

Суперкомпиляция: идеи и методы

Илья Ключников

klyuchnikov@fprog.ru

Аннотация

Суперкомпиляция (*supervising compilation*) — техника преобразования программ, основанная на построении *полной и самодостаточной модели* программы.

В статье описываются основные идеи и методы суперкомпиляции на примере работающего суперкомпилятора SC Mini для простейшего чисто функционального языка.

Supercompilation (supervising compilation) is a program transformation technique based upon constructing a self-sufficient model of the program.

The paper describes the main ideas and methods of supercompilation through a series of examples performed by the simple supercompiler SC Mini.

Обсуждение статьи ведется по адресу:

<http://fprog.ru/2011/issue7/ilya-klyuchnikov-supercompilation/discuss/>.

7.1. Идея суперкомпиляции

Суперкомпиляция была придумана в СССР в 1970-х годах В. Ф. Турчиным. Сам он объяснял название и идею так [31]:

A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the machine, let us call it M_1 , which is represented by the program. In observing the operation of M_1 , the supercompiler COMPILES a program which describes the activities of M_1 , but it makes shortcuts and whatever clever tricks it knows, in order to produce the same effect as M_1 , but faster. The goal of the supercompiler is to make the definition of this program (machine) M_2 , self-sufficient. When this is achieved, it outputs M_2 , and simply throws away the (unchanged) machine M_1 .

A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between possible configurations of the computing system ... in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one¹.

Суммируем кратко:

- 1) Программе P_1 ставится в соответствие машина M_1 , моделирующая в общем виде выполнение программы P_1 .
- 2) Контролируя и наблюдая (SUPERvises) работу машины M_1 , суперкомпилятор создает (компилирует, COMPILEs) другую машину M_2 , которая полностью описывает M_1 .
- 3) Машина M_2 далее может быть представлена в виде программы P_2 .

7.2. Цель данной статьи

Среди людей, занимающихся именно практическим функциональным программированием (да и не только функциональным), отношение к суперкомпиляции (среди тех, кто о ней слышал) очень разное. Наиболее часто встречающаяся реакция — недоверие и даже отторжение по той причине, что реального промышленного суперкомпилятора в природе пока еще не существует. А те, что существуют — экспериментальные. И в большинстве случаев экспериментальные суперкомпиляторы плодотворно используются только их авторами.

¹Приблизительный перевод:

Программа рассматривается в виде некоторой машины. Смысл извлекается из наблюдения за ее действиями. Суперкомпилятор не преобразует программу шаг за шагом; он управляет и наблюдает за машиной (назовем ее M_1), которая представлена в виде программы. Наблюдая за работой машины M_1 , суперкомпилятор конструирует другую машину M_2 , которая описывает действия машины M_1 . При построении M_2 суперкомпилятор пользуется различными трюками, чтобы M_2 работала так же, как и M_1 , но быстрее. Цель суперкомпилятора — сделать M_2 самодостаточной. По достижении этой цели суперкомпилятор отбрасывает исходную машину M_1 и выдаёт M_2 .

Суперкомпилятор запускает M_1 в общем виде (с неизвестными значениями переменных), строит граф состояний и переходов между различными конфигурациями вычислительной системы. Такой график полностью описывает поведение системы. Таким образом, новая программа становится самодостаточной моделью исходной программы.

Идея суперкомпиляции в некотором смысле универсальна, но полезными и работоспособными оказываются достаточно узко специализированные изделия, использующие идеи суперкомпиляции. А пользователи от универсальной идеи ожидают универсального инструмента. Еще одна причина недоверия — то, что на базе суперкомпиляции нет еще того, что называется killer app.

Достаточно распространено заблуждение, что суперкомпиляция — техника оптимизации программ. Суперкомпиляция — это метод преобразования программ. Бывают разные цели преобразований программ. Целью может быть оптимизация программы, а может быть и ее анализ. Методы суперкомпиляции используются и для того, и для другого. Большинство работ посвящены применению суперкомпиляции для оптимизации, но это не значит, что нужно рассматривать суперкомпиляцию однобоко только с этой позиции.

Необходимо различать понятия *суперкомпиляция* и *суперкомпилятор*. Многие статьи описывают различные технические трудности, возникающие при построении конкретного суперкомпилятора, и способы их преодоления, отодвигая основные методы суперкомпиляции на задний план. В большинстве случаев части, из которых состоит конкретный суперкомпилятор, очень тонко и старательно подогнаны друг под друга, из-за чего может возникать ощущение сложной монолитной конструкции².

Основная цель данной статьи — на примере «минимального» суперкомпилятора попытаться четко и обозримо проиллюстрировать приведенную цитату В. Ф. Турчина, описывающую идею суперкомпиляции. Я постараюсь сделать акцент на вычленении и объяснении основных логических частей и показать, как эти части могут быть запрограммированы и соединены достаточно простым и понятным способом.

Как устроена статья и что в ней есть: Главный материал этой статьи — это приложение (pdf-файл) с полным кодом игрушечного³ суперкомпилятора **SC Mini**⁴ и подробными комментариями. Сам текст статьи стоит рассматривать как подготовку к чтению приложения. В статье вводятся и разбираются основные понятия и на примерах показывается «механика» работы SC Mini. Примеры, приведенные далее, не являются сложными, но и простыми их назвать, наверное, тоже нельзя. Читателю следует подготовиться к некоторому «погружению» в примеры.

Чего нет в данной статье: Даже поверхностное сравнение суперкомпиляции с современными техниками оптимизации и/или анализа программ было бы противоречивым и беспредметным, поэтому я даже не пытаюсь это сравнение сделать⁵.

²«Если посмотреть, как устроен суперкомпилятор, то видно, что он состоит из нескольких частей, сплеленных в один запутанный комок грязи. Когда я попытался по-настоящему понять, как работают суперкомпиляторы, это оказалось очень трудно.» (С. Пейтон Джонс. Интервью // «ПФП», № 6.)

³250 строк основного кода + 200 строк вспомогательного кода (различные утилиты) на Хаскеле.

⁴<https://github.com/ilya-klyuchnikov/sc-mini>

⁵Это сравнение в данной статье было бы хоть как-то возможным, если бы в предыдущих выпусках «ПФП» уже были бы обзорные статьи по методам оптимизации и анализа программ — тогда можно было бы сравнить суперкомпиляцию с описанными методами. Нужно вначале прийти к соглашению, какие методы и инструменты мы хотим сравнить. Иначе данная статья будет объектом всевозможных упреков в том, что метод А или Б несправедливо обойден вниманием. Однако автор не уходит от ответственности и готов ответить на конкретные вопросы в дискуссии в ЖЖ.

7.3. Методы суперкомпиляции на примерах

Достоинства всякого формализованного языка определяются не только тем, сколь он удобен для непосредственного использования человеком, но и тем, в какой степени тексты на этом языке поддаются формальным преобразованиям.

В. Ф. Турчин [43]

Было бы хорошо, если бы все статьи по информатике сопровождались изложением в формальной машинно-читаемой форме (в виде программ или других формальных спецификаций), чтобы не возникала проблема воспроизводимости результатов.

Суперкомпилятор автора SC Mini — формальное изложение основных методов суперкомпиляции, описанных в данной статье. За основу суперкомпилятора SC Mini взят суперкомпилятор, описанный в изумительной магистерской диссертации Мортена Сёренсена [26]⁶.

При описании суперкомпиляции так или иначе приходится затрагивать такие базовые, но непростые понятия, как смысл (семантика) программы, результат вычислений, состояние вычислений. Все эти понятия формально описаны (= запрограммированы) в тексте SC Mini, что помогает избежать двусмысленности (по крайней мере, в приложении).

Суперкомпилятор SC Mini преобразует программы, написанные на игрушечном языке SLL.⁷ Перефразируя высказывание В. Ф. Турчина, приведенное в качестве эпиграфа, можно сказать, что достоинствами языка SLL являются:

- 1) Простота описания языка.
- 2) Простота описания суперкомпилятора SLL-программ.

Несмотря на то, что SC Mini — суперкомпилятор для конкретного языка SLL, методы, на которых он основан, достаточно универсальны и присутствуют в той или иной степени практически во всех суперкомпиляторах.

Далее мы на примерах покажем, что SC Mini способен оптимизировать программы (предварительно формально определив, что такое оптимизатор) и может быть использован для доказательства простых теорем о программах.

Статья содержит замечания, набранные мелким шрифтом, которые при первом прочтении можно пропустить или просто обратить внимание на вопросы, которые там затрагиваются, но к которым можно вернуться после знакомства с текстом SC Mini. Цель замечаний — указать на некоторые очень серьёзные «тонкости», которые обеспечивают корректность преобразований. Именно такие тонкие и сперва неприметные моменты привносят основные сложности в создание суперкомпиляторов.

Следующий раздел, в котором описывается язык SLL, носит формальный характер.

Рис. 7.1. SLL: абстрактный синтаксис

$P ::= d_1 \dots d_n$	программа
$d ::= f(v_1, \dots, v_n) = e;$	«безразличная» функция
$g(p_1, v_1, \dots, v_n) = e_1;$	«любопытная» функция
...	
$g(p_m, v_1, \dots, v_n) = e_m;$	
$e ::=$	выражение
v	переменная
$C(e_1, \dots, e_n)$	конструктор
$f(e_1, \dots, e_n)$	вызов функции
$p ::= C(v_1, \dots, v_n)$	образец

Рис. 7.2. prog1: работа с числами Пеано

```

add(z(), y) = y;
add(s(x), y) = s(add(x), y);

mult(z(), y) = z();
mult(s(x), y) = add(y, mult(x, y));

sqr(x) = mult(x, x);

even(z()) = True();
even(s(x)) = odd(x);

odd(z()) = False();
odd(s(x)) = even(x);

add'(z(), y) = y;
add'(s(v), y) = add'(v, s(y));

```

7.3.1. Объектный язык SLL

Определить язык — значит описать его синтаксис и семантику. Синтаксис языка традиционно описывается бесконтекстной грамматикой. Описанием семантики может быть реализация языка. То, что дальше описывается словами, гораздо проще и яснее описано в приложении в виде программы на Хаскеле.

Синтаксис

Абстрактный синтаксис SLL приведен на рис. 7.1. Выражения языка SLL:

- переменная,
- конструктор, аргументами которого являются SLL-выражения,
- вызов функции, аргументами которого являются SLL-выражения.

Согласимся, что имена конструкторов начинаются с большой буквы, а имена функций и переменных — с маленькой.

⁶Сёренсен только описал, но не реализовал свой суперкомпилятор.

⁷SLL — Simple Lazy Language. В диссертации Сёренсена рассматриваются языки M_1 , $M_{1/2}$, M_0 . SLL в точности соответствует языку M_0 . Название SLL введено только для удобства речи.

Рис. 7.3. SLL: контекст и редекс

<i>con</i>	$\ ::= \langle \rangle \mid g(con, \dots)$	контекст
<i>red</i>	$\ ::= f(e_1, \dots, e_n) \mid g(C(e_1, \dots, e_n), \dots)$	редекс

SLL-выражение, состоящее только из конструкторов, называется *значением*. SLL-выражение без переменных называется *замкнутым выражением*. SLL-выражение со свободными переменными называется *конфигурацией*.

Программа на языке SLL состоит из определений функций. Функции бывают двух видов — «безразличные» и «любопытные».⁸ Безразличные функции только передают свои аргументы другим функциям и конструкторам: определение безразличной функции — одно предложение. Любопытные функции осуществляют ветвление по первому аргументу: определение любопытной функции состоит из нескольких предложений (по одному предложению на каждый вариант первого аргумента).

Никаких встроенных типов данных (булевых значений, чисел и т. д.) в языке SLL нет. Однако их можно определить, например, так:

- константы `True()` и `False()` — логические значения,
- константа `Z()` — 0, `S(Z())` — 1, `S(S(Z()))` — 2 и т. д. Если константа *n* соответствует натуральному числу *n*, то константа `S(n)` соответствует натуральному числу *n* + 1 — так называемые числа Пеано.

На рис. 7.2 приведена программа, в которой определяются функции для работы с числами Пеано. В этой программе только одна безразличная функция — возвведение в квадрат (`sqr`). Все остальные функции разбирают свой первый аргумент и являются любопытными.

Подстановка связывает переменные v_1, v_2, \dots, v_n с выражениями e_1, e_2, \dots, e_n и записывается как список пар $\{v_1 := e_1, \dots, v_n := e_n\}$. Применение подстановки к выражению e определяется стандартным образом и записывается $e/\{v_1 := e_1, \dots, v_n := e_n\}$.

Упражнение 1 В приложении применение подстановки рассматривается как список пар. Применение подстановки s к выражению e определяется как $e // s$. Найдите такие $e, v1, e1, v2$ и $e2$, что $e // [(v1, e1), (v2, e2)] \neq e // [(v1, e1)] // [(v2, e2)]$.

Семантика

Семантика⁹ языка SLL определяется через переписывающий SLL-интерпретатор с нормальным порядком редукции. SLL-интерпретатор \mathcal{I}_p программы p для замкнутого SLL-выражения пошагово вычисляет SLL-значение (или зацикливается)¹⁰ следующим образом.

С точки зрения редукции есть два вида замкнутых выражений:

- 1) $e = C(e_1, \dots, e_n)$ — конструктор «выталкивается» наружу, дальше происходит редукция аргументов.

⁸Это просто синтаксический трюк, чтобы избежать конструкции `case e ...` и не возиться со связанными переменными в выражениях. Исторически функции первого вида назывались f-функциями, а второго — g-функциями.

⁹Со способами описания семантик можно познакомиться, например, по книге [19]; там же описываются понятия редекс и контекст редукции, используемые на рис. 7.3 и 7.4 без объяснений.

¹⁰Или выдаёт ошибку, но такие случаи мы здесь для простоты не рассматриваем. Их рассмотрение не привносит ничего принципиально нового.

- 2) $e \neq C(e_1, \dots, e_n)$ — тогда в выражении находится самый левый редуцируемый вызов функции (редекс, см. рис. 7.3) и редуцируется в соответствии с определением в программе. Вызов функции является редуцируемым, если это 1) вызов безразличной функции или 2) вызов любопытной функции, первым аргументом которого является конструктор.

Формально редукционная семантика SLL-выражений дана на рис. 7.4. Запись $e_1 \stackrel{p}{=} e_2$ означает, что в программе p есть определение $e_1 = e_2$.

Правила вычисления SLL-выражений обладают приятным и важным свойством «композиционности». А именно, если выражение $e_1/\{v := e_2\}$ замкнутое, то:

$$\mathcal{I}_p[e_1/\{v := e_2\}] = \mathcal{I}_p[e_1/\{v := \mathcal{I}_p[e_2]\}]$$

Заметим, что \mathcal{I}_p можно рассматривать, как машину, описывающую программу p .

Упражнение 2 В SC Mini интерпретатор определяется как функция `eval`. Запись $\mathcal{I}_p[e]$ соответствует вызову `eval p e`. Докажите, что результатом вычисления

`eval p (e1 // [(v, e2)]) == eval p (e1 // [(v, eval p e2)])` не может быть `False`.

Пример вычисления выражения `even(sqr(S(Z())))` интерпретатором для программы `prog1` представлен на рис. 7.5. Редексы подчеркнуты.

SLL-задание¹¹ — пара (e, p) из выражения e и программы p . Вычисление задания (e, p) на аргументах $args$ определяется как:¹²

$$\mathcal{R}_{(e,p)}[args] = \mathcal{I}_p[e / args]$$

Оптимизатор

Определим, что такое оптимизатор SLL-заданий. SLL-оптимизатор получает задание (e, p) и выдаёт новое другое задание (e', p') такое, что на любых аргументах $args$ результаты вычисления старого и нового заданий совпадают¹³:

$$\mathcal{R}_{(e,p)}[args] = \mathcal{R}_{(e',p')}[args]$$

и при вычислении нового задания интерпретатор делает *не больше* шагов (редукций), чем при вычислении старого.

7.3.2. Обзор устройства суперкомпилятора SC Mini

Суперкомпилятор SC Mini, преобразуя задание (e, p) в задание (e', p') , действует следующим образом:

- 1) Конструирует машину \mathcal{M}_p , которая описывает работу *одного шага* интерпретатора \mathcal{I}_p программы p в «общем виде» — над конфигурациями.
- 2) Осуществляет конечное (в общем случае, достаточно большое) число запусков машины \mathcal{M}_p , и анализирует результаты этого запуска.
- 3) Строит конечный граф конфигураций, описывающий работу \mathcal{M}_p во время пробных запусков.

¹¹Понятие SLL-задания вводится для упрощения изложения и является в некоторой степени аналогом функции `main`.

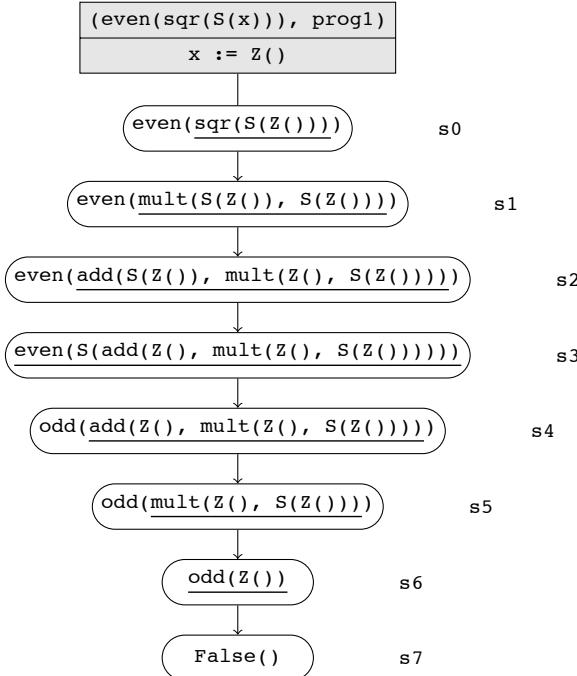
¹²В SC Mini этому соответствует `sll_run (e, p) args`.

¹³То есть SLL-система либо выдаёт равные SLL-значения для обоих запусков, либо зацикливается для обоих запусков.

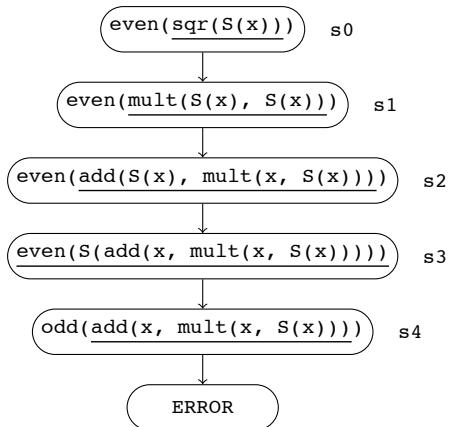
Рис. 7.4. SLL: интерпретатор программы \mathcal{I}_p для программы p

$\mathcal{I}_p[e]$	$\Rightarrow e$	(I_1)
$\mathcal{I}_p[C(e_1, \dots, e_n)]$	$\Rightarrow C(\mathcal{I}_p[e_1], \dots, \mathcal{I}_p[e_n])$	(I_2)
$\mathcal{I}_p[con(f(e_1, \dots, e_n))]$	$\Rightarrow \mathcal{I}_p[con(e/\{v_1 := e_1, \dots, v_n := e_n\})] (I_3)$	
	if $f(v_1, \dots, v_n) \stackrel{p}{=} e$	
$\mathcal{I}_p[con(g(C(e_1, \dots, e_m), e_{m+1}, \dots, e_n))]$	$\Rightarrow \mathcal{I}_p[con(e/\{v_1 := e_1, \dots, v_n := e_n\})] (I_4)$	
	if $g(C(v_1, \dots, v_m), v_{m+1}, \dots, v_n) \stackrel{p}{=} e$	

Рис. 7.5. Пример вычисления задания



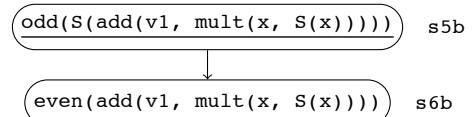
- 4) Упрощает граф конфигураций и превращает его в новое задание (e', p') .



Интерпретатор не ожидает переменной в первом аргументе вызова функции `add`. Однако наличие переменной `x` не помешало интерпретатору сделать несколько начальных шагов.¹⁴

«Расширим» интерпретацию \mathcal{I}_p :¹⁵ вместо аварийного завершения рассмотрим различные варианты дальнейших действий интерпретатора. Программа `prog1` для рассматриваемой ситуации предусматривает два варианта развития: когда $x = Z()$ и когда $x = S(x)$ (см. определение функции `add`): рис. 7.6.

Теперь мы можем продвинуться в вычислении правого выражения (и т. д.):



Интерпретатор \mathcal{I}_p рассчитан на пошаговое вычисление замкнутых выражений. Результатом запуска интерпретатора $\mathcal{I}_p[e]$ является некоторое (итоговое) SLL-значение.

Машина \mathcal{M}_p будет вычислять конфигурации. Результатом запуска машины $\mathcal{M}_p[c]$ будет моделирование шага интерпретатора.

Рассматриваем следующие модели шагов:

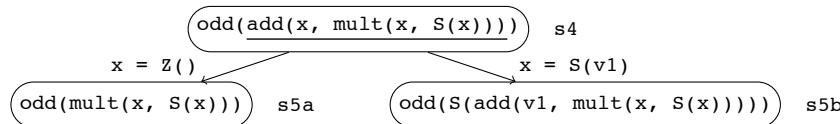
- 1) Транзитный шаг — ближайший шаг интерпретации не зависит от конфигурационных переменных. Пример — переход из состояния $s0$ в состояние $s1$.

¹⁴При передаче параметров по имени (call-by-name), действительно, интерпретатор способен «кое-что» вычислить. При передаче параметров по значению (call-by-value) заставить интерпретатор продвинуться в вычислении выражения со свободными переменными нетривиально. Именно поэтому суперкомпиляцию легче объяснять для языка с передачей параметров по имени.

¹⁵Многим это напоминает абстрактную интерпретацию [2]. Однако, абстрактная интерпретация (по отношению к обычной) есть сужение области значений. Здесь же никакого сужения области значений не происходит.

SLL-интерпретатор устроен таким образом, что, в принципе, он «способен» вычислять и выражения со свободными переменными:

Рис. 7.6. SLL: Рассмотрение вариантов при прогонке



- 2) Остановка. Дальнейшее моделирование невозможно. Возникает, если данное выражение является переменной или константой.
- 3) Декомпозиция. Часть результата уже известна, например, в выражении $S(sqr(x))$ внешний конструктор уже никуда не денется. Переходим к обработке подвыражений.
- 4) Разбор вариантов. Однозначное моделирование дальше невозможно. Однако можно рассмотреть *все варианты шагов интерпретатора, которые описаны в программе*. Пример — переходы из состояния s_4 в состояния s_5a и s_5b .

Такое моделирование в суперкомпиляции называется *прогонкой* (driving). То, что описано выше — шаг прогонки.

Упражнение 3 В SC Mini конструирование машины \mathcal{M}_p есть `driveMachine p`. Покажите, что `driveMachine p` достаточно для вычислений замкнутых выражений. То есть можно написать такую функцию f , что

`f (driveMachine p) e = eval p e`

7.3.4. Дерево конфигураций

Дерево конфигураций для SLL-задания (e, p) строится следующим образом. Создается машина \mathcal{M}_p , моделирующая p . Вначале дерево состоит из одного узла, в который помещена стартовая конфигурация e (цель задания). Затем для каждого листа дерева n , в котором находится некоторая конфигурация c , производится «запуск» $\mathcal{M}_p[c]$ (т. е. запуск одного шага интерпретации) и результаты запуска подвешиваются к n в виде дочерних узлов. Затем для них также запускается машина и т. д. Получается дерево конфигураций (в каждом узле дерева находится конфигурация).

В общем случае дерево конфигураций, конструируемое таким образом, будет бесконечным. Но мы всегда можем заглянуть в него на конечную глубину (в предыдущем разделе мы рассмотрели построение верхушки дерева конфигураций для задания $(even(sqr(S(x))), prog1)$).

Упражнение 4 В SC Mini построение дерева конфигураций для задания (e, p) происходит так:

`buildTree (driveMachine p) e`

Попробуйте как-то охарактеризовать класс заданий, для которых строятся конечные деревья конфигураций.

Хотя сама концепция дерева конфигураций имеет небольшую практическую ценность,¹⁶ она имеет «важную теоретическую ценность» и помогает понять механизм работы суперкомпилятора.

Предположим на минуту, что мы можем строить (и как-то хранить) бесконечные деревья конфигураций. Тогда, если мы для задания (e, p) с помощью машины \mathcal{M}_p построили дерево конфигураций t , то мы можем отбросить и программу p , и машину \mathcal{M}_p , и работать только с деревом t . Дерево t полностью

описывает выполнение задания (e, p) , абстрагируясь от текста программы p .

Упражнение 5 Чтобы последнее утверждение не было голословным, в приложении приведен «древесный» вычислитель `intTree`, который умеет вычислять задания, используя только дерево конфигураций, и не нуждается в исходной программе. Если t — дерево конфигураций для задания (e, p) , то `intTree t args` есть результат вычисления задания для аргументов $args$. Убедитесь, что значением выражения

`eval (e, p) args == intTree (buildTree (driveMachine p) e) args`

не может быть `False`.

7.3.5. Свертка

Infinite driving graphs are useful in many ways, but to use a graph as an executable program it must be finite.

В. Ф. Турчин [32]

Мы можем посчитать `even(sqr(S(x)))` для любого x , конструируя дерево конфигураций и затем используя «древесный» интерпретатор `intTree`. Однако соответствующее дерево конфигураций получается бесконечным. Цель *свертки* (folding) — превратить бесконечное дерево в конечный объект, из которого при желании восстанавливается исходное бесконечное дерево.

Как осуществляется свертка? Пусть в строящемся дереве конфигураций возникла ветвь, спускаясь по которой, мы встречаем узлы $n_0 \rightarrow \dots \rightarrow n_i \rightarrow \dots \rightarrow n_j \rightarrow \dots$. Пусть конфигурация в узле n_j является переименованием (отличается только именами переменных) конфигурации в некотором узле n_i ($i < j$). Тогда делается следующий вывод: поддерево, «растущее» из узла n_j можно получить из поддерева, «растущего» из узла n_i , если последовательно переименовывать переменные в соответствующих конфигурациях.

Рассмотрим дерево конфигураций для задания $(even(sqr(x)), prog1)$, изображенное на рис. 7.7.

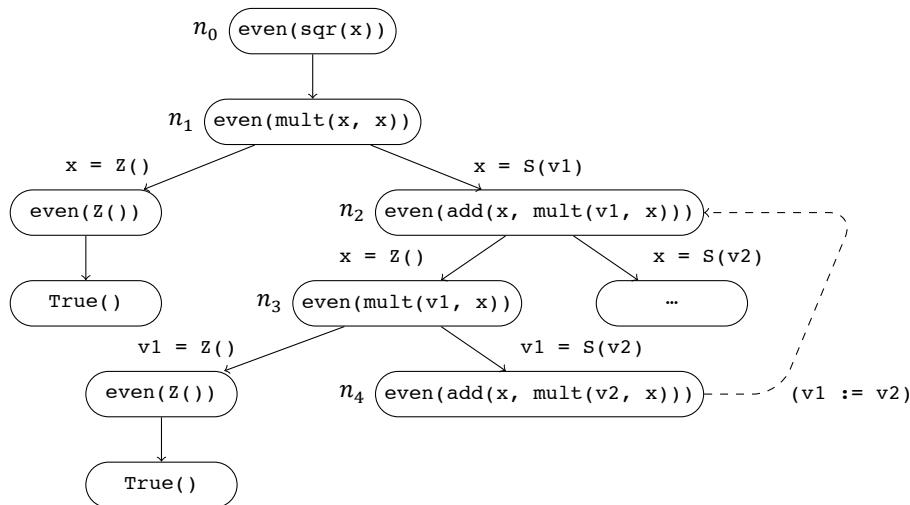
Конфигурация в узле n_4 является переименованием конфигурации в узле n_2 . Поддеревья (бесконечные!), растущие из n_2 и n_4 , отличаются только именами переменных в соответствующих узлах. Поэтому мы можем запомнить, что поддерево n_4 получается из поддерева n_2 , и не строить его прямо сейчас, а отложить построение до момента, когда это поддерево действительно понадобится. Важным результатом является то, что для получения поддерева n_4 из поддерева n_2 нам не требуется программа (= машина \mathcal{M}_p).

Для обозначения этого факта будем использовать специальную дугу, ведущую из нижнего узла в верхний. В результате дерево конфигураций перестает уже быть деревом и превращается в *граф конфигураций*.

С деревом конфигураций для задания $(even(sqr(x)), prog1)$ нам везет — оно сворачивается в граф¹⁷. Таким обра-

¹⁶Из-за этого ни дерево конфигураций, ни граф конфигураций (см. дальше) почти не упоминаются в статьях последних лет (особенно западных авторов), обсуждающих создание практического суперкомпилятора.

¹⁷Граф получается достаточно большим — он приведен полностью в приложении, убедитесь, что для этого примера действительно получается конечный граф без бесконечных ветвей.

Рис. 7.7. Дерево конфигураций для задания $(\text{even}(\text{sqr}(x)), \text{prog1})$ 

зом, мы можем превращать некоторые бесконечные деревья в конечные графы. Получившийся граф и будет той самой новой самодостаточной *конечной* машиной для задания. Будем обозначать граф для задания t как \mathcal{G}_t .

Упражнение 6 Стоит совсем немного изменить древесный интерпретатор `intTree`, чтобы он умел вычислять задания с помощью соответствующего свернутого графа конфигураций. Посмотрите, как это сделано в SC Mini.

Граф \mathcal{G}_t для задания $t = (e, p)$ можно представить в виде нового SLL-задания (e', p') . Полученная таким образом программа p' называется *остаточной* (*residual*), а задание (e', p') будем по аналогии называть *остаточным*.

Упражнение 7 В приложении описана функция `residuate`: `residuate g` преобразует граф конфигураций g в новое задание (e', p') .

7.3.6. Обобщение: превращение дерева в сворачиваемое дерево

В рассмотренном примере нам повезло в том, что дерево конфигураций удалось свернуть в граф. Как было бы замечательно, если бы все деревья конфигураций сворачивались! К сожалению, не все деревья можно превратить в графы. В программе `prog1` есть функция `add'`, определяющая сложение с помощью накапливающего параметра. Дерево конфигураций для задания $(\text{add}'(x, y), \text{prog1})$ строится, как показано на рис. 7.8.

Это дерево не сворачивается. Действительно, конфигурации могут различаться именами переменных только тогда, когда они (конфигурации) одинакового размера. В данном же примере, идя по «самой правой» ветке, мы встречаем конфигурации, постоянно увеличивающиеся в размере. Однако если бы размер конфигураций в дереве был ограничен, то такое дерево было бы сворачиваемым.¹⁸

Будем действовать по принципу *divide et impera*. Ограничим размер конфигураций в узлах некоторой константой `sizeBound`, и если некоторая конфигурация превысит допустимый размер — разделим ее на меньшие составляющие, чтобы их можно было рассматривать *независимо*. Однако мы

¹⁸Рассмотрим переименование как отношение эквивалентности, разбивающее множество выражений на классы эквивалентности. Множество SLL-выражений, в которых присутствуют конструкторы и функции только из программы p (= конечная сигнатура) и размер которых ограничен константой, разбивается на *конечное число классов*.

должны так представить конфигурацию, чтобы изучив поведение ее частей, мы могли бы сказать, как ведет себя целое.

Напомню, что вычисление замкнутых SLL-выражение обладает следующим свойством. Если выражение $e_1/\{v := e_2\}$ замкнутое, то:

$$\mathcal{I}_p[e_1/\{v := e_2\}] = \mathcal{I}_p[e_1/\{v := \mathcal{I}_p[e_2]\}]$$

Суперкомпилятор SC Mini ограничивает размер конфигураций в дереве и строит сворачиваемые (в граф) деревья так: если при конструировании дерева возникает конфигурация e , являющаяся вызовом функции,¹⁹ размер которой больше чем `sizeBound`, то эта конфигурация представляется в виде $e = e_1/(v := e_2)$, где e_2 — самое большое по размеру подвыражение конфигурации e , а e_1 — это выражение e , из которого «вынули» e_2 и заменили на переменную v . Для представления такого разделения будем использовать `let`-выражение `let v = e2 in e1`.²⁰ Затем конфигурации e_1 и e_2 рассматриваются отдельно.

Конфигурация e_1 является по отношению к исходной конфигурации e более общей. А e по отношению к e_1 является, соответственно, частным случаем (*instance*). Описанный выше шаг при конструировании дерева конфигураций называется *обобщением* (*generalization*).

Теперь, если мы ограничим рост конфигураций (по размеру), для любого задания мы можем построить (потенциально бесконечное) дерево конфигураций, которое будет *гарантированно* сворачиваться в *конечный* граф конфигураций.

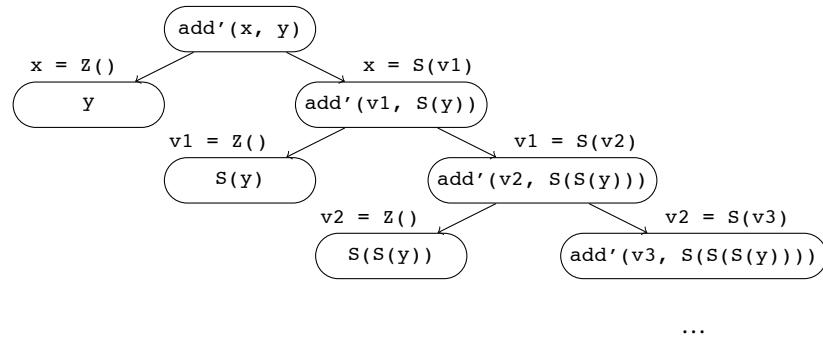
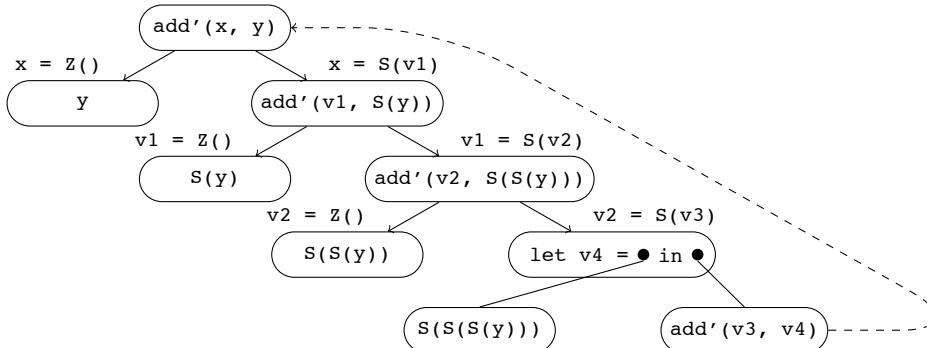
Например, если мы поставим максимальный размер выражения `sizeBound` равным 5, то в результате обобщения самой правой конфигурации в рассматриваемом примере получим дерево, изображенное на рис. 7.9.

Упражнение 8 Какими свойствами должно обладать задание, чтобы дерево конфигураций было сворачиваемым и без обобщений?

Упражнение 9 Интерпретатор `intTree` очень просто дополняется обработкой конструкции `let`. Посмотрите, как это сделано в приложении.

¹⁹Когда мы встречаем конструктор, то осуществляем декомпозицию, тем самым уменьшая размер конфигураций.

²⁰Конструкция `let v = e2 in e1` — синтаксический трюк, чтобы машина M_p в качестве следующих шагов вернула $e1$ и $e2$, а не $e1 / (v := e2)$.

Рис. 7.8. Дерево конфигураций для задания $(\text{add}'(x, y), \text{prog1})$ Рис. 7.9. Дерево конфигураций для задания $(\text{add}'(x, y), \text{prog1})$ 

Идея обобщения конфигураций — очень важная составляющая суперкомпиляции. Мы рассмотрели, пожалуй, самый примитивный способ обобщения.

Часть суперкомпилятора, определяющая когда нужно сделать обобщение, на жаргоне суперкомпиляции исторически называется *свистком*. Свистки в большинстве существующих суперкомпиляторов устроены гораздо сложнее, чем рассмотренный здесь свисток.

Термин «свисток» (whistle), несмотря на низкий слог, всем полюбился и стал общепринятым. Свисток является эвристикой, главная задача которой — сигнализировать об опасности появления в дереве конфигураций бесконечных (несворачиваемых) ветвей. Сама задача точного распознавания бесконечных ветвей является алгоритмически неразрешимой (сводится к проблеме останова). Любой свисток лишь делает приближенную оценку и может ошибиться — просвистеть в «нескольких шагах» от хорошего зацикливания.

7.3.7. Прообраз суперкомпилятора

Итак, мы добились того, что для любого задания (e, p) мы умеем конструировать конечный граф конфигураций g . В графике g содержится вся необходимая информация для вычисления задания с любыми аргументами. В требованиях к оптимизатору в разделе 7.3.1 мы поставили задачу получить новое задание (e', p') такое, что для любых аргументов s :

```
sll_run(e, p) s = sll_run(e', p') s
```

В итоге мы конструируем следующий преобразователь программ (см. приложение):

```
transform :: Task → Task
transform (e, p) =
```

```
residuate $ foldTree $ buildFTree
  (driveMachine p) e
```

Функция `buildFTree` строит сворачиваемое (foldable) дерево. И внутри нее, можно сказать, и происходит самое интересное: машина M_p не просто запускается — результат ее запуска анализируется и, возможно, принимается решение сделать обобщение.

Можно сказать, что данный преобразователь программ формально подходит под определение суперкомпилятора из цитаты (см. раздел 7.1). Хотя этот преобразователь практически ничего не делает, в нем уже есть основа для будущего суперкомпилятора. Будем использовать этот преобразователь как своеобразную точку отсчета, с которым мы сравним описываемые далее преобразователи `deforest` и `supercompile`.

Упражнение 10 Попробуйте показать, что для любого задания (e, p) и для любой подстановки s вычисление $\text{sll_run}(e', p') s$ (где e', p' — соответствующее остаточное задание) требует ровно столько же шагов интерпретатора, что и вычисление $\text{sll_run}(e, p) s$. То есть преобразователь `transform` не ухудшает, но и не улучшает задание с точки зрения производительности.

Но изменяется ли в ходе данного преобразования программа? Да, вот пример:²¹

```
(even(sqrt(x)), prog1)  $\xrightarrow{\text{transform}}$  (f1(x), prog1T)
```

Где `prog1T`:

²¹Остаточная программа приведена здесь точно в таком виде, как она выдаётся преобразователем `transform`. В качестве имен новых функций используются идущие в некотором порядке идентификаторы. Конечно, было бы замечательно, если бы при такого рода преобразованиях функциям давались понятные имена.

```

f1(x) = g2(x, x);
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
f3() = True();
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
f10() = False();
g11(Z(), x, v3) = g9(v3, x);
g11(S(v4), x, v3) = f12(v4, v3, x);
f12(v4, v3, x) = g13(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
f19() = True();

```

Видно, как минимум, одно свойство такого преобразования — в программе теперь нет вложенных вызовов функций: получилась «плоская», более «бесчеловечная» программа.²²

7.3.8. КМП-тест

В двух следующих разделах мы добавим к преобразователю `transform` два «трюка», которые являются неотъемлемыми составляющими любого суперкомпилятора, и именно они «ускоряют» остаточное задание. Каждый из трюков можно было бы разобрать на отдельных небольших примерах, однако именно вместе они дают «синергетический эффект». И этот эффект сложно прочувствовать, рассматривая совсем игрушечные программы. Так называемый КМП-тест,²³ рассматриваемый далее, является относительно «реалистичной» программой.

Это тест позволит не только ощутить эффект трюков, но и почувствовать «глубину» преобразования, осуществляемого суперкомпилятором. Один из простых методов оценки оптимизирующего преобразования — посмотреть, может ли преобразование породить некоторый общеизвестный эффективный алгоритм из «наивного» и неэффективного. Эффект суперкомпиляции хорошо демонстрируется на примере генерации из наивного алгоритма сопоставления с образцом и зафиксированного образца эффективного алгоритма сопоставления — такого же, как выдаёт алгоритм Кнута — Морриса — Пратта [16] (отсюда и название)²⁴

²²Эту программу легче исполнить компьютеру (она в «итеративной» форме) и сложнее понять человеку (вы могли бы догадаться, что эта программа делает?).

²³Ставший в своем роде классическим примером в суперкомпиляции, так как на этом примере оказалось легче всего сравнивать суперкомпиляцию с другими методами преобразований, такими как дефорестация, частичные вычисления и т. п. [26, 28].

²⁴Сам алгоритм КМП в общем случае случае в данном примере не выводится, выводится лишь его экземпляр для конкретного паттерна. То есть проис-

Рис. 7.10. `prog2`: поиск подстроки в строке

```

match(p, s) = m(p, s, p, s);

-- matching routine
-- current pattern is empty, so match succeeds
m("", ss, op, os) = True();
-- proceed to match first symbol of pattern
m(p:pp, ss, op, os) = x(ss, p, pp, op, os);

-- matching of the first symbol
-- current string is empty, so match fails
x("", p, pp, op, os) = False();
-- compare first symbol of pattern with first
-- symbol of string
x(s:ss, p, pp, op, os) =
    if(eq(p, s), m(pp, ss, op, os), n(os, op));

-- failover
-- current string is empty, so match fails
n("", op) = False();
-- trying the rest of the string
n(s:ss, op) = m(op, ss, op, ss);

-- equality routines
eq('A', y) = eqA(y); eqA('A') = True();
eqB('A') = False();
eq('B', y) = eqB(y); eqA('B') = False();
eqB('B') = True();
-- if/else
if(True(), x, y) = x;
if(False(), x, y) = y;

```

Рассмотрим программу на рис. 7.10:²⁵ функция `match(p, s)` проверяет, содержится ли строка `p` в строке `s`. Для упрощения мы рассматриваем только строки над алфавитом из двух символов — ‘А’ и ‘В’. Строки представляются списком символов. Далее для экономии места будем использовать стандартные сокращения для записи строк и списков.

Рассмотрим проверку `match(«AAB», s)`, содержит ли подстрока (паттерн) «AAB» в строке `s`. Наша программа будет вычислять такое задание следующим образом: сравнивает ‘A’ с первым символом строки `s`, ‘A’ — со вторым, ‘B’ — с третьим. Если какое-то сравнение заканчивается неудачей, перезапускает серию сравнений для хвоста строки. Однако такая стратегия совсем не оптимальна. Допустим, что строка `s` начинается с «AAA...». Первые два сравнения будут успешными, последнее завершится неудачей. Неэффективно повторять это с хвостом «AA...», так как заранее известно, что первые два сопоставления «AAB» с «AA...» завершаются успехом. Детерминированный конечный автомат, получаемый по алгоритму Кнута — Морриса — Пратта, рассматривает каждый символ строки `s` ровно один раз. В итоге (с помощью преобразователя `supercompile`) мы получим задание, соответствующее такому ДКА.

Наше простое преобразование `transform` дает следующий результат:

ходит эффективная специализация задания.

²⁵Автор извиняется за короткие имена функций — они укорочены только для того, чтобы следующие далее графы конфигураций были разумной ширины.

```
(match("AAB", s), prog2)
  =====> (f1(s), prog2T)
```

где `prog2T`:

```
f1(s) = f2(s);
f2(s) = g3(s, s);
g3("", s) = False();
g3(v1:v2, s) = f4(v1, v2, s);
f4(v1, v2, s) = g5(v1, v2, s);
g5('A', v2, s) = f6(v2, s);
g5('B', v2, s) = f22(v2, s);
f6(v2, s) = f7(v2, s);
f7(v2, s) = g8(v2, s);
g8("", s) = False();
g8(v3:v4, s) = f9(v3, v4, s);
f9(v3, v4, s) = g10(v3, v4, s);
g10('A', v4, s) = f11(v4, s);
g10('B', v4, s) = f20(v4, s);
f11(v4, s) = f12(v4, s);
f12(v4, s) = g13(v4, s);
g13("", s) = False();
g13(v5:v6, s) = f14(v5, v6, s);
f14(v5, v6, s) = g15(v5, v6, s);
g15('A', v6, s) = f16(v6, s);
g15('B', v6, s) = f18(v6, s);
f16(v6, s) = g17(s);
g17("") = False();
g17(v7:v8) = f2(v8);
f18(v6, s) = f19(v6, s);
f19(v6, s) = True();
f20(v4, s) = g21(s);
g21("") = False();
g21(v5:v6) = f2(v6);
f22(v2, s) = g23(s);
g23("") = False();
g23(v3:v4) = f2(v4);
```

Данный большой по объему результат приведен только для того, чтобы послужить «точкой отсчета». Отметим, что в преобразованной программе нет вложенных вызовов функций, но символы из `s`, как и в исходном задании, рассматриваются несколько раз.

7.3.9. Устранение транзитных шагов

Очень часто в графах конфигураций есть участки следующего вида:

```
... c1 → c2 → c3 ...
```

где шаг `c2 → c3` является транзитным (понятие транзитного шага было определено в разделе 7.3.3). Такой участок можно заменить на:²⁶

```
... c1 → c3 ...
```

Например, верхняя часть графа, построенного для нашего КМП-теста преобразователем `transform`, выглядит как показано на рис. 7.11.

Удаляемые транзитные шаги показаны зигзагами. Если от них избавиться, то верхушка «подчищенного» графа будет выглядеть как на рис. 7.12.

Преобразователь `deforest` — модификация преобразователя `transform`, удаляющая транзитные дуги и соответствующие узлы из графа конфигураций (см. приложение).

²⁶Это корректно для вычислений языка SLL, где нет никаких побочных действий. Если бы на шаге $c_1 \rightarrow c_2$ совершался бы, допустим, ввод-вывод, то удаление этого шага было бы некорректным.

Пропустим наш тест через дефорестатор:

```
(match("AAB", s), prog2)
  =====> (f1(s), prog2D)
```

где `prog2D` =

```
f1(s) = g2(s, s);
g2("", s) = False();
g2(v1:v2, s) = g3(v1, v2, s);
g3('A', v2, s) = g4(v2, s);
g3('B', v2, s) = g10(s);
g4("", s) = False();
g4(v3:v4, s) = g5(v3, v4, s);
g5('A', v4, s) = g6(v4, s);
g5('B', v4, s) = g9(s);
g6("", s) = False();
g6(v5:v6, s) = g7(v5, v6, s);
g7('A', v6, s) = g8(s);
g7('B', v6, s) = True();
g8("") = False();
g8(v7:v8) = f1(v8);
g9("") = False();
g9(v5:v6) = f1(v6);
g10("") = False();
g10(v3:v4) = f1(v4);
```

В дефорестированной программе в 2 раза меньше функций, чем в просто преобразованной (10 вместо 23) — были устранины вызовы «промежуточных» функций (грубо говоря, произошел инлайнинг). Хоть остаточная программа и упростилась значительно, мы пока еще не добились желаемого эффекта.

Удаление транзитных дуг есть упрощение графа конфигураций. Это упрощение — один из двух основных механизмов оптимизации с помощью суперкомпиляции. Наиболее эффективным он оказывается, когда применяется вместе со вторым механизмом — распространением информации.

Упражнение 11 Можно устраниять транзитные шаги не из графа конфигураций, а сразу из дерева (до свертки). Какие очень нехорошие «подводные камни» есть у этого варианта?

Немного о дефорестации

Есть отдельный метод преобразования программ под называнием **дефорестация** [37, 3]. Цель дефорестации: уменьшить создание промежуточных структур данных — списков, деревьев (отсюда и название) и т. д., которые возникают при композиции функций. В работе [3] рассматривается дефорестация для языка SLL.²⁷ Классическая дефорестация происходит без обобщения — рассматриваются программы в специальной синтаксической форме (которая гарантирует, что дерево «свернется»). Практическое достоинство дефорестации — то, что для многих синтаксических форм есть так называемая сокращенная дефорестация (shortcut deforestation), для которой можно сразу выписать результат преобразований.

7.3.10. Распространение информации

Рассмотрим граф конфигураций для нашего КМП-теста, порождаемый преобразователем `deforest` и изображенный на рис. 7.13.

В нем есть шаги, где разбирается строка `s` (пустая/непустая — подчеркнуто), а дальше — ниже по дереву — строка `s` опять разбирается (дважды подчеркнуто), хотя уже сделано предположение, что список — непустой. Такое двойное тестирование избыточно.

²⁷Правда, там ему не дается никакого имени.

Рис. 7.11. Верхняя часть графа КМП-теста, транзитные шаги отмечены зигзагами

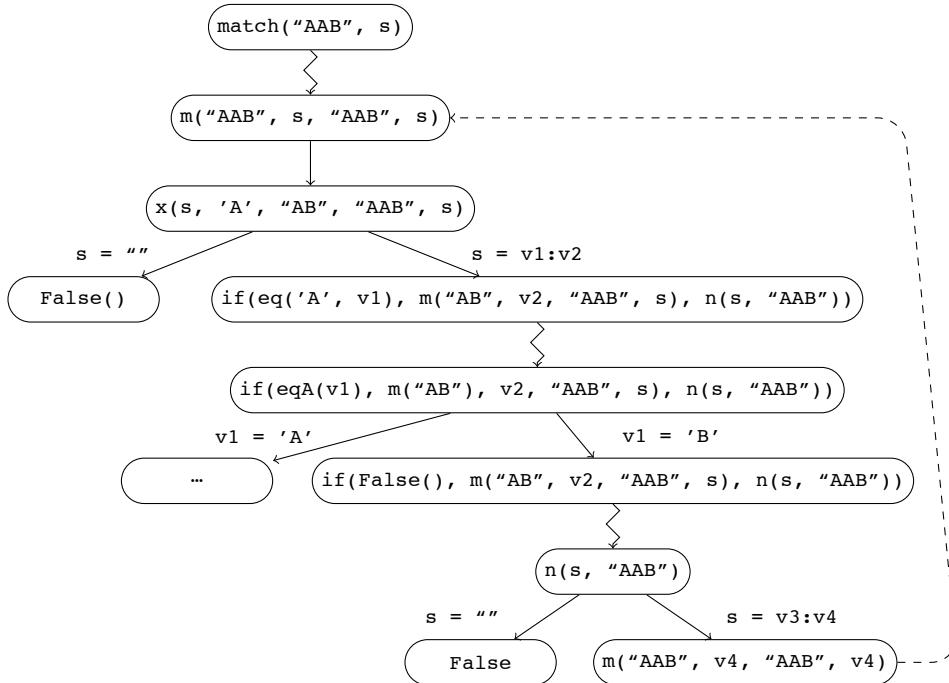


Рис. 7.12. Верхняя часть графа КМП-теста после удаления транзитных шагов

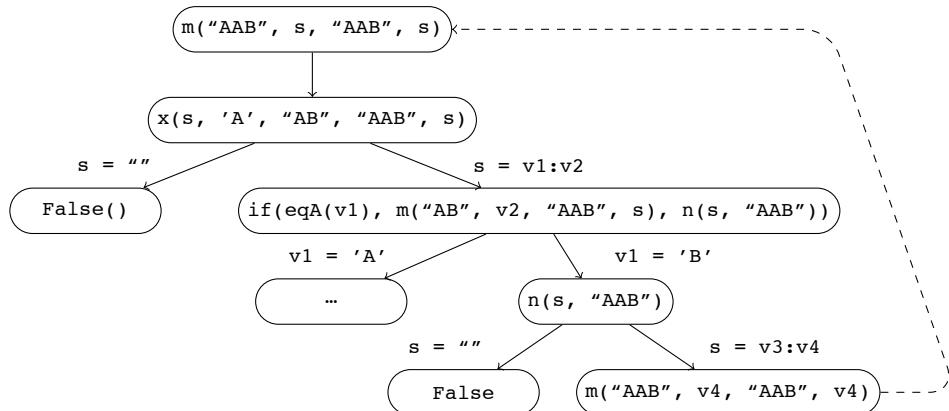
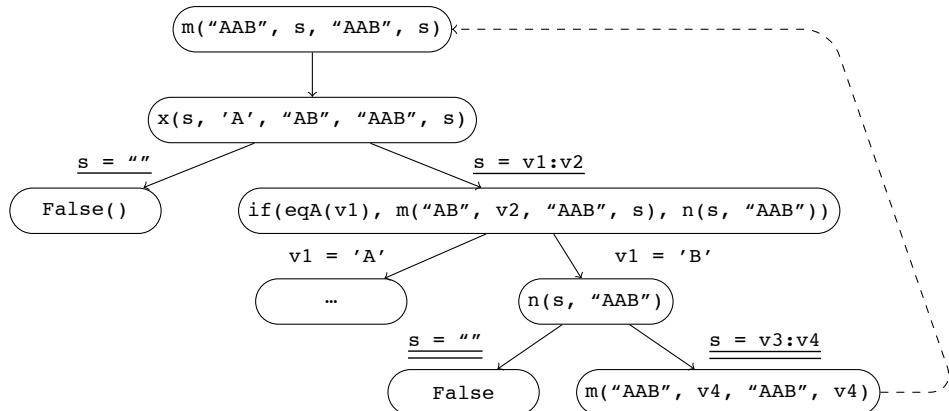


Рис. 7.13. Граф конфигураций КМП-теста после преобразования deforest



В суперкомпиляции такая избыточность устраняется так: всякий раз, когда мы разбираем различные условия — тести-

руем какую-то свободную в выражении, мы распространяем результат тестирования дальше (вниз по соот-

всего в поддереве).

Распространим информацию о том, что строка `s` — не пустая: получится граф на рис. 7.14.

Выигрыш не только в том, что мы избавились от лишней проверки, но мы также выявили транзитный переход, который можно затем удалить. В результате получаем граф, верхушка которого показана на рис. 7.15 (граф приведен полностью в приложении).

Преобразователь `supercompile` отличается от преобразователя `deforest` тем, что на каждом шаге прогонки результат тестирования распространяется на все рассматриваемое состояние. В языке SLL предусмотрены проверки только одного вида — что аргумент соответствует некоторому образцу. Соответственно, мы можем распространить информацию только одного вида — что некоторая конфигурационная переменная соответствует образцу. Это называется *распространением позитивной информации*. Соответственно, суперкомпилятор SC Mini — позитивный суперкомпилятор. Если бы в нашем языке были проверки, что аргумент *не соответствует* образцу, мы могли бы распространять и негативную информацию (о несоответствии образцу). Суперкомпилятор, распространяющий и позитивную, и негативную информацию, называется *перфектным*.²⁸

Результат суперкомпиляции нашего КМП-теста таков:

```
(match("AAB", s), prog2)
  supercompile
  =====> (f1(s), prog2S)
```

где `prog2S` =

```
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

Желаемый эффект достигнут — каждый символ из строки `s` теперь рассматривается не более одного раза.

Упражнение 12 Докажите это утверждение.

Упражнение 13 В получившейся программе есть «транзитные» функции — `f1`, `f6`. Можно безопасно осуществить их инлайнинг — ведь в программе заменить `f1`, `g2` и `f6` на `g7`, а затем выкинуть `f1` и `f6`. Посмотрите внимательно на граф конфигураций, получившийся при суперкомпиляции КМП-теста, приведенный в приложении. Почему SC Mini не осуществил инлайнинг в этом случае?

Если разрешить в языке SLL использовать вложенные образцы (как это делается в Хаскеле), то получившийся результат можно переписать как `(match(s), prog2S')`, где `prog2S'`:

```
match("") = False();
match('A':s) = matchA(s);
match('B':s) = match(s);
matchA("") = False();
```

²⁸Чаще всего встречается промежуточный вариант, когда распространяется только часть негативной информации.

```
matchA('A':s) = matchAA(s);
matchA('B':s) = match(s);
matchAA("") = False();
matchAA('A':s) = matchAA(s);
matchAA('B':s) = True();
```

По такой программе строится конечный автомат, приведенный на рис. 7.16.

Распространение информации — второй механизм оптимизации с помощью суперкомпиляции (первый — упрощение графа конфигураций через удаление транзитных дуг).

Распространение информации имеет тройной эффект:

- 1) Переменная не тестируется дважды — очевидная оптимизация.
- 2) Как следствие, появляются транзитные дуги, которые затем удаляются.
- 3) В графе конфигураций исчезают ветви, по которым никогда не пойдет вычисление — в остаточной программе удаляется «мертвый» код.

В рассмотренном выше примере в графе, построенном преобразователем `deforest`, есть такой путь:

```
{s = v1:v2} → ... → {s = ""} →
```

Ясно, что никакое конкретное вычисление не может пройти по этому пути. Однако при дефорестации этот путь переходит в остаточную программу — как следствие, в дефорестированной `prog2d` (см. выше) программе есть функция `g10`, первый аргумент которой никогда не может быть пустой строкой:

```
g10("") = False();
g10(v3:v4) = f1(v4);
```

В результате суперкомпиляции КМП-теста получилась программа, в которой отсутствует мертвый код. Однако в общем случае получить программу без мертвого кода нельзя — это алгоритмически неразрешимая задача.²⁹

Упражнение 14 В большинстве работ по суперкомпиляции распространение информации рассматривается как обязательная операция. В SC Mini распространение информации выделено как отдельный логический шаг — модификация машины \mathcal{M}_p :

```
addPropagation :: Machine Conf → Machine Conf
```

7.3.11. Квинтэссенция суперкомпилятора SC Mini

Данная глава получилась очень длинной, поэтому будет полезно взглянуть на изложенный в ней материал под несколько иным углом «для закрепления пройденного».

Суперкомпилятор SC Mini является попыткой описания минимального суперкомпилятора для программиста на Хаскеле. Главная цель SC Mini — внятно выделить основные составляющие, присутствующие в любом суперкомпиляторе, и показать, как эти составляющие сочетаются друг с другом.

Давайте проследим, как легким движением руки преобразователь `transform` (не улучшающий, но и не ухудшающий) превращается в `deforest`, который, в свою очередь, превращается в `supercompile`.

Безликий преобразователь `transform`:

²⁹То есть невозможно сконструировать преобразователь, который бы для любого SLL-задания выдавал программу без лишнего кода. Лишний код появляется в результате обобщения.

Рис. 7.14. Граф конфигураций КМП-теста после распространения информации о непустоте s

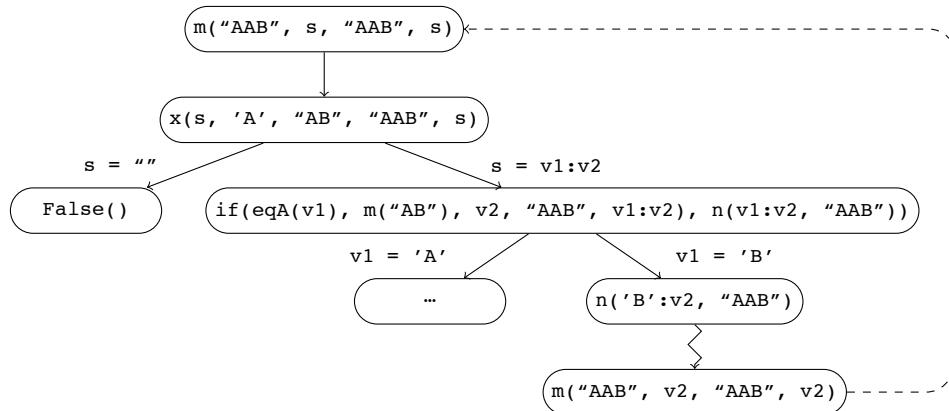


Рис. 7.15. Граф конфигураций КМП-теста после распространения информации о непустоте s и удаления транзитного перехода

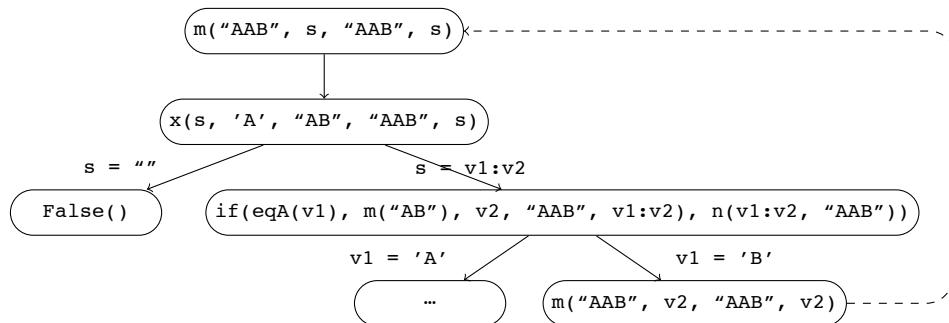
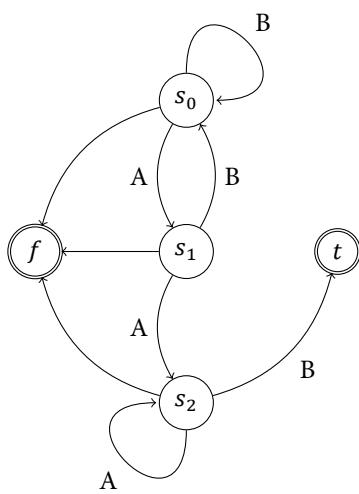


Рис. 7.16. Конечный автомат, соответствующий результату суперкомпиляции КМП-теста

```
deforest :: Task → Task
deforest (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (driveMachine p) e
```



Добавляем распространение информации:

```
supercompile :: Task → Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation $ driveMachine p) e
```

За счет распространения результатов тестов на всю конфигурацию и дальнейшего упрощения графа конфигураций наш минимальный суперкомпилятор SC Mini оказался способен свести неоптимальный наивный алгоритм поиска подстроки в строке к хорошо известному эффективному КМП-алгоритму.

7.3.12. Не только оптимизация

Суперкомпилятор SC Mini способен доказывать теоремы о программах. Покажем, что операция `add`, определенная в программе `prog1`, ассоциативна, то есть для любых аргументов `args`:

```
sll_run (add(add(x,y),z), p1) args ==
        sll_run (add(x,add(y,z)), p1) args
```

Можно доказать это по индукции (рассматривая все правила операционной семантики SLL). А можно рассмотреть преобразования этих заданий. Суперкомпилятор SC Mini приводит оба эти задания к одной и той же *текстовой* форме:

```
transform :: Task → Task
transform (e, p) =
    residuate $ foldTree $
        buildFTree (driveMachine p) e
```

Добавляем упрощение графа конфигураций посредством удаления транзитных шагов:

```
(add(add(x, y), z), p1)  $\xrightarrow{\text{supercompile}}$ 
  (g1(x, y, z), prog3S')
(add(x, add(y, z)), p1)  $\xrightarrow{\text{supercompile}}$ 
  (g1(x, y, z), prog3S')
```

где `prog3S'`:

```
g1(z(), y, z) = g2(y, z);
g1(s(v1), y, z) = s(g1(v1, y, z));
g2(z(), z) = z;
g2(s(v1), z) = s(g2(v1, z));
```

Отсюда сразу следует ассоциативность операции `add`, так как:

```
sll_run (add(add(x, y), z), p1) args ==
sll_run (g1(x, y, z), prog3S') args ==
sll_run (add(x, add(y, z)), p1) args
```

7.4. Проблемы

В этом разделе я постараюсь выявить основные недостатки нашего суперкомпилятора с точки зрения промышленного программирования. В той или иной степени все перечисленные моменты присутствуют во всех существующих суперкомпиляторах.

7.4.1. Непредсказуемость результатов

Рассмотренный КМП-тест был на самом деле, что называется, образцово-показательным примером. Нам повезло в том, что свисток не сработал и удалось построить граф конфигураций без обобщений.

Однако в большинстве других случаев размер конфигураций в графе по мере его построения будет увеличиваться, и рано или поздно придется делать обобщение — чтобы гарантировать завершаемость.

Вернемся к ранее рассмотренному заданию:

```
(even(sqrt(x)), prog1)
```

При дефорестации этой программы удается построить график конфигураций, не прибегая к обобщениям. При суперкомпиляции же этого задания без обобщения не обойтись — это плата за распространение информации: распространяя информацию, мы «раздуваем» и усложняем конфигурации. В приложении даны результаты дефорестации и суперкомпиляции этого задания, а также сравнение скорости работы дефорестированных и суперкомпилированных программ. На основе материала, приведенного в приложении, делаются следующие выводы:

- 1) Размер суперкомпилированной программы намного больше, чем размер дефорестированной программы.
- 2) Для малых чисел `x` суперкомпилированная программа работает быстрее, чем дефорестированная. Однако для больших чисел `x` суперкомпилированная программа работает медленнее.

Это — антипод КМП-теста: отрицательный пример, показывающий основные слабости суперкомпилятора SC Mini. (Будем для краткости называть его анти-КМП-тест. Такой пример при желании находится для любого из существующих суперкомпиляторов.)

Если с тем, что суперкомпилятор может упускать некоторые возможности оптимизации, мириться можно, то с тем, что суперкомпилятор склонен раздувать остаточную программу, мириться труднее. Опасность раздухания остаточного кода

(code explosion) — большая проблема суперкомпиляции. В нашем случае мы ограничиваем размер выражений в узлах графа конфигураций — то есть, в какой-то степени ограничиваем рост графа «в ширину» и гарантируем завершаемость. Однако мы не ограничиваем рост «в глубину» — граф (даже после удаления транзитных дуг) получается очень глубоким и, как следствие — раздутая остаточная программа.

Упражнение 15 Попробуйте изменить SC Mini, ограничив рост графа в глубину. Что получится?

Упражнение 16 Можно ли оценить размер графа для конкретной программы?

Самое плохое, что непредсказуемость суперкомпилятора может заметно проявляться даже для небольших программ.

Суперкомпилятор SC Mini всегда выдаёт программу, которая не менее эффективна, чем исходная, если считать эффективность в шагах интерпретатора. Однако эффективность заключается не только в скорости работы, но и в использовании памяти. Практически все суперкомпиляторы могут в некоторых случаях выдавать программы, которые будут быстрее, чем исходные, но будут потреблять намного больше памяти.

7.4.2. Плохая масштабируемость

Суперкомпилятор имитирует поведение программы, стараясь учесть взаимодействие всех частей программы. К сожалению, размер модели — графа конфигураций — очень нелинейно зависит от размера программы. То есть, увеличение размера исходной программы на 30% может привести к тому, что суперкомпилятор будет работать в десятки раз дольше.

Упражнение 17 Попробуйте найти такие примеры для SC Mini.

Более того, у подавляющего большинства суперкомпиляторов время работы нелинейно зависит от размера графа конфигураций.

Упражнение 18 Покажите, что последнее утверждение верно и для суперкомпилятора SC Mini.

А это значит, что суперкомпиляция больших программ — весьма непредсказуемый процесс: в худшем случае суперкомпилятор будет очень долго работать и выдаст огромную программу, которая будет работать ненамного быстрее, чем исходная.

Очевидная область исследований — суперкомпиляция программы «по частям», при которой программа рассматривается как совокупность модулей, которые суперкомпилируются по отдельности. К сожалению, это пока еще никто не исследовал.

7.4.3. А как отлаживаться?

Суперкомпиляция преобразует исходные тексты программ — то, с чем потом работает обычный компилятор (или интерпретатор). Можно ли отлаживать суперкомпилированный код? Конечно, можно, если об этом заранее позаботиться. Другое дело, что чтобы обеспечить возможность такой отладки, потребуется сделать большой объем технической работы. Ясно, что пока нет востребованного суперкомпилятора, мало кто захочет такими исследованиями заниматься.

7.4.4. Детали, детали, детали...

Мы рассмотрели очень маленький, «игрушечный» суперкомпилятор для игрушечного языка, который обладает заметными недостатками. Сконструировать суперкомпилятор для игрушечного языка, лишенный перечисленных недостатков —

7.5. История вопроса

большая и серьезная, можно даже сказать, фундаментальная задача, достойная научной степени.³⁰

Если говорить о суперкомпиляторах для промышленных языков программирования, то сразу же возникает масса деталей, которые ведут к усложнениям.

Глобальное состояние, побочные эффекты Рассмотренный суперкомпилятор использует предположение о композициональности вычислений, позволяющее корректно делать обобщение:

$$\mathcal{I}_p[e_1/\{v := e_2\}] = \mathcal{I}_p[e_1/\{v := \mathcal{I}_p[e_2]\}]$$

Выражение e_2 можно вычислить вне контекста, в котором оно используется.

Однако если в языке есть глобальное состояние и/или побочные эффекты (а в каком промышленном языке этого нет?), то понятие композициональности в лучшем случае усложняется, а в худшем — теряет всякий смысл. Как с этим бороться? — необходим дополнительный анализ, когда можно распространять информацию, а когда — нет.³¹ Еще сложнее становится, когда появляется многопоточность. Однако сейчас наблюдается такая положительная тенденция в современных (прогрессивных) языках программирования, как возможность выделения «функционального кода» (например, это уже сделано в языке D 2.0 и планируется сделать в Scala). Самое распространенное и прямолинейное решение для суперкомпилятора — оставлять «опасные» куски кода без изменений. В таком случае оптимизируется только чистый функциональный код.

Борьба со строгостью и ленивостью Суперкомпилятор SC Mini оказался таким простым изнутри, потому что мы выбрали простую и элегантную (с точки зрения теории) семантику для языка SLL — ленивые вычисления с вызовом по имени (call-by-name). Именно поэтому прогонка (моделирование интерпретатора) оказалась достаточно простой. Однако семантика вызова по имени плоха с практической точки зрения и почти не используется в современных языках программирования. В мире функционального программирования используются семантики call-by-value и call-by-need. Суперкомпилятор, сохраняющий все тонкости вычисления для таких семантик, устроен намного сложнее.

Упражнение 19 Напишите интерпретатор SLL для семантики call-by-value. Найдите пример программы, смысл которой исказится после суперкомпиляции суперкомпилятором SC Mini.

Упражнение 20 Напишите интерпретатор SLL для семантики call-by-need. Насколько он оказался сложнее?

7.4.5. Почему же ей все-таки занимаются?

Букет проблем суперкомпиляции немал, результаты (пока что) непредсказуемы. Почему же тогда интерес к суперкомпиляции в последние три года существенно растет?

Во-первых, потому, что под определенным углом зрения многие методы оптимизации программ являются частным случаем суперкомпиляции. Это и уже упомянутая дефорестация, и частичные вычисления, и сплавка (fusion) разных видов,

³⁰Речь, конечно же, не идет о тривиальных суперкомпиляторах, не меняющих программу: $Sc\ p = p$.

³¹В случае языка SLL распространяемая информация не меняется, однако в случае изменяющего состояния необходимо следить, что распространяемая информация не изменилась.

и инлайнинг, и дефункционализация, и т. д.³² Это сильно впечатляет, когда простой суперкомпилятор может обрабатывать некоторые программы не хуже, чем сложный узкоспециализированный инструмент.

Во-вторых, из-за концептуальной простоты — идея суперкомпиляции не привязана к какому-то конкретному языку программирования. Она не привязана даже к группе языков, хотя и разработана на нынешний момент лучше всего для функциональных языков.

В-третьих, из-за многогранности идеи — идеи суперкомпиляции применимы не только к оптимизации программ, но и к анализу программ [17, 18, 15, 40]. Очень заманчива перспектива получить с помощью суперкомпиляции «два в одном» — оптимизацию и анализ программ в одном флаконе.

В-четвертых, из-за философии и фигуры автора идеи суперкомпиляции — В. Ф. Турчина:³³ Суперкомпиляция — лишь верхушка большого айсberга, в центре которого находится теория метасистемного перехода. Об этом — кратко в следующем разделе.

7.5. История вопроса

Валентина Федоровича Турчина (1931—2010) — автора идеи суперкомпиляции — я бы кратко охарактеризовал как Программист-Философ.

«В. Ф. Турчин родился в 1931 г. в Москве. Окончил физический факультет МГУ и с 1953 по 1964 г. работал под Москвой в Обнинске в Физико-энергетическом институте, где изучал рассеяние медленных нейтронов в жидкостях и твердых телах и защитил докторскую диссертацию. В 33 года он уже был известным физиком-теоретиком с большими перспективами. И тем не менее в 1964 г. В. Ф. Турчин оставляет физику, переходит в Институт прикладной математики АН СССР (ныне Институт им. М. В. Келдыша) и погружается в информатику. ... Он оставил науку ради мета науки.³⁴»

«В разделе “Сумасшедшие теории и мета наука” я высказал мысль, что для того, чтобы разрешить трудности в современной теории элементарных частиц, надо разработать методы “мета науки”, т. е. теории о том, как строить теории. Причины я усматривал в том, что основные понятия физики на ранних стадиях ее развития брались из нашей интуиции макроскопического мира. Но для познания законов микромира (а точнее, для построения математических моделей этого мира) наша “макроскопическая” интуиция неадекватна. Если интуиция не дает нам прямую тех “колесиков”, из которых можно строить модели микромира, то нам нужны какие-то теории о том, как эти колесики выбирать и как модели строить. Это и будет мета наука.³⁵»

В 1966 году Турчин придумывает РЕФАЛ³⁶ — язык, сильно отличающийся от существовавших тогда (да и от существующих ныне тоже) языков программирования и предназначенный в первую очередь для описания и обработки других языков. Не буду вдаваться в детали языка РЕФАЛ — он описан в

³²Во многих статьях это утверждение делается авторами искренно, но неформально. Было бы крайне интересно увидеть исследование, где это показывается убедительно — например чью-нибудь дипломную работу.

³³По моему личному ощущению большинство специалистов по суперкомпиляции поставили бы эту причину в итоге на первое место.

³⁴В. С. Штаркман Предисловие редактора к первому русскому изданию книги «Феномен науки» [44].

³⁵В. Ф. Турчин. Предисловие к книге «Феномен науки», написанное в 1990 [44].

³⁶REFAL = REcursive Functions Algorithmic Language.

отдельной статье в данном номере журнала ПФП. Турчин «зарождает» своими идеями молодежь — начинает работать «подпольный» РЕФАЛ-семинар (каждую неделю по вторникам на квартире у С. А. Романенко, вплоть до 77 года).

Следующие 5—6 лет — работа над эффективной реализацией РЕФАЛА. Сначала над интерпретатором, а затем и над компилятором. Компилятор РЕФАЛА — программа на РЕФАЛе, обрабатывающая программы на РЕФАЛЕ, и выдающая программы на языке сборки. Но ничто не мешает выдавать программы не на языке сборки, а на том же РЕФАЛЕ. Так в 1970-е годы возникает идея прогонки, которую Турчин описывает как «эквивалентные преобразования функций на языке РЕФАЛ». В 1974 прогонка описывается уже в контексте теории метасистемных переходов.

В 77 году Турчин вынужден покинуть СССР, в том же году в США в переводе на английский публикуется один из главных трудов Турчина — книга «Феномен науки», написанная еще 1970-м году в СССР, сверстанная и уже готовая к печати к 1973-му, но так и не вышедшая в силу политических обстоятельств.

Живет и работает в Нью-Йорке, вначале в Courant Institute, затем в City College. С 1989 года, пока позволяет здоровье, ежегодно приезжает в Россию.

В этом году Валентину Федоровичу исполнилось бы 80 лет.

В истории суперкомпиляции можно условно выделить 3 периода:³⁷

1970-е–1990-е. Суперкомпиляция РЕФАЛА С 1979 года Турчин публикует (с соавторами) не один десяток работ по суперкомпиляции и метавычислениям. Как мы сейчас понимаем, в этих работах рассыпано огромное количество интересных и плодотворных идей. Однако многие концепции описаны недостаточно формально, фрагментарно и чаще всего в терминах языка РЕФАЛ, поскольку для Турчина суперкомпиляция была неотделима от языка РЕФАЛ. К сожалению, для неподготовленного человека даже описание прогонки выглядело очень сложно и непонятно. Также, ни в одной работе, посвященной суперкомпиляции Рефала, алгоритм суперкомпиляции не приведен полностью. По этим причинам, несмотря на значительное количество опубликованных работ, к началу 1990-х суперкомпиляция не обрела признания за пределами узкого круга «посвященных». Более того, практически все основные составляющие суперкомпиляции описывались без должной доли формализма — зачастую туманно и расплывчато. Я соглашусь со следующим высказыванием о работах Турчина — оно применимо и к работам по суперкомпиляции: «В. Ф. практически не пишет ничего легкодоступного. Внимательно прочитавших и осмысливших [его] работы, можно и сейчас, наверное, пересчитать по пальцам³⁸...» Важные работы этого периода: [43, 30, 31, 33, 34, 36, 35].

1990-е. Суперкомпиляция функциональных языков первого порядка Работа Андрея Климова и Роберта Глюка «Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree» [5]

³⁷Следующее далее разделение на периоды и выделение наиболее значимых работ субъективны.

³⁸Н. С. Работнов. Давно... В шестидесятые // «Индекс / Досье на цензуру» 1997/2

о сущности прогонки — по сути, первая работа, нацеленная на понимание суперкомпиляции как общего метода — без привязки к языку Рефал. В 1994 Мортен Сёренсен сделал очень важную вещь — в своей магистерской диссертации «Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation» [26] пересказал и сформулировал основные идеи суперкомпиляции для простого функционального языка (SLL). Это была первая работа, в которой были описаны все существенные части суперкомпилятора. После этого последовала лавина работ, «объясняющих» суперкомпиляцию, сравнивающих ее с другими методами преобразований программ. Важные работы этого периода: [5, 26, 38, 27, 6, 29, 28, 39].

2000-е. Суперкомпиляция функциональных языков высшего порядка Первая половина нулевых — некоторое затишье. Затем — всплеск интереса к суперкомпиляции функций высших порядков. РЕФАЛ — язык первого порядка, работы девяностых годов также не затрагивали высший порядок. В случае высшего порядка открылось много интересных деталей, возможностей и проблем. Проводятся международные семинары (workshops) META-2008 [4] и META-2010 [25], главной темой которых является суперкомпиляция. Оценить важность работ этого периода можно будет лет через десять.

7.5.1. Текущие направления

Вначале предполагалось, что в данной статье будет два отдельных раздела — «Подходы» и «Текущие направления», где будет дан обзор того, что сейчас происходит в суперкомпиляции. Оказалось, что это достаточно сложно сделать: современный ландшафт суперкомпиляции очень пестр и многие работы посвящены достаточно узко специализированным темам. Деятельность, связанная с суперкомпиляцией, с одной стороны чрезвычайно многогранна. С другой стороны, многие интересные идеи там еще не выкристаллизовались, их объяснение было бы туманным и сложным. Поэтому я ограничусь кратким обзором существующих суперкомпиляторов (см. следующий раздел).

Можно с уверенностью обозначить лишь главный лейтмотив современных исследований — *создание практически полезного суперкомпилятора*:³⁹

Любопытным читателям, желающим все-таки узнать полное положение дел в суперкомпиляции, рекомендую почитать соответствующие обзоры в диссертациях по суперкомпиляции: [26, 24, 23, 20, 42, 10, 40, 11].

7.6. Существующие суперкомпиляторы

Все перечисленные в данном разделе суперкомпиляторы являются экспериментальными и очень разными. Заинтересованный читатель может ознакомиться с литературой, ссылки на которую даются ниже.

SCP4⁴⁰ Суперкомпилятор SCP4 для языка РЕФАЛ-5 является своего рода итогом работ по суперкомпиляции РЕФАЛА. SCP4 использует свойства языка РЕФАЛ (ассоциативность конкатенации) и, помимо методов суперкомпиляции, включает дополнительные инструменты: распознавание частично рекурсивных константных функций, распознавание частично рекурсивных мономов конкатенации, нахождение и анализ выходных форматов. Основная информация о SCP4 собрана

³⁹Как сказали бы раньше, применимого в народном хозяйстве.

⁴⁰<http://www.botik.ru/pub/local/scp/refal5/refal5.html>

7.7. Вместо заключения

в работе [42] и монографии [41]. Стоит отметить, что суперкомпилятор SCP4 может расширять область определения программы в следующем смысле: если на каких-то входных данных исходная программа завершалась с ошибкой или зацикливалась, то преобразованная программа может завершаться и выдавать некоторое значение.

Суперкомпилятор языка TSG⁴¹ Язык TSG – упрощенный «плоский ЛИСП» (без вложенных вызовов функций). Помимо учебного суперкомпилятора языка TSG, описанного в [39], для языка TSG реализованы методы метавычислений, которые изначально возникли в контексте суперкомпиляции, но которым не уделялось (и, к сожалению, не уделяется сейчас) должного внимания — окрестностный анализ, окрестностное тестирование, инверсное программирование, нестандартные семантики (описаны в [38]).

Jscp⁴² Суперкомпилятор для языка Java. Первый суперкомпилятор для нефункционального языка. Один из основных результатов работы: суперкомпилировать нефункциональные языки — сложно. Описан в [12]. Исходные тексты Jscp закрыты.

Суперкомпилятор языка Timber⁴³ Timber — чистый (pure) объектно-ориентированный язык с передачей параметров по значению. Суперкомпилятор реализован для функциональной части языка Timber. Главная цель проекта — добиться того же эффекта оптимизации, что и в случае суперкомпиляции для языка с передачей параметров по имени, полностью сохраняя семантику программы. Описан в [10, 11].

Supero⁴⁴ Суперкомпилятор для подмножества Хаскеля. Первый суперкомпилятор для языка с семантикой call-by-need. Главная цель — сохранить ленивость call-by-need при суперкомпиляции. Описан в [20, 21].

SPSC⁴⁵ SPSC — учебный суперкомпилятор для рассмотренного здесь языка SLL. Цель проекта — реализовать суперкомпилятор, соответствующий позитивному суперкомпилятору, описанному (но не реализованному) в работах [29, 28]. Описание SPSC дано в [14].

HOSC⁴⁶ HOSC — суперкомпилятор подмножества языка Хаскеля. В отличие от других суперкомпиляторов, главная цель которых — оптимизация программ, основная цель суперкомпилятора HOSC — анализ программ. Помимо обычной суперкомпиляции, суперкомпилятор HOSC выполняет двухуровневую суперкомпиляцию, основанную на выявлении и применении улучшающих лемм. Описан в [40, 13].

Optimusprime⁴⁷, пакет на hackage⁴⁸ Оптимизирует функциональные программы, которые предполагается выполнять на FPGA-программируемом процессоре (Reduceron). Описан в [22].

CHSC⁴⁹ Суперкомпилятор подмножества Хаскеля. Цель — встроить суперкомпиляцию в компилятор GHC. Главная особенность — в отличие от всех других суперкомпиляторов, не делает λ -лифтинг вложенных определений. Описан в [1].

Дистиллятор⁵⁰ В большинстве суперкомпиляторов конфигурации представлены в виде выражений со свободными переменными. В дистилляции конфигурации представлены графами (то есть возникают такие сложные конструкции как графы, к узлах которых находятся графы): свертка и обобщение выполняются на графах. Дистилляция описана в [7, 8].

7.7. Вместо заключения

Главная цель, которую я перед собой ставил: взятно, обозримо и модульно описать на языке Хаскель основные составляющие минимального суперкомпилятора и показать, как они соединяются вместе. В итоге получилось следующее:

```
supercompile :: Task → Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $
        buildFTree (addPropagation $ driveMachine p) e
```

Данная же статья, по сути, лишь попытка объяснить, что делают эти составляющие, и какой, в итоге, может получиться эффект.

Я надеюсь, что читатель заглянет в приложение к данной статье, где приведен полный код **суперкомпилятора SC Mini⁵¹** с разъяснением некоторых технических моментов.

7.7.1. Куда двигаться дальше?

Если вас заинтересовала тема суперкомпиляции и вы не знаете с чего начать, я бы порекомендовал следующее:

- 1) [Видеозаписи лекций⁵²](#) Сергея Абрамова по метавычислениям.
- 2) [Блог Сергея Романенко⁵³](#) посвященный суперкомпиляции.
- 3) В работах [28, 29] доступным языком описываются классические методы суперкомпиляции, оставшиеся за рамками данной статьи — использование в качестве свистка отношения гомеоморфного вложения (*homeomorphic embedding*), тесное обобщение конфигураций (*most specific generalization*).

7.7.2. Приглашение к сотрудничеству

История суперкомпиляции напоминает раскачивание маятника — от простого к сложному, от сложного к простому.

Первый период — суперкомпиляция РЕФАЛА — развитие от простого к сложному: сделать хоть какой-нибудь (сложный) суперкомпилятор, лишь бы он автоматически работал и выдавал разумные результаты. Второй период — суперкомпиляция функциональных языков первого порядка — осмысление полученных результатов, их схематизация и отчуждение. Нынешний период — это опять движение от простого (осмысленного) к сложному (пока еще непонятному). Это что касается практики.

С другой стороны, в области, близкой к суперкомпиляции — частичных вычислениях — очень важные результаты были получены при экспериментах на совсем игрушечных, не имеющих практической ценности, частичных вычислителях для игрушечных языков [9].

⁴¹<http://www.botik.ru/abram/mca/>

⁴²<http://supercompilers.ru>

⁴³<http://timber-lang.org/>

⁴⁴<http://community.haskell.org/ndm/supero/>

⁴⁵<http://code.google.com/p/spsc/>

⁴⁶<http://code.google.com/p/hosc/>

⁴⁸<http://hackage.haskell.org/package/optimusprime>

⁴⁹<https://github.com/batterseapower/chsc>

⁵⁰<https://github.com/barnacle/dhc>

⁵¹<https://github.com/ilya-klyuchnikov/sc-mini>

⁵²<http://vimeo.com/channels/metacomputation>

⁵³<http://metacomputation-ru.blogspot.com>

Одно из крайне интересных для меня направлений для исследований — проведение различного рода экспериментов на простых и понятных суперкомпиляторах. Проблема, что таких суперкомпиляторов пока нет. Данная статья — это попытка движения в этом направлении. Предложенный суперкомпилятор SC Mini невелик, но некоторые части в нем написаны не очень элегантным способом. Буду рад конструктивным предложениям и замечаниям по его улучшению.

Литература

- [1] Bolingbroke M., Peyton Jones S. L. Supercompilation by evaluation // Haskell 2010 Symposium. — 2010.
- [2] Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. — 1977.
- [3] Ferguson A., Wadler P. When Will Deforestation Stop? // 1988 Glasgow Workshop on Functional Programming. — 1988.
- [4] First International Workshop on Metacomputation in Russia / Program Systems Institute. — Pereslavl-Zalesky2008. — July 2–5.
- [5] Glück R., Klimov A. Occam's razor in metacomputation: the notion of a perfect process tree // WSA '93: Proceedings of the Third International Workshop on Static Analysis. — London, UK: Springer-Verlag, 1993. — Pp. 112–123.
- [6] Glück R., Sørensen M. H. A roadmap to metacomputation by supercompilation // Selected Papers From the International Seminar on Partial Evaluation. — Vol. 1110 of LNCS. — 1996. — Pp. 137–160.
- [7] Hamilton G. W. Distillation: extracting the essence of programs // Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation / ACM Press New York, NY, USA. — 2007. — Pp. 61–70.
- [8] Hamilton G. W. A graph-based definition of distillation // Second International Workshop on Metacomputation in Russia. — 2010.
- [9] Jones N. D., Gomard C. K., Sestoft P. Partial evaluation and automatic program generation. — Prentice-Hall, Inc., 1993.
- [10] Jonsson P. — Positive supercompilation for a higher-order call-by-value language. — Licentiate thesis, Luleå University of Technology, 2008.
- [11] Jonsson P. Time- and Size-Efficient Supercompilation: Ph.D. thesis / Luleå University of Technology. — 2011.
- [12] Klimov A. An approach to supercompilation for object-oriented languages: the java supercompiler case study // First International Workshop on Metacomputation in Russia. — 2008.
- [13] Klyuchnikov I. Towards effective two-level supercompilation: Preprint 81: Keldysh Institute of Applied Mathematics, 2010.
- [14] Klyuchnikov I., Romanenko S. SPSC: a Simple Supercompiler in Scala // PU'09 (International Workshop on Program Understanding). — 2009.
- [15] Klyuchnikov I., Romanenko S. Proving the equivalence of higher-order terms by means of supercompilation // Perspectives of Systems Informatics. — Vol. 5947 of LNCS. — 2010. — Pp. 193–205.
- [16] Knuth D. E., Morris J. H., Pratt V. R. Fast pattern matching in strings // SIAM Journal on Computing. — 1977. — Vol. 6. — P. 323.
- [17] Lisitsa A., Nemytykh A. Verification as a parameterized testing (experiments with the scp4 supercompiler) // Programming and Computer Software. — 2007. — Vol. 33, no. 1. — Pp. 14–23.
- [18] Lisitsa A., Nemytykh A. Reachability analysis in verification via supercompilation // International Journal of Foundations of Computer Science. — 2008. — Vol. 19, no. 4. — Pp. 953–969.
- [19] Mitchell J. C. Foundations for programming languages. — MIT Press, 1996. — Русский перевод: Митчелл Дж. Основания языков программирования. — М.-Ижевск: НИЦ «Регулярная и хаотическая динамика», 2010.
- [20] Mitchell N. Transformation and Analysis of Functional Programs: Ph.D. thesis / University of York. — 2008.
- [21] Mitchell N. Rethinking supercompilation // ICFP 2010. — 2010.
- [22] Reich J. S., Naylor M., Runciman C. Supercompilation and the Reducer // Second International Workshop on Metacomputation in Russia. — 2010.
- [23] Secher J. Driving-Based program transformation in theory and practice: Ph.D. thesis / Department of Computer Science, Copenhagen University. — 2002.
- [24] Secher J. P. — Perfect Supercompilation. — Master's thesis, Department of Computer Science, University of Copenhagen, 1999.
- [25] Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia / Program Systems Institute. — Pereslavl-Zalesky2010. — July 1–5.
- [26] Sørensen M. H. — Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. — Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.
- [27] Sørensen M. H., Glück R. An algorithm of generalization in positive supercompilation // Logic Programming: The 1995 International Symposium / Ed. by J. W. Lloyd. — 1995. — Pp. 465–479.
- [28] Sørensen M. H., Glück R. Introduction to supercompilation // Partial Evaluation. Practice and Theory. — Vol. 1706 of LNCS. — 1998. — Pp. 246–270.
- [29] Sørensen M. H., Glück R., Jones N. D. A positive supercompiler. // Journal of Functional Programming. — 1996. — Vol. 6, no. 6. — Pp. 811–838.

- [30] *Turchin V. F. The Language Refal: The Theory of Compilation and Metasystem Analysis.* — Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.
- [31] *Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems (TOPLAS).* — 1986. — Vol. 8, no. 3. — Pp. 292–325.
- [32] *Turchin V. F. Program transformation by supercompilation // Programs as Data Objects.* — Vol. 217 of *LNCS*. — Springer, 1986. — Pp. 257–281.
- [33] *Turchin V. F. The algorithm of generalization in the supercompiler // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop.* — 1988.
- [34] *Turchin V. F. Program transformation with metasystem transitions // Journal of Functional Programming.* — 1993. — Vol. 3, no. 03. — Pp. 283–313.
- [35] *Turchin V. F. Metacomputation: Metasystem transitions plus supercompilation // Partial Evaluation.* — Vol. 1110 of *Lecture Notes in Computer Science*. — Springer, 1996. — Pp. 481–509.
- [36] *Turchin V. F. Supercompilation: Techniques and results // Perspectives of System Informatics.* — Vol. 1181 of *LNCS*. — Springer, 1996.
- [37] *Wadler P. Deforestation: Transforming programs to eliminate trees // ESOP '88.* — Vol. 300 of *LNCS*. — Springer, 1988. — Pp. 344–358.
- [38] Абрамов С.М. Метавычисления и их применение. — Наука-Физматлит, 1995.
- [39] Абрамов С.М., Пармёнова Л.В. Метавычисления и их применение. Суперкомпиляция. — Институт программных систем РАН, 2006.
- [40] Ключников И.Г. Выявление и доказательство свойств функциональных программ методами суперкомпиляции: Кандидатская диссертация / Институт прикладной математики им. М.В. Келдыша РАН. — 2010.
- [41] Немытых А.П. Суперкомпилятор SCP4. Общая структура. — Москва: ЛКИ, 2007.
- [42] Немытых А.П. Специализация функциональных программ методами суперкомпиляции: Кандидатская диссертация / Институт программных систем РАН. — 2008.
- [43] Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛЕ: Труды ЦНИПИАСС 6: ЦНИПИАСС, 1974.
- [44] Турчин В.Ф. Феномен науки: Кибернетический подход к эволюции. — Москва: ЭТС, 2000.