всегда однозначно определяется типом узла, из которого они исходят.)

Теперь, после декомпозиции `S(add(add(a1, b), c))`, граф конфигураций (= дерево процессов) принимает вид, показанный на Рис. 4.

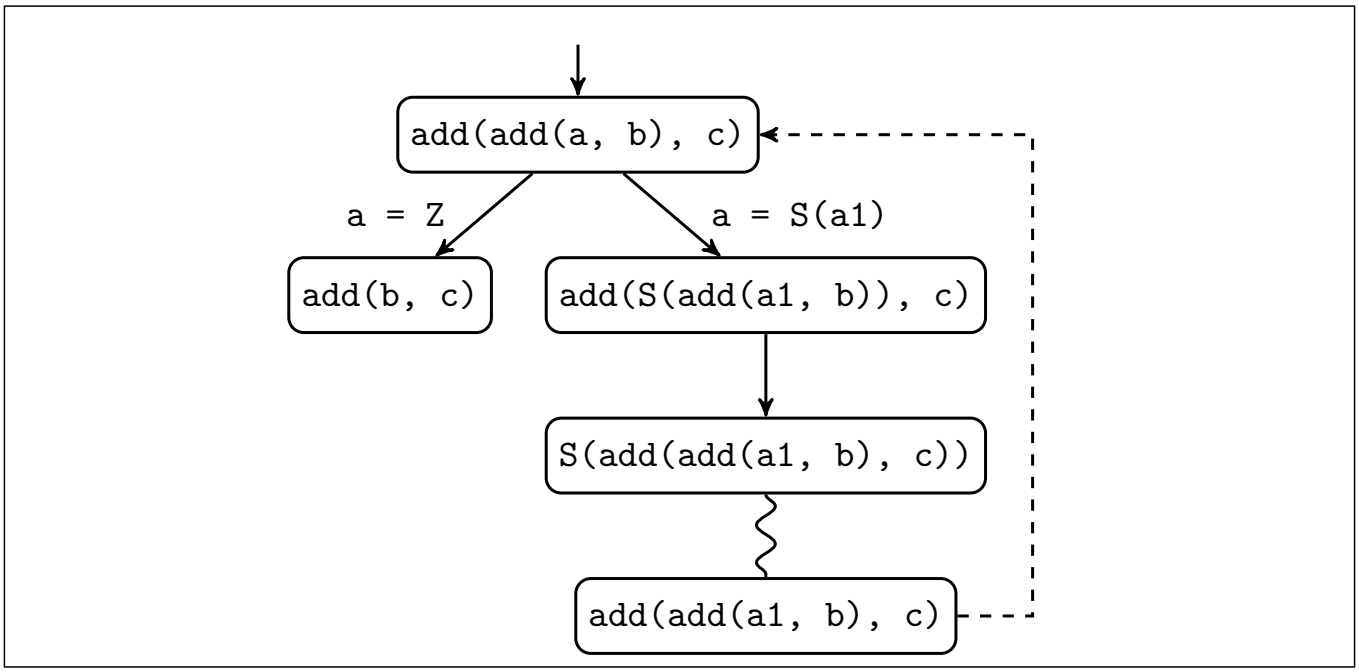

Рис. 5: Результат «зацикливания».

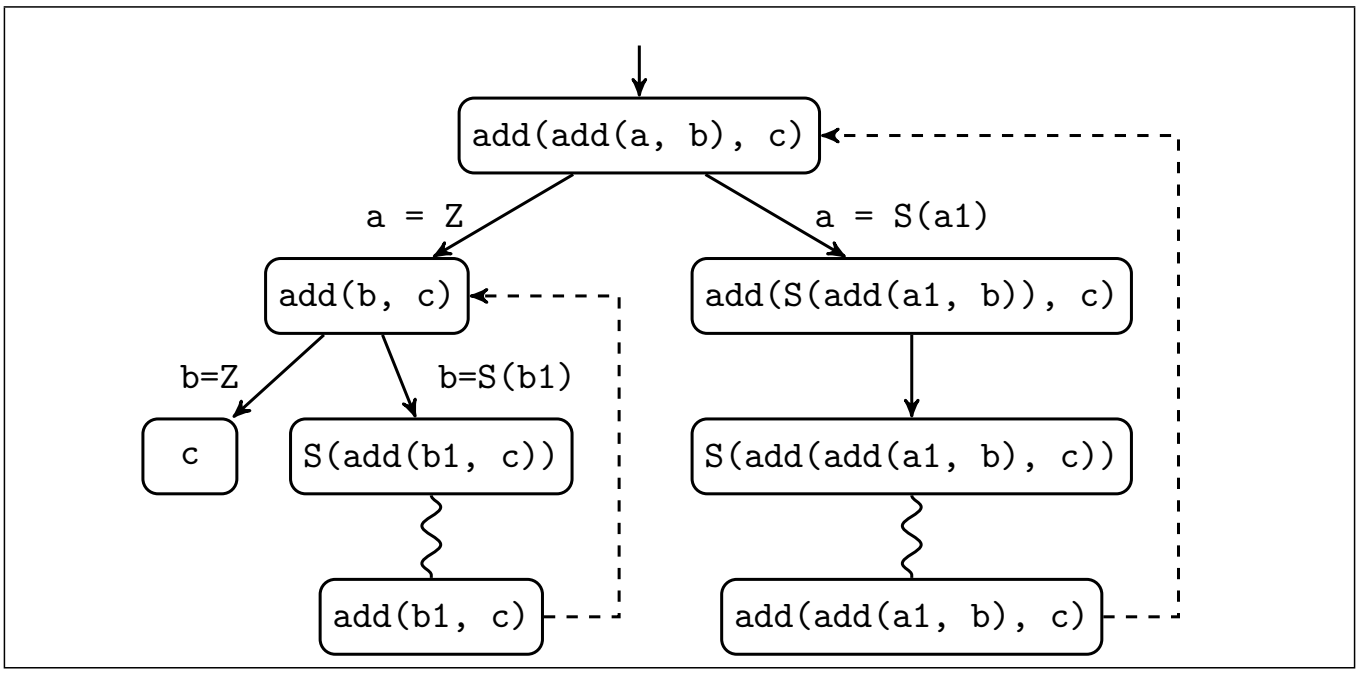

Рис. 6: Окончательное состояние графа.

4.6 Зацикливание (приведение к эквивалентному)

И тут возникает интересная ситуация! Присмотревшись к выражению `add(add(a1, b), c)`, мы видим, что оно совпадает (с точностью до переименования переменных) с исходным выражением `add(add(a, b), c)`. Это означает, что продолжать анализировать `add(add(a1, b), c)` просто глупо, поскольку ничего нового и интересного мы не получим. Всё, что можно было узнать, мы уже узнали при анализе выражения `add(add(a, b), c)`.

Поэтому мы делаем *зацикливание*, добавив к графу «обратную» пунктир-

ную стрелку от `add(add(a1, b), c)` к `add(add(a, b), c)`, после чего оставляем `add(add(a1, b), c)` в покое. В результате получается граф, показанный на Рис. 5.

Теперь нужно заняться выражением `add(b, c)`. Здесь требуется разобрать два случая:

$$\texttt{add(b, c)} \xrightarrow{\texttt{b=Z}} \texttt{c}$$
$$\texttt{add(b, c)} \xrightarrow{\texttt{b=S(b1)}} \texttt{S(add(b1, c))}$$

Извлекаем `add(b1, c)` из `S(add(b1, c))`

$$\texttt{S(add(b1, c))} \rightsquigarrow \texttt{add(b1, c)}$$

После чего видим, что `add(b1, c)` совпадает, с точностью до переименования переменных, с `add(b, c)`. Получается граф, показанный на Рис. 6, который изображает *все возможные пути* вычисления выражения `add(add(a, b), c)` для любых `a`, `b` и `c`.

Интересно, что случаи аварийного завершения вычисления тоже отражены в графе, хотя и неявно. Просто, если в аргументе вызова функции появляется некий конструктор, для которого в графе нет стрелки, считается, что вычисление завершается аварийно.

4.7 Извлечение остаточной программы

Итак, в некоторых случаях удается построить граф конфигураций, который содержит *полную* информацию о возможных путях вычисления. И тогда эту информацию можно превратить в *программу* (конечного размера). Эта программа и будет окончательным результатом работы суперкомпилятора!

Способ генерации программы из графа конфигураций (дерева процессов) довольно прямолинеен.

Каждый узел в графе можно превратить в функцию. Для этого пометим каждый узел в графе неким идентификатором, который будет служить именем функции. Посмотрим, какие переменные появляются в конфигурации. Эти переменные и будут аргументами функции. Например, если для узла придумано имя `g1` и в узле написано `add(add(a, b), c)`, то этому узлу будет соответствовать функция `g1(a, b, c)`.

Теперь надо посмотреть, какие стрелки выходят из узла. Допустим, выходят стрелки, соответствующие проверкам. Например:

$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=Z}} \texttt{add(b, c)}$$
$$\texttt{add(add(a, b), c)} \xrightarrow{\texttt{a=S(a1)}} \texttt{add(S(add(a1, b)), c)}$$

Тогда для каждой из стрелок генерируется правило, выполняющее проверку

```
g1(Z, b, c) = ...
g1(S(a1), b, c) = ...
```

Теперь нужно сгенерировать правые части правил. Для этого надо посмотреть, что находится в тех узлах, на которые направлены стрелки. В случае правила с левой частью `g1(S(a1), b, c)` стрелка идёт в узел, в котором находится выражение `add(S(add(a1, b)), c)`. Допустим, этому узлу присвоено имя `f1`. Тогда ему соответствует функция `f1(a1, b, c)`. Ну так эту функцию и нужно вызвать в правой части правила:

```
g1(S(a1), b, c) = f1(a1, b, c)
```

Теперь займёмся узлом `f1`. Из него выходит только одна стрелка:

$$\texttt{add(S(add(a1, b)), c)} \longrightarrow \texttt{S(add(add(a1, b), c))}$$

и на этой стрелке нет проверки условий. Отлично! Значит, нужно просто вызвать функцию, соответствующую узлу `S(add(add(a1, b), c))`. Допустим, ему присвоено имя `f2`. Стало быть, ему соответствует функция `f2(a1, b, c)`. Получается правило:

```
f1(a1, b, c) = f2(a1, b, c)
```

Понятно, что на самом деле функция `f1` — лишняя, поскольку из `g1` можно было бы вызвать `f2` напрямую, минуя `f1`. Но это уже — оптимизация. А нам сейчас важно разобраться с генерацией программы из графа, так сказать, на идейном уровне.

Теперь надо рассмотреть узел, в котором написано `S(add(add(a1, b), c))`. Ему, как упоминалось выше, соответствует функция `f2(a1, b, c)`. А стрелка, выходящая из этого узла, идёт в узел, в котором находится `add(add(a1, b), c)`, и означает, что нужно вычислить `add(add(a1, b), c)`, а на то, что получится, надеть конструктор `S`. Пусть узлу `add(add(a1, b), c)` присвоено имя `f3` и соответствует функция `f3(a1, b, c)`. Тогда всему вышесказанному соответствует правило:

```
f2(a1, b, c) = S(f3(a1, b, c))
```

И теперь мы можем, наконец, рассмотреть узел `add(add(a1, b), c)`, которому, как сказано выше, соответствует функция `f3(a1, b, c)`. Из этого узла идёт обратная стрелка в узел `add(add(a, b), c)`, соответствующий функции `g1(a, b, c)`. Очень хорошо! Это можно изобразить в виде правила:

```
f3(a1, b, c) = g1(a1, b, c)
```

Как узнать, каким способом следует сформировать аргументы в вызове функции `g1`? Очень просто! Нужно просто наложить друг на друга два выражения

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

и убрать совпадающие части. Вот и получатся два списка переменных:

```
a, b, c
a1, b, c
```

из которых видно, какие переменные каким соответствуют.

Продолжая в том же духе (и удаляя лишние промежуточные функции), получаем остаточную программу:

```
start(a, b, c) = g1(a, b, c);
g1(Z, a, b) = g2(a, b);
g1(S(a), b, c) = S(g1(a, b, c));
g2(S(a), b) = S(g2(a, b));
g2(Z, a) = a;
```

Рассмотренный пример можно пропустить через реально работающий суперкомпилятор SPSC, описанный в Приложении А. Примеры заданий на суперкомпиляцию находятся в папке `tasks` в файлах с расширением `task`. Задание для данного примера — в файле `add_add_a_b_c.task`.

5 Пример: сложение с накоплением

В Разделе 4.1 разбирался пример, в котором всё получалось хорошо и гладко. Как легко догадаться, это свидетельствует вовсе не о том, что суперкомпиляция способна «колоть как орехи» любые проблемы, а о том, что пример был тщательно подобран.

Поэтому, чтобы компенсировать допущенную «нечестность» и «необъективность», давайте разберём пример, в котором суперкомпиляция сталкивается с некоторыми проблемами. Некоторые из этих проблем можно решать с помощью «обобщения» конфигураций.

5.1 Сложение чисел с накоплением результата

Итак, рассмотрим такую забавную функцию:

```
addAcc(0, n) = n;
addAcc(m + 1, n) = addAcc(m, n + 1);
```

Эта функция выдаёт результат сложения двух её аргументов. Но, в отличие от функции `add`, рассмотренной в Разделе 4.1, она долго трудится, ничего

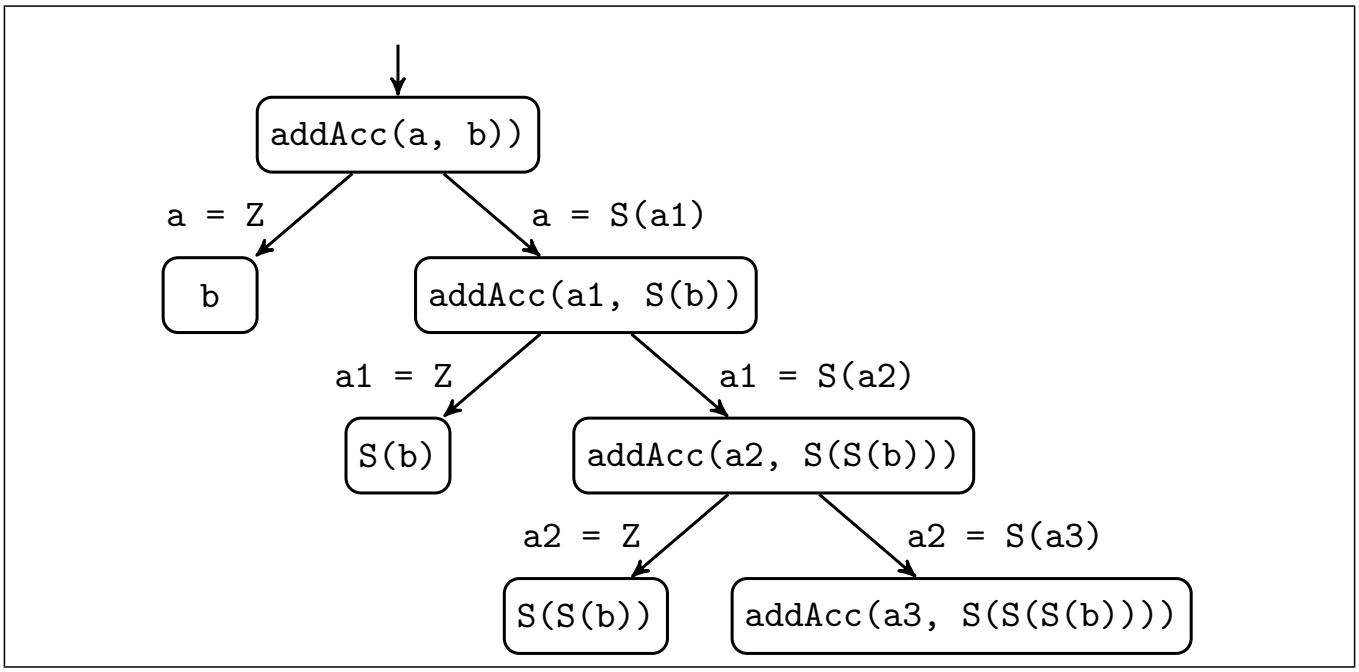

Рис. 7: Дерево процессов для `addAcc(a, b)`.

не выдавая наружу: перекладывает единички из первого аргумента во второй, а потом сразу выдаёт готовый результат. В отличие от функции `add`, выдававшей результат своей работы частями.

Если перейти к представлению натуральных чисел конструкторами `Z` и `S`, определение функции `addAcc` принимает вид:

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

Теперь попробуем просуперкомпилировать `addAcc(a, b)`. Получается дерево процессов, показанное на Рис. 7.

Видно, что это дерево бесконечно, поскольку всё время получаются конфигурации, не совпадающие с предыдущими:

```
addAcc(a, b)
addAcc(a1, S(b))
addAcc(a2, S(S(b)))
addAcc(a3, S(S(S(b))))
...
```

Если и дальше продолжать в таком же духе, то процесс построения дерева никогда не закончится!

С другой стороны, хотя конфигурации и не совпадают, между ними всё же есть и нечто общее. А именно, каждая из конфигураций в последовательности является «частным случаем» предыдущей конфигурации (кроме, разумеется, самой первой из них).

Сейчас постараемся разобраться более точно, что именно означают слова «частный случай».

5.2 Сравнение конфигураций на общность

Начнём с уточнения терминологии. Выражения общего вида, которые могут (но не обязаны) содержать конструкторы, вызовы функций и переменные, мы будем называть «конфигурациями». А выражения, которые могут содержать только конструкторы и вызовы функций (то есть не содержат переменных), мы будет называть «рабочими выражениями». Таким образом, рабочие выражения — это вырожденные конфигурации, не содержащие переменных.

Можно считать, что каждое рабочее выражение изображает некоторое «конкретное» состояние вычислительного процесса. А каждая конфигурация может рассматриваться как изображение некоторого множества рабочих выражений или, другими словами, возможных состояний вычислительного процесса. А именно, берём конфигурацию, подставляем вместо её переменных всевозможные рабочие выражения и получаем разные рабочие выражения из множества, изображаемого конфигурацией.

А если конфигурация не содержит переменных — то считаем, что она изображает множество из одного элемента: рабочего выражения, совпадающего с самой конфигурацией.

Можно подойти к делу и с другого конца. Допустим, у нас есть некая конфигурация X и некое рабочее выражение A. Можно ли проверить, принадлежит ли A к множеству, изображаемому X? Или, другими словами, можно ли переменные в X заменить на такие рабочие выражения, что X после этого совпадёт с A? Поиск таких значений переменных часто называют «сопоставлением A с образцом X».

Найти подходящие значения переменных (если таковые существуют) можно, наложив A и X друг на друга. Например, сопоставим `addAcc(S(Z), S(Z))` с `addAcc(a1, S(b))`. Удалим те части двух выражений, которые попарно совпадают, и сразу становится видно, что `S(Z)` накладывается на `a1`, а `Z` на `b`. Или, выражаясь более формально:

$$A = X\ \texttt{\{a1 := S(Z), b := Z\}}$$

где X `{a1 := S(Z), b := Z}` — это результат применения к X подстановки `{a1 := S(Z), b := Z}`. Впрочем, для подстановок часто применяют и другие обозначения, например, такое: `[S(Z)/a1, Z/b]`. Как говорится, «на вкус и цвет товарищей нет»...

Теперь рассмотрим две конфигурации X_1 и X_2. Предположим, что конфигурацию X_1 можно превратить в конфигурацию X_2, заменив переменные, входящие в X_1, на некоторые выражения (которые могут содержать переменные). Или, говоря формально,

$$X_1\ S = X_2$$

где S — некоторая подстановка. В этом случае будем говорить, что X_2 является частным случаем X_1, и будем записывать это как $X_1 \Subset X_2$. Знак $\Subset$

(похожий на $\leq$) здесь используется из-за того, что X_1 не может быть «толще», чем X_2. Ведь переменные в X_1 заменяются либо на переменные, либо на что-то более «массивное». Например,

```
addAcc(a1, S(b)) ⋐ addAcc(a2, S(S(b)))
```

поскольку

```
addAcc(a1, S(b)) {a1 := a2, b := S(b)} ⋐ addAcc(a2, S(S(b)))
```

Можно доказать такую теорему: если $X_1 \Subset X_2$, то множество рабочих выражений, изображаемых X_2, полностью покрывается множеством рабочих выражений, изображаемых X_1. В этом смысле, X_2 и является частным случаем X_1, то есть X_1 «накрывает» X_2.

А теперь мы можем вернуться к графу конфигураций, который рассматривали в начале. Теперь мы видим, что при построении дерева у нас получилась последовательность конфигураций

```
addAcc(a, b) ⋐ addAcc(a1, S(b)) ⋐ addAcc(a2, S(S(b))) ⋐ ...
```

в которой каждая последующая является частным случаем предыдущей.

Поэтому возникает такая идея: если в процессе суперкомпиляции нам сначала попадается конфигурация X_1, а затем — конфигурация X_2, которая является частным случаем X_1, то зачем изучать дальнейшее течение вычислительного процесса для X_2? Всё, что включено в X_2, входит и в X_1, а все пути вычислений, выходящие из X_1, мы и так изучим. Поэтому нельзя ли как-то привести X_2 к такому виду, чтобы она совпала с X_1 (с точностью до имён переменных) и провести в графе «обратную» стрелку от X_2 к X_1?

5.3 Обобщение конфигураций

Сведение X_2 к «более общей» конфигурации X_1 в процессе суперкомпиляции называется «обобщением X_2 до X_1». А именно, заменяем X_2 на конструкцию

```
let v1 = e1, ..., vk = ek in e0
```

где e_0 `{`v_1 `:=` e_1`, ...,` v_k `:=` e_k`}` `=` X_2, а e_0 совпадает с X_1 с точностью до переименования переменых. При этом в качестве $v_1, \ldots, v_k$ выбираются такие переменные, которые нигде больше не используются в дереве процессов (чтобы случайно не возникло путаницы с другими переменными).

Выглядит всё это устрашающе, но суть дела проста. Поскольку $X_1 \Subset X_2$, то просто накладываем X_1 на X_2 и смотрим, какие куски из X_2 наложатся на переменные из X_1. Если X_1 содержит k переменных, то эти переменные наложатся на какие-то подвыражения $e_1, \ldots, e_k$ из X_2. Придумаем для них свежие уникальные имена $v_1, \ldots, v_k$. После этого извлекаем $e_1, \ldots, e_k$ из X_2 и заменяем их на $v_1, \ldots, v_k$ соответственно. Результат всех этих действий изображаем в виде let-выражения

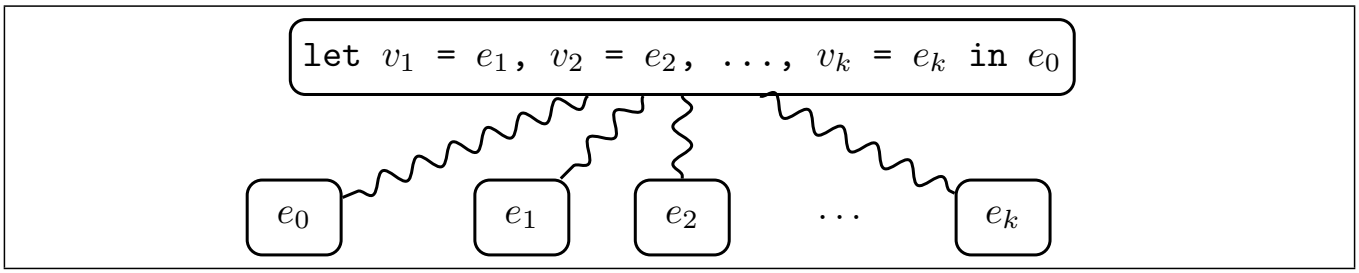

Рис. 8: Декомпозиция let-узла.

`let` v_1 `=` e_1`, ...,` v_k `=` e_k `in` e_0

Например, сопоставляем `addAcc(a, b)` и `addAcc(a1, S(b))` и видим, что

```
addAcc(a, b) { a := a1, b := S(b)} = addAcc(a1, S(b))
```

Отлично! Значит, e_1 и e_2 — это `a1` и `S(b)`. Придумываем для них имена `v1` и `v2`. После чего переписываем `addAcc(a1, S(b))` в виде

```
let v1 = a1, v2 = S(b) in addAcc(v1, v2)
```

Теперь мы видим, что `addAcc(v1, v2)` совпадает с `add(a, b)` с точностью до переименования переменных. Но, чтобы можно было добавить к графу конфигураций обратную стрелку, нужно совершить ещё одно действие: *декомпозицию* или, говоря проще, разваливание let-узла на части, чтобы выражение `addAcc(v1, v2)` оказалось в отдельном узле, который мы и соединим с узлом, в котором находится `addAcc(a, b)`.

При рисовании графов декомпозиция let-узла будет изображаться так, как показано на Рис. 8.

Это очень похоже на декомпозицию узла, содержащего конструктор на верхнем уровне (Раздел 4.5). И, как и в том случае, порядок, в котором расположены стрелки, существен. Но есть и различие: при декомпозиции вызова функции, содержимое узла стирается и заменяется на результат декомпозиции.

Теперь мы можем, в качестве результата суперкомпиляции `addAcc(a, b)`, построить конечный граф конфигураций, изображенный на Рис. 9. Состояние узла до декомпозиции мы показываем на рисунке в виде перечеркнутого узла, но это — не более чем комментарий, поскольку в графе, выдаваемом суперкомпилятором, реально сохраняется только состояние узла после переписывания.

Если из этого графа построить остаточную программу, то оказывается, что она совпадает с исходной программой. Таков, прямо скажем, жалкий результат всех наших мучений и титанических усилий...

Ну что же... Это говорит только о том, что возможности «чистой суперкомпиляции» ограничены. С чем-то она справляется, а с чем-то — нет. А, как уже было сказано, в суперкомпиляторах разрешается использовать не

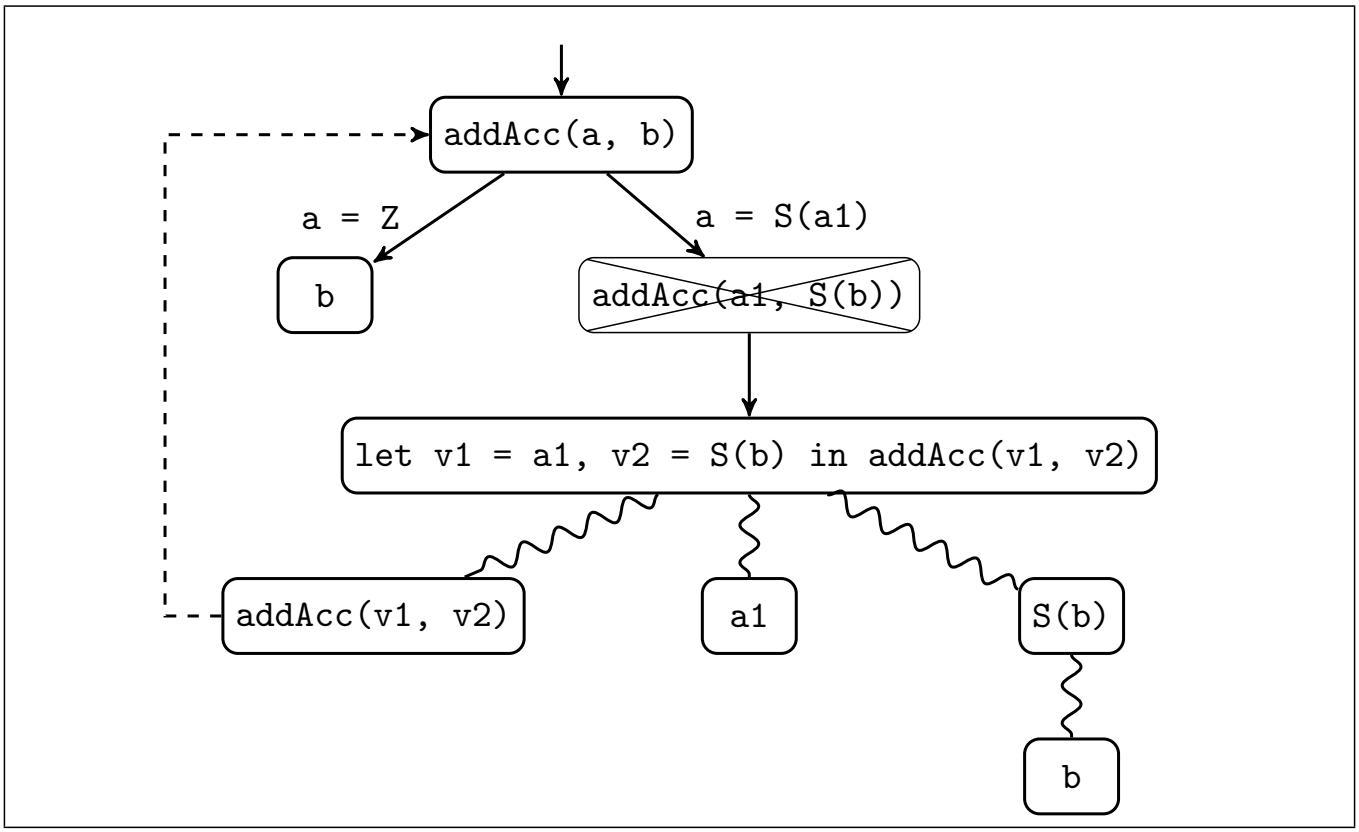

Рис. 9: Результат суперкомпиляции `addAcc(a, b)`.

только суперкомпиляцию как таковую, но и любые дополнительные методы, увеличивающие способности суперкомпилятора.

Можно посмотреть, как справляется с этим примером суперкомпилятор SPSC (см. Приложение A), пропустив через него задания `addAcc_a_b` и `addAcc_a_Z`.

6 Суперкомпиляция как метод специализации программ

Общеизвестной истиной является то, что следует различать *цели* (решаемые задачи) и *средства* (методы решения задач), ибо, с одной стороны, одна и та же задача может решаться различными методами, а, с другой стороны, один и тот же метод может быть полезен при решении многих различных задач.

Суперкомпиляция является *методом*, который может применяться для решения *различных* задач. Одной из таких задач является *специализация программ*.

Таким образом, хотя суперкомпилятор может использоваться в качестве специализатора, не всякий специализатор является суперкомпилятором (поскольку он может быть, например, частичным вычислителем [5]).

6.1 Что такое специализация программ?

Абстрактно задачу, решаемую специализатором программ, можно описать следующим образом.

Пусть P — *исходная* программа, а C — ограничения на условия эксплуатации P. Тогда на вход специализатора подается (P, C), а задача специализатора — породить *остаточную* программу P', удовлетворяющую следующим требованиями.

- P' эквивалентна P, если выполнены условия C.
- P' получается путем устранения из P тех частей и действий, которые становятся ненужными, в результате наложения условий C.

Надежды и ожидания, связанные со специализацией P, состоят в следующем.

- P' будет *эффективнее*, чем P.
- P' будет *меньше*, чем P.
- P' будет *проще*, чем P.

И эти ожидания *иногда* сбываются [5].

6.2 Пример: специализация «интерпретатора»

Рассмотрим следующее задание на суперкомпиляцию

```
eq(S(S(Z)), x)
where

eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(x)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

которое можно найти в папке `tasks` под именем `eq_SD`. Пропустим это задание через суперкомпилятор SPSC (как описано в Приложении А).

```
eqS1(x)
where

eqZ3(Z)=True;
eqZ3(S(v))=False;
eqS2(Z)=False;
eqS2(S(v))=eqZ3(v);
eqS1(Z)=False;
eqS1(S(v))=eqS2(v);
```

Очевидно, что `eq(S(S(Z)), x)` выдает `True` тогда и только тогда, когда `x` равно `S(S(Z))`. Тем же свойством обладает и выражение `eqS1(x)`. Таким образом, `eq` эквивалентна `eqS1` *при условии*, что первый аргумент `eq` равен `S(S(Z))`. При этом `eqS1` — проще, чем `eq` (не содержит рекурсивных вызовов).

С формальной точки зрения, eq можно считать «интерпретатором», первый аргумент которого является «программой», которая предписывает, что интерпретатор должен делать со вторым аргументом.

7 Ранняя история суперкомпиляции

«Откуда есть пошла» суперкомпиляция? Сейчас мы уже как-то привыкли к тому, что всё новое придумывают иностранцы, что они же это новое изготавливают и продают, а мы потом всё это покупаем за нефть. Однако в случае с суперкомпиляцией дело обстоит не так: она была изобретена в России. Ну, точнее, в СССР. Причём придумал её не программист, а физик: Валентин Фёдорович Турчин[8] [33].

Сначала в 1966 году он придумал «метаалгоритмический язык» Рефал, предназначенный для обработки алгоритмов, записанных на каких-то языках программирования [34, 35].

Статью [35] можно найти в оцифрованном виде, а что касается первой статьи [34], то где её можно добыть — неизвестно (хотя когда-то давно авторы и держали её в руках).

Через некоторое время Турчин задумался над такой мыслью: если Рефал, в принципе, годится для обработки программ на любых алгоритмических языках, то, по определению, должен быть пригоден и для обработки программ на нём же самом. (Задним числом эта мысль кажется очевидной.)

Правда, «обработка» программ тоже бывает разная... Например, раскраска ключевых слов в программе — это, можно считать, тоже «обработка» программы. Турчина же заинтересовал особый вид «обработки», вытекающий из идеи «метасистемного перехода». Что понимает Турчин под «метасистемным переходом», можно прочитать в его книге «Феномен науки» [38].

Тема эта — интересная, но очень обширная, поэтому мы сейчас в неё не будем углубляться, а рассмотрим только один частный случай, относящийся к исполнению программ.

Рассмотрим некоторый алгоритмический язык. (Заметим в скобках, что понятие «алгоритмический язык» не совсем совпадает с понятием «язык программирования». Например, язык машин Тьюринга очевидным образом является «алгоритмическим языком», но «языком программирования» его обычно не называют.) Допустим, у нас есть программа на этом языке, которую нам хочется «выполнить», применив к некоторым «исходным данным». (Впрочем, в некоторых языках, например в λ-исчислении, разницы между «данными» и «программой» не существует.)

[8] `http://refal.net/author.html`

Понятно, что «выполнить» программу можно только в том случае, если есть некий субъект, который её «понимает» и умеет выполнять. Этим субъектом может быть «железным» устройством (вроде процессора компьютера), программой или человеком. Для краткости будем называть этого субъекта «интерпретатором».

Итак, берём интерпретатор, программу и исходные данные и всё это запускаем. Интерпретатор начинает задумчиво «жевать» программу и данные. Всю эту (до боли знакомую) ситуацию назовём «основной системой», находящейся на «основном уровне».

А теперь представим, что над основным уровнем есть ещё «метауровень», на котором находится другой субъект, который с большим интересом наблюдает за тем, что происходит в основной системе, и, быть может, что-то с ней делает. Вот это всё и называется «метасистемой».

Ну, а переход от ситуации, когда есть только основная система, к ситуации, когда возникает метасистема, как и следует ожидать, называется «метасистемным переходом».

Имеем ли мы дело с метасистемными переходами в программировании? На самом деле, программисты-практики сталкиваются с ними каждый день. Ну, например, что представляет из себя процесс отладки? Отладчик исполняет программу + данные по шагам, а над ними тяжело дышит программист, который тщетно пытается понять, что происходит. Этот программист и находится на «метауровне» и, стало быть, является «метасистемой» (или её частью). Однако же старая идея «автоматизации программирования» подразумевает, что было бы хорошо, если бы всех программистов можно было повыгонять и заменить на бездушные машины и/или программы.

То есть в идеале поведение одной программы должна была бы изучать другая программа, а не человек. А над этой программой, естественно, можно было бы поставить другую программу (сидящую на мета-мета-уровне), и т.д. Но, как говорится, «гладко было на бумаге, да забыли про овраги». Светлая и завлекательная мечта об «автоматизации программирования» так мечтой и осталась. Хотя кое-что сделать всё же удалось...

Итак, Турчин задумался о том, как можно было бы построить метасистему (в виде программы), наблюдающую за Рефал-программой, обрабатывающей какие-то данные. И вскорости он обнаружил, что на метауровне имеются кое-какие возможности изучать поведение программы «в общем виде», то есть для целых классов входных данных, а не только для какого-то конкретного набора исходных данных (как в случае отладки). Говоря по-простому, обнаружилась возможность перейти от «арифметики» к «алгебре» (в школьном понимании этого слова). Например, то, что $3*(2+1) = 3*2+3*1$, — это факт из арифметики. А то, что для любых a, b и c верно $a*(b+c) = a*b+a*c$, — уже факт из алгебры.

Так появилась система преобразований Рефал-программ, получившая название «прогонка» (driving). Слово «прогонка» появилось из-за того, что «конкретные» данные проходят через процесс вычислений «естественным»

путём, а вот в случае «обобщённых» данных, изображающих целые классы «конкретных» данных, дело обстоит хуже: их приходится пропихивать через процесс вычислений буквально пинками, для которых, кстати, в прогонке используются прямо-таки садистские названия: «сужение» и «расщепление». Сразу на ум приходит сцена в духе «Преступления и наказания»: стоит на метауровне субъект с топором и то обтёсывает «обобщенные данные», то расщепляет...

Прогонка была описана Турчиным в статьях [36, 37], опубликованных в 1972 и 1974 году. В них описана система преобразований Рефал-программ (прогонка), а также её применение для решения «обратных» задач. А именно, допустим, что на Рефале описана некая функция f, которая для любого входного x выдаёт `True` или `False` (если завершается). И вот Турчин показывает, что с помощью прогонки можно решать «обратную задачу»: подбирать для f такие значения x, что $f(x) =$ `True`.

С точки зрения чистой теории в этом нет ничего удивительного, поскольку известно, что область определения любой рекурсивной функции является рекурсивно-перечислимой. Интерес был в том, что прогонка позволяла искать решения не тупым полным перебором, а вполне разумным способом. Например, в статьях рассматривается вопрос о решении уравнений вида $a + x = b$, где a и b — натуральные числа, представленные в двоичной системе счисления, а сложение определено как сложение «в столбик» в виде функции на Рефале. И получилось, что поиск x с помощью прогонки делается не перебором, а вычитанием «в столбик». Получалось, что компьютер, получив алгоритм сложения в виде программы, автоматически находил разумный алгоритм вычитания.

Нужно сказать, что описанное в статьях было тогда же реализовано в виде программ на Рефале (хотя в самих статьях это и не отражено). Турчин написал реализацию прогонки на Рефале, а «хождение на машину» (как это тогда называлось) и отладку поручил (или, скажем более точно, доверил ☺ С. А. Романенко). К слову, программы тогда набивались на перфокартах[9], а «машиной» была могучая БЭСМ-6[10], которая, правда, несколько уступала американской CDC-6600[11]. Впрочем, в настоящее время, и БЭСМ-6 и CDC-6600, с точки зрения вычислительной мощности, выглядят как какие-то карлики по сравнению со смартфоном в кармане читателя...

Как бы то ни было — всё заработало по теории: обратные задачи решались... А решения печатались на широченных лентах бумаги.

Однако же, если внимательно вчитаться в статью [36] 1972 года, то у тех, кто знаком с Прологом, наверняка возникнет ощущение чего-то знакомого... Ну да, «обобщённое отождествление» — это «унификация». Стало быть, Турчин ничего нового не придумал: просто взял Пролог, перекрасил и выдал за своё. Но это не совсем так. Или, точнее, совсем не так.

[9] `https://en.wikipedia.org/wiki/Punched_card`

[10] `http://www.computer-museum.ru/histussr/28-1.htm`

[11] `http://en.wikipedia.org/wiki/CDC_6600`

Ведь, как утверждают сведущие люди, язык Пролог был задуман группой Алена Колмероэ в Марселе, Франция, в начале 1970-х годов, а первая Пролог-система был разработана в 1972 году Колмероэ с Филиппом Руселем[12].

То есть в 1972 году Колмероэ сидел и придумывал Пролог как раз в то самое время, когда Турчин придумывал прогонку. И, естественно, они друг о друге они ничего не знали.

Также есть и различия на содержательном уровне:

- Пролог основан на «унификации», а унификация симметрична по отношению к своим двум аргументам. А прогонка основана на «обобщённом алгоритме отождествления», который асимметричен и представляет собой сопоставление выражения, содержащего переменные с образцом. При этом «сужения» (подстановки) над образцом не выполняются, а переменные из образца «принимают значения».

- Турчин следовал своей методологии «метасистемного перехода». Сначала — основной уровень (то есть Рефал и обычные вычисления), затем — метауровень (на котором наблюдается и изучается поведение основной системы). Поэтому прогонка — это надстройка над «обыкновенным» функциональным языком программирования (Рефалом). Программист может сначала написать и отладить программу на традиционном языке и традиционными методами. А после этого — засунуть эту программу в прогонку.

- Колмероэ пошёл другим путём. Он объявил, что нужно отказаться от традиционных (неправильных?) языков и методов программирования, и заменить их на новый (правильный?) язык Пролог. Наверное, он при этом надеялся, что программистское сообщество удастся уговорить свернуть с неправильной кривой дорожке на правильную. И многих уговорить удалось! Но не всех... И для этих «не всех» подход Турчина, наверное, выглядит не таким уж экстремистским, как у Колмероэ.

Понятно, что прогонка — это только первый шаг на пути построения интересной метасистемы. И Турчин активно занялся дальнейшим продвижением на пути к суперкомпиляции. Но вскоре возникли небольшие затруднения, связанные не с самой научной работой, а, скажем так, с «объемлющей метасистемой» (в виде тогдашних начальников СССР). В результате Турчина сначала выгнали с работы, а потом и вовсе предложили выбирать: либо поехать далеко-далеко на Восток, либо поехать далеко-далеко на Запад. Вот так Турчин и превратился из советского/российского учёного в американского...

Но не будем отвлекаться от темы, то есть суперкомпиляции.

---

[12] `https://en.wikipedia.org/wiki/Prolog`: The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Итак, Турчин был изгнан с работы. В нынешнее время, когда стало модно получать зарплату «в конвертиках», даже трудно осознать суть проблемы. Нет «официальной работы» — всегда можно найти частную лавочку, в которой — понятно что... Но в эпоху «развитого социализма» «выгнать с работы» означало на самом деле ещё и лишить возможности найти работу в другом месте.

Нет работы — нет зарплаты. А если рассматривать научный аспект, то нет работы — нет возможности что-то опубликовать, поскольку опубликовать научную статью можно было, только проделав некоторые обязательные действия *по месту работы*!

Поэтому между 1974 и 1979 годами — публикаций нет. Точнее, в 1977 году Турчину удалось-таки (с помощью маленькой военной хитрости) опубликовать аж 4 (!) страницы, посвящённых суперкомпиляции. Конкретно, это были страницы 92-95 в [26].

Анекдотическая подробность этого дела заключалась в том, что книга [26], в довершение всего, была издана вообще анонимно! В ссылке [26] перечислены ее реальные авторы, которых нет в выходных данных книги. Все авторы получили официальные справки, что да, они являются авторами таких-то и таких-то страниц. У авторов до сих пор лежат эти справки[13] (в качестве диковинки, передающий «аромат эпохи»)...

Чтобы пояснить современному читателю, как такое может быть, поясним, что Турчина выгоняло «высокое начальство», а те люди, которые издавали книгу, выписывали и подписывали эти справки, делали как раз всё, что могли, чтобы хоть как-то протащить книгу в печать.

Что касается 4 страниц, вставленных потихоньку в книгу в процессе её «редакторской правки», так они — просто шедевр научной прозы. В эти 4 страницы Турчину удалось втиснуть:

- объяснение сущности суперкомпиляции;
- объяснение того, что сейчас известно как «три проекции Футамуры».

«Проекции Футамуры» — вопрос тонкий и имеющий свою интересную историю [2]. Его нужно рассматривать отдельно. Поэтому изложим его сейчас чисто формально и мимоходом, не рассчитывая, что это описание будет кому-нибудь понятно (кроме тех, кто уже заранее знает, о чём речь).

Допустим, у нас есть программа f, исходные данные для которой разбиты на две части: «статическую» часть x и «динамическую» часть y. Обозначим через $f(x, y)$ результат применения программы f к исходным данным (x, y). Тогда «специализатором» называется такая программа s, что

$$s(p, x)(y) = p(x, y)$$

[13] `https://pat.keldysh.ru/~roman/doc/Turchin/1977-CNIPIASS--Spravka_ob_avtorstve_S_A_Romanenko.pdf`

Допустим, что p — программа, результатом применения которой к исходным данным d является $p(d)$. Тогда «интерпретатором» называется такая программа i, что

$$i(p, d) = p(d)$$

Теперь, используя свойства s и i, мы можем построить такую цепочку равенств:

$$p(d) = i(p, d) = s(i, p)(d) = s(s, i)(p)(d) = s(s, s)(i)(p)(d)$$

из которой «легко» видно, что

- $s(i, p)$ — скомпилированная программа;
- $s(s, i)$ — скомпилированный компилятор для языка, который интерпретирует i;
- $s(s, s)$ — скомпилированный компилятор компиляторов (преобразующий интерпретаторы в компиляторы).

Результат специализации функции `eq`, рассмотренный в Разделе 6.2, можно считать примером применения первой проекции Футамуры, ибо `eq` — это «интерпретатор», первый аргумент которого — «программа», при исполнении которой «интерпретатор» проверяет, что «программа» равна исходным данным.

Впрочем, как потом выяснилось, Ёсихико Футамура додумался до $s(i, p)$ и $s(s, i)$ ещё в 1971 году [2], а в отчете 1973 года [3] привел и $s(s, s)$. Поэтому в соответствии с научными традициями и принципами справедливости, эти штуки и называются «проекциями Футамуры».

Однако же, когда Футамура опубликовал свою статью в 1971, на неё никто не обратил внимания. И только когда другие «дозрели» и начали додумываться до того же самого, эта статья была замечена и оценена.

Можно ли считать, что Футамура изобрёл и суперкомпиляцию? Вопрос тонкий. Для практической реализации проекций Футамуры полноценная суперкомпиляция не нужна: вполне достаточно «частичных вычислений». А с точки зрения суперкомпиляции, частичные вычисления — это некоторый экстремальный, вырожденный случай суперкомпиляции. Ну а для особого частного случая можно использовать и особые методы, специально «заточенные» под этот особый случай. Вот и получается, что по решаемой задаче частичные вычисления — это частный случай суперкомпиляции, а по применяемым методам — различаются. Впрочем, это — отдельная интересная тема... А мы вернёмся к суперкомпиляции.

К сожалению, с 1979 года приходится читать уже на английском языке...

Оказавшись в Нью-Йоркском Городском университете[14], Турчин, первым делом, занялся публикацией того, что ему не удалось втиснуть в 4 страницы, пока он находился в СССР, в результате чего в 1979–1982 гг. появилось несколько работ: [15, 16, 17, 22].

[14] `http://portal.cuny.edu/`

На чем, можно считать, и завершилась *ранняя* история суперкомпиляции. Хороший обзор этого периода можно найти у Сёренсена [11]. А сам Турчин подвел итоги своей работы в двух статьях 1996 года [20, 21].

В те годы многих авторов [4, 11, 13, 12, 14, 23] интересовал следующий вопрос. У Турчина суперкомпиляция всегда рассматривалась применительно к языку Рефал. Верно ли, что суперкомпиляция применима только к Рефалу? Как и следовало ожидать, суперкомпиляция применима не только к Рефалу, и основы суперкомпиляции можно объяснять и рассматривать и на примере более простого языка (как мы и сделали в этой работе, использовав язык SLL, описанный в Приложении А). Однако основные проблемы, которые возникают при суперкомпиляции Рефал-программ, проявляются и при суперкомпиляции SLL-программ. Поэтому и имеет смысл сначала разобраться с суперкомпиляцией SSL-программ, а уже после этого переходить к суперкомпиляции для более сложных языков.

8 Заключение

Был продемонстрирован базовый круг понятий метода преобразования программ, называемого «суперкомпиляцией». Это только «первый этаж» вокруг алгоритма прогонки, порождающей потенциально бесконечное дерево конфигураций и процессов. Мы немного заглянули на второй этаж, показав простейшие правила конфигурационного анализа — свертки этого дерева в конечный граф путем зацикливания и обобщения. Однако приведенные методы обобщения еще недостаточны для построения конечного графа в общем случае.

Если удается получить конечный граф, он содержит в себе достаточно информации для порождения «остаточной» программы, эквивалентной исходной. На примерах было видно, что по сравнению с исходной она в определенной степени проспециализирована по информации, имеющейся в теле программы, в частности по константам. В некоторых случаях этого достаточно для компиляции программы путем специализации интерпретатора. Такая фигура называется «первой проекцией Футамуры».

В. Ф. Турчин, будучи сторонником постепенного, эволюционного подхода, строил суперкомпиляцию пошагово как последовательность метасистемных переходов [38]. С его точки зрения и исходя из его эволюционной теории сложные системы только так и возникают. На примере суперкомпиляции он искал и отрабатывал приемы совершения метасистемных переходов в формально-языковых системах вообще, нацеливаясь на пересмотр методов обработки математических теорий на компьютерах и даже оснований математики [19].

В этих терминах представленный здесь материал посвящен первому метасистемному переходу над программами, первому шагу разработки метавычислений, с небольшими заготовками на второй шаг. В следующем препринте совершим (вслед за В. Ф. Турчиным) второй метасистемный переход и

займемся проблемой, как гарантировать завершаемость суперкомпиляции и построением конечного графа в общем случае путем анализа конфигураций и путей в графе.

К сожалению, суперкомпиляторов (да и других специализаторов) еще нет в широкой программистской практике. Например, суперкомпиляторы для языков Рефал [32], Java, Haskell не вышли из экспериментальной стадии, и реально ими могут пользоваться только их авторы.

В данной цикле работ мы не замахиваемся на суперкомпиляцию для развитых практических языков, а рассматриваем методы в «дистиллированном», «чистом» виде для просто модельного функционального языка.

Изложенные алгоритмы реализованы в простом суперкомпиляторе SPSC. Его исходные коды вместе с коллекцией примеров предоставлены в открытом доступе. С одной из версий можно экспериментировать через веб-интерфейс. Ссылки на эти версии суперкомпилятора SPSC и руководство по их использовании даны в Приложении А.

А «Учебный» суперкомпилятор SPSC

Все примеры суперкомпиляции, рассматриваемые в данной работе, могут быть пропущены через суперкомпиялтор SPSC (= A Small Positive Supercompiler).

SPSC был создан с «учебными» целями: (1) для того чтобы можно было пропускать через него реальные примеры и (2) чтобы исходные тексты суперкомпилятора были умеренного размера и были достаточно легко читаемыми.

Таким образом, предполагалось рассеять ореол таинственности, одно время возникший вокруг суперкомпиляции, и показать, что нет ничего непостижимого ни в самой суперкомпиляции, ни в её реализациях.

Была поставлена задача изготовить «минималистский» суперкомпилятор, такой, чтобы *полный* текст его реализации (с объяснениями) можно было уместить в одной статье среднего размера. И эта задача была решена! Правда, в статью удалось втиснуть совсем «урезанный» вариант SPSC [9, 31], который был назван SPSC Lite. Но этот вариант отличался от «полного» SPSC только в тех частях, которые не имеют отношения к суперкомпиляции как таковой, связанных с лексическим и синтаксическим анализом исходной программы, проверкой контекстных ограничений на исходную программу, форматированием для печати графа конфигураций и остаточной программы.

Первоначальные версии SPSC и SPSC Lite были написана на языке Scala, но затем были изготовлены их версии и на нескольких других языках, таких как Python, Ruby, Haskell. В настоящее время исходные тексты реализаций выложены в общедоступном репозитории[15].

Версия SPSC, написанная на языке Scala, в настоящее время доступна в

[15] `https://github.com/sergei-romanenko/spsc`

виде веб-приложения[16], и через неё можно пропускать задания на суперкомпиляцию. Достоинство веб-приложения в том, что примеры можно вводить и запускать через браузер, ничего не устанавливая на своем компьютере. С другой стороны, часто хочется всё своё иметь при себе, не выставляя свои примеры публично. Да и вообще, сегодня веб-приложение существует, а завтра оно может и исчезнуть. Поэтому была изготовлена версия SPSC на языке Idris[17], которая может быть установлена локально из репозитория `spsc-idris`[18] и которую можно использовать для выполнения всех заданий на суперкомпиляцию, рассматриваемых в данной работе.

Теоретические принципы и алгоритмы, на которых основана работа SPSC, изложены в работах [11, 14, 12].

A.1 SLL — входной язык суперкомпилятора

Входным языком суперкомпилятора SPSC является SLL — «ленивый» функциональный язык первого порядка.

$task$	`::=` e `where` $prog$	задание
$prog$	`::=` d_1 `...` d_n	программа
d	`::=` f`(`x_1`, ... ,` x_n`) = `e`;`	равнодушная функция
	`|` g`(`p_1`,` x_1`, ... ,`x_n`) = `e_1`;`	любопытная функция
	`...`	
	g`(`p_m`,` x_1`, ... ,`x_n`) = `e_m`;`	
e	`::=` x	переменная
	`|` C`(`e_1`, ... ,`e_n`)`	конструктор
	`|` f`(`e_1`, ... ,`e_n`)`	вызов равнодушной функции
	`|` g`(`e_1`, ... ,`e_n`)`	вызов любопытной функции
p	`::=` C`(`v_1`, ... ,`v_n`)`	образец

Рис. 10: Синтаксис SLL-программ.

В языке SLL используется перечислимое множество символов для представления переменных $x \in X$, конструкторов $c \in C$ и имен функций $f \in F$ и $g \in G$. Все символы имеют фиксированное число аргументов (арность). SLL-программы обрабатывают данные, представляющие собой конечные деревья, построенные с помощью конструкторов. Программа на языке SLL состоит из набора определений функций (Рис. 10). Причем функции делятся на два класса: «равнодушные» (f-функции) и «любопытные» (g-функции) [12].

Определение равнодушной функции состоит из одного правила, в котором все аргументы являются переменными. Равнодушные функции не интересуются структурой своих аргументов. Исполнение вызова равнодушной функции $f(e_1, \ldots, e_n)$ сводится к тому, что этот вызов заменяется на правую

[16] `https://spsc.appspot.com`
[17] `https://www.idris-lang.org/`
[18] `https://github.com/sergei-romanenko/spsc-idris`

часть правила, в котором параметры функции заменяются на выражения e_1, ..., e_n.

Определение любопытной функции состоит из одного или нескольких правил, при этом в каждом правиле присутствует образец, с помощью которого анализируется структура первого аргумента. Если вызов функции имеет вид g`(`C`(`e_{00}`, ..., `e_{0k}`), `e_1`, ..., `e_n`)`, то в программе отыскивается правило, левая часть которого имеет вид g`(`C`(`x_{00}`, ..., `x_{0k}`), `x_1`, ..., `x_n`)`, и вызов функции заменяется на правую часть правила, в котором параметры функции заменяются на выражения e_{00}, ..., e_{0k}, e_1, ..., e_n. Если же не обнаруживается подходящего правила, вычисление аварийно завершается.

Исполнению подлежат не программы (наборы определений функций), а *задания* вида e `where` *prog*. При этом выражение e, вообще говоря, может содержать переменные, вместо которых, в случае обычного исполнения задания, следует подставить конкретные значения. Но это же самое задание может рассматриваться и как *задание на суперкомпиляцию*. В этом случае значения переменных в e считаются неизвестными.

На программы накладываются следующие контекстные ограничения.

- Имя конструктора должно начинаться с заглавной буквы, а имя функции или переменной — со строчной.
- Множества имен параметров, равнодушных функций и любопытных функций не должны пересекаться.
- В определении любопытной функции левые части различных правил должны содержать разные конструкторы.
- В правой части правила можно использовать только параметры функции (перечисленные в левой части).
- В правой части правила можно вызывать только те функции, для которых в программе имеется определение.
- В задании e `where` *prog*, все функции, которые вызываются в e, должны быть определены в *prog*.
- Все вхождения конструктора или функции в задании должны иметь одинаковую арность.

Если в строке встречается `--`, то остаток строки считается комментарием и игнорируется.

A.2 Примеры заданий

A.2.1 Сложение трех натуральных чисел

Будем считать, что неотрицательные целые числа представлены следующим образом. Ноль — как `Z` (где `Z` — конструктор без аргументов), а число, следующее за числом n, — как `S(`n`)` (где `S` — унарный конструктор).

Тогда следующее задание вычисляет сумму трех чисел `a`, `b` и `c`.

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

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

При этом любопытная функция `add`, определяемая в программе, вычисляет сумму двух чисел.

A.2.2 Проверка натуральных чисел на равенство

Следующее задание проверяет, что a равно 2.

```
eq(a, S(S Z))
where

eq(Z, y) = eqZ(y);
eq(S(x), y) = eqS(y, x);
eqZ(Z) = True;
eqZ(S(x)) = False;
eqS(Z, x) = False;
eqS(S(y), x) = eq(x, y);
```

Заметим, что в SLL не допускаются правила вроде

```
eq(S(x), S(y)) = eq(x, y);
```

из-за чего определение функции `eq` использует две вспомогательные функции `eqZ` и `eqS`.

A.3 Исполнение заданий на суперкомпиляцию

Первым делом следует установить компилятор для языка Idris[19] и клонировать репозиторий `spsc-idris`[20].

Затем — переходим в папку, в которой находится клонированный репозиторий и строим исполняемый файл `spsc_idris`:

```
idris --build spsc_idris.ipkg
```

После этого (при желании) пропускаем набор тестов:

```
idris --testpkg spsc_idris.ipkg
```

[19] `https://www.idris-lang.org/`
[20] `https://github.com/sergei-romanenko/spsc-idris`

Теперь переходим в папку `tasks`, в которой находятся примеры заданий на суперкомпиляцию (в файлах с расширением `task`).

Чтобы просуперкомпилировать задание `name`, исполняем команду

```
../spsc_idris name
```

В результате SPSC читает задание из файла `name.task` и записывает остаточное (просуперкомпилированное) задание в файл `name.res`. При этом получившийся граф конфигураций (в текстовом виде) записывается в файл `name.tree`.

Например, в файле `add_add_a_b_c.task` находится задание

```
-- Transforming a 2-pass algorithm to a 1-pass one
-- for binary functions.

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

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

то, в результате исполнения этого задания, в файл `add_add_a_b_c.res` записывается остаточное задание

```
add1(a,b,c)
where

add2(Z,c) = c;
add2(S(v2),c) = S(add2(v2,c));
add1(Z,b,c) = add2(b,c);
add1(S(v1),b,c) = S(add1(v1,b,c));
```

а в файл `add_add_a_b_c.tree` записывается граф конфигураций в текстовом представлении (которое мы не описываем, поскольку оно достаточно очевидно).

Список литературы

[1] Bolingbroke M., Peyton Jones S. Supercompilation by Evaluation // Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell 2010). — ACM, 2010. — P. 135–146.

[2] Futamura Y. Partial Evaluation of Computation Process — An Approach to a Compiler-Compiler // Systems, Computers, Controls. — 1971. — Vol. 2, no. 5. — P. 45–50.

[3] Futamura Y. EL1 Partial Evaluator (Progress Report). — Center for Research in Computing Technology, Harvard University, January 1973. — URL: http://fi.ftmr.info/PE-Museum/EL1.PDF (accessed: 25.05.2018).

[4] Glück R., Klimov And. V. Occam's Razor in Metacompuation: The Notion of a Perfect Process Tree // Static Analysis. — Vol. 724 of Lecture Notes in Computer Science. — Springer, 1993. — P. 112–123.

[5] Jones N. D., Gomard C. K., Sestoft P. Partial Evaluation and Automatic Program Generation. — Prentice Hall, 1993. — ISBN 0-13-020249-5. — URL: http://www.itu.dk/~sestoft/pebook/pebook.html (accessed: 25.05.2018).

[6] Jonsson P. A., Nordlander J. Positive Supercompilation for a Higher-Order Call-By-Value Language // Logical Methods in Computer Science. — 2010. — Vol. 6, no. 3.

[7] Klimov And. V. An Approach to Supercompilation for Object-Oriented Languages: The Java Supercompiler Case Study // First International Workshop on Metacomputation in Russia (META 2008). — Pereslavl-Zalessky : Ailamazyan University of Pereslavl, 2008. — P. 43–53.

[8] Klimov And. V. A Java Supercompiler and Its Application to Verification of Cache-Coherence Protocols // Perspectives of Systems Informatics, PSI 2009. — Vol. 5947 of Lecture Notes in Computer Science. — Springer, 2009. — P. 185–192.

[9] 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.

[10] Mitchell N. Rethinking Supercompilation // Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010). — ACM, 2010. — P. 309–320.

[11] Sørensen M. H. B. Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation : Master's Thesis. — Department of Computer Science, University of Copenhagen, 1994.

[12] 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.

[13] Sørensen M. H. B., Glück R. Introduction to Supercompilation // Partial Evaluation — Practice and Theory. — Vol. 1706 of Lecture Notes in Computer Science. — Springer, 1998. — P. 246–270.

[14] Sørensen M. H. B., Glück R., Jones N. D. A Positive Supercompiler // Journal of Functional Programming. — 1996. — Vol. 6, no. 6. — P. 811–838.

[15] Turchin V. F. A Supercompiler System Based on the Language REFAL // ACM SIGPLAN Notices. — 1979. — Vol. 14, no. 2. — P. 46–54.

[16] Turchin V. F. The Language Refal: The Theory of Compilation and Metasystem Analysis. Technical Report no. 20. — Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

[17] Turchin V. F. The Use of Metasystem Transition in Theorem Proving and Program Optimization // Automata, Languages and Programming. — Vol. 85 of Lecture Notes in Computer Science. — Springer, 1980. — P. 645–657.

[18] Turchin V. F. The Concept of a Supercompiler // ACM Transactions on Programming Languages and Systems. — 1986. — Vol. 8, no. 3. — P. 292–325.

[19] Turchin V. F. A Constructive Interpretation of the Full Set Theory // The Journal of Symbolic Logic. — 1987. — Vol. 52, no. 1. — P. 172–201.

[20] Turchin V. F. Metacomputation: Metasystem Transitions plus Supercompilation // Partial Evaluation. — Vol. 1110 of Lecture Notes in Computer Science. — Springer, 1996. — P. 481–509.

[21] Turchin V. F. Supercompilation: Techniques and Results // Perspectives of System Informatics (PSI 1996). — Vol. 1181 of Lecture Notes in Computer Science. — Springer, 1996. — P. 227–248.

[22] Turchin V. F., Nirenberg R. M., Turchin D. V. Experiments with a Supercompiler // Proceedings of the 1982 ACM Symposium on LISP and Functional Programming (LFP 1982). — ACM, 1982. — P. 47–55.

[23] Абрамов С. М. Метавычисления и их приложения. — М. : Наука, 1995. — 128 с.

[24] Абрамов С. М. Метавычисления. Часть I. — Переславль-Залесский : «Университет города Переславля», 2016. — 128 с. — ISBN 978-5-901795-26-2. — URL: `http://site.u.pereslavl.ru/Studentu/uchebniki/Metavychisleniya/part_I` (дата обращения: 25.05.2018).

[25] Абрамов С. М., Пармёнова Л. В. Метавычисления. Часть II. — Переславль-Залесский : «Университет города Переславля», 2016. — 72 с. — ISBN 978-5-901795-27-9. — URL: `http://site.u.pereslavl.ru/Studentu/uchebniki/Metavychisleniya/part_II` (дата обращения: 25.05.2018).

[26] Базисный РЕФАЛ и его реализация на вычислительных машинах / Анд. В. Климов, Арк. В. Климов, А. Г. Красовский, С. А. Романенко, Е. В. Травкина, В. Ф. Турчин, В. Ф. Хорошевский, И. Б. Щенков. — М. : ЦНИПИАСС, 1977. — 258 с.

[27] Гречаник С. А. Полипрограммы как представление множеств функциональных программ и преобразования над ними // Препринты ИПМ им. М. В. Келдыша. — 2017. — № 5. — 31 с.

[28] Климов Анд. В. Введение в метавычисления и суперкомпиляцию // Будущее прикладной математики: Лекции для молодых исследователей. — М. : КомКнига, 2008. — С. 343–368.

[29] Ключников И. Г. Суперкомпиляция функций высших порядков // Программные системы: теория и приложения. — 2010. — Т. 3, № 3. — С. 37–71.

[30] Ключников И. Г. Суперкомпиляция: идеи и методы // Практика функционального программирования. — 2011. — № 7. — С. 152–188.

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

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

[33] Отставнов М. Е. Валентин Турчин: «Проплыть между Сциллой и Харибдой» // Компьютерра. — 2001. — № 25.

[34] Турчин В. Ф. Метаязык для формального описания алгоритмических языков // Цифровая вычислительная техника и программирование. — М.–Л. : Советское радио, 1966.

[35] Турчин В. Ф. Метаалгоритмический язык // Кибернетика. — 1968. — № 4. — С. 116–124.

[36] Турчин В. Ф. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ // Теория языков и методы построения систем программирования. — Киев-Алушта, 1972. — С. 31–42.

[37] Турчин В. Ф. Эквивалентные преобразования программ на РЕФАЛе // Автоматизированная система управления строительством. — М. : ЦНИПИАСС, 1974. — С. 36–68.

[38] Турчин В. Ф. Феномен науки: Кибернетический подход к эволюции. Изд. 2-е. — М. : ЭТС, 2000.

Содержание

1 Введение **3**

2 Цель и составные части суперкомпиляции **4**

3 Смысл терминов «суперкомпиляция» и «суперкомпилятор» **5**

4 Пример суперкомпиляции: сложение чисел **6**
4.1 Сложение чисел Пеано 6
4.2 Символические вычисления (редукция) 7
4.3 Вложенные вызовы функций 7
4.4 Прогонка (редукция + разбор случаев) 8
4.5 Декомпозиция вызовов конструкторов 9
4.6 Зацикливание (приведение к эквивалентному) 11
4.7 Извлечение остаточной программы 12

5 Пример: сложение с накоплением **14**
5.1 Сложение чисел с накоплением результата 14
5.2 Сравнение конфигураций на общность 16
5.3 Обобщение конфигураций 17

6 Суперкомпиляция как метод специализации программ **19**
6.1 Что такое специализация программ? 20
6.2 Пример: специализация «интерпретатора» 20

7 Ранняя история суперкомпиляции **21**

8 Заключение **27**

A «Учебный» суперкомпилятор SPSC **28**
A.1 SLL — входной язык суперкомпилятора 29
A.2 Примеры заданий 30
A.2.1 Сложение трех натуральных чисел 30
A.2.2 Проверка натуральных чисел на равенство 31
A.3 Исполнение заданий на суперкомпиляцию 31

Список литературы **32**