А. П. Немытых

СУПЕРКОМПИЛЯТОР SCP4

ОБЩАЯ СТРУКТУРА

<SCP4 .... ( )>
<Int e.data ( )> SourceInt
<Go ! > Prog

URSS

А. П. Немытых

СУПЕРКОМПИЛЯТОР SCP4

Общая структура

URSS
МОСКВА

ББК 32.81 32.973–018

Настоящее издание осуществлено при финансовой поддержке Российского фонда фундаментальных исследований (проект № 07–07–07002).

Немытых Андрей Петрович

Суперкомпилятор SCP4: Общая структура. — М.: Издательство ЛКИ, 2007. — 152 с.

На основе технологии суперкомпиляции автор реализовал преобразователь функциональных программ SCP4. SCP4 реализован на функциональном языке программирования Рефал-5. Этот же язык является и входным языком для SCP4. В работе мы рассматриваем общую структуру суперкомпилятора SCP4 и показываем несколько примеров преобразований посредством SCP4.

Nemytykh Andrei Petrovich
The Supercompiler SCP4: General Structure

The author constructed a transformer SCP4 of functional programs. The transformer uses the technology known as Turchin's supercompilation. SCP4 was implemented in a functional language Refal-5. The input language for SCP4 is also Refal-5. In the book we consider the general structure of the supercompiler and give a number of examples of transformations.

Издательство ЛКИ. 117312, г. Москва, пр-т Шестидесятилетия Октября, д. 9.
Формат 60×90/16. Печ. л. 9,5. Зак. № 1193.

Отпечатано в ООО «ЛЕНАНД».
117312, г. Москва, пр-т Шестидесятилетия Октября, д. 11А, стр. 11.

ISBN 978–5–382–00365–8

НАУЧНАЯ И УЧЕБНАЯ ЛИТЕРАТУРА

E-mail: URSS@URSS.ru
Каталог изданий в Интернете:
http://URSS.ru
Тел./факс: 7 (495) 135–42–16
URSS Тел./факс: 7 (495) 135–42–46

5029 ID 55806

Оглавление

Предисловие

Технология программирования естественно развивается в сторону оперирования понятиями задачи, которая стоит перед программистом, а не понятиями универсального прибора, на котором программа будет исполняться. Это стимулирует развитие языков программирования высокого уровня, позволяющих адекватно отражать объектную область задачи. К таким языкам, например, относятся функциональные и логические языки (LISP, REFAL, PROLOG, HASKELL, ML, SCHEME и др.), а также различные языки, специализированные на конкретную область их применения. С другой стороны, аппаратная реализация современных широко используемых ЭВМ поддерживает фоннеймановскую модель вычислений, что приводит к неэффективной реализации таких языков — посредством интерпретации — более того, часто не прямой, а косвенной — через другую интерпретацию. К подобной неэффективности приводит и любое структурное программирование само по себе, ибо его целью является создание гибких, легко понимаемых и изменяемых программ. Всё чаще программы вычисляются другими программами, а потому естественно ожидать, что первые будут содержать простейшие структуры, ведущие к накладным расходам, которые никогда бы не допустил квалифицированный программист.

Методы автоматической оптимизации структурированных программ высокого уровня (а не программ, отшлифованных профессиональными программистами на языках программирования низкого уровня) и призваны предоставить свободу развития новым технологиям программирования.

Одним из активно развивающихся здесь направлений является автоматическая специализация программ. Предположим, что вы купили дистрибутив операционной системы LINUX. В момент её установки на вашем компьютере вы должны указать его аппаратные характеристики, т. е. эти характеристики являются аргументами программы-установщика. Возникает желание максимально настроить LINUX на ваше «железо», ибо в другом контексте он вам не понадобится. В этом и состоит задача специализации. Операционную систему вы устанавливаете однажды, и потому стоит предложить поработать автоматическому специализатору, даже если его работа достаточно продолжительна во времени.

Суперкомпиляция есть набор методов автоматической специализации программ, написанных на функциональных языках. Основной механизм суперкомпиляции — метаинтерпретация. Основополагающие идеи суперкомпиляции, как и сам термин, были предложены В. Ф. Турчиным в 70-х годах XX века. Но первый реально работающий свободно распространяемый экспериментальный суперкомпилятор был создан относительно недавно. Описанию его структуры и принципов работы и посвящена предлагаемая читателю книга.

Сам факт существования такого суперкомпилятора является значительным шагом в направлении внедрения технологии суперкомпиляции в практику программного обеспечения современных компьютеров.

Чтобы не усложнять задачи, мы здесь отвлечёмся от разнородности орудий. Пусть теперь количество пороха остаётся всегда одним и тем же, угол же возвышения орудия непрерывно меняется, и пусть ищутся дальность полёта и продолжительность движений ядра в воздухе.

Леонард Эйлер

Введение

В семидесятых годах В. Ф. Турчин предложил ряд идей по автоматическому преобразованию программ, которые назвал суперкомпиляцией [1]. Он поставил задачу создать инструменты для наблюдения за операционной семантикой программы, когда фиксирована функция F, вычисляемая этой программой.

Результатом таких наблюдений должно стать построение нового алгоритмического определения некоторого продолжения функции F. Новый алгоритм строится с целью более быстрого вычисления F на конкретных аргументах.

Позже рядом авторов эти идеи В. Ф. Турчина изучались и в той или иной мере доводились до алгоритмов.

Нам удалось построить экспериментальный суперкомпилятор, предметной областью которого является функциональный язык программирования РЕФАЛ-5. Демонстрация суперкомпилятора доступна на Web-странице в режиме on-line [58]. Обсуждению структуры и принципов работы нашего преобразователя программ и посвящена данная работа. Мы показываем большое количество результатов преобразований нашим суперкомпилятором и комментируем эти примеры. Основные принципы построения суперкомпилятора SCP4 обсуждались с Валентином Фёдоровичем Турчиным. Более того, он инициировал и поддерживал нашу работу.

Само понятие «более быстрого вычисления», безусловно, требует уточнения. Мы имеем в виду некоторое логическое время, хотя на практике часто оно отражает физическое время. Вопрос о соотношении этих времён рассматривается в одном из разделов нашей работы.

Перед прочим, мы позволим себе переиначить высказывание С. А. Романенко: «При решении всякой задачи полезно не терять из виду различие между целью, которая должна быть достигнута, и методами, которые приходится применять для её достижения. В контексте данной работы *оптимизация* программ является целью, а *суперкомпиляция* — одним из методов её достижения. Однако не из чего не следует, что *оптимизация* программ может достигаться только *суперкомпиляцией*. Полезны и другие средства (в числе которых находятся и традиционные методы)» [14]. (*Выделенные слова заменены.*)

---

[1] По нашему мнению, название выбрано весьма неудачно. Суперкомпиляция не является компиляцией, подобно многозначной функции, которая не является функцией (или векторному полю, которое не является полем). Английский вариант "supercompilation" немного более премлем, и было бы правильно «переводить» его словом «надкомпиляция».

Языком реализации нашего оптимизатора также является РЕФАЛ-5. Язык программирования РЕФАЛ (В. Ф. Турчин) — функциональный язык первого порядка с аппликативной (вызовы по значению) семантикой. Грубо говоря, программа на РЕФАЛе представляет собой систему переписывания термов. Предложения упорядочены, и выбор предложения происходит посредством сопоставления с образцом. Для построения термов используются два конструктора. Первый конструктор — конкатенация [2] — бинарный, ассоциативный и используется в инфиксной записи, что позволяет опускать его скобки. Знак пробела служит для обозначения этого конструктора. Второй конструктор одноместный. Синтаксически он обозначается только его скобками, т. е. без имени. Функциональный вызов оформляется угловыми скобками; причём имя вызываемой функции записывается непосредственно после открывающей скобки. В РЕФАЛе все функции являются одноместными, термы принято называть выражениями. Пустая последовательность принадлежит к множеству базисных константных термов и называется «пустым выражением». По определению, это единица конкатенации (левая и правая). Все остальные базисные константные термы называются «символами». Базисные неконстантные термы (переменные): `e.name`, `s.name` и `t.name`. Значением `e`-переменной может быть любое константное выражение, значением `s`-переменной — любой символ, значением `t`-переменной — любой символ или выражение в круглых скобках (указанный выше одноместный конструктор). Ассоциативность конкатенации делает множество РЕФАЛ-термов более выразительным, по сравнению с множеством Lisp-термов.

Пример.

```
$ENTRY Go {
  = <Zip (Church Markov McCarthy Post Steel Turchin Turing)
         (Lambda-calculus Markov-algorithm Lisp
          Post-system Scheme Refal Turing-machine)
    >;
}
/*
```

Результатом работы этой программы является РЕФАЛ-выражение:

```
(Curch Lambda-calculus) (Markov Markov-algorithm)
(McCarthy Lisp) (Post Post-system) (Steel Scheme)
(Turchin Refal) (Turing Turing-machine)

*/
Zip {
   (s.name e.names) (t.value e.values)
            = (s.name t.value) <Zip (e.names) (e.values)>;
       () () = ;
}
```

[2] Приписывание.

В левой части функции Go мы видим пустое выражение (*ничто*). Ниже мы иногда будем использовать метасимвол [] для его обозначения. Правая часть функции Go показывает ассоциативность конкатенации. В левой части второго предложения функции Zip аргументами конструкторов круглых скобок являются пустые выражения, правая часть этого предложения совпадает с пустым выражением.

Самоописание некоторого подмножества РЕФАЛа-5 читатель может найти в разделе 16.2 данной книги. Детальное описание языка доступно в электронном виде [69].

Глава 1. Схема структуры преобразователя программ SCP4

Здесь мы дадим набросок структуры нашего суперкомпилятора. Последующие разделы работы посвящены уточнению этой схемы, понятий и описанию принципов работы некоторых инструментов преобразований.

Под программой мы будем понимать конечную последовательность[3] конечных последовательностей инструкций. Множество номеров первых элементов этих последовательностей назовём именами функций и обозначим Entry.

Языком программирования L назовём четвёрку: Entry, множество программ Prog, множество данных Data и частичную вычислимую функцию $U\colon Prog \times Entry \times Data \mapsto Data_{\perp}$. Ниже мы будем обозначать элементы множеств Entry, Prog, Data именами, совпадающими с именами самих множеств (возможно, с приставкой или/и окончанием). Пусть во множестве программ Prog фиксировано некоторое множество термов — «вызовов функций». Символы < > будут обозначать структурные скобки вызова функции. Имя вызываемой функции будем писать сразу за открывающей угловой скобкой. Функцию U будем называть универсальной функцией языка L или его семантикой, вызов функции <Entry Data> — входной точкой. Реализацией языка L назовём рекурсивное определение его универсальной функции.

Нас будут интересовать языки, допускающие реализацию U, определение которой является некоторым уточнением рекурсивной схемы вида:

```
U(Prog, <Entry Data>)
   = Int(Prog,UpdateStack(4),Step(Prog,<Entry Data>,[]^n) ));
Int(Prog, , Data^n) = Final(Data^n);
Int(Prog, <Name args> Stack, Data^n )
   = Int(Prog,UpdateStack(Stack,
                          Step(Prog,<Name args>,Data^n) ));
UpdateStack(Stack, Stack1, Data^n) = (Stack1 Stack, Data^n);
```

[3] Мы встаём на операционную точку зрения, т. е. предполагаем, что задан алгоритм поиска конкретного определения в программе.

[4] Здесь и в аналогичных местах ниже пробел обозначает пустую строку.

```
Definition: Prog×Name ↦ SubprogramBody
Step: Prog×Task ↦ Stack×Data^n
Final: Data^n ↦ Data

Task ::= <Name args>×Data^n
Stack ::= <Name args> Stack | []
args ::= arg,* arg | []
arg ::= Data | <Name args>
```

Здесь $n \in \mathbb{N}$ и мы подчеркнули синтаксические понятия, а не их семантические значения.

Мы потребуем, чтобы: рекурсивное определение частичной функции `Step` не содержало бесконечных циклов для любой пары `(Name, args)`; таким образом, существует алгоритм, вычисляющий значение этой функции на любых конкретных аргументах либо завершающийся сообщением «функция `Step` на данных аргументах неопределена». Кроме того, число доступов к программе внутри одного шага (осуществляемых через функцию `Definition`) должно быть равномерно ограничено по входным данным. Последние n аргументов $\texttt{Data}^n$ являются данными, изменяющимися только напрямую посредством базисных синтаксических конструкторов. Текущие значения этих аргументов есть куски данных, полностью вычисленные программой, которые могут быть лишь достроены.

Этот алгоритм назовём шагом реализации U, или шагом L-машины U. А саму реализацию U-пошаговой. Алгоритм её вычисления является последовательным вычислением шагов L-машины, он выдаёт результат, когда стек будет исчерпан. Таким образом, программа представляет собой динамическую систему с дискретным временем (временем исполнения функции `Step`).

Пусть дана программа на некотором языке и параметризованная входная точка этой программы, тогда такая пара определяет частичную функцию. По определению, суперкомпилятор преобразует такие пары.

Первым шагом суперкомпилятор переводит программу во входное подмножество языка РЕФАЛ-графов. Язык РЕФАЛ-графов ориентирован на адекватное описание временно́й эффективности и является выходным языком основной стадии преобразований. Это язык более низкого уровня по отношению к РЕФАЛу, но работает с теми же данными. Большая часть алгоритмов оптимизации делает преобразования в терминах РЕФАЛ-графов, тем не менее некоторые свойства преобразуемых алгоритмов проще сформулировать в понятиях самого РЕФАЛа, и соответствующие инструменты используют эти понятия.

В нашей работе мы ограничимся только фрагментом входного подмножества языка РЕФАЛ-графов, допускающим пошаговую реализацию (далее везде в этом разделе мы имеем в виду только этот алгоритмически полный фрагмент). (В частности, мы не рассматриваем примитивы — встроенные функции.) Зафиксируем U-пошаговую реализацию и рассмотрим шаг РЕФАЛ-граф машины `RGMStep`.

Подчеркнём, что *везде далее мы будем предполагать, что реализация фиксирована*. То есть под РЕФАЛом-5 мы имеем в виду не только язык, но и кон-

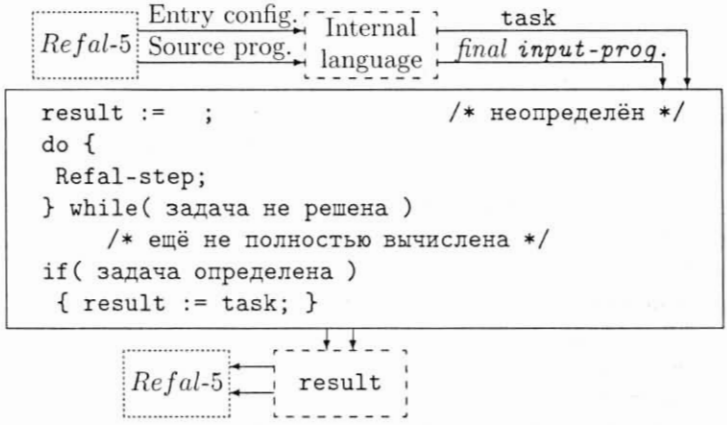


Рис. 1. Общая структура интерпретатора РЕФАЛа

кретную реализацию [75]. На рис. 1 дана блок-схема этой реализации интересующего нас подмножества РЕФАЛа-5.

Прогонкой (`driving`) называется модификация алгоритма `RGMStep`. Она определена на `Prog_Name`×`P_Task`, где `P_Task` — множество параметрических описаний подмножеств множества задач `Task` (определённого нами выше) в некотором фиксированном языке `Param` первого порядка и `Task` ⊆ `P_Task`. Результат прогонки — ориентированное от корня к листьям дерево возможных вычислений `RGMStep` на заданном подмножестве множества `Task`. Будем называть это дерево гроздью (`cluster`). Ветвления в грозди определяются синтаксисом и операционной семантикой преобразуемой программы, ветви перенумерованы. Узлы грозди поименованы временем их создания и содержат информацию о данных и стеке функций, определяющую дальнейший ход возможных вычислений. Таким образом, каждый путь в грозди — от корня к листу — является описанием шага РЕФАЛ-граф машины, когда зафиксированы конкретные подходящие данные, если такие данные существуют. Одной из целей прогонки является исполнение шагов U-пошаговой реализации равномерно по входным данным.

Метааналогом `Int` — основного цикла реализации универсальной функции — служит алгоритм развёртки (`unfolding`), который строит потенциально бесконечные деревья возможных вычислений посредством прогонки, которые связаны, в свою очередь, ориентированным метадеревом последовательных вычислений.

Следующий инструмент — свёртка (`folding`) — факторизует это дерево деревьев в конечный граф посредством структурной индукции, порождая индукционные гипотезы в языке `Param`; пытается их доказать и, если не удаётся, строит более слабые гипотезы и (или) разбивает задачу на несколько подзадач (`P_Tasks`). Эти подзадачи являются корнями деревьев возможных вычислений.

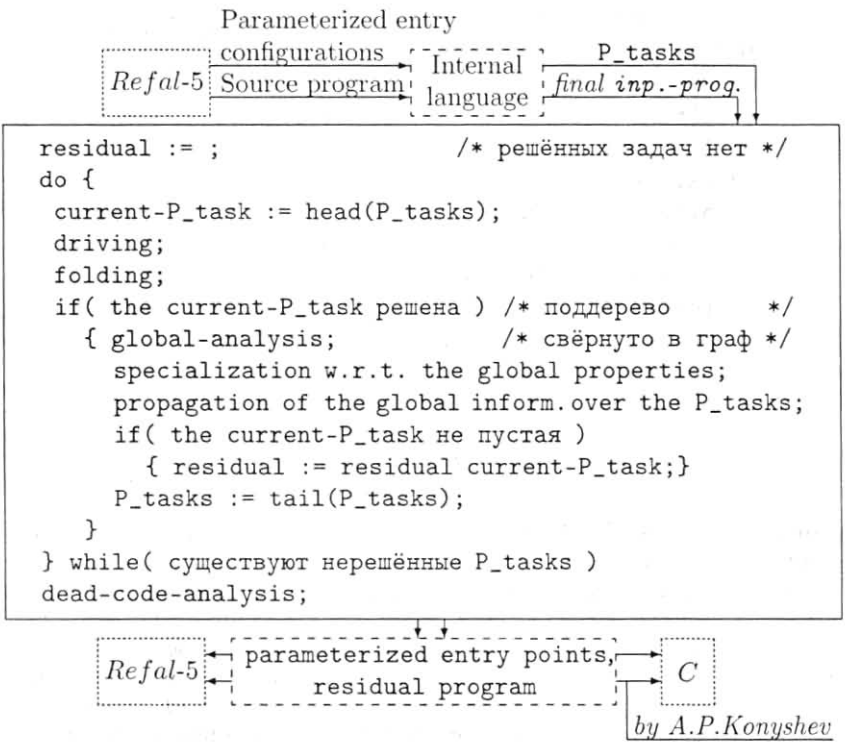


Рис. 2. Общая структура суперкомпилятора SCP4

После каждой удачной попытки полной факторизации некоторой компоненты дерева вычислений анализируются её глобальные свойства; в частности, вычисляется общий структурный формат всех выходов, который, в свою очередь, использует повторная прогонка при выяснении недостижимости конкретного ветвления. Построение нетривиального выходного формата может привести к понижению порядка временно́й сложности алгоритма. Другим важным инструментом глобального анализа является распознавание некоторого достаточного условия тождественности частичных функций. Такой механизм позволяет динамически типизировать рекурсивные данные на этапе преобразований без потери эффективности результата преобразований, в том случае, когда эта типизация остаётся после преобразований посредством других инструментов.

Профакторизованная компонента (`residual`) именуется, заменяется в метадереве своим именем и является кандидатом на оформление в виде функции в конечной программе-результате.

Суперкомпилятор SCP4 допускает ручную разметку в преобразуемой программе, которая может способствовать интересным преобразованиям.

В простейшем случае основная часть преобразований может быть грубо очерчена следующей схемой:

```
SCP4( Prog, <Entry P_Data> )
  = Unfolding(Prog,
              GlobalAnalysis(Folding( ,Driving(
                                Prog,<Entry P_Data>,[]^n))), );

Unfolding( Prog, , P_Data^n, , Residual, Residuals)
  = Final(P_Data^n,Residuals,Residual);
Unfolding( Prog, <Name P_args>, P_Data^n, MetaTree,
           Residual, Residuals)
  = Unfolding(Prog,
              GlobalAnalysis(Folding( MetaTree,
                   Driving(Prog,<Name P_args>,P_Data^n) )),
              Residuals [5] Residual);

Folding: MetaTree×Cluster ↦ P_Task×MetaTree×Residual
GlobalAnalysis:
  P_Task×MetaTree×Residual ↦ P_Task×MetaTree×Residual
Final: P_Data^n×Residuals×Residual ↦ Residuals

Driving: Prog×P_Task ↦ Cluster

P_Task ::= <Name P_args>×P_Data^n
P_args ::= P_arg,* | P_arg
P_arg ::= P_Data | <Name P_args>
```

На рис. 2 показана блок-схема этой рекурсии.

Глава 2. ***Язык параметров***

Входная точка программы определяет начальное состояние преобразуемой части «оперативной памяти» абстрактной машины, которую в РЕФАЛе принято называть полем зрения. Шаг U-пошаговой реализации — логически замкнутое преобразование этого поля зрения, обеспечивающее корректность его структур. Естественно возникает последовательность

```
ViewField(0) = <Entry_Data>;
ViewField(n+1) = Step( Program, ViewField(n) );
```

описывающая эволюцию динамической системы с дискретным временем, когда заданы начальные условия. Мы хотим дать параметрический язык описания множеств полей зрения.

[5] Здесь пробел обозначает операцию приписывания.

2.1. Параметризованные множества данных

Сначала определим язык описания множеств РЕФАЛ-данных: его удобно представить парой pd_+ и pd_-. Первая часть представляет «положительную» информацию о данных, вторая — «отрицательную».

pd ::= ⟦pd_+ , pd_-⟧, здесь жирные скобки являются метасимволами.
pd_+ ::= pterm pd_+ | []
pterm ::= param | (pd_+) | data
param ::= s-parameter | e-parameter | t-parameter
s-parameter ::= s.name
t-parameter ::= t.name
e-parameter ::= e.name
data ::= SYMBOL data | (data) data | []

Параметры типа s – s.name представляют множество всех РЕФАЛ-символов, параметры типа e – e.name представляют множество всех константных выражений (данных РЕФАЛа), параметры типа t – t.name представляют объединение множества всех РЕФАЛ-симолов и множества всех константных выражений в круглых скобках (см. Введение). data — данное РЕФАЛа. Остальные термы из pd_+ интерпретируются по их индукционному построению.

Подчеркнём, что синтаксическое совпадение языка pd_+ с языком образцов РЕФАЛа достаточно случайно и для другого суперкомпилятора язык pd мог бы быть иным (например, мог бы иметь выразительные возможности описания регулярных множеств). Чтобы исключить путаницу между переменными и параметрами, мы будем именовать параметры числами, а переменные идентификаторами (если нет явного указания).

pd_- ::= restriction, pd_- | [] | ∅
restriction ::= $\text{s-parameter}_1 \neq \text{s-parameter}_2$
| s-parameter ≠ SYMBOL | SYMBOL ≠ s-parameter
| $\text{SYMBOL}_1 \neq \text{SYMBOL}_2$ | e-parameter ≠ []

Поясним семантику: $\text{s.name}_1 \neq \text{s.name}_2$ — ограничение множеств значений параметров: два множества не пересекаются; s.name ≠ SYMBOL и SYMBOL ≠ s.name — ограничение множества значений параметра s.name (SYMBOL не принадлежит этому множеству); $\text{SYMBOL}_1 \neq \text{SYMBOL}_2$ — пустое множество, если символы SYMBOL_1 и SYMBOL_2 совпадают; или тавтология, если эти символы графически различны; e.name ≠ [] — ограничение множества значений параметра e.name (это множество не содержит пустого РЕФАЛ-выражения); [] — пустое выражение представляет тавтологию. Графически совпадающие параметры в одной паре pd представляют одно и то же множество. Запятая здесь и в паре, определяющей pd, интерпретируется как знак пересечения множеств. ∅ — обозначает пустое множество.

Множество термов pd предупорядочено отношением включения описываемых его элементами множеств данных. Терм-пара вида ⟦e.name, ⟧ описывает всё множество данных, и потому сравним с любым другим термом-парой и не меньше его. Существуют бесконечные неубывающие цепи различных пар-термов pd, несводящихся друг к другу переименованием параметров.

Пример.
`[ , ], [e.11, ], [e.21 e.22, ], ... [e.N1 e.N2 ... e.NN, ], ...`

Скажем, что замена разных параметров $p_1, \ldots, p_n$ в паре-терме `Pair` из `pd` соответственно термами $t_1, \ldots, t_n$ из `pd` корректна, если $\forall\, i\ (1 \leqslant i \leqslant n)$ в термах t_i нет параметров, графически совпадающих с параметрами из `Pair`, и множество данных, описываемое p_i, включает в себя множество данных, описываемое t_i.

Результат корректной замены параметров в паре-терме `Pair` сравним с `Pair` и не больше его. Приведённый выше пример показывает бесконечное множество графически разных, но равных в смысле нашего частичного предпорядка термов.

2.2. Параметризованные множества полей зрения (стеков) и РЕФАЛ-выражений

Язык параметров, используемый SCP4, является языком первого порядка: нет параметров, в область определения которых входили бы имена функций или конструктор вызова функции. Все имена функций представлены непосредственно сами собой, конструкторы функциональных вызовов `<...>` — константной кодировкой в данных (`Call...`). Внутри скобок — информация о вызове, в частности, описание области определения аргументов и области значения этого вызова на языке описания данных $\mathtt{pd}_+$. Графически совпадающие параметры в аргументах разных вызовов в стеке представляют одно и то же множество. Связи между элементами стека описываются через формальные выходные переменные `out.name`, которые являются лишь фиксированными метаименами значений соответствующих функциональных вызовов. «Отрицательная» часть описания параметров является общей для всех вызовов в стеке.

Пример. `<F s.1>` $\overset{a}{\leftarrow}$ `out.1; <G <F e.2> out.1>` $\overset{a}{\leftarrow}$ `out.0;` (здесь знак $\overset{a}{\leftarrow}$ означает присваивание левой части правой. В более привычных терминах этот пример может быть переписан как **let** `out.1 := <F s.1>;` **in** `out.0 := <G <F e.2> out.1>;`).

Иногда мы будем писать `<F e.2>` $\overset{a}{\leftarrow}$ `e.4`, имея в виду, что область определения правой части ограничивается образом функции `F`.

Глава 3. ***Язык РЕФАЛ-графов***

Под языком РЕФАЛ-графов мы понимаем язык, позволяющий показать структуру промежуточного состояния РЕФАЛ-графа в процессе его преобразований. Входная часть этого языка допускает U-пошаговую реализацию и является собственным подмножеством не только всего языка РЕФАЛ-графов, но и его выходной части.

3.1. Синтаксис

```
scp-view-filed ::= + meta-branch (FOREST forest)
meta-branch ::= graph-call meta-branch|walk-segment meta-branch| []
forest ::= current-graph roots | []
graph-call ::= assignments graph output
current-graph ::= block
roots ::= root*
root ::= (NODE node-cont)
node-cont ::= node-id colour restriction stack basics
colour ::= PIVOT | SECONDARY | PASSIVE
node-id ::= INTEGER
```

Здесь `restriction stack` — параметризованное описание стека функций. Имя `name` каждого узла уникально; `basics` будет уточнён позже.

Описанные выше структуры определяют промежуточное состояние преобразуемого графа. Первая из них — поле зрения суперкомпилятора. Содержимое поля зрения однозначно определяет дальнейшие шаги преобразований. `meta-branch` содержит «более преобразованные» («более остаточные») структуры; первая подструктура `current-graph` структуры `forest` является текущим объектом преобразований, а «корни» `roots` «леса» — это последовательность подзадач, преобразование которых будет происходить позже. Эти корни представляют собой тривиальные графы, состоящие из одной вершины, и могут иметь общие параметры.

```
graph ::= input-format NAME { branching } output-format
input-format ::= (INPF id-assigns)
output-format ::= residual-output-format | inductive-output-format
residual-output-format ::= (OUTF id-assigns)
inductive-output-format ::= (IND-OUTF id-assignment*)
id-assignment ::= variable <-a- variable;

branch ::= + walk-name walk-segment end-of-segment
walk-segment ::= contraction walk-segment | call walk-segment | []
end-of-segment ::= call | fork | output
call ::= restriction* assignments function-call output; | block;
       | restriction* assignments inductive-function-call
         inductive-output;
```

```
function-call ::= <NAME>;
inductive-function-call ::= ≪NAME≫;
block ::= assignments { branching } output
        | assignments { branching } inductive-output
fork ::= node :{ branching };
node ::= (NODE node-cont)
branching ::= branch*
output ::= restriction* assignments
assignments ::= assignment*
inductive-output ::= {{ assignment* }}
assignment ::= Refal-expression ←a variable;
```

[6)]

```
contraction ::= variable →c elementary-pattern;
Refal-expression ::= t-expr Refal-expression | []
t-expr ::= (Refal-expression) | SYMBOL | variable
restriction ::= s-variable1 ≠ s-variable1 | s-variable ≠ SYMBOL
              | e-variable ≠ []
variable ::= s-variable | t-variable | e-variable
walk-name ::= INTEGER
```

Реально структурные скобки представлены в некоторой метакодировке. Мы опускаем её (см. главу 15). `id-assignment` — тождественная подстановка.

3.1.1. Синтаксис входного подмножества

```
input-graph ::= NAME { input-branch+ }
input-branch ::= + walk-name input-walk-segment end-of-input-segment
input-walk-segment ::= contraction*
end-of-input-segment ::= end-of-branch | input-fork
end-of-branch ::= node | leaf
leaf ::= node
end-of-input-segment ::= input-fork | output
input-fork ::= node :{ input-branch+ };
```

Не определённые здесь структуры даны выше по тексту.

[6)] В разделе 2.2 мы уже поясняли семантику знака $\xleftarrow{a}$ — это определение подстановки (присваивания) переменной `variable` РЕФАЛ-выражения. Мы пользуемся «языком стрелок» ($\xleftarrow{a}$, $\xrightarrow{c}$) вслед за В. Ф. Турчиным [66, 70, 72, 73]. Стрелка $\xrightarrow{c}$ означает оператор сужения значения переменной из левой части соответствующей `contraction` до вида образца, указанного в правой части. В случае интепретации, когда значением переменной может быть только конкретное данное (точка), этот оператор совпадает с оператором отождествления (сопоставления) РЕФАЛ-выражения с указанным образцом: **case** `variable` **of** `elementary-pattern...` В. Ф. Турчин объяснял автору такой выбор обозначений «двойственностью» (он использует термин «сопряжённость» — `conjuction`) операторов сужения и подстановки: результатом удачного сужения, в общем случае, являются подстановки (одна или несколько); в то же время подстановку можно рассматривать как сужение произвольного (неопределённого) значения переменной до подставляемого выражения. (См. также раздел 3.2.)

3.2. Семантика

Семантика языка РЕФАЛ-графов проецируется из РЕФАЛа:

- При интерпретации значениями переменных являются константные выражения. В этом случае `contraction` (`variable` $\xrightarrow{c}$ `pattern`) является предикатом «значение переменной имеет вид», переменная из левой части должна быть определена к моменту выполнения этой контракции. Множество переменных из правой части разбито на множество переменных, определённых и не определённых к данному моменту. Каждая истинная контракция определяет значения своих не определённых переменных из правой части. В случае преобразований (метаинтерпретация), значением переменной может быть любое параметризованное поле зрения (см. главу 4). В этом случае контракция является оператором сужения этого поля зрения, согласно правой части контракции (см. подробности в главе 4) [7]. Контракции исполняются последовательно.
- `restriction` интерпретируется аналогично параметрическому `restriction` (см. раздел 2.1). Последовательность рестрикций есть конъюнкция соответствующих предикатов.
- Ветви в ветвлении `branching` перенумерованы и упорядочены для отождествления сверху вниз.
- Ветвления функциональны (т. е. если отождествление внутри некоторого ветвления неудачно, то частичная функция, описанная данным графом на рассматриваемых конкретных входных данных, неопределена: ветвления безоткатны).
- Блоком называется непоименованный граф.
- Заключённое в фигурные скобки множество присваиваний выполняется одновременно (параллельно). Все переменные из левых частей присваиваний (`left` $\xleftarrow{a}$ `variable`) должны быть определены к моменту подстановки.
- В отличие от РЕФАЛа-5, функции в языке РЕФАЛ-графов многоместны. Это относится как к числу аргументов функции, так и к количеству её выходов.
- Начальная среда шага РЕФАЛ-граф машины определяется конкретным функциональным вызовом `call` и его входным форматом `assignments`.
- Результатом вычисления шага РЕФАЛ-граф машины является среда. Она определяется выходным форматом `output` и подстановкой на конце выбранной ветви.

Конкретное множество элементарных образцов `elementary-pattern` для нас неважно. Это множество должно обладать свойством полноты: каждый образец РЕФАЛа должен быть представим композицией «элементарных» образцов. Мы будем называть это множество базисом, хотя некоторые элементарные образцы могут быть представлены композицией нескольких других.

[7] Читателю, знакомому с языком PROLOG, указанный механизм хорошо известен.

Выходные форматы и вызовы функций (графов) раскрашены в два цвета (см. определение синтаксиса): выходной формат объявляется индуктивным сразу после его построения в процессе преобразований (посредством выдвижения гипотезы и её доказательства по индукции); вызовы такого графа объявляются индуктивными сразу после создания (использования) их реальных выходных параметров. Мы будем опускать эту раскраску, когда она несущественна. Имя каждого узла n_i уникально в поле зрения суперкомпилятора `scp-view-filed`. Имена ребер `walk-name` уникальны в каждом ветвлении.

Наконец, необходимо отметить, что данное промежуточное представление графов является объектом преобразований. Это есть причина того, что в наших рассмотрениях синтаксис РЕФАЛ-графов может незначительно меняться, как и семантика конкретных компонент факторизации, если это не приводит к изменению частичной функции (определённой преобразуемой программой) на её области определения. В конкретных рассмотрениях мы будем опускать структуры, не являющиеся существенными в данном контексте.

3.3. Язык РЕФАЛ-5 и язык РЕФАЛ-графов

Реализация РЕФАЛа-5 [75] не является U-пошаговой, но она становится таковой после её ограничения на базисный РЕФАЛ-5 — алгоритмически полный язык [75]. Перед трансляцией в язык РЕФАЛ-графов мы переводим РЕФАЛ программу в базисный РЕФАЛ: программа на базисном РЕФАЛе адекватно отображается во входном подмножестве РЕФАЛ-графов (с фиксированной U-пошаговой реализацией).

3.3.1. О неравномерности шагов РЕФАЛ-машины

С точки зрения преобразований представляет интерес вопрос о равномерном по входным данным времени исполнения реализацией шагов базисной РЕФАЛ-5 машины. Следующие два примера показывают, что не существует равномерной реализации отождествления абстрактной РЕФАЛ-5 машины.

Пример 1.

```
Equal {
 (e.x) (e.x) = True;
 (e.x) (e.y) = False;
}
```

Кратное вхождение `e.x` в левую часть первого предложения требует при отождествлении цикл по данным, размер которых может быть произвольным.

Пример 2.

```
Search {
 e.x Pointer e.y = True;
 e.x = False;
}
```

Во втором примере отождествление, в первом предложении, — это поиск первого вхождения терма `Pointer` в конечной последовательности неизвестной длины. Его временна́я сложность линейна по этой длине.

В основной части нашей книги мы ограничимся рассмотрением алгоритмически полного подмножества базисного РЕФАЛа-5 («ограниченного» РЕФАЛа), который не допускает в образцах:

a) повторных вхождений графически совпадающих (*«повторных»*) `t`- и `e`-переменных;

b) двух `e`-переменных на одном скобочном уровне (*«открытых»*) [69].

Назовём такие образцы строгими.

В РЕФАЛе-5 реализация строгих образцов равномерна по входным данным: время отождествления данных с конкретным образцом P (в определении F) ограничено константой c, не зависящей от размера входных данных. c зависит лишь от последовательности образцов из определения F, находящихся не ниже P. Все известные автору реализации диалектов РЕФАЛа обладают тем же свойством.

Аналогичный вопрос относительно реализации правых частей РЕФАЛ-предложений более тонкий.

Пример 3.

```
$ENTRY Copy {
 t.x = <Combine t.x t.x>;
 e.x = e.x e.x;
}

Combine {(e.x) (e.y) = e.x e.y;}
```

Первый шаг абстрактной РЕФАЛ-машины (функция `Copy`):

a) РЕФАЛ-5 — время построения правых частей обоих предложений линейно по размеру данных;

b) FLAC[8] [22,23] и РЕФАЛ-6 — правая часть первого предложения строится равномерно (за счёт механизма счётчика ссылок), а время построения правой части второго предложения — линейно.

Второй шаг абстрактной РЕФАЛ-машины (функция `Combine`):

a) РЕФАЛ-5 — правая часть строится за единицу времени;

b) FLAC и РЕФАЛ-6 — время построения правой части линейно зависит от размера данных.

В реализации РЕФАЛа+ [3], основанной на массивном представлении выражений, время копирования выражения равномерно ограничено по его размеру, но операция приписывания одного выражения к другому (конкатенация), в общем случае, таковой не является. Более того, проблемы

[8] Диалект РЕФАЛа, ориентированный на компьютерную алгебру.

неравномерности частично сдвинуты на уровень сборки мусора, и потому существенно глобальны. Равномерность или неравномерность конкретного шага машины зависит не только от размера оперативной памяти конкретного компьютера, но от её конфигурации в данный момент времени. Сдвиг проблемы неравномерности копирования в конкатенацию, как показывает наш пример, частично произошёл и во FLAC-е и в РЕФАЛе-6. В некотором смысле реализация РЕФАЛа-5 наиболее функциональна: равномерность или неравномерность по входным данным времени исполнения РЕФАЛ-шага зависит лишь от синтаксиса соответствующей функции, а не от контекста её вызова, т. е. это свойство синтаксическое и локальное.

Известно, что нижняя оценка временно́й сложности алгоритма копирования строки длины n на одноленточной машине Тьюринга равна $C * n^2$. В моделях вычислений, преобразующих РЕФАЛ-данные, — эта оценка линейна; обойти проблему нельзя, — можно лишь сдвинуть её.

Наш преобразователь программ рассматривает время исполнения одного шага абстрактной РЕФАЛ-машины (машины РЕФАЛ-графов) с точностью до времени копирования. И в этом смысле мы говорим о логическом времени исполнения программы, а не о физическом. Ниже мы ещё вернёмся к этому вопросу в главе 17.

3.3.2. Дерево отождествления в языке РЕФАЛ-графов

Повсюду в данной работе мы будем предполагать, что дерево развивается слева направо. То есть везде под деревом мы имеем в виду ориентированное слева направо корневое помеченное дерево с раскрашенными и упорядоченными сверху вниз ребрами.

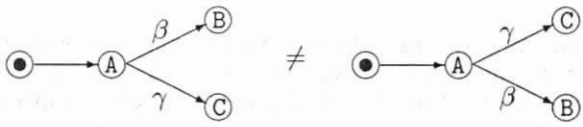


Рис. 3. Два разных дерева

Транслятор из РЕФАЛа-5 в язык графов строит дерево разбора последовательности образцов конкретной «функции» — подпрограммы. Ветвления в этом дереве безоткатные (см. 3.2). Дерево обеспечивает одновременное сопоставление с общими частями образцов нескольких подряд идущих предложений. Структура дерева зависит от конкретного базиса элементарных образцов. Каждая ветвь содержит «отрицательную» информацию о значениях переменных (которые определены последовательностью элементарных образцов), выразимую в языке рестрикций `restrictions` (см. определение синтаксиса 3.1) и зависящую от предикатов на вышестоящих ветвях.

Пример.

```
F {                          | F {
A e.x    = e.x;              | +[1] e.inp -c-> s.v e.u;
s.y e.x = s.y e.x;           |  :{ +[1] s.v -c-> A;{e.u <-a- e.out;};
         = ;                 |     +[2] s.v≠A; {s.v e.u <-a- e.out;};
}                            |   };
                             | +[2] e.inp -c-> []; {[] <-a- e.out;};
                             | }
```

Цифры в квадратных метаскобках являются номерами ветвей в конкретном ветвлении.

Приведём результат этой трансляции в более привычных терминах:

```
F {
case e.inp of
                s.v e.u   ⟶ case s.v of
                                    A ⟶ e.out := e.u;
                                    not A ⟶ e.out := s.v e.u;
                []        ⟶ e.out := [];
}
```

Здесь стрелка показывает ребро дерева.

Глава 4. ***Прогонка***

Рассмотрим школьный алгоритм вычисления корней квадратного уравнения, данный в неформальном языке. Будем считать числовые коэффициенты мономов данными, а оставшуюся часть синтаксической структуры уравнения — программой. Таким образом, наш алгоритм превратился в «интерпретатор» — решатель программы-уравнения, в котором нет циклов (мы предполагаем, что арифметические операции являются базисными примитивами).

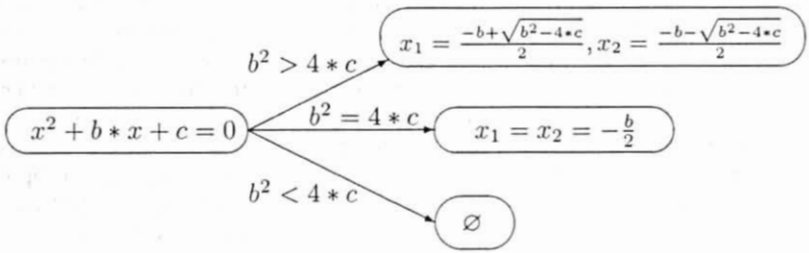


Рис. 4. Прогонка вычисления корней квадратного уравнения

Теперь рассмотрим квадратные уравнения с параметрами: некоторые из данных-коэффициентов стали параметрами. Школьник, решающий такие уравнения, владеет понятием алгоритма прогонки. Ответ, естественно, всегда выражается в терминах параметров и представляет собой дерево разбора этих параметров, в котором исполнены алгебраические действия, равномерные по значениям параметров. Напомним, что ранее (см. главу 1) мы назвали результат прогонки гроздью.

Выбрав язык параметров, мы можем уточнить понятие прогонки во входном подмножестве языка РЕФАЛ-графов. В данном параграфе мы рассматриваем только это подмножество.

4.1. Общая структура прогонки

Напомним, что

`Step: Prog×<Name args>×Data`$^n \mapsto$ `Stack×Data`^{n} .

Шаг абстрактной машины языка РЕФАЛ-графов состоит из двух логических этапов. На первом этапе выбирается ветвь (путь) от корня графа `Name` до его «листа» `end-of-branch`, на втором этапе — происходит модификация стека в соответствии с синтаксисом `end-of-branch` и вычисленной на первом этапе средой.

Рассмотрим интерпретацию подробнее.

На первом этапе происходит ряд сопоставлений значений переменных из текущей среды с элементарными образцами. (Задача сопоставления РЕФАЛ-выражения `Expr` с образцом `Pattern` является задачей решения уравнения `Pattern = Expr`. Результат решения есть значения переменных из образца `Pattern` либо информация о том, что уравнение корней не имеет.)

a) При удачных сопоставлениях среда изменяется и, если ребро, помеченное отождествлёнными образцами-предикатами, пройдено, то машина достигла либо точку ветвления, в которую ведёт данное ребро, либо `end-of-branch`. Во втором случае путь выбран и отождествление завершено. В первом случае текущая среда сохраняется в этой точке ветвления (в узле `node`) и машина переходит к последовательному сопоставлению полученных значений переменных с образцами, которыми помечено первое ребро, выходящее из данного ветвления.

b) При неудачах отождествления происходит возвращение на ближайшую пройденную точку ветвления, где восстанавливается среда (заранее сохранённая в этом узле `node`) и следующая ветвь в этом ветвлении рассматривается как возможный путь достижения некоторого `end-of-branch`.

c) Если ближайшее ветвление исчерпано (неудача произошла на последней ветви этого ветвления), то частичная функция, соответствующая графу, объявляется неопределённой в данной входной среде. Переход на ветвление, предшествующее к ближайшему, не происходит.

Замечание 1. Из безоткатности ветвлений РЕФАЛ-графа следует, что любой узел в нём, с точки зрения интерпретации, равноправен с корнем — точкой

входа в граф: на пару (имя-графа, имя-узла) можно смотреть как на самостоятельную «функцию» — подграф. Её местность определяется числом переменных в среде данного узла. Далее мы будем обозначать корень этого подграфа так: <Name$_{|nodeId|}$ args>, подразумевая под args среду.

Driving: Prog × <Name P_args> × P_Data^n ↦ Cluster

В отличие от интерпретатора, прогонка получает на вход не конкретную (одноточечную), а параметризованную среду P_args, значениями переменных в которой могут быть любые параметризованные выражения (см. главу 2). Попадая в очередную точку ветвления, Driving, как и Step, сохраняет в ней текущее состояние среды, выбирает первое исходящее из неё ребро и пытается последовательно решать уравнения с параметрами

Pattern = Parameterized_Expr

Pattern — элементарный образец из очередного сужения — contraction (см. 3.1), а Parameterized_Expr — параметризованное значение переменной из левой части этого сужения.

Утверждение 1. *Для любого строгого образца* (*см.* 3.3.1) Pattern *и любого параметризованного* (*в нашем языке параметров* pd) *выражения*

Parameterized_Expr

существует алгоритм «решения» уравнения с параметрами

Pattern = Parameterized_Expr,

который в результате своей работы выдаёт:

a) *либо ответ «в явном виде» в языке* pd;

b) *либо дерево разбиения параметров на подслучаи* (*подмножества*) *"if* $P(\ldots)$ ***then*** *...* ***else if*** $Q(\ldots)$ *...", где предикаты* $P(\ldots)$ *и* $Q(\ldots)$, *... представимы коньюнкцией композиций строгих образцов, — в каждом случае ответ выписан «в явном виде» в языке* pd. *Мы называем это дерево гроздью, чтобы отличать его от дерева отождествления в преобразуемом РЕФАЛ-графе*;

c) *либо информацию о том, что корней нет при любых значениях параметров*;

d) *либо параметризованный вызов функции, от значения которого зависит разрешимость уравнения в языке* pd.

Доказательство. Очевидно и может быть найдено в [66, 67].

Следствие 1. *Предикаты, которыми помечены ребра грозди, выразимы в языке элементарных строгих образцов.*

Нижеследующие примеры вполне проясняют данное утверждение.

Пример 1.
Уравнение: s.x e.y = ⟦s.3 A e.2 <F e.1>, s.3 ≠ B, e.1 ≠ []⟧
Ответ: Без дополнительных условий на параметры
⟦{s.x = s.3, e.y = A e.2 <F e.1>}, s.3 ≠ B, e.1 ≠ []⟧

Пример 2.
Уравнение: `s.x e.y = ⟦e.1 A e.2 <F e.1>, e.2 ≠ []⟧`
Ответ:
Если `e.1` $\xrightarrow{c}$ `[]`, то
`⟦{s.x = A, e.y = e.2 <F >}, e.2 ≠ []⟧`
иначе, если `e.1` $\xrightarrow{c}$ `s.11 e.12`, то
`⟦{ s.x = s.11, e.y = e.12 A e.2 <F s.11 e.12> }, e.2 ≠ []⟧`
иначе корней нет.

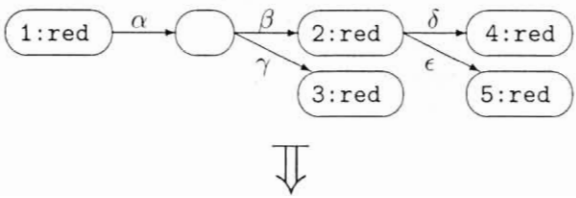
Пример 3.
Уравнение: `s.x s.x e.y = ⟦s.3 s.4 e.2 <F e.1> , s.4 ≠ A ⟧`
Ответ:
Если `s.4` $\xrightarrow{c}$ `s.3` , то
`⟦{ s.x = s.3, e.y = e.2 <F e.1> }, s.3 ≠ A ⟧`
иначе корней нет.

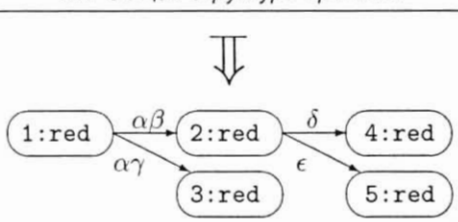
Пример 4.
Уравнение: `e.y s.x = ⟦s.3 A e.2 <F e.1>, e.2 ≠ [] , s.3 ≠ A⟧`
Ответ: Разрешимость зависит от возможных значений вызова `<F e.1>`.

Разбиение параметров на подмножества зависит от конкретного алгоритма решения уравнения с параметрами. Дальнейшие действия прогонки зависят от результатов исполнения этого алгоритма. Рассмотрим схему возможных результатов (см. Утверждение 1) и последующих действий.
Для каждого узла $\texttt{<Name}_{[nodeId]}\texttt{ P\_args>}$ РЕФАЛ-графа:

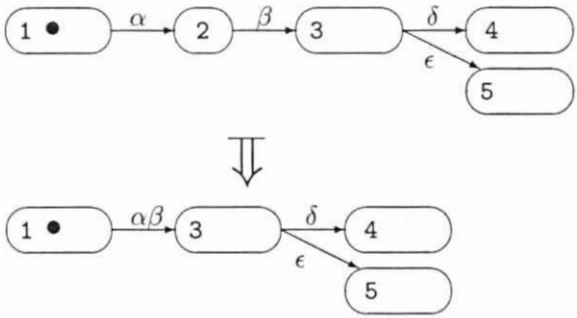
```
{
: Узел РЕФАЛ-графа объявляется стартовым узлом Node грозди
: ветвления параметров и помечается красным цветом.
: Для каждого ребра, исходящего из текущей узла РЕФАЛ-графа:
: {
: : Для каждого сужения на текущем ребре РЕФАЛ-графа:
: : {
: : : a) Если значения переменных (уравнения, соответствующего
: : :      сужению и текущей параметризованной среде) определились
: : :      синтаксически — без дополнительных условий на параметры,
: : :      тогда параметризованная среда корректируется. Переход к
: : :      следующему сужению.
: : : b) Иначе, если построены гроздь разбиения параметров и
: : :      значения переменных на листьях этой грозди:
: : : Для каждого листа построенной грозди:
: : :    {
: : :        — параметризованная среда в листе корректируется;
: : :        — последовательно решаются уравнения, соответствующие
: : :            этой построенной среде и сужениям contractions непрой-
: : :            денной части текущего ребра дерева РЕФАЛ-графа
: : :            (происходит развитие грозди);
```

: : : — лист заменяется этим последующим развитием гроздиразбиения параметров, — т. е. красным узлом, если развитие не пусто.
: : : }
: : : c) Иначе, если корней нет при любых значениях параметров:
: : : Соответствующая ветвь грозди удаляется — частичная функция, соответствующая графу, неопределена ни при каких значениях параметров из текущей среды.
: : : d) Иначе, если указан параметризованный вызов функции
: : : <$F_{[nodeId_1]}$ P_args_1>, от значений которого зависит разрешимость текущего уравнения: подгрозд, соответствующая текущей точке ветвления преобразуемого графа, удаляется.
: : : Работает алгоритм декомпозиции параметризованной среды из текущей точки ветвления преобразуемого графа, который корректирует параметризованное описание — вычисляет аргументы последующего вызова прогонки:
: : : `Driving(Prog,`
: : : `Decomposition( Prog, P_environment,`
: : : <$F_{[nodeId_1]}$ P_args_1>,
: : : <$Name_{[nodeId]}$ `P_args`>, P_Data^n)
: : :)
: : : }
: : Ребро РЕФАЛ-графа пройдено:
: : a) Если значения переменных определились равномерно — без дополнительных условий на параметры (дерево ветвлений тривиально), тогда из текущего узла `Node` строится ребро грозди, не содержащее сужений, которое входит в узел (лист) — параметризованную среду, являющуюся решением последовательности уравнений вдоль рассматриваемого ребра РЕФАЛ-графа. Все последующие ребра в текущем узле `Node` РЕФАЛ-графа не рассматриваются.
: : b) Иначе, если построена гроздь ветвления параметров, соответствующая этому ребру:
: : — гроздь преобразуется в семантически эквивалентную (посредством «расщепления» — см. рисунок ниже), в которой узлами являются только красные узлы преобразуемой грозди.
: : Пример расщепления:





: : — после чего её ветви приклеиваются к текущему стартовому
: : узлу Node ниже ветвей, соответствующих предыдущим
: : пройденным ребрам РЕФАЛ-графа.
: }
: Ребра, исходящие из текущего узла РЕФАЛ-графа, исчерпаны —
: нашим алгоритмом построена гроздь:
: — Если эта гроздь пуста (из стартового узла не выходит ни одной
: ветви), то стартовый узел удаляется вместе с входящим в него
: ребром (если такое существует).
: — Если из какого-то узла этой грозди выходит единственное
: ребро, то этот узел называется полутранзитным; все полутран-
: зитные узлы, кроме стартового (корня грозди), удаляются:
: входящее в узел и исходящее из него ребро соединяются, т. е.
: из двух ребер строится одно.
: Пример удаления:



: Стартовый полутранзитный узел может быть удалён или
: оставлен в графе, в зависимости от пожеланий пользователя.
: — Если стартовый узел из грозди не удалён, то алгоритм
: InputFormatStrategy помечает его или как опорный, или как
: вторичный.
}

Утверждение 2. *Для каждого узла в грозди, построенной прогонкой, существует ровно один прообраз — узел в преобразуемом графе. То же утверждение верно для листьев грозди. Все узлы в грозди безоткатны.*

Доказательство: По построению.

Замечание 2. Узел грозди может иметь большее, меньшее или равное количество исходящих ребер, чем его прообраз.

Замечание 3. На ребре в грозди может быть построено сужение, которого нет на ребрах преобразуемого графа.

Замечание 4. В грозди может существовать узел, все исходящие ребра которого имеют общее сужение.

Пример 5 (пример прогонки).

```
/*
$ENTRY Go { e.1 s.2 (e.3) = <Sqrt e.1 s.2 e.3>; }

Sqrt {
 0   = 0;
 s.Discriminant = (Sqrt s.Discriminant);
 '-' s.D = "The discriminant is negative: no solutions!";
}
*/
```

На вход подаётся пара:

```
{ e.41 s.102 e.101 <-a- e.inp; };
Sqrt {
:{ '+' e.inp -c-> s.v2 e.v1;
      :{ '+' e.v1 -c-> [];
                :{ '+' s.v2 -c-> 0; {0 <-a- e.out;};
                   '+' s.v2 ≠ 0; {(Sqrt s.v2) <-a- e.out;};
                 };
         '+' s.v2 -c-> '-'; e.v1 -c-> s.v3 e.v11; e.v11 -c-> [];
           {"The discriminant is negative: no solutions!" <-a- e.out;};
       };
 };
}
```

⇓

Гроздь прогонки:

```
{ e.41 <-a- e.41; s.102 <-a- s.102; e.101 <-a- e.101; };
:{ '+' e.41 -c-> s.103 e.42; s.103 -c-> '-'; e.42 -c-> []; e.101 -c-> [];
        {"The discriminant is negative: no solutions!" <-a- e.out;};
    '+' e.41 -c-> [];
      :{ '+' e.101 -c-> [];
                    :{ '+' s.102 -c-> 0; { 0 <-a- e.out; };
```

```
                    '+' s.102 ≠ 0; { (Sqrt s.102) <-a- e.out;};
                  };
      '+' s.102 -c-> '-'; e.101 -c-> s.105 e.104; e.104 -c-> [];
     {"The discriminant is negative: no solutions!" <-a- e.out;};
    };
}
```

4.2. Перестройка стека функций

Если, чтобы решить уравнение с параметрами в процессе прогонки стека с вершиной <Current$_{[nodeId]}$ P_args>, необходимо знать некоторые свойства результата параметризованного вызова (см. выше пример 4) <F$_{[nodeId_1]}$ P_args$_1$>, то стек перестраивается:

```
Decomposition(Prog, P_environment, <F[nodeId1] P_args1>,
                                   <Current[nodeId] P_args>, P_Data )
```

— вызов <F$_{[nodeId_1]}$ P_args$_1$> выталкивается на вершину стека в ближайшей точке ветвления (см. 4.1) и далее прогонка работает уже с определением F$_{[nodeId_1]}$. Точка остановки прогонки вызова <Current$_{[nodeId]}$ P_args> по РЕФАЛ-графу запоминается, чтобы далее, при необходимости, продолжить прогонку <Current$_{[nodeId]}$ P_args> с этой точки.

Следствие 1. *Гроздь прогонки может содержать поддеревья, соответствующие разным подграфам-«функциям» преобразуемой программы.*

Замечание 1. Прогонка может стартовать с любого узла преобразуемого РЕФАЛ-графа, а не только от корня.

Утверждение 1. *Функция прогонки определена для любого РЕФАЛ-графа и для любого параметризованного стека (нет бесконечных циклов).*

Доказательство. Утверждение следует из двух замечаний:

a) число перестроек стека функций ограничено сверху общим числом вызовов функций в данном стеке;

b) число узлов в преобразуемом РЕФАЛ-графе конечно.

4.3. Стратегия выбора входного формата

Узлы графа, объявленные прогонкой опорными, с точки зрения последующих инструментов преобразований, являются поименованными узлами: алгоритм факторизации дерева возможных вычислений сворачивает это потенциально бесконечное дерево в граф — перенаправляя ребра дерева в опорные вершины. Безоткатность узлов грозди позволяет любой узел объявить

корнем некоторой остаточной компоненты-подграграфа и далее ссылаться на него по имени, сохраняя семантику отождествления.

Ребра грозди прогонки сужают параметризованное множество стеков. Таким образом, структура стека в узле вхождения ребра уточняется: множество параметров, описывающих стек, определяет местность частичной функции, описываемой начинающимся в этом узле поддеревом возможных вычислений.

Алгоритм `InputFormatStrategy` принимает решение об опорности конкретного узла. Выбор множества опорных узлов отвечает как за качество преобразований, так и за принципиальную возможность факторизации дерева возможных вычислений в граф.

В суперкомпиляторе SCP4 используется следующая стратегия выбора опорных узлов:

- если стеки во всех листьях грозди прогонки исчерпаны (нет вызовов функций), тогда в этой грозди опорных узлов нет;
- иначе, если в грозди существуют точки ветвления параметров, то самая левая (самая ранняя) из них является опорной;
- иначе, если точек ветвления в грозди нет, то решение объявлять или нет корень грозди опорным узлом принимает пользователь перед вызовом суперкомпилятора;
- все остальные узлы вторичные (неопорные).

4.4. К вопросу о целях преобразований

Вернёмся к основной цели наших преобразований — повышению временно́й эффективности программы. Рассмотрим соотношение свойств грозди прогонки со свойствами преобразуемого РЕФАЛ-графа.

Гроздь представляет собой дерево разбора ветвлений параметров и перестройки среды — от узла к узлу, до листьев. По определению прогонки, каждый узел грозди есть один из образов[9)] некоторого узла РЕФАЛ-графа. Ребра грозди — последовательно склеенные образы ребер РЕФАЛ-графа.

При удалении из грозди полутранзитного узла (см. 4.1) — исчезают действия, связанные с сохранением среды и с возможной перестройкой стека в этом узле. Полутранзитный узел в грозди назовём транзитным, если единственное ребро, из него исходящее, не содержит сужений. Если удаляется транзитный узел, тогда

- сопоставления с образцами на прообразе ребра, исходящего из этого узла, выполнены в грозди равномерно по параметрам;
- синтаксически равномерно выполнена подстановка значений переменных в узле `Node` вхождения удалённого (стёртого) ребра: данное действие является семантически равномерным по входным (в удалённый узел) данным в том случае, когда параметризованный стек узла `Node` не содержит повторных вхождений графически совпадающих `e`- и `t`-параметров.

[9)] Отображения прогонки.

Параметризованное значение какой-то переменной из среды может содержать вызов функции, и возможна ситуация когда уравнения вдоль всех ребер, исходящих из некоторого узла РЕФАЛ-графа, разрешимы равномерно по возможным значениям этого вызова. В этом случае значения этого вызова не используются при выборе отождествлением исходящего ребра из этого узла — вызов может быть оставлен невычисленным совсем, либо его вычисление сдвинуто в последующие узлы.

С другой стороны, синтаксически равномерная подстановка может привести к повторению последующих вычислений, если переменная среды, содержащая в своём значении вызов функции, имеет кратные вхождения в выражение, куда её значения подставляются. Исключение синтаксически явных перевычислений производится после основной стадии преобразований (см. главу 11).

Пример 1 («ленивое» отождествление).

```
/*
$ENTRY Go { e.x = <Fa A <G e.x>>; }
Fa {
 A   e.x = A;
 s.1 e.x = <G e.x>;
}
G { ... }
*/
```

На вход подаётся пара:

```
{ A <G e.1> <-a- e.input; };
Fa { '+' e.input -c-> s.v2 e.v3;
                      :{ '+' s.v2 -c-> A; { A <-a- e.out; };
                         '+' s.v2 ≠ A; { <G e.v3> <-a- e.out; };
                       };
}
```

На выходе получаем гроздь прогонки: `:{ '+' {A <-a- e.out;};}`

К сожалению,

— число исходящих из узла грозди ребер может быть больше, чем у его прообраза в РЕФАЛ-графе;

— число элементарных сужений вдоль ребра грозди также может быть больше, чем у его прообраза.

Причиной являются некоторые невыполненные конкатенации значений e-параметров во входной среде. Эти обстоятельства, конечно, приводят к снижению временно́й эффективности отождествления в данном ветвлении. (При реализации отождествления на параллельной машине этой проблемы не существует: «лишние» ребра (или сужения) независимы и могут выполняться одновременно.) Число возможных невыполненных конкатенаций зависит

не только от синтаксиса преобразуемого РЕФАЛ-графа, но и от процесса преобразований — других инструментов суперкомпилятора SCP4; и может анализироваться только на более глобальном уровне.

Пример 2 (полутранзитный узел переходит в нетривиальное ветвление).

```
/*
$ENTRY Go { e.1 = <F e.1 A>; }
F { A  e.x = e.x; }
*/
```

На вход подаётся пара:

```
{ e.1 A <-a- e.input; };
F { '+' e.input -c-> A e.x; { e.x <-a- e.out; }; }
```

На выходе получаем гроздь прогонки:

```
:{
    '+' e.1 -c-> A e.2; { e.2 A <-a- e.out; };
    '+' e.1 -c-> []; { [] <-a- e.out; };
  };
```

Глава 5. ***Свёртка***

Свёртка (folding) не имеет аналога в абстрактной РЕФАЛ-граф машине: она факторизует метадерево возможных вычислений и вызывается непосредственно после прогонки. (При «полностью ленивых» вычислениях аналогом может служить просмотр: вычисляли ли мы уже в точности такой же вызов или нет.)

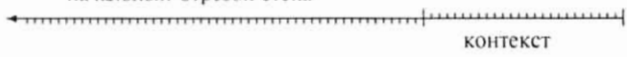
Точнее: свёртка пытается профакторизовать, непрофакторизованную к данному моменту, часть пути от корня метадерева возможных вычислений до опорного узла в грозди прогонки, если такой узел существует (назовём его текущим — CurNode).

Свёртка естественно разделяется на два логически замкнутых инструмента: вложение и обобщение. Оба эти инструмента используют не только синтаксические, но и семантические свойства стека функций. Идея семантической части алгоритмов была впервые рассказана В. Ф. Турчиным в Обнинске в 1989 году.

5.1. Вложение

Напомним, что мы работаем в языке параметров первого порядка. Будем считать, что функциональный стек растёт справа налево — самый левый вызов является активным. Пусть стек функций разделён на некоторый его начальный отрезок ненулевой длины и оставшуюся часть, тогда вычисления,

определяемые этим стеком, являются последовательным выполнением этого начального участка; и затем оставшейся части — в среде, вычисленной на первом этапе.



Расщепление стека

Вложение ищет вдоль указанного выше пути другой опорный узел (`PrevNode`), такой что его множество функциональных стеков, описанное параметрами, не только синтаксически содержит в себе множество функциональных стеков начального отрезка `StSeg` текущего опорного узла, но и существует подстановка параметров, сводящая описание предыдущего стека к описанию рассматриваемого отрезка текущего стека `StSeg`[10]. Таким образом, все возможные вычисления, определяемые этим начальным отрезком, посредством подстановки (быть может, определённой через вызов функций) можно свести к возможным вычислениям, определённым предыдущим опорным узлом. Оставшаяся часть текущего параметризованного стека является входной точкой отдельной задачи на суперкомпиляцию: результат вычисления рассматриваемого начального отрезка `StSeg` объявляется произвольным. То есть его выходная среда полностью неопределена — описывается новыми `e`-параметрами (см. примеры в конце данного раздела).

Пример. Рассмотрим два стека `[<F t.10 e.11>, ]` и `[<F e.1 t.2>, ]`. Множества, определяемые этими параметрическими описаниями, совпадают, но не существует подстановки параметров, сводящей одно описание к другому.

Если подходящего предыдущего опорного узла не существует, то метадерево возможных вычислений не меняется, иначе текущий стек расщеплён на два: часть задачи на развитие метадерева потенциальных вычислений, которая определена в текущем узле, сведена к одной из ранее рассмотренных задач. Сделан индукционный шаг по структуре описания функционального стека. Кроме того, как разбиение стека, так и сводящая подстановка определяют множество подзадач и связи между ними: произошла декомпозиция задачи. Эта декомпозиция оформляется в виде последовательности задач развития деревьев возможных вычислений, которые решаются далее одна за другой. С точки зрения суперкомпилятора, эти задачи являются входными точками отдельных преобразований, которые происходят в контексте построенного к данному моменту метадерева возможных вычислений (см. 5.2).

Если описанное выше сведение стеков произошло, то мы будем говорить что `CurNode` частично сводится к (или частично вкладывается в) `PrevNode`.

Утверждение 1. *Отношение частичного вложения на параметризованных стеках (или узлах) является предпорядком.*

[10] То есть множество стеков наделено структурой их описания и вложение «уважает» эту структуру.

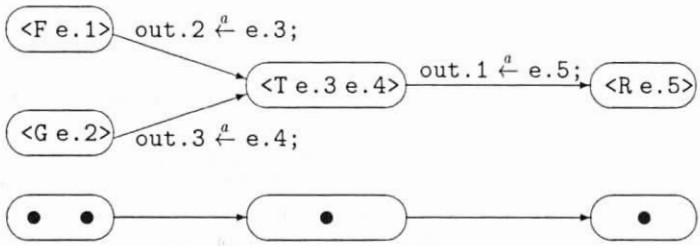
Доказательство. По построению.

Схема вложения:

```
Reduce(CurNode, PrevNode ) = (FOREST Tasks; ReducedPart; Context;)

Tasks ::= Task*
Context ::= Task | []
Task ::= input; { Node; }; new-output;
ReducedPart ::= assignments; { PrevNode; }; new-output;
input ::= id-assigns
new-output ::= out.name <-a- e-var;
```

Сведение к одному из предыдущих опорных узлов части текущей задачи сделано в предположении, что задачи `Tasks` будут решены, а `Context` — это узел, определяющий возможные вычисления после вычислений, определяемых `ReducedPart`. Структура `FOREST` является ориентированным ациклическим подграфом — «перевёрнутым» деревом, ребра которого ориентированы от листьев к корню `Context` (или `ReducedPart`, если `Context` отсутствует). Нам удобно будет смотреть на этот подграф как на ветвь — проекцию общего положения этого дерева на «линию вдоль его ребер». Ребра этой ветви помечены входными и выходными параметризованными средами. `ReducedPart` показана нами символически: этот узел — совпадает с уже существующим узлом левее в метадереве, а не является его второй копией.



Пример структуры FOREST и её схематическая «проекция»

Выходная среда каждой подзадачи на данном этапе одноместна и тривиальна, т. е. представляет собой произвольные РЕФАЛ-данные. Позже она может быть уточнена инструментами глобального анализа (см. главу 9). Местность входной среды подзадачи определяется числом различных параметров, описывающих её стек. Аргументы определяются тождественной подстановкой для всех подзадач, кроме сведённой части. Задача связана с другими частями метадерева только через входную и выходную среды: все изменения параметров в ней локализованы фигурными скобками, ограничивающими корень задачи.

Пример 1. Предыдущий стек:

`⟦<F <G e.1>> <-a- out.1; <G out.1> <-a- out.0; , ⟧`

Текущий стек:

$$[\![\underline{\texttt{<F <G <F e.2>>>}} \overset{a}{\leftarrow} \texttt{out.5;}\ \underline{\texttt{<G out.5>}} \overset{a}{\leftarrow} \texttt{out.4;}\ \underline{\texttt{<F out.4>}} \overset{a}{\leftarrow} \texttt{out.3; , }]\!]$$

Частичное сведение:
Частичная факторизация произведена в предположении, что будет решена задача (эта задача не решена в рамках предыдущего стека):

$$\{\ \texttt{e.2} \overset{a}{\leftarrow} \texttt{e.2;}\ \}\ \{\ [\![\underline{\texttt{<F e.2>}} \overset{a}{\leftarrow} \texttt{out.6;}\ ]\!]\}\ \{\ \texttt{out.6} \overset{a}{\leftarrow} \texttt{e.10;}\ \}$$

Профакторизованная часть (ReducedPart):

$$\{\texttt{e.10} \overset{a}{\leftarrow} \texttt{e.1;}\}$$
$$\{[\![\underline{\texttt{<F <G e.1>>}} \overset{a}{\leftarrow} \texttt{out.1;}\ \underline{\texttt{<G out.1>}} \overset{a}{\leftarrow} \texttt{out.0;}]\!]\}\ \{\texttt{out.0} \overset{a}{\leftarrow} \texttt{e.12;}\}$$

Узел, определяющий дальнейшие возможные вычисления:

$$\{\ \texttt{e.12} \overset{a}{\leftarrow} \texttt{e.12;}\ \}\ \{\ [\![\underline{\texttt{<F e.12>}} \overset{a}{\leftarrow} \texttt{out.4}\ ]\!]\}\ \{\ \texttt{out.4} \overset{a}{\leftarrow} \texttt{e.14;}\ \}$$

Пример 2. Предыдущий стек:

$$[\![\underline{\texttt{<F s.3 <G e.1>>}} \overset{a}{\leftarrow} \texttt{out.1;}\ \underline{\texttt{<G s.4 out.1>}} \overset{a}{\leftarrow} \texttt{out.0; , s.3} \neq \texttt{A; s.3} \neq \texttt{s.4; }]\!]$$

Текущий стек:

$$[\![\underline{\texttt{<F B <G e.1>>}} \overset{a}{\leftarrow} \texttt{out.1;}\ \underline{\texttt{<G A out.1>}} \overset{a}{\leftarrow} \texttt{out.4; , }]\!]$$

В данном случае сведение является полным (ReducedPart):

$$\{\ \texttt{B} \overset{a}{\leftarrow} \texttt{s.3;}\ \texttt{A} \overset{a}{\leftarrow} \texttt{s.4;}\ \}$$
$$\{\ [\![\underline{\texttt{<F s.3 <G e.1>>}} \overset{a}{\leftarrow} \texttt{out.1;}\ \underline{\texttt{<G s.4 out.1>}} \overset{a}{\leftarrow} \texttt{out.0; , s.3} \neq \texttt{A; s.3} \neq \texttt{s.4; }]\!]\}$$
$$\{\ \texttt{out.0} \overset{a}{\leftarrow} \texttt{e.5;}\ \}$$

Пример 3. Предыдущий стек:

$$[\![\underline{\texttt{<F s.3 e.1 s.2>}} \overset{a}{\leftarrow} \texttt{out.0; , s.3} \neq \texttt{s.2; }]\!]$$

Текущий стек:

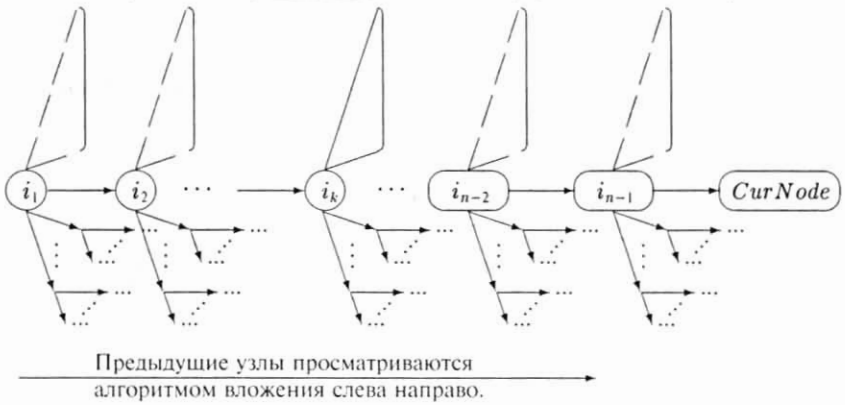
$$[\![\underline{\texttt{<F B e.4 B>}} \overset{a}{\leftarrow} \texttt{out.1; , e.4} \neq [\,]\texttt{; }]\!]$$

В этом случае частичного сведения не существует.

Опорный узел `PrevNode`, в который построено ребро из подграфа `FOREST`, назовём базисным. Только базисные узлы имеют входящие ребра, факторизующие метадерево возможных вычислений. Эти узлы (и корень всего метадерева) могут быть объявлены точками входа поименованных подграфов-«функций» остаточной программы (конечного результата преобразований).

5.2. Стратегия обхода дерева при факторизации

Как уже отмечено, факторизуется путь от корня метадерева до текущего узла. Опорные узлы просматриваются последовательно вдоль этого пути, согласно его ориентации — слева направо (другими словами: в порядке времени их создания). Перед попыткой вложения текущего узла в очередной



предыдущий опорный узел `PrevNode`, алгоритм вложения последовательно пробует свести текущий узел к базисным узлам полностью профакторизованных к данному моменту поддеревьев с корнями на концах ребер, исходящих из `PrevNode` (выше на рисунке узел i_k) и отличных от ребра, ведущего в текущий опорный узел (на рисунке соответствующая профакторизованная часть [11]) обозначена треугольником, касающимся узла i_k). Просмотр базисных узлов происходит также в порядке времени их создания.

Утверждение 1. *Алгоритм вложения частично сводит* `CurNode` *к одному из минимальных узлов (в смысле предпорядка частичного вложения) из множества всех опорных и базисных узлов, существующих в частично профакторизованном метадереве в момент работы алгоритма вложения.*

Доказательство. По построению алгоритма.

Текущий узел, сведенный к предыдущему опорному узлу с именем `Name`, заменяется в метадереве структурой `call`, соответствующей `function-call` (т. е. `<Name>`). Если же текущий узел сведён к базисному узлу, тогда он заменяется структурой `call` соответствующей `inductive-function-call` (т. е. `≪Name≫`) (см. определение синтаксиса в разделе 3.1). Выходные подстановки мы уточним ниже.

5.3. Обобщение

С точки зрения вложения, множество параметрических стеков опорных узлов в частично профакторизованном потенциально бесконечном метадереве является множеством индукционных гипотез о том, что на каждом пути в этом

[11] То есть та часть, из которой рассматриваются базисные узлы. Сам же граф показан нами схематично: в общем случае ребра графа могут идти из «правых треугольников» в более «левые».

дереве встретится опорный узел грозди прогонки, стек которого будет целиком сведён к стекам других базисных или опорных узлов (быть может, к нескольким из них), существующих в метадереве в момент попытки вложения.

Алгоритм обобщения служит инструментом обеспечения конечности числа этих гипотез, которые удастся «доказать», развивая дерево. Если вложение не обнаружило узла, к которому частично сводится текущий узел, тогда обобщение ищет на пути от корня метадерева до текущего узла опорный узел, «похожий» на текущий.

5.3.1. Отношение «похожести»

Отношение «похожести» связывает семантическое проявление цикла в метадереве с синтаксическими структурами в этом дереве. Удобно представить это отношение как пересечение отношения «похожести» чисто функциональной структуры стеков (последовательности имён функций) и «похожести» параметрического описания аргументов разных вызовов одной функции.

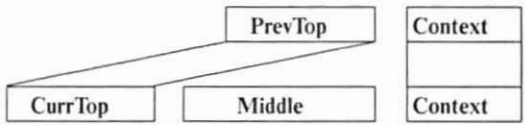
5.3.1.1. Обнинское условие выделения цикла

В этом разделе мы будем обращать внимание лишь на имена вызовов функций. Точнее, на имена вызовов, синтаксически являющихся самыми верхними вызовами в каждом элементе стека (вызовы функций могут быть пассивными аргументами элементов стека [см. 4.4, 6.2]). В этом разделе мы будем рассматривать только такие вызовы, если не оговорено противное.

В 1989 году в Обнинске В. Ф. Турчин предложил нижеследующее условие синтаксической аппроксимации цикла в частично профакторизованном метадереве потенциально возможных вычислений.

Припишем каждому вызову функции F его время t создания в процессе развития метадерева F_t. Пусть даны два стека (помеченных временами) — предыдущий `Prev` и текущий `Curr`, тогда будем считать, что эта пара указывает на цикл, если эти стеки можно представить в виде

```
Prev = PrevTop; Context;
Curr = CurrTop; Middle; Context;
```



Здесь дно стеков `Context` есть общая часть максимальной длины (возможно, нулевой), т. е. никак не участвовало в процессе развития стека на пути от предыдущего узла к текущему, а вершины стеков

$$(\texttt{PrevTop} = \texttt{F1}_{pt_1}, \ldots \texttt{Fn}_{pt_n} \text{ и } \texttt{CurrTop} = \texttt{F1}_{ct_1}, \ldots \texttt{Fn}_{ct_n})$$

совпадают, с точностью до времён создания вызовов. Вершину `PrevTop` назовём точкой входа в предполагаемый цикл. Тело цикла определяет путь от предыдущего опорного узла к текущему, который накапливает в стеке часть `Middle`. `Context` определяет вычисления после завершения этого цикла.

Обозначим данное отношение между стеками так: `Prev` $\propto$ `Curr`.

Данное определение отражает предполагаемое дальнейшее развитие стека в виде `PrevTop; Middle`^{n}`; Context`, если `Middle` и `PrevTop` рассматривать с точностью до времён создания вызовов. В случае если «похожесть» рассматриваемых узлов в метадереве подтвердится другими критериями (см. ниже), тогда ветви метадерева, исходящие из предыдущего узла `Prev`, выкидываются, а задача на преобразования, описанная в этом узле, разбивается на подзадачи, соответствующие циклу и контексту.

Пример 1. Рассмотрим определение функции факториала в унарной системе счисления.

```
$ENTRY IncFact { e.n = <Plus (I) (<Fact (e.n)>)>; }

Fact { ()  = I;
       (e.n) = <Times (e.n) (<Fact (<Minus (e.n) (I)>)>)>; }
Times { (e.u) () = ;
        (e.u) (I e.v) = <Plus (<Times (e.u) (e.v)>) (e.u)>; }
Plus { (e.u) () = e.u;
       (e.u) (I e.v) = I <Plus (e.u) (e.v)>; }

Minus { (I e.u) (I e.v) = <Minus (e.u) (e.v)>;
        (e.u)   ()      = e.u; }
```

«Ленивое» (см. 4.4) развитие стека по главной ветке даёт следующую последовательность:

```
[1]: IncFact1;
[2]: Plus2;
[3]: Fact4; Plus3;
[4]: Times5; Plus3;
[5]: Fact7; Times6; Plus3;
                ...
```

Здесь обнинское условие указывает, что пятый шаг формирует стек, «похожий» на стек, созданный на третьем шаге. Выдвигается гипотеза о том, что далее стек будет иметь вид `Fact; Times`^{n}`; Plus`$_3$`;`. Причём `Fact; Times`^{n} будет циклом [12], а `Plus`$_3$ — представляет вычисления после окончания этого цикла. Заметим, что аргументы вызовов `Plus`$_2$, `Times`$_5$, `Fact`$_7$ содержат другие вызовы.

Определение 1. Пусть дан конечный алфавит A `::=` $\{\mathtt{a}_j\}$. Пусть W — множество конечных слов над A, включая пустое слово. Пусть даны функция $\mathtt{G} : A \mapsto W$ и некоторое слово $s \in W$. Определим последовательность (быть может, конечную) с записью времён рождения $\mathtt{stack}_n$:

1. $\mathtt{stack}_0$ `::= (s,0)`.
2. пусть определен $\mathtt{stack}_i$ `::=` $(\mathtt{a}_i$ `, t)` $\mathtt{u}_i$, где $\mathtt{a}_i \in A$, $\mathtt{t} \in \mathbb{N}$, а $\mathtt{u}_i$ — оставшаяся часть последовательности $\mathtt{stack}_i$.

[12] `Fact` — точка входа, `Times` — тело цикла.

Тогда

$\mathtt{stack}_{i+1}$ ::= Time(G($\mathtt{a}_i$), MaxTime) $\mathtt{u}_i$,

где MaxTime есть максимум по вторым компонентам пар из $\mathtt{stack}_i$.

Time(a w, time) ::= (a,time+1) Time(w, time+1);
Time(, time) ::= ; , где $\mathtt{a} \in A$, $w \in W$.

Последовательность $\mathtt{stack}_n$ назовём временны́м развитием стека (G,s).

Утверждение 1 (Обнинское условие, В. Ф. Турчин [68]). *Пусть дано временно́е развитие стека* $\mathtt{stack}_n$. *Тогда оно либо конечно, либо существует пара номеров* k, m, *такая что* $k > m$ *и*

$\mathtt{stack}_m$ = ($\mathtt{a}_1$,$\mathtt{t}_{m_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{m_i}$) context
$\mathtt{stack}_k$ = ($\mathtt{a}_1$,$\mathtt{t}_{k_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{k_i}$) middle context

middle *и* context *конечные последовательности пар вида* ($\mathtt{a}_j$,$\mathtt{t}_j$), *возможно, нулевой длины.*

Часть стека ($\mathtt{a}_1$,$\mathtt{t}_{m_1}$)...($\mathtt{a}_i$,$\mathtt{t}_{m_i}$), отвечающую за вход в цикл, далее будем называть префиксом.

Напомним, что каждый опорный узел в метадереве имеет узел-прообраз в преобразуемом графе (см. 4.1, Утверждение 2). Каждый узел во входном графе имеет уникальное имя (см. определение входного синтаксиса 3.1.1). Суперкомпилятор SCP4 рассматривает в качестве алфавита функциональных имён, с которым работает обнинское условие, конечное множество имён $\mathtt{FuncName}_{[nodeId]}$, которые запоминаются в узлах в процессе прогонки и перестройки стека (см. 4.1 и 4.2).

$\mathtt{FuncName}_{[nodeId]}$ полностью определяет структуру среды — множество переменных, от значений которых зависит последующее вычисление этого вызова. Итак,

Утверждение 2. *Пусть*

($\mathtt{a}_1$,$\mathtt{t}_{m_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{m_i}$) *и* ($\mathtt{a}_1$,$\mathtt{t}_{k_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{k_i}$)

префиксы выделенные обнинским условием, тогда для любого j $(1 \leqslant j \leqslant i)$ *местность сред и соответствующие имена переменных сред* ($\mathtt{a}_j$,$\mathtt{t}_{m_j}$) *и* ($\mathtt{a}_j$,$\mathtt{t}_{k_j}$) *совпадают.*

5.3.1.2. Условие упрощающего отношения

Если обнинское условие выделило пару, определяющую предполагаемый цикл в метадереве, тогда верхняя функциональная структура стеков этой пары имеет вид:

$\mathtt{stack}_m$ = ($\mathtt{a}_1$,$\mathtt{t}_{m_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{m_i}$) context
$\mathtt{stack}_k$ = ($\mathtt{a}_1$,$\mathtt{t}_{k_1}$) ... ($\mathtt{a}_i$,$\mathtt{t}_{k_i}$) middle context .

Причём структура сред ($\mathtt{a}_j$,$\mathtt{t}_{m_j}$) и ($\mathtt{a}_j$,$\mathtt{t}_{k_j}$) совпадает для всех j, где $1 \leqslant j \leqslant i$.

Условие упрощающего отношения определяет отношение «похожести» той части параметрического описания сред пар ($\mathtt{a}_j$,$\mathtt{t}_{m_j}$), ($\mathtt{a}_j$,$\mathtt{t}_{k_j}$), которая даёт «положительную» информацию о данных (см. главу 2).

Определение 2. Упрощающее отношение $\trianglerighteq$ на множестве термов $\mathtt{pd}_+$ (см. главу 2) определим рекурсивно. Это наименьшее отношение, удовлетворяющее свойствам:

(1) $\mathtt{parameter}_1 \trianglerighteq \mathtt{parameter}_2$, $\mathtt{INTEGER}_1 \trianglerighteq \mathtt{INTEGER}_2$,
$\mathtt{SYMBOL} \trianglerighteq \mathtt{s.name}$, $\mathtt{SYMBOL} \trianglerighteq \mathtt{t.name}$,
$\mathtt{s.name} \trianglerighteq \mathtt{SYMBOL}$, $\mathtt{t.name} \trianglerighteq \mathtt{SYMBOL}$,
$(\mathtt{pd}_+) \trianglerighteq \mathtt{t.name}$, $(\mathtt{pd}_+\ ) \trianglerighteq \mathtt{e.name}$;

(2) $(\mathtt{pd}_+) \trianglerighteq \mathtt{pd}_+$,
$\mathtt{pd}'_+\ \mathtt{pd}_+ \trianglerighteq \mathtt{pd}'_+\ \mathtt{pd}_+$, $\mathtt{pd}_+\ \mathtt{pd}'_+ \trianglerighteq \mathtt{pd}_+\ \mathtt{pd}'_+$;

(3) если $\mathtt{pd}'_+ \trianglerighteq \mathtt{pd}''_+$,
то $(\mathtt{pd}'_+) \trianglerighteq (\mathtt{pd}''_+)$, $\mathtt{pd}_+\ \mathtt{pd}'_+ \trianglerighteq \mathtt{pd}_+\ \mathtt{pd}''_+$, $\mathtt{pd}'_+\ \mathtt{pd}_+ \trianglerighteq \mathtt{pd}''_+\ \mathtt{pd}_+$;

(4) если $\mathtt{pd}_+ \trianglerighteq \mathtt{pd}'_+\ \mathtt{pd}''_+$, то $\mathtt{pd}_+ \trianglerighteq \mathtt{pd}'_+\ \mathtt{e.name}\ \mathtt{pd}''_+$

Следствие 1.

(1) $\mathtt{pd}_+ \trianglerighteq \mathtt{pd}_+ \trianglerighteq \square$;

(2) *если* $\exists\, i_1, \ldots, i_j$, *такие что* $1 \leqslant i_1 < i_2 < \ldots < i_j \leqslant n$, *то* $t_1 \ldots t_n \trianglerighteq t_{i_1} \ldots t_{i_j}$, *где* t_k *— термы из* $\mathtt{pd}_+$.

Теорема 1 (Higman, Kruskal [36, 46]). *Если множество нечисловых базисных символов конечно, тогда в любой бесконечной последовательности термов* t_n *из* $\mathtt{pd}_+$ *найдётся пара* t_i *и* t_k, *такая что* $k > i$ *и* $t_k \trianglerighteq t_i$.

Мы используем в SCP4 некоторый вариант этого отношения, технические детали которого мы опускаем (см. исходные тексты суперкомпилятора [58]). Свойство (2) есть монотонность конструкторов построения РЕФАЛ-выражений относительно отношения $\trianglerighteq$, свойство (3) — его согласованность с этими конструкторами.

Данное выше определение является некоторым уточнением следующего отношения между термами. Пусть два терма `Prev` и `Cur` написаны мелом на доске. Скажем, что терм `Cur` сложнее или равен терму `Prev`, если `Prev` можно получить из `Cur`, стирая тряпочкой некоторые базисные термы и конструкторы. При этом наша аппроксимация понятия цикла заключается в предположении, что этот цикл строит стёртые нами термы и конструкторы.

Пример 2. Термы `(A B)` и `(B A)` несравнимы.

Пример 3. `(A B (C))` $\trianglerighteq$ `A C`.

Пример 4. `A B C` $\trianglerighteq$ `A e.1 C`.

5.3.1.3. Другие условия «похожести»

Существуют и другие дополнительные условия «похожести». Часть из них связана с чисто техническими деталями реализации (например, с поднятием параметров — см. ниже), другая часть взята *ad hoc* — на основании опыта авторов реализации.

5.3.2. Обобщение конфигураций

Если на пути от корня метадерева до текущего узла нашёлся опорный узел, «похожий» на текущий, тогда параметризованное описание стека в найденном (предыдущем) опорном узле разбито на две части (как результат обнинского условия): первая часть является точкой входа в предполагаемый цикл, вторая часть — контекст вычислений этого цикла.

`stack_prev` = (a_1, t_{m_1}) ... (a_i, t_{m_i}) `context`
`stack_curr` = (a_1, t_{k_1}) ... (a_i, t_{k_i}) `middle context`

Наше предположение заключается в том, что, начиная с текущего узла, процесс возможных вычислений будет «повторяться». Оба префикса являются точками входа в цикл: предыдущий — в первую итерацию цикла, текущий — во вторую. Чтобы профакторизовать метадерево, необходимо свести с помощью подстановки параметров параметрическое описание среды текущего префикса к среде предыдущего. Если такой подстановки не существует, тогда описания этих двух сред «обобщаются», т. е. строится параметризованная среда, которую можно свести к обеим средам префиксов с помощью подстановок параметров. Предыдущий префикс заменяется на обобщённый, и, таким образом, утверждение о возможности факторизации метадерева посредством алгоритма вложения становится более слабым: все параметризованные описания стеков, которые можно было подстановкой `Subst` свести к предыдущему стеку, можно свести и к обобщённому посредством композиции подстановки, построенной обобщением, и подстановки `Subst`.

По определению, верхняя[13] функциональная и временна́я структура предыдущего и текущего контекстов совпадают. Следовательно, среды контекстов будут совпадать при любом конкретном потенциальном вычислении. Но их параметризованные описания (описания значений переменных из сред) могут отличаться: описание параметризованной среды текущего контекста является некоторым уточнением параметризованной среды предыдущего контекста. Это уточнение определяется путём от предыдущего узла до текущего. Каждое уточнение может произойти лишь посредством подстановки значений параметров и формальных выходных переменных в $\mathtt{pd}_+$-часть описания среды и соответствующего изменения $\mathtt{pd}_-$-части, так как во всех других случаях изменения параметризованного контекста его временна́я структура, по определению, изменяется. Если в процессе уточнений (сужений) параметрически описанной среды контекста не было ни одной подстановки значений формальных выходных переменных `out.name`, тогда описание среды текущего контекста сводится к описанию среды предыдущего контекста композицией этих подстановок параметров. Иначе необходимо «обобщить» описания сред контекстов аналогично обобщению префиксов.

Ветви метадерева с корнем в предыдущем опорном узле отбрасываются. Задача разбивается на две: обобщённые префикс и контекст. Узлы, описывающие эти задачи и задачи, определяющиеся (возможно) функциональной сводящей подстановкой, а также связи между этими задачами, помещаются

[13] То есть верхний уровень синтаксического дерева функциональных вызовов.

на место предыдущего узла. Результат вычисления построенных задач объявляется произвольным, т. е. выходная среда в этот момент преобразований неопределена — описывается новыми e-параметрами. Как и в случае вложения, эти задачи являются входными точками отдельных преобразований.

Зафиксируем сказанное в виде утверждений.

Утверждение 1. *Частичная функция, определяемая обобщённым параметризованным стеком, является расширением частичной функции, которая определена стеком заменяемого предыдущего опорного узла.*

Утверждение 2. *Результат подстановки параметров, построенной обобщением предыдущего стека, в параметризованные вызовы функций в обобщённом стеке определяет частичную функцию заменяемого предыдущего опорного узла (функцию, которую определял этот предыдущий узел).*

Пример 1. Рассмотрим вариант примера 1 из 5.3.1. Пусть факториал считает количество своих шагов.

```
$ENTRY IncFactStep { e.n = <Plus (I) (<FactStep (e.n) ()>)>; }

FactStep {
  ()     (e.s) = I;
  (e.n) (e.s) =
          <Times (e.n) (<FactStep (<Minus (e.n) (I)>) (e.s I)>)>;
}
```

Здесь обнинское условие отработает аналогично примеру 1 из 5.3.1. Рассмотрим ситуацию подробнее.

```
    ...
[3]: [[<Fact4 (e.1) ()> <-a out.1; <Plus3 (I) (out.1)> <-a out.0; , ]]
    ...
[5]: [[<Fact7 (e.1) (I)> <-a out.2;
     <Times6 (I e.1) (out.2)> <-a out.1;
     <Plus3 (I) (out.1)> <-a out.0; ,
     ]]
```

Префикс стека [5] не вкладывается в стек [3]. Условие упрощающего отношения говорит, что стеки «похожи». Требуется обобщение сред префиксов. В результате обобщения получаем:

```
[Gener([prefix 3],[prefix 5])]: [[<Fact8 (e.1) (e.2)> , ]]
```

Подстановка { e.1 <-a e.1; []<-a e.2; } сводит обобщённую конфигурацию к [3]. Подстановка { e.1 <-a e.1; I <-a e.2; } сводит обобщённую конфигурацию к [5].

Развитие дерева правее [3] выбрасывается. Задача [3] разбивается на две:

```
[Gener-3]: { e.1 <-a e.1; []<-a e.2; }
           { [[<Fact8 (e.1) (e.2)> <-a out.3; , ]] } { out.3 <-a e.3; };
           /* закончилось описание первой задачи */
```

```
{ e.3 <-a- e.3; } { [<Plus9 (I) (e.3)> <-a- out.0; , ]}
{ out.0 <-a- e.4; };
/* закончилось описание второй задачи */
```

Замечание 1. Любопытный читатель, проследив ситуацию при реальном использовании SCP4, обнаружит, что обобщение срабатывает правее по рассматриваемой ветке. Причина этого явления в том, что опущенные нами другие условия «похожести» отметили два рассматриваемые нами префикса как «непохожие».

Пример 2. Рассмотрим простейший пример обобщения контекстов.
`$ENTRY Go { (e.numb1) (e.numb2) = <Plus (e.numb1) (e.numb2)>; }`
Развитие стека по рекурсивной ветке:

```
[1]: [<Go1 (e.1) (e.2)> <-a- out.0; , ]
[2]: [<Plus2 (e.1) (e.2)> <-a- out.1; out.1 <-a- out.0; , ]
[3]: [<Plus3 (e.1) (e.3)> <-a- out.1; I out.1 <-a- out.0; , ]
```

Контексты конфигураций [2] и [3] необходимо обобщить:

```
[Gener( [out.1 , ], [I out.1 , ] )]: [e.4 out.1 , ]
```

Подстановка { [] <-a- e.4; } сводит обобщённую конфигурацию к [2].

Развитие дерева правее [2] выбрасывается. Строится новая задача:

```
[Gener-2]: { [] <-a- e.4; }
           { [<Plus2 (e.1) (e.2)> <-a- out.1; e.4 out.1 <-a- out.0 , ]}
           { out.0 <-a- e.5; };
```

5.3.3. Обобщение параметризованных выражений

Укажем на некоторые принципы обобщения параметризованных выражений, используемые в SCP4 [14]. Наш суперкомпилятор стремится построить: 1) минимальное выражение относительно предпорядка, описанного в 2.1, которое не содержит двух или более e-параметров на одном и том же скобочном уровне, и 2) подстановки, сводящие построенное выражение к обобщаемым выражениям. Обобщаемые выражения в параметризованных средах обобщаются последовательно, с учётом возможной разделяемости параметров между значениями переменных среды. Пара значений одноимённых переменных этих сред обобщается на каждом скобочном уровне терм за термом; для обобщения выбираются самая левая и самая правая пары термов, и обобщается та из них, обобщение которой приводит к наименьшей потере информации об обобщаемых термах. Более детальное описание алгоритма может быть найдено в исходных текстах SCP4 [58].

[14] Специализацию можно понимать как частичную проверку типов: остаточные программы всегда применяются к аргументам частично правильного типа — его не надо проверять в момент исполнения. Аналогично, обобщение можно рассматривать как систему вывода параметрических («полиморфных») типов.

Утверждение 1. *Для любого параметризованного выражения* `Expr` *существует не более чем конечное число обобщающих его параметризованных выражений, которые не содержат двух или более* `e`*-параметров на одном и том же скобочном уровне, не совпадающих с* `Expr`. (*В данном случае равенство выражений рассматривается по модулю имён параметров.*)

Пример 1. `GenerExpr(A B A, A B X Y A B A) = A B e.1 A`
Сводящие подстановки: { `[]` $\overset{a}{\leftarrow}$ `e.1;` } и { `X Y A B` $\overset{a}{\leftarrow}$ `e.1;` }

Пример 2. `GenerExpr(2 <F e.1> , 3 <G e.2>) = s.3 e.4`
Сводящие подстановки: { `2` $\overset{a}{\leftarrow}$ `s.3; <F e.1>` $\overset{a}{\leftarrow}$ `e.4;` } и
{ `3` $\overset{a}{\leftarrow}$ `s.3; <G e.2>` $\overset{a}{\leftarrow}$ `e.4;` }

Пример 3. `GenerExpr(2 <F A> , 3 <F B A>) = s.1 <F e.2 A>`
Сводящие подстановки: { `2` $\overset{a}{\leftarrow}$ `s.1; []` $\overset{a}{\leftarrow}$ `e.2;` } и
{ `3` $\overset{a}{\leftarrow}$ `s.1; B` $\overset{a}{\leftarrow}$ `e.2;` }

5.3.4. Обобщение и построение «отрицательной» информации

После обобщения $\texttt{pd}_+$-частей параметрических описаний стеков необходимо обобщить их $\texttt{pd}_-$-описания. Напомним, что «отрицательная» часть описания является общей для всех вызовов функций из данного стека (см. 2.2). На входе алгоритм получает как сводящие подстановки, построенные при обобщении $\texttt{pd}_+$, так и сами описания.

```
GenerRestr(PrevAssignment, PrevPd_, CurrAssignment, CurrPd_)
                                                = GeneralizedPd_
```

`PrevPd_` ограничивает множества допустимых значений параметров из предыдущего стека, `CurrPd_` ограничивает О. Д. З. определений параметров из текущего стека.

Назовём параметризованное выражение $\texttt{pexpr}_1$ подвыражением параметризованного выражения $\texttt{pexpr}_2$, если последнее можно представить: 1) в виде $\texttt{pexpr}_3\ \texttt{pexpr}_1\ \texttt{pexpr}_4$, либо 2) в виде $\texttt{pexpr}_3\ (\texttt{pexpr}_5)\ \texttt{pexpr}_4$, где $\texttt{pexpr}_1$ является подвыражением выражения $\texttt{pexpr}_5$.

Каждое описание сводящей подстановки параметров определяет отображение `Subst` из множества параметров обобщающей конфигурации во множество параметризованных подвыражений обобщённой конфигурации. Обозначим `CurrSubst` и `PrevSubst` отображения, соответствующие текущей и предыдущей конфигурациям.

Рассмотрим схему алгоритма обобщения отрицательной информации, используемого в SCP4:

```
Для каждой из двух сводящих подстановок
Subst (CurrSubst и PrevSubst):
 {
 :  — для всех пар неравных символов SYMBOL1, SYMBOL2, принадлежащих
 :    образу отображения Subst:
```

: { построим множество формальных тавтологий
: $\texttt{SYMBOL}_1 \neq \texttt{SYMBOL}_2$
: }
: — для каждого ограничения на s-параметры из обобщённого описания
: pd_ (соответствующего рассматриваемой сводящей подстановке
: Subst) и каждой построенной тавтологии
: $\texttt{SYMBOL}_1 \neq \texttt{SYMBOL}_2$:
: { построим множество ограничений на параметры из обобщающей
: конфигурации (построенной на этапе обобщения «положительной»
: информации): $\texttt{ProImage}_1 \neq \texttt{ProImage}_2$, где
: $\texttt{ProImage}_1$ и $\texttt{ProImage}_2$ пробегают всё множество прообразов
: отображения Subst соответствующих частей (левой или правой)
: рассматриваемого ограничения (или тавтологии).
: }
: — для каждого ограничения на s-параметры из обобщённого описания
: pd_ (соответствующего рассматриваемой сводящей подстановке) вида
: s.name ≠ SYMBOL, SYMBOL ≠ s.name:
: { построим множество ограничений на параметры из обобщающей
: конфигурации: ProImage ≠ SYMBOL (соответственно,
: SYMBOL ≠ ProImage), где ProImage пробегает всё множество
: прообразов отображения Subst, соответвующих левой
: (соответственно, правой) части рассматриваемого ограничения.
: }
}

Для каждой из двух сводящих подстановок
Subst (CurrSubst и PrevSubst):
{
— для каждого обобщающего e-параметра e.gname (из области
определения отображения Subst), такого что множество,
описываемое образом Subst(e.gname), не может содержать
пустого выражения:
{ построим ограничение e.gname ≠ [] }
}

Рассмотрим пересечение двух построенных множеств элементарных ограничений на область определения отображения Subst. GeneralizedPd_ строится связыванием ограничений из этого пересечения связками конъюнкции.

Утверждение 1. *Области истинности предикатов*

PrevSubst(GeneralizedPd_) *и* PrevPd_

совпадают. Области истинности предикатов

CurrSubst(GeneralizedPd_) *и* CurrPd_

совпадают.

Доказательство. По построению.

Пример 1.

```
PrevAssignment: { C ←a s.40; D ←a s.41; e.1 (e.2) e.3 ←a e.43; }
PrevPd_:                    /* пустое описание */
CurrAssignment: { A ←a s.40; B ←a s.41; s.4 ←a e.43; }
CurrPd_:                    /* пустое описание */
GeneralizedPd_: s.40 ≠ s.41, e.43 ≠ []
```

Пример 2.

```
PrevAssignment: { C ←a s.40; D ←a s.41; s.2 ←a s.42;
                  D ←a s.43; s.3 ←a s.44; [] ←a e.45; }
PrevPd_: s.3 ≠ s.2, s.3 ≠ C
CurrAssignment: { A ←a s.40; s.4 ←a s.41; s.5 ←a s.42;
                 B ←a s.43; D ←a s.44; s.6 ←a e.45; }
CurrPd_: s.4 ≠ s.5, s.4 ≠ A, s.6 ≠ C
GeneralizedPd_: s.44≠s.40, s.40≠s.41, s.40≠s.43, s.40≠D
```

5.3.5. Стратегия обхода метадерева при обобщении

Обобщение просматривает опорные узлы вдоль «пути» от текущего узла до корня метадерева, т. е. в направлении, противоположном ориентации (противоположном просмотру этих узлов алгоритмом вложения) — справа налево (см. рисунок в разделе 5.2).

Сформулируем очевидные свойства инструментов обобщения.

Утверждение 1. *Пусть* $\propto$ *есть отношение обнинского выделения цикла* (*см.* 5.3.1.1). *Пусть даны три параметризованных стека из опорных узлов вдоль пути от корня метадерева до узла текущего, в порядке времени их создания* $t_1 < t_2 < t_3$, $\texttt{Stack}_{t_1}$, $\texttt{Stack}_{t_2}$, $\texttt{Stack}_{t_3}$ *и пусть* $\texttt{Stack}_{t_1} \propto \texttt{Stack}_{t_3}$ *и* $\texttt{Stack}_{t_2} \propto \texttt{Stack}_{t_3}$. *Тогда*

1) $\texttt{Stack}_{t_1} \propto \texttt{Stack}_{t_2}$;
2) *длины префиксов* $\texttt{Prefix}_1$, $\texttt{Prefix}_{3A}$, *выделяемых условием* $\texttt{Stack}_{t_1} \propto \texttt{Stack}_{t_3}$, *не больше длин префиксов* $\texttt{Prefix}_2$, $\texttt{Prefix}_{3B}$, *выделяемых условием* $\texttt{Stack}_{t_2} \propto \texttt{Stack}_{t_3}$;
3) *длина контекста* $\texttt{Context}_1$, *выделяемого условием* $\texttt{Stack}_{t_1} \propto \texttt{Stack}_{t_3}$, *не больше длины контекста* $\texttt{Context}_2$, *выделяемого условием* $\texttt{Stack}_{t_2} \propto \texttt{Stack}_{t_3}$.

Таким образом, указанная стратегия выбирает цикл наименьшей длины. Точка входа в цикл представлена функциональным стеком наибольшей длины, что оставляет возможность оптимизации этой композиции внутри цикла. С другой стороны, наибольшая длина контекста отражает семантику вычислений, которые не имеют отношения к циклу и будут профакторизованы отдельно.

5.3.6. Обнинское условие и транзитные вершины

Рассмотрим программу, реализующую функцию бинарного вычитания; цифры натурального числа в бинарной системе счисления расположены справа налево.

Пример 1. Поставим суперкомпилятору задачу:

```
$ENTRY The_task { t.numb = <BSub t.numb t.numb>; };
/* Ближайшее предыдущее число. */
BPred { (e.xs) = (<P e.xs>) };
P {
   '1'       = ;
   '1' e.xs = '0' e.xs;
   '0' e.xs = '1' <P e.xs>;
            = ;
}
/* Вычитание по единице в цикле. */
BSub {
   (e.xs) () = (e.xs);
   (e.xs) (e.ys) = <BPred <BSub (e.xs) <BPred (e.ys)>>>;
}
```

Стартовые шаги «ленивого» развития стека по первой рекурсивной ветке выглядят так:

```
[1]: <The_task t.1>;
[2]: <BSub (e.2) (e.2)>;
[3]: <BPred <BSub (e.2) <BPred (e.2)>>>;
[4]: <BSub (e.2) <BPred (e.2)>> <-a- out.0; <BPred out.0>;
[5]: <BPred (e.2)> <-a- out.1; <BSub (e.2) out.1> <-a- out.0; <BPred out.0>;
[6]: <BSub (e.2) (<P e.2>)> <-a- out.0; <BPred e.out>;
[7]: <P e.2> <-a- out.1; <BSub (e.2) (out.1)> <-a- out.0; <BPred out.0>;
                ...
```

Здесь точками ветвления являются только [2] и [7]. Можно показать, что, начиная с седьмого шага, стек в точках ветвления имеет структуру

$$\mathtt{P}^n;\ \mathtt{BSub};\ \mathtt{BPred}^{(2^n-1)};$$

где n — номер точки ветвления (считая от [7]). Знак степени после имён функций относится к повторению операции приписывания (конкатенации).

Сильная безусловность метадерева возможных вычислений приводит к невозможности его факторизации по обнинскому условию, если рассматривать последнее только на точках ветвления. Это утверждение является немедленным следствием структуры стека: она не позволяет разрезать стек на контекст и префикс. Кроме того, она показывает, что между $(n+1)$-й и n-й точками ветвления существует не менее чем 2^n транзитных шагов. Здесь мы считаем за «шаг» каждую перестройку стека.

Замечание. Отметим ещё одну проблему факторизации метадерева. Любая программа определяет синтаксическое дерево (соответствующее грамматике программы: каждое предложение в программе есть правило вывода) и семантическое дерево (дерево реальных вычислений). Текущее состояние частично профакторизованного метадерева является приближением к семантическому дереву. Развертка (прогонка) может продвигаться по синтаксической ветви этого дерева, которая недостижима ни при каких значениях данных и не содержит формальных точек ветвления. Следовательно, на ней нет опорных узлов (при некоторых стратегиях прогонки 4.3).

5.4. К вопросу о целях преобразований

Рассмотрим свойства результатов вложения и обобщения с точки зрения временно́й эффективности последующего выполнения результата преобразований.

Выбор опорных и базисных узлов для факторизации продиктован целью увеличения числа действий в программе, которые можно проинтерпретировать равномерно по параметрам (см. 4.3, 4.4). Каждое вложение есть оформление цикла в графе, который далее может быть объявлен отдельной «функцией» в остаточной программе (результате); узел вложения (куда вкладывается) и вложенные узлы (или часть их описаний при частичном вложении) представляют вызовы этой «функции». Их уменьшение есть уменьшение числа шагов остаточной программы на некоторых данных. Местность (входной формат) этой «функции» определяется числом графически различных параметров в среде узла вложения. Константные структуры (конструкторы данных и вызовов функций) среды узла вложения, по существу, представляют имя остаточной функции; они не входят в определение функции, все зависящие от них действия произошли на этапе преобразований, остаточная функция проспециализирована по рассматриваемым структурам. Аналогичную роль играет и информация-«конструктор» повторных вхождений параметра. Вынос за скобки максимального числа общих сужений параметров (см. 4.3) даёт ещё одну возможность распространения информации по стеку при разделяемости значений этих параметров разными элементами стека. Среды вложенных узлов могут описывать более узкие множества данных, чем узел вложения (их константная структура более точна) — все конструкторы, вошедшие в образ подстановки сведения, в общем случае, будут разбираться на этапе интерпретации. Может существовать возможность вложения одного и того же узла в несколько узлов. Стратегия обхода дерева при факторизации (см. 5.2) продиктована данными выше рассуждениями. Кроме того, когда опорный узел вложения выбран, сводящая подстановка, вообще говоря, определяется неоднозначно; местность остаточной функции зависит от алгоритма построения этой подстановки.

В случае частичного сведения параметризованного стека или рекурсивной сводящей подстановки (содержащей вызовы функций в значениях её среды), задача из текущего узла разбивается на подзадачи, одна из которых профакторизована. Описываются связи между этими подзадачами; выходная

среда оставшихся (непрофакторизованных) подзадач объявляется тривиальной (описывается e-параметром), и, таким образом, происходит полная потеря информации об областях значений соответствующих частичных функций, которая потенциально могла бы быть обнаружена прогонкой при сохранении композиционной структуры между задачами. Если профакторизованая задача свелась к базисному узлу, который является точкой входа полностью профакторизованного к данному моменту (и, более того, полностью просуперкомпилированного) подграфа, тогда выходная среда этой задачи несёт информацию, вскрытую инструментами глобального анализа, пока не рассмотренными нами (см. главу 9). Иначе, выходная среда профакторизованной части стека тривиальна.

Далее нас будут интересовать свойства механизма обобщения.

Обобщение строит новые более общие параметризованные стеки и соответствующие подстановки, сводящие описания более конкретных стеков к построенным: задача на преобразование заменяется на более общую — происходит частичная потеря информации о структуре стека и, следовательно, принципиальная потеря возможности некоторых специализаций. Появляются более общие опорные узлы, в которые может быть вложено большее число других узлов (см. текст выше). Естественно возникают задачи сокращения числа обобщений и, если решение об обобщении принято, построения более точного описания обобщённых задач. Первая проблема является предметом рассмотрения алгоритма, реализующего отношения «похожести» (см. 5.3.1). Например, условие упрощающего отношения запрещает обобщать пару стеков, если параметризованная среда текущего стека отличается от предыдущего лишь отсутствием некоторых конструкторов, предполагая, что эти конструкторы были «вычислены» (использованы в процессе вычислений). Стеки не будут обобщаться и тогда, когда их параметризованные среды описаны существенно разными конструкторами, либо композиционная структура конструкторов «существенно» разная. Не будут обобщаться и стеки с разными форматами сред (см. 4.3).

В процессе обобщения часть информации о параметризованных средах может быть потеряна тремя принципиально разными способами:

а) разбиением задачи на подзадачи и описанием связей между ними;

б) обобщением параметризованных выражений с построением нерекурсивных (без функциональных вызовов) сводящих подстановок, которые остаются в метадереве возможных вычислений;

в) обобщением параметризованных выражений с построением рекурсивных сводящих подстановок, которые остаются в метадереве возможных вычислений.

Случай б) является наиболее простым с точки зрения аппроксимации задачи минимальной потери информации. Примеры из разделов 5.3.3 и 5.3.4 показывают, каким образом сохраняется часть конструкторов и строятся ограничения на множество значений некоторых параметров.

В случае а) описания значений подзадач строятся тривиальным образом: никакая информация через связи между задачами не передаётся: они описываются e-параметрами (см. выше комментарии к частичному вложению).

Случай в) содержит в себе проблемы случаев а) и б).

Нелокальность (по отношению к текущей ветви метадерева) базисных конфигураций, которые алгоритм вложения рассматривает при попытке вложить в них текущий узел (см. раздел 5.2), с одной стороны, служит для построения более компактных остаточных программ (результатов преобразований) и может существенно сократить сам процесс суперкомпиляции; с другой стороны, приводит к построению ребер от узлов, лежащих на одной ветви метадерева (от корня до текущего узла) возможных вычислений, к узлам на другой ветви. Последнее, в свою очередь, является серьёзным препятствием при анализе глобальных свойств компонент факторизации, которые мы рассмотрим ниже (в главе 9).

5.4.1. Изменение местности параметризованной среды при её обобщении

Здесь мы рассматриваем обобщение, в результате которого стек не разбивается на подзадачи. В этом случае имеет смысл вопрос об изменении местности параметризованной среды в процессе обобщения — среды, которая будет заменена в метадереве на обобщённую, т. е. предыдущей среды. (Чтобы избежать путаницы, подчеркнём, что речь идёт не о входном формате вызовов функций в стеке — они измениться не могут, а о количестве разных параметров, описывающих этот стек.) Местность этой среды может как возрасти, так и уменьшиться.

По определению, любое изменение местности, при таком обобщении, влечёт потерю информации[15] о структуре параметризованных выражений, входящих в значение этой среды (см. примеры из 5.3.3).

Изменение местности может отвечать природе преобразуемого алгоритма.

Пример 1. На вход суперкомпилятору подаётся пара:

```
{ <Fa () e.1> <-a- e.input } и
Fa { + e.input -c-> (e.x) e.y;
                 :{+ e.y -c-> s.f s.y1; {<Fa (e.x B) e.y> <-a- e.out;};
                   + { e.x <-a- e.out; };
                  };
   }
```

В этом примере произойдёт обобщение:

`GenerExpr(<F () e.3>, <F (B) e.4> ) = <F (e.5) e.6>`

Сводящие подстановки: { `[]` <-a- `e.5; e.3` <-a- `e.6;` } и

{ `B` <-a- `e.5; e.4` <-a- `e.6;` }.

Без такого обобщения нельзя построить конечную остаточную программу.

В следующем примере аналогичное обобщение необходимо для свёртки метадерева возможных вычислений, но оно оставляет в результате лишний

[15] Повсюду под потерей информации мы имеем в виду (если не оговорено противное) потерю некоторой части этой информации.

аргумент (если не предпринимать дополнительных преобразований (см. главу 11)) и связанные с ним накладные расходы.

Пример 2. На вход суперкомпилятору подаётся пара:

```
{ <Fa1 () e.1> <-a- e.input } и
Fa1 { + e.input -c-> (e.x) e.y;
                :{ + e.y -c-> s.f s.y1; {<Fa1 (e.x B) e.y> <-a- e.out;};
                   + { e.y <-a- e.out; };
                 };
    }
```

Глава 6. ***Развёртка***

Абстрактная РЕФАЛ-граф машина, после окончания очередного шага, модифицирует стек функций — готовит его синтаксическую структуру для следующего шага. Модификация входной точки программы — тождественна. Входная точка однозначно определяет результаты последовательности отождествлений. Аналогом развёртки при интерпретации является выбор активного вызова функции согласно семантике языка.

Абстрактная развёртка (unfolding):

а) перестраивает структуру параметризованных стеков функций в листьях грозди прогонки, согласно некоторой стратегии `StackDevelopmentStrategy`;

б) выбирает в поле зрения суперкомпилятора лист, стек которого будет предметом преобразования следующего вызова прогонки.

Поле зрения суперкомпилятора представляет собой частично профакторизованное дерево возможных вычислений РЕФАЛ-граф машины: некоторые из его ветвей профакторизованы, другие нет.

Параметризованная входная точка `<Entry pd>` преобразуемой программы рассматривается как лист тривиальной грозди.

6.1. Стратегия развития дерева

Выбор очередного листа для последующей прогонки осуществляется TreeDevelopmentStrategy.

В нашем преобразователе TreeDevelopmentStrategy выбирает лист самой верхней непрофакторизованной ветви из тех ветвей, на которых есть хотя бы один вызов функции. Таким образом, метадерево факторизуется последовательно ветвь за ветвью: выше текущей ветви находится профакторизованная часть, ниже находятся ветви, развитие которых и факторизация будет происходить позже. (Напомним, что дерево развивается слева направо.) Такую стратегию принято называть стратегией развития в глубину.

6.2. Стратегии развития стека функций

Напомним, что стек растёт справа налево.

StackDevelopmentStrategy перестраивает описание параметризованного функционального стека. После прогонки каждый элемент стека представлен параметризованным выражением (грубо говоря).

Реализованы две стратегии развития стека:

а) `Lazy STRATEGY` соответствует ленивой (вызов по необходимости) семантике функциональных языков. Параметризованная среда элементов стека может содержать вызовы функций. Данное развитие стека во время суперкомпиляции не противоречит семантике РЕФАЛа, ибо может лишь расширить область определения частичных функций (см. Введение и главу 1).

б) `Applicative STRATEGY` соответствует аппликативной семантике (вызов по значению). Вершина стека полностью декомпозируется.

Конструкторы верхнего уровня элементов стека, т. е. те конструкторы, которые не являются частью аргументов вызовов функций, сдвигаются вправо по стеку. Самый правый элемент стека (его дно) является общим контекстом всех вычислений в конкретной задаче и накопителем конструкторов.

Пример.
Стек `[A <G B <Fe.1>>←out.1; <He.1 out.1>←out.2; ,]`
при ленивой стратегии будет преобразован к виду
`[<G B <Fe.1>>←out.3; <He.1 A out.3>←out.2; ,]`
при аппликативной стратегии к
`[<Fe.1>←out.3; <G B out.3>←out.4; <He.1 A out.4>←out.2; ,]`

С точки зрения развития нашего преобразователя SCP4, большой интерес представляет реализация следующих возможностей:

а) Вычисление всех полутранзитных шагов элементов стека (а не только его вершины). И, как уточнение, вычисление каждого элемента стека до первой точки ветвления параметров (см. [61]).

б) Вынесение «за скобку» кратных вхождений функциональных вызовов (минимизация их количества). То есть оформление стека в виде ациклического графа, а не только дерева. В актуальной версии суперкомпилятора такая минимизация делается после основной стадии преобразований (см. главу 11).

6.3. К вопросу о целях преобразований

Вернёмся к основному вопросу. От выбора очередной ветви стратегии развития метадерева зависит порядок обобщения стеков (как и сам факт обобщения конкретных узлов в дереве) и, следовательно, результат обобщения конкретного узла, т. е. качество остаточной программы: в следующем примере при перестановке РЕФАЛ-предложений частичная функция, определяемая программой, не меняется, однако при перестановке местами двух

первых предложений функции F структуры остаточных программ (результатов суперкомпиляций) существенно разные.

Пример 1.

```
$ENTRY Go {
 e.1 = <G <F (e.1) >>;
}

F {
 (B e.1) e.2 = <F (e.1) e.2 A>;
 (C e.1) e.2 = <F (e.1) (e.2)>;
 (D e.1) e.2 = e.2;
}

G {
 (e.1) e.2 = e.1;
  e.1  e.2 = s.1;
}
```

Анализ этого примера мы оставляем читателю, остаточные программы можно получить, преобразовав программу посредством SCP4.

Разделяемые разными функциональными вызовами параметры могут передать информацию о сужении множества своих значений для специализации функций-подграфов, отличных от той функции, где данное сужение произошло.

При выборе пользователем Lazy STRATEGY развития стека количество композиционных циклов в программе может сократиться.

Пример 2. Программа:

```
$ENTRY Go {
 e.input = <Replace (B C) <Replace (A B) e.input>>;
}
Replace {
(s.what s.with) s.what e.where
                =  s.with <Replace (s.what s.with) e.where>;
(s.what s.with) s.1 e.where
                =  s.1 <Replace (s.what s.with) e.where>;
(s.what s.with) =  ;
}
```

* преобразуется к :

```
* InputFormat: <Go e.41>
$ENTRY Go {
 e.41  = <F7 e.41>;
}
```

```
* InputFormat: <F7 e.41>
F7 {
 A e.41  = C <F7 e.41>;
 B e.41  = C <F7 e.41>;
 s.101 e.41 = s.101 <F7 e.41>;
        = ;
}
```

Глава 7. ***Подграф — компонента факторизации***

Предположим, что в процессе преобразований какой-то опорный узел нашего метадерева был объявлен базисным (см. 5.1) и исходящее из него поддерево G полностью профакторизовано. Ниже мы будем называть этот узел входной точкой в G. В этот момент начинается анализ глобальных свойств G. Нижеследующие параграфы, если не оговорено противное, рассматривают инструменты анализа и преобразований профакторизованной компоненты.

В процессе свёртки могут быть построены три типа компонент:

1) самодостаточные — все ребра, исходящие из вершин таких подграфов, заканчиваются только в вершинах данного подграфа;
2) компоненты, содержащие только вершины, исходящие ребра из которых заканчиваются либо в вершинах данного подграфа, либо в вершинах компонент факторизации, построенных ранее (далее мы будем говорить, что подграф ссылается на построенные подграфы);
3) компоненты, в которых существуют вершины с исходящими из них ребрами, которые заканчиваются на базисных узлах, являющихся корнями поддерева, факторизация которого ещё не завершена (далее в этом случае мы будем говорить, что подграф ссылается на непрофакторизованную часть метадерева).

Подчеркнём динамическую природу данной классификации: она определяется для конкретной процедуры свёртки и зависит от неё.

Глава 8. ***Чистка экранируемых ветвей***

Используемый суперкомпилятором SCP4 язык описания «отрицательной» информации (см. 2.1 и 3) очень узок. Он не позволяет описать предикаты выбора ветвей РЕФАЛ-графа независимо (одной ветви от другой), т. е. локально на конкретной ветви; в общем положении, множество истинности предиката на конкретной ветви лишь включает в себя множество достижимости данной ветви, а не равно ему. Здесь под множеством достижимости мы понимаем множество достижимости в контексте одного шага РЕФАЛ-машины. Следовательно, в процессе преобразований графа могут появиться семантически недостижимые ветви. Некоторые инструменты нашего суперкомпилятора работают локально вдоль каждой ветви (т. е. используют лишь информацию, обнаруженную на самой ветви). Например, прогонка. Таким образом, возникает необходимость в удалении таких недостижимых ветвей; их существование в графе не только несёт дополнительные накладные расходы при отождествлении, но и не позволяет обнаружить специфические свойства остаточного подграфа при его глобальном анализе, что влечёт за собой огрубление последующих преобразований метадерева возможных вычислений. (Последнее обстоятельство более существенно.)

Экранируемые ветви чистятся с использованием инструментов-«деталей» прогонки.

Глава 9. Глобальный анализ

После чистки экранируемых ветвей происходит анализ глобальных свойств построенной компоненты факторизации. Некоторые свойства удобнее анализировать в языке РЕФАЛ-графов, другие — на уровне языка РЕФАЛ. С точки зрения входной программы, которая преобразуется, эти свойства достаточно тривиальны и малоинтересны, но рассматриваемая подграф-компонента построена автоматически — это результат специализации и упрощения композиционной структуры программы описанными выше методами. Как показывают эксперименты, любые тривиальные преобразования на этом этапе крайне желательны и, часто, критичны, ибо от их результатов зависят последующие преобразования ещё не профакторизованной части метадерева. Принципиальным моментом является построение выходного формата компоненты факторизации; как мы уже отмечали выше (см. 5), разбиение задачи на подзадачи приводит к потере всей информации об области значения подзадач, — общий вид структуры области значения и представляют собой индуктивные выходные форматы. После глобального анализа эта информация передаётся по связям между задачами и далее используется.

Глобальный анализ и нелокальные преобразования в метадереве на основании обнаруженных свойств логически разделены. Соответствующие преобразования в метадереве будут рассмотрены в последующих разделах.

9.1. Анализ в терминах языка РЕФАЛ-графов

Пусть дан РЕФАЛ-граф `graph` и его входная точка `EntryName`.

9.1.1. Пустые подграфы

Определение 1. Вызов `call` РЕФАЛ-графа `graph` назовём вызовом входной точки `EntryName`, если он имеет вид (см. определение синтаксиса 3.1)

- `restriction* assignments <EntryName> output;`
- или `restriction* assignments ≪EntryName≫ output;`
- или `block`, где в блоке
  `block = assignments { branching } output` все ветви из `branching` ссылаются на данную входную точку `EntryName`.

Скажем, что ветвь
`branch = walk-name walk-segment end-of-segment` ссылается на входную точку `EntryName`, если

- `walk-segment` можно представить в виде
  $\texttt{walk-segment}_1$ `call` $\texttt{walk-segment}_2$, где вызов `call` является вызовом входной точки `EntryName`;
- или `end-of-segment = call`, где `call` — вызов входной точки `EntryName`;
- или `end-of-segment = node : { branching }`, где все ветви из `branching` ссылаются на входную точку `EntryName`.

Определение 2. РЕФАЛ-граф
`graph = input-format EntryName { branching } output-format`
называется синтаксически пустым графом, если каждая ветвь из `branching` ссылается на входную точку `EntryName`.

Утверждение 1. *Синтаксически пустой РЕФАЛ-граф определяет частичную функцию с пустой областью определения.*

Доказательство. По определению.

Если прогонка удаляет недостижимые ветви метадерева на основании свойств параметризованного стека в конкретной точке ветвления, то данный механизм является инструментом удаления недостижимых ветвей на основании глобальных свойств компоненты факторизации. И, в этом смысле, является метаинструментом.

Пример.

```
$ENTRY Go { e.input = <F e.input B>; }

F {
 (e.1) e.2 s.3 = (<F e.1 s.3>) <F e.2 s.3> ;
 s.1   e.2 s.3 = s.1 <F e.2 s.3>;
               A = ;
  }
```

/* После специализации по контексту вызова «функция» F будет пустой; что и обнаружит алгоритм, который содержит в себе определение синтаксически пустого графа. */

9.1.2. Выходные форматы

Напомним, что суперкомпилятор SCP4 принимает входное подмножество языка РЕФАЛ-графов, в котором выходная среда любого графа описывается тривиальным образом (см. раздел 3.1.1). К рассматриваемому нами моменту преобразований метадерева возможных вычислений выходная среда никак не преобразовывалась. Таким образом, достаточно рассмотреть только указанный случай, хотя его ограничения в данном контексте непринципиальны.

Определение 1. Будем называть конец сегмента `end-of-segment` концом ветви (см. 3.1), если он имеет один из двух следующих видов: `call` или `output` (см. 3.1).

Определение 2. Под синтаксическим индуктивным выходным форматом подграфа `G` мы будем понимать описание параметризованного множества данных $[\![\mathtt{pd}_+, \mathtt{pd}_-]\!]$ (см. 2.1), такое что для каждой ветви в подграфе `G`,

- если её конец имеет вид
  $\mathtt{output} = \mathtt{restriction}^* \{\mathtt{Refal\text{-}expression} \overset{a}{\leftarrow} \mathtt{variable}\}$,
  тогда существует подстановка параметров из $\mathtt{pd}_+$, сводящая индуктивный выходной формат к $[\![\mathtt{Refal\text{-}expression}, \mathtt{restriction}^*]\!]$;

- если её конец имеет вид <Name>, где Name есть имя одного из базисных узлов, тогда существует подстановка параметров из $\mathtt{pd}_+$, сводящая индуктивный выходной формат $[\![\mathtt{pd}_+, \mathtt{pd}_-]\!]$ к индуктивному выходному формату ранее построенного подграфа с входной точкой Name;
- если её конец имеет вид <Name>, где Name есть имя опорного небазисного узла и не равно EntryName, тогда индуктивный выходной формат $[\![\mathtt{pd}_+, \mathtt{pd}_-]\!]$ равен тривиальному — $[\![\mathtt{e.name},]\!]$;
- если её конец имеет вид `assignments branching inductive-output`, тогда существует подстановка параметров из $\mathtt{pd}_+$, сводящая индуктивный выходной формат $[\![\mathtt{pd}_+, \mathtt{pd}_-]\!]$ к `inductive-output`, который, в свою очередь, определяется индуктивно по концам ветвей `branching`.

Замечание. Кроме указанных случаев, к рассматриваемому моменту преобразований в компоненте связности G концы ветвей могут иметь вид <EntryName>. `Refal-expression` из первого случая синтаксически не содержит функциональных вызовов (ребер подграфа). Во втором случае Name всегда не равно EntryName.

Наше определение содержит простейший алгоритм построения индуктивного выходного формата, который используется в суперкомпиляторе SCP4: выходной формат строится только инструментами обобщения, описанными нами выше в 5.3.3 и 5.3.4, которые применяются последовательно к концам ветвей, — обобщая выходной формат ранее рассмотренных ветвей и описания конца текущей ветви (результат, конечно, может зависеть от последовательности рассмотрения ветвей). Среди всех индуктивных форматов наиболее интересны возможно более точные описания индуктивных выходных форматов.

Пример. Рассмотрим на входе следующую РЕФАЛ-программу:

```
$ENTRY Go { e.number = <UnarySum (I I I) (e.number)>; }

UnarySum {
 (e.numb1) (I e.numb2) = I <UnarySum (e.numb1) (e.numb2)>;
 (e.numb1) () = e.numb1;
}
```

/* Определение функции унарного сложения UnarySum, рассматриваемое отдельно от контекста вызова функции, не несёт никакой информации о структуре выхода этой функции. К моменту построения её выходного формата, после специализации по контексту вызова, в терминах языка РЕФАЛ-графов компонента факторизации будет иметь нижеследующий вид. */

```
(INPF e.41 <-a- e.41;)
F7 {
  +  e.41 -c-> I e.41;
     {e.41 <-a- e.41;} <F7 e.41> {e.out <-a- e.101;};
     I e.101 <-a- e.out;
```

```
  +  e.41 -c-> []; I I I <-a- e.out;
} (OUTF e.out <-a- e.out;)
```

/* Мы видим, что появилась возможность построения нетривиального выходного формата. Описанный выше алгоритм построит индуктивный выходной формат I e.102, хотя более точный анализ (семантический) мог бы дать идеальный вариант I I I e.102. Внимательный читатель заметит другие интересные свойства подграфа F7, к которым мы обратимся позже. */

Описание подграфа уточняется следующим образом:

```
(INPF e.41 <-a- e.41;)
F7 {
   + e.41 -c-> I e.41;
      {e.41 <-a- e.41;} ≪F7 e.41≫ {{I e.out1 <-a- e.101;}};
      I e.101 <-a- e.out;
   + e.41 -c-> []; I I I <-a- e.out;
}
(IND-OUTF I e.out1 <-a- e.out;)
```

Сокращение IND-OUTF (inductive output format) отражает суть метода построения этого формата.

9.1.3. Графы, определяющие константу

Важным предельным случаем выходного индуктивного формата является его полная определённость, т. е. когда область значений частичной функции состоит из одной точки. В этом случае мы можем заменить все обращения к данной функции её значением (возможно, расширяя её область определения).

9.1.4. Проекции

Предыдущий случай является частным случаем подграфа, значение которого может быть вычислено равномерно по значению переменных из его входной среды (входных формальных параметров). Мы выделили этот случай отдельным разделом с целью подчеркнуть тот факт, что необходим анализ зависимостей переменных внутри подграфа. Мы ограничимся здесь лишь простым примером; подробности могут быть найдены в исходных текстах суперкомпилятора SCP4 [58].

Пример. На вход суперкомпилятору SCP4 подаётся программа:

```
$ENTRY Go { e.input = <H e.input>; }

H {
  (e.1)       (e.3) (e.4) = e.1 e.4 e.1;
  (e.1) A e.2 (e.3) (e.4) = <H (e.1) e.2 (e.3 e.1) (e.4)>;
  (e.1) B e.2 (e.3) (e.4) = <H (e.1) e.2 (e.3 e.4) (e.4)>;
  (e.1) e.2   (e.3) (e.4) = e.1 e.4 e.1;
}
```

```
/* После выделения входного формата (см. 4.3)
<H (e.1) e.2 (e.3) (e.4)> будет обнаружено, что определение частичной
функции может быть преобразовано в одношаговое: */
* InputFormat: <Go e.41>
$ENTRY Go
       { (e.101) e.41 (e.103) (e.102) = e.101 e.102 e.101; }
```

9.2. Анализ в терминах языка РЕФАЛ

Вполне естественно, что алгоритмы анализа, их прозрачность и простота реализации зависят от языка представления объектов анализа. Часть глобального анализа свойств компонент факторизации происходит в терминах более высокого (по отношению к языку РЕФАЛ-графов) уровня — в терминах языка РЕФАЛ.

9.2.1. Тождественность

Здесь мы намерены синтаксически выделить очень простое подмножество РЕФАЛ-программ, которые определяют тождественные частичные функции. Обнаружение таких функций нашим преобразователем позволяет динамически типизировать данные во время преобразований без потери качества остаточной программы (см. 10.3).

Нас будет интересовать следующее подмножество базисного РЕФАЛа (назовём его Refal_1):

```
program ::= definition+
definition ::= function-name { body } | $ENTRY function-name { body }
body ::= sentence;+
sentence ::= left-side = right-side
left-side ::= pattern
right-side ::= expr
expr ::= empty | term expr | function-call expr
function-call ::= <function-name arg>
arg ::= expr
pattern ::= term*
term ::= SYMBOL | variable | (expr)
variable :: e.var-name | t.var-name | s.var-name
empty ::= []
function-name ::= IDENTIFIER
var-name ::= VARIABLE-NAME
```

Уточнение не определённых выше понятий для нас несущественно. В каждом предложении `sentence` множество переменных левой части `left-side` включает в себя множество переменных соответствующей правой части `right-side`; других ограничений на использование переменных не накладывается. Словарь имён функциональных вызовов `function-call` является подмножеством словаря имён определений `definition`.

Определение 1. Назовём Refal$_1$ программу синтаксически тождественной, если после формальной замены в ней каждого функционального вызова `<function-name arg>` его аргументом `arg` левая часть `left-side` каждого предложения `sentence` будет графически совпадать с правой частью этого предложения.

Пример 1 (синтаксически тождественной Refal$_1$ программы).

```
$ENTRY Go { e.1 = <F e.1>; }

F {
 s.1  e.expr = s.1 <F e.expr>;
(e.1) e.expr = (<F e.1>) <F e.expr>;
             = ;
}
```

Утверждение 1 (достаточное условие тождественности). *Каждое определение* `definition` *из синтаксически тождественной Refal$_1$ программы определяет частичную функцию F, тождественную на своей области определения*: $F(x) = x$.

Доказательство. Очевидно. В самом деле, рассмотрим произвольную точку x_0 из области определения функции F. Тогда вызов `<`$F\ x_0$`>` будет вычислен РЕФАЛ-машиной за конечное число шагов n. (Напомним, что шагом РЕФАЛ-машины называется следующая последовательность действий: отождествление, замена переменных в правой части их значениями — результатом отождествления, замена вызова `<`$F\ x_0$`>` в стеке функций (в поле зрения) полученной правой частью и выбор активного вызова на вершине изменившегося стека для следующего шага РЕФАЛ-машины.)

Будем вести математическую индукцию по числу шагов n. Если $n = 1$, тогда правая часть соответствующего предложения в определении F не содержит функциональных вызовов и равна левой части. База индукции доказана.

Пусть $n > 1$. Предположим, что утверждение доказано для всех определений из программы и всех вызовов функций, вычисляемых за число шагов $m < n$. Тогда правая часть `right-side` предложения, выбранного отождествлением, содержит хотя бы один вызов функции. Каждый вызов `<`$G\ arg$`>` из `right-side` вычисляется РЕФАЛ-машиной за число шагов, меньшее n, и, следовательно, по предположению индукции `<`$G\ arg$`>`$= arg$. Утверждение доказано.

Следствие 1. *Каждый вызов функции из синтаксически тождественной Refal$_1$ программы может быть заменён в процессе преобразований своим аргументом.*

Доказательство. По определению, суперкомпилятор имеет право расширять область определения частичной функции, определение которой он изменяет.

Пример 2.

```
$ENTRY Go { e.1 = <F e.1>; }
```

```
F {
 s.1 = s.1;
 s.1 e.string  = <F s.1> <F <F e.string>>;
     = ;
}
```

* В результате преобразований получим:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41  = e.41; }
```

Временна́я сложность исходной программы $O(2^n)$, преобразованной программы — $O(1)$ (n — размер входных данных).

9.2.2. Мономы конкатенации

Хотя в РЕФАЛе можно определить только одноместные функции, язык РЕФАЛ-графов шире по своим выразительным возможностям. К рассматриваемому моменту анализа и преобразований, как мы видели выше, входная среда компоненты факторизации может быть нетривиальной, т. е. местность определяемой ею функции может быть больше единицы. Существует естественный простой аналог понятия тождественности для многоместных функций. Напомним, что конструктор приписывания (конкатенации) в РЕФАЛе ассоциативен.

Определение 1. Пусть дана n-местная частичная функция

$$F(x_1, x_2, \ldots, x_n) : DATA^n \mapsto DATA_\perp,$$

где $DATA$ — множество данных РЕФАЛа. Скажем, что F частичный моном приписывания, если существует такая конечная последовательность натуральных чисел $k_1, k_2, \ldots, k_m$, что на области определения функции верно тождество

$$F(x_1, x_2, \ldots, x_n) = x_{j_1}^{k_1} x_{j_2}^{k_2} \ldots x_{j_m}^{k_m},$$

где $\forall\, s : (0 < s \leqslant m)\ 0 < j_s \leqslant n$. Здесь операция умножения — приписывание, возведение в степень относится к этой же операции.

Пример 1. $Copy(x) = xx = x^2$. Функция дублирования является мономом приписывания.

Пример 2. $Concat(x, y) = xy$. Функция присоединения является мономом приписывания.

Рассмотрим расширение языка Refal_1, введя возможность определения многоместных функций (назовём это расширение Refal_n).

```
program ::= definition+
definition ::= function-name { body } | $ENTRY function-name { body }
body ::= sentence;+
sentence ::= left-side = right-side
left-side ::= patterns
right-side ::= expr
```

```
expr ::= empty | term expr | function-call expr
function-call ::= <function-name args>
args ::= expr,* expr
patterns ::= pattern,* pattern
pattern ::= term*
term ::= SYMBOL | variable | (expr)
variable :: e.var-name | t.var-name | s.var-name
empty ::= []
function-name ::= IDENTIFIER
var-name ::= VARIABLE-NAME
```

Дополнительные ограничения на синтаксис: 1) пусть дано определение `definition`, тогда все его предложения содержат равное количество образцов; 2) число аргументов в каждом функциональном вызове `<F args>` равно числу образцов в определении F. Расширение семантики: значение n-го аргумента функционального вызова `<F args>` отождествляется с n-ми образцами предложений из определения F. Множества переменных из разных образцов данного предложения могут пересекаться.

Refal$_n$ есть лишь вариант подмножества языка РЕФАЛ-графов в терминах более высокого уровня. Компонента факторизации в рассматриваемый нами здесь момент преобразований всегда определена на этом подмножестве. Разделяемость переменных между образцами, в контексте наших рассмотрений, будет семантически значимой только для s-переменных, разделяемость которых допустима и в языке РЕФАЛ-графов. Разделяемость образцами t- и e-переменных мы будем использовать лишь исключительно в синтаксическом контексте (см. ниже).

Пример 3. Данное выше определение `UnarySum` (см. пример в 9.1.2) может быть адекватно переведено с языка Refal на Refal$_n$:

```
$ENTRY Go {
 e.number1, e.number2 = <UnarySumBN e.number1, e.number2>;
}

UnarySumBN {
 e.numb1, I e.numb2 = I <UnarySumBN e.numb1, e.numb2>;
 e.numb1,           = e.numb1;
}

/* Определение UnarySumBN описывает частичный моном приписывания
<UnarySumBN e.numb1,e.numb2> = e.numb2 e.numb1 . */
```

Пример 4. Данное ниже определение `Copy` описывает частичный моном приписывания `<Copy e.x> = e.x e.x`.

```
$ENTRY Copy { e.x = <Double e.x,  >; }

$ENTRY Double {
 s.1 e.x, e.y  = s.1 <Double e.x, e.y s.1>;
       , e.y  = e.y;
}
```

Определение 2. Пусть дана Refal_n программа P, такая что местности всех определений из P совпадают и равны n. Пусть $F_i(x_1, x_2, \ldots, x_n)$ входные форматы определений из P. И пусть дана конечная последовательность натуральных чисел $k_1, k_2, \ldots, k_n$, где $k_j > 0$ для всех j. Пусть $[y]^j$ обозначает $y, \ldots, y$, где y повторено j раз. Формальным повышением местности программы P по отношению к n-ке $k_1, k_2, \ldots, k_n$ назовём её следующее преобразование:
для всех i

- каждую левую часть `left-part` $= p_1, \ldots, p_n$ каждого предложения из P преобразуем к виду $[p_1]^{k_1}, \ldots, [p_n]^{k_n}$;
- каждый функциональный вызов $<F_i\ arg_1, \ldots, arg_n>$ программы P преобразуем к виду $<F_i\ [arg_1]^{k_1}, \ldots, [arg_n]^{k_n}>$.

Построенную программу обозначим $P^{k_1,k_2,\ldots,k_n}$ и назовём $(k_1, k_2, \ldots, k_n)$-местной по отношению к P. (Определения из $P^{k_1,k_2,\ldots,k_n}$ будем обозначать аналогично: $F_i^{k_1,k_2,\ldots,k_n}$.)

Пример 5. Рассмотрим программу из примера 4 без определения `Copy`. Формальное повышение местности по отношению к паре `2,1` даёт программу:

```
* Input format: <Double^2,1 e.x1, e.x2, e.y>
$ENTRY Double^2,1 {
 s.1 e.x, s.1 e.x, e.y  = s.1 <Double^2,1 e.x, e.x, e.y s.1>;
        ,         , e.y  = e.y;
}
```

Утверждение 1. *Пусть дана Refal_n программа P и пусть F_j определения из P. Пусть определена $P^{k_1,k_2,\ldots,k_n}$ для некоторой n-ки $k_1, k_2, \ldots, k_n$. Для всех j рассмотрим новые Refal_n определения*

$$G_j\ \{\ \texttt{e.1,...,e.n} = <F_j^{k_1,k_2,\ldots,k_n}\ \texttt{[e.1]}^{k_1}, \ldots, \texttt{[e.n]}^{k_n}>;\ \},$$

где $F_j \neq G_i$ для всех j и i. Пусть функция, описанная определением F_j, определена на n-ке $d_1, \ldots, d_n$, где каждое d_s некоторое данное языка РЕФАЛ, тогда верно семантическое равенство

$$<G_j\ d_1, \ldots, d_n> = <F_j\ d_1, \ldots, d_n>.$$

Причём число шагов Refal_n машины, требуемых для вычисления $<G_j\ d_1, \ldots, d_n>$, на единицу больше числа шагов Refal_n машины, требуемых для вычисления $<F_j\ d_1, \ldots, d_n>$.

Доказательство. По построению программы $P^{k_1,k_2,\ldots,k_n}$. Будем вести индукцию по числу шагов Refal_n машины, вычисляющей вызов $<F_j\ d_1, \ldots, d_n>$.

База индукции: пусть $<F_j\ d_1, \ldots, d_n>$ вычисляется за один шаг. Никаких новых условий на выбор предложений в программе $P^{k_1,k_2,\ldots,k_n}$ мы не добавили, так как в позициях разделяемых переменных в построенных повторных образцах программы $P^{k_1,k_2,\ldots,k_n}$ всегда стоят равные данные.

Следовательно, при вычислении вызова

$$<F_j^{k_1,k_2,\ldots,k_n}\ [d_1]^{k_1},\ldots,[d_n]^{k_n}>$$

при отождествлении будет выбрано предложение `sentence`, являющееся образом того предложения из программы P (при формальном повышении местности), которое будет выбрано при вычислении вызова $<F_j\ d_1,\ldots,d_n>$. Более того, среды, определяемые двумя этими отождествлениями, совпадают.

С другой стороны, правая часть предложения sentence совпадает с правой частью своего прообраза, так как вызов $<F_j\ d_1,\ldots,d_n>$ вычисляется за один шаг и, следовательно, правая часть прообраза не содержит вызовов функций.

Таким образом, результаты вычисления обоих вызовов совпадают, но для вычисления $<G_j\ d_1,\ldots,d_n>$ требуется один дополнительный шаг. Доказательство базы индукции закончено. Заметим, что мы доказали большее: пусть значение произвольного вызова $<F_j\ d_1,\ldots,d_n>$ из программы P вычисляется за один шаг, тогда за один шаг вычисляется вызов

$$<F_j^{k_1,k_2,\ldots,k_n}\ [d_1]^{k_1},\ldots,[d_n]^{k_n}>$$

из программы $P^{k_1,k_2,\ldots,k_n}$ и их значения совпадают.

Шаг индукции: предположим для всех i $(0 < i < m)$ доказано, что если значение каждого вызова $<F_j\ d_1,\ldots,d_n>$ вычисляется за i шагов, тогда определено значение вызова

$$<F_j^{k_1,k_2,\ldots,k_n}\ [d_1]^{k_1},\ldots,[d_n]^{k_n}>$$

и оно также вычисляется за i шагов. Пусть значение вызова $<F_t\ d_1,\ldots,d_n>$ определено и вычисляется за $m > 1$ шагов, тогда буквальное повторение рассуждения из базового случая показывает, что отождествление во время первого шага вычисления вызова

$$<F_t^{k_1,k_2,\ldots,k_n}\ [d_1]^{k_1},\ldots,[d_n]^{k_n}>$$

выберет `sentence` — образ предложения из P, которое выбирается первым шагом вычисления вызова $<F_t\ d_1,\ldots,d_n>$, и две среды после этих отождествлений будут совпадать. Каждый вызов функции из правой части прообраза предложения sentence вычисляется за количество шагов, меньшее m. По предположению индукции, его значение совпадает со значением образа этого вызова и требует для своего вычисления столько же шагов. Утверждение доказано.

Пример 6.

```
G { e.x = <Double^2,1 e.x, e.x, >; }
```

* G и Copy из примера 4 определяют одно и то же частичное отображение.

Определение 3. Пусть дана Refal_n программа P, такая что местности всех определений из P совпадают и равны n. Пусть $F_i(x_1, x_2, \ldots, x_n)$ входные форматы определений из P. И пусть дана перестановка σ элементов последовательности $1,\ldots,n$. Формальной перестановкой параметров программы P посредством σ назовём её следующее преобразование:

для всех i

- каждую левую часть `left-part` $= p_1, \ldots, p_n$ каждого предложения из P преобразуем к виду $p_{\sigma(1)}, \ldots, p_{\sigma(n)}$;
- каждый функциональный вызов <F_i $arg_1, \ldots, arg_n$> программы P преобразуем к виду <F_i $arg_{\sigma(1)}, \ldots, arg_{\sigma(n)}$>.

Построенную программу обозначим ${}^{\sigma}P$ и назовём σ-переставленной по отношению к P. (Определения из ${}^{\sigma}P$ будем обозначать аналогично: ${}^{\sigma}F_i$.)

Пример 7.

* Пусть дана перестановка σ последовательности 1, 2, 3:
* $\sigma(1) = 1$, $\sigma(2) = 3$, $\sigma(3) = 2$.
* Пусть P программа из примера 5. Ниже дана программа ${}^{\sigma}P$.
* Input format: <${}^{\sigma}$Double^{2,1} e.x1, e.y, e.x2>

```
$ENTRY σDouble2,1 {
 s.1 e.x, e.y, s.1 e.x  = s.1 <σDouble2,1 e.x, e.y s.1, e.x>;
        , e.y,          = e.y;
}
```

Из определения 3 вытекает очевидное

Следствие 1. *В обозначениях определения 3 для всех i и всех РЕФАЛ-данных $d_1, \ldots, d_n$ имеет место семантическое равенство*

$$<{}^{\sigma}F_i\ d_{\sigma(1)}, \ldots, d_{\sigma(n)}> = <F_i\ d_1, \ldots, d_n>.$$

Определение 4. Назовём Refal$_n$ программу P синтаксическим мономом, если существует конечная последовательность натуральных чисел $k_1, \ldots, k_n$ (где $k_j > 0$ для всех j), для которой определена программа $P^{k_1,k_2,\ldots,k_n}$, и существует такая перестановка σ последовательности $1, \ldots, m$ $(m = k_1 + \ldots + k_n)$, что после формальной замены в программе ${}^{\sigma}P^{k_1,k_2,\ldots,k_n}$

- каждого функционального вызова

  <`function-name` $arg_{\sigma(1)}, \ldots, arg_{\sigma(m)}$>

  конкатенацией его аргументов $arg_{\sigma(1)}\ \ldots\ arg_{\sigma(m)}$,
- каждой левой части

  $$pattern_{\sigma(1)}, \ldots, pattern_{\sigma(m)}$$

  каждого предложения конкатенацией его образцов

  $$pattern_{\sigma(1)}\ \ldots\ pattern_{\sigma(m)},$$

  левая часть `left-side` каждого предложения `sentence` программы ${}^{\sigma}P^{k_1,k_2,\ldots,k_n}$ будет графически совпадать с правой частью этого предложения.

Пример 8. Программа из примера 5 даёт пример синтаксического монома.

Утверждение 2 (достаточное условие мономиальности). *В обозначениях определения* 4 *пусть* $F_j(x_1, \ldots, x_n)$ *есть входной формат определения* `definition` F_j. *Пусть* y_i *обозначает* i*-й член последовательности* $[x_1]^{k_1}, \ldots, [x_n]^{k_n}$, *тогда для всех* j — F_j *из синтаксического монома* P *определяет частичную функцию* F_j, *являющуюся частичным мономом* $y_{\sigma(1)} \ldots y_{\sigma(m)}$:

$$F_j(x_1, \ldots, x_n) = y_{\sigma(1)} \ldots y_{\sigma(m)}.$$

Доказательство. Буквальным повторением рассуждений из доказательства достаточного условия тождественности (см. 9.2.1) получаем, что

$$<{}^{\sigma}F_j^{k_1,\ldots,k_n}\ y_{\sigma(1)}, \ldots, y_{\sigma(m)}> = y_{\sigma(1)} \ldots y_{\sigma(m)}.$$

Далее следствие 1 (из текущего раздела) даёт семантическое равенство

$$<{}^{\sigma}F_j^{k_1,\ldots,k_n}\ y_{\sigma(1)}, \ldots, y_{\sigma(m)}> = <F_j^{k_1,\ldots,k_n}\ y_1, \ldots, y_m>.$$

По утверждению 1, верно семантическое равенство

$$F_j(x_1, \ldots, x_n) = <F_j^{k_1,\ldots,k_n}\ y_1, \ldots, y_m>.$$

Следовательно, $F_j(x_1, \ldots, x_n) = y_{\sigma(1)} \ldots y_{\sigma(m)}$. Что и требовалось доказать.

Пример 9. Данное ниже определение есть синтаксический моном, причём `<Double e.x, e.y> = e.x e.y e.x`.

```
$ENTRY Double {
 s.1 e.x, e.y  = s.1 <Double e.x, e.y s.1>;
       , e.y  = e.y;
}
```

9.2.3. Стратегия выбора гипотезы мономиальности

Определения и утверждения предыдущего раздела дают простой алгоритм проверки, является ли Refal_n программа P некоторым данным синтаксическим мономом. При выполнении достаточного условия, программа P может быть преобразована в одношаговую. Возникает необходимость в алгоритме построения монома-гипотезы или отказа от проверки на мономиальность. Мы предлагаем строить такую гипотезу на основании формального синтаксического выхода из рекурсии, определяемой анализируемой программой P. То есть по предложениям программы, правые части которых не содержат вызовов функций.

Пусть программа P содержит ровно одно определение F и пусть $<F\ x_1, \ldots, x_n>$ есть его входной формат. Рассмотрим все пассивные предложения [16] из F: $p_{1_i}, \ldots, p_{n_i}$= `right-side`$_i$. Пусть d_{j_i} обозначают некоторые фиксированные данные РЕФАЛа. Предположим, что для каждого i существует хотя бы одна точка $d_{1_i}, \ldots, d_{n_i}$, такая что определение F описывает частичную функцию, при вызове которой $<F\ d_{1_i}, \ldots, d_{n_i}>$ на первом же шаге машины отождествление произойдёт в предложении $p_{1_i}, \ldots, p_{n_i}$= `right-side`$_i$. То есть набор данных $d_{1_i}, \ldots, d_{n_i}$ представляет собой один из базисных случаев

[16] Правые части которых не содержат вызовов функций.

рекурсивного определения F. Правая часть $\texttt{right-side}_i$, по определению, не будет изменена в предполагаемом процессе анализа на мономиальность. Следовательно, если P частичный синтаксический моном, тогда существует моном M_i от формальных переменных $x_1, \ldots, x_n$, такой что результат подстановки вместо этих переменных соответственно $p_{1_i}, \ldots, p_{n_i}$ есть $\texttt{right-side}_i$. Для всех i и j должно выполняться графическое равенство

$$M_i(x_1, \ldots, x_n) = M_j(x_1, \ldots, x_n).$$

Описанная выше стратегия порождения гипотезы обладает одним существенным недостатком: построение гипотезы неоднозначно.

Пример 1. В примере 9 из раздела 9.2.2 единственный синтаксический выход из рекурсии во втором предложении позволяет построить бесконечно много гипотез вида `e.x`^{k} `e.y e.x`^{l}, где k и l произвольные натуральные числа. Причина — в существовании в левой части образца нулевой длины.

Из ограниченности длины правой части любого предложения немедленно следует

Утверждение 1. *В обозначениях данного раздела (см. выше), пусть в программе P существует хотя бы одно предложение $p_{1_i}, \ldots, p_{n_i}$= $\texttt{right-side}_i$, такое что $\texttt{right-side}_i$ не содержит вызовов функций и для всех k образцы p_{k_i} отличны от пустых выражений. Тогда может существовать лишь конечное число гипотез, которые можно построить в рамках описанной выше стратегии.*

Пример 2.

```
* Input Format <F s.x, e.y, s.z>
F { A , e.y, A = A e.y e.y A; }
```

```
/* Определение F есть синтаксический частичный моном, реализующий,
например, как моном s.x e.y e.y s.z, так и моном s.z e.y e.y s.x. */
```

Причина неоднозначности — в разделяемости синтаксических единиц разными образцами левой части предложения. С нашей точки зрения, на практике предложения с таким свойством встречаются редко, и алгоритм может либо рассмотреть все гипотезы, либо ограничиться первой построенной. Пустое же выражение в левой части есть естественное завершение рекурсии, по определению данных РЕФАЛа, и этот случай встречается очень часто. Чем больше пассивных предложений в программе, тем меньше вариантов для гипотез мономиальности.

Пример 3.

```
* Input Format <G e.x, e.y>
G {     , e.2 = e.2;
    e.1,      = e.1;
}
```

```
/* Определение G есть синтаксический частичный моном, реализующий
ровно два монома e.y e.x и e.x e.y . */
```

Дополнительные синтаксически базовые случаи рекурсивного определения можно получить разверткой (прогонкой) правых частей рекурсивных предложений программы P.

Пример 4. После развёртки определения примера 9 из 9.2.2 получаем программу:

```
$ENTRY Double1 {
s.1 s.2 e.x, e.y  = s.1 s.2 <Double1 e.x, e.y s.1 s.2>;
s.1             , e.y  = s.1 e.y s.1;
                , e.y  = e.y;
}
```

* второе предложение которой сразу даёт единственную верную гипотезу.

При построении гипотезы мономиальности по нескольким пассивным предложениям могут быть использованы описанные выше инструменты обобщения параметризованных выражений (см. 5.3.3).

9.2.4. Частичные выражения

Наше ограничение (см. Определение 2 из 9.2.2) на моном конкатенации $x_{j_1}^{k_1} x_{j_2}^{k_2} \dots x_{j_m}^{k_m}$ о положительности степеней k_j, при проверке на синтаксическую мономиальность, может быть без труда снято. То есть можно допустить нулевые степени формальных переменных. Пусть $k_j = 0$ для некоторого j, тогда формальное «повышение» местности по переменной x_j будет заключаться в вычёркивании этой формальной переменной и соответствующих ей образцов в программе. (Формальное повышение местности определений становится формальным понижением по некоторым формальным переменным и повышением по другим.)

Пример 1.

```
* Input Format <F e.x, e.y>
$ENTRY F {
s.1 e.x, s.2 e.y  = s.1 <F e.x, e.y>;
    e.x, e.y  = e.x;
}
```

/* Гипотеза мономиальности однозначно определяется вторым предложением. После понижения местности имеем синтаксический моном `e.x` . */

```
* Input Format <G e.x, e.y>
G { e.x, e.y = <F^1,0 e.x>; }
```

```
* Input Format <F^1,0 e.x>
$ENTRY F^1,0 {
s.1 e.x = s.1 <F^1,0 e.x>;
    e.x = e.x;
}
```

Более того, алгоритм можно применить не только для проверки на синтаксическую мономиальность, но и для проверки на синтаксическую одношаговость программы.

Определение 1. Пусть дана n-местная частичная функция

$$F(x_1, \ldots, x_n) : \; DATA^n \mapsto DATA_\perp,$$

где DATA — множество данных РЕФАЛа. Пусть

R-expr$(x_1, \ldots, x_n)$ ::= empty | R-term$(x_1, \ldots, x_n)$ R-expr$(x_1, \ldots, x_n)$
R-term$(x_1, \ldots, x_n)$ ::= SYMBOL | variable | (R-expr$(x_1, \ldots, x_n)$)
variable ::= x_1 | ... | x_n
empty ::= []

Скажем, что F есть частичное выражение, если существует R-expr$(x_1, \ldots, x_n)$, такое что на области определения функции F верно тождество

$$F(x_1, \ldots, x_n) = \text{R-expr}(x_1, \ldots, x_n).$$

Заметим, что $x_1, \ldots, x_n$ любые формальные переменные, а не только переменные языка РЕФАЛ. В этих обозначениях каждый образец и любая пассивная часть каждого предложения Refal$_n$ программы P являются мономами конкатенации, если на все базисные константы, входящие в них, смотреть как на формальные переменные. В частности, левая структурная скобка (является самостоятельной синтаксической единицей — формальной переменной с именем (. Аналогично, правая структурная скобка. Теперь, если в Refal$_n$ программе P допустить такое расширение множества «переменных» и формально расширить входные форматы (и, соответственно, образцы) всеми базисными константами из P, то буквальное повторение всех определений и утверждений из разделов 9.2.2 и 9.2.3 даст алгоритм проверки на синтаксическую мономиальность программ с расширенным множеством переменных, а следовательно, проверки является ли Refal$_n$ программа P синтаксическим выражением. Мы ограничимся примером (см. [52]).

Пример 2.

```
* Input Format <F e.x, e.y>
F {
 e.y, s.1 e.x = <F e.y s.1, e.x>;
 e.y,     e.x = (e.y e.x (A));
}

/* После формального расширения входного формата базисными
константами имеем: */
* Input Format <FF e.x, e.y, (, ), A>
FF {
e.y, s.1 e.x, (, ), A = <FF e.y s.1, e.x, (, ), A>;
e.y,     e.x, (, ), A = (e.y e.x (A));
}
```

```
/* Гипотеза мономиальности однозначно определяется вторым
предложением FF. После повышения местности и подходящей перестановки
форматных переменных имеем синтаксический моном (e.y e.x (A))
формальных переменных e.x, e.y, (, ), A, т.е. РЕФАЛ-выражение. */
* InputFormat <G e.x, e.y>
G { e.x, e.y = <FF (, e.y, e.x, (, A, ), )>; }

* Input Format <FF (, e.y, e.x, (, A, ), )>
FF {
(, e.y, s.1 e.x, (, A, ), )= <FF(,e.y,s.1 e.x,(,A,),)>;
(, e.y,     e.x, (, A ,), )= (e.y e.x (A));
}
```

Сделаем последнее

Замечание. Описанный выше алгоритм (см. 9.2.2–9.2.4) наиболее широко применим к программам с «хвостовой рекурсией» (каждое предложение имеет не более одного функционального вызова). После упрощения композиционной структуры программы, к моменту её глобального анализа другими инструментами, как показывает наш опыт, компонента факторизации часто обладает свойством «хвостовой рекурсивности».

9.3. Чистка поглощаемых ветвей

Пример 1. В определении Test первое предложение может быть удалено. Прогонка оставит три предложения. Используя инструменты вложения РЕФАЛ-выражений (5.1), SCP4 определит, что второе предложение можно свети подстановкой к первому. Предложения рассматриваются формально как РЕФАЛ-выражения с дополнительным конструктором функционального вызова.

```
$ENTRY Go { e.input = <Test e.input>; }

Test {
 A   e.2 ()     = () <Test e.2>;
 s.1 e.2 (e.3) = (e.3) <Test e.2>;
                = B;
}
```

В результате преобразований имеем:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41 = <F11 e.41> ; }

* InputFormat: <F11 e.41>
F11 {
 s.101 e.41 (e.102) = (e.102) <F11 e.41> ;
  = B ;
}
```

Глава 10. Использование результатов глобального анализа

Глобальный анализ компоненты факторизации полностью завершён. В структуре `basics` (списке базисных узлов, см. описание в 3.1) сохраняется информация о свойствах компоненты факторизации: узел — точка входа, индуктивный выходной формат и другие свойства, обнаруженные глобальным анализом. Согласно стратегии обхода дерева при факторизации (см. 5.2), каждый узел `Node` в метадереве содержит список базисных узлов полностью профакторизованных к данному моменту ветвей, исходящих из `Node`. Список базисных узлов удаляется из метадерева вместе с узлом, его содержащим — когда удаляется `Node`. Таким образом, список базисных узлов не содержит узлов, не достижимых (удалённых) из метадерева. Входная точка компоненты стала базисным узлом.

10.1. Одношаговые подграфы

Если результаты глобального анализа позволяют заменить компоненту факторизации одношаговым вариантом (константа, проекция, тождественная частичная функция, частичный моном конкатенации), тогда входная точка заменяется соответствующим пассивным узлом. Например, в случае тождественной частичной функции, пассивное параметризованное множество полей зрения совпадает с параметризованным значением среды входной точки. При последующем сведении к одношаговой компоненте (вложении в её входную точку, ставшую базисным узлом) подзадач преобразований, сводимый опорный узел также заменяется пассивным узлом, в соответствии с семантикой одношаговости конкретного базисного узла.

10.2. Пустые подграфы

Пустые подграфы удаляются из метадерева вместе с ребрами, входящими в их входные точки. Соответственно, удаляются узлы (ребра, входящие в них), вычисление параметризованных полей зрений которых (или их частей) сведено к вычислению параметризованного поля зрения пустого базисного узла. Следовательно, в метадереве могут появиться новые полутранзитные узлы (см. 4.1). Неопорные полутранзитные узлы удаляются из метадерева возможных вычислений (т. е. их входное ребро склеивается с их выходным ребром).

10.3. Рекурсивные подграфы. Повторная специализация

Рассмотрим основной случай. Пусть в результате анализа рекурсивной компоненты факторизации `P` построен её индуктивный выходной формат (`IND-OUTF assigns`): структура выходной среды рассматриваемого подграфа `P` уточнилась. Мы снова находимся в стартовом положении относительно

специализации подграфа P, ибо он может содержать ребра-ссылки на свою входную точку и, следовательно, информация о структуре выходной среды никак не была использована при построении P.

Пример 1. Рассмотрим вариант примера 1 из раздела 5.3.1.1.

```
$ENTRY Go { e.n = <Fact1 (e.n)>; }

Fact1 {
 () = (I);
 (e.numb)
    = (<Times (e.numb) <Fact1 (<Minus (e.numb) (I)>)>>);
}

/* Определения Times и Minus смотри в 5.3.1.1. Ниже приведён подграф —
компонента факторизации, соответствующая этой задаче после глобального
анализа. Ссылка F16 на другую компоненту для нас несущественна и опре-
деление этой компоненты мы опускаем. */
(INPF e.41 <-a- e.41;)
F7 {
    +  e.41 -c-> []; (I) <-a- e.out;
    +  e.41 -c-> I e.41;
       { e.41 <-a- e.41; } <F7 e.41> { e.out <-a- e.102; }
       e.102 -c-> (e.116);
       { e.41 <-a- e.41; e.116 <-a- e.103; }
       <F16 e.41 , e.103> { e.out1 <-a- e.113; }
       { (e.113) <-a- e.out; }
} (IND-OUTF (e.119) <-a- e.out;)

/* Мы видим, что вторая ветвь может быть содержательно проспециализиро-
вана по структуре индуктивного выходного формата нашего подграфа. */
```

Повторная специализация подграфа специализирует компоненту факторизации по структуре выходных форматов подграфов F_i, на которые эта компонента ссылается (т. е. существует ребро с началом в некотором узле этой компоненты и концом во входной точке одного из F_i). В нашем примере F7 специализируется по структуре выходных форматов F7 и F16. Если выходной формат какого-то подграфа F_i ещё не построен, то в момент специализации он считается находящимся в общем положении (нет информации о его структуре). Мы намеренно называем это преобразование специализацией, а не прогонкой, так как никакие новые развёртки не делаются — подграф лишь чистится. Хотя, специализация производится посредством инструментов из алгоритма прогонки.

Пример 1 после специализации выглядит следующим образом:

```
(INPF e.41 <-a- e.41;)
F7 {
    +  e.41 -c-> []; { I <-a- e.119; }
```

```
    +  e.41 -c-> I e.41;
       { e.41 <-a- e.41; } ≪F7 e.41≫ {{ e.out <-a- e.123; }};
       { e.41 <-a- e.41; e.123 <-a- e.103; }
       ≪F16 e.41 , e.103≫ {{ e.out1 <-a- e.113; }};
       { e.113 <-a- e.119; }
} (OUTF e.119 <-a- e.out;)
```

В синтаксисе мы сохранили информацию о том, что специализация по структуре выходных форматов произведена (двойные фигурные и угловые скобки).

Следующий пример показывает, что местность выходной среды компоненты может увеличиться.

Пример 2. На вход SCP4 подаётся программа:

```
$ENTRY Go {
 (e.names s.name) (e.values t.value)
          = <Zip (e.names s.name) (e.values t.value)>;
}

/* Где функция Zip определена во Введении. */

/* Перед повторной специализацией подграф, соответствующий параметризованному полю зрения <Zip (e.101 s.102)(e.103 t.104)>: */

(INPF e.101 <-a- e.101; s.102 <-a- s.102; e.103 <-a- e.103; t.104 <-a- t.104;)
F18 {
   +  e.101 -c-> s.105 e.101; e.103 -c-> t.106 e.103;
      { e.101 <-a- e.101; s.102 <-a- s.102; e.103 <-a- e.103; t.104 <-a- t.104; }
       <F18 e.101, s.102, e.103, t.104> { e.out <-a- e.107; }
       (s.105 t.106) e.107 <-a- e.out;
   +  e.101 -c-> []; e.103 -c-> []; (s.102 t.104) <-a- e.out;
} (IND-OUTF (s.123 t.124) e.125 <-a- e.out;)

/* После специализации имеем: */

(INPF e.101 <-a- e.101; s.102 <-a- s.102; e.103 <-a- e.103; t.104 <-a- t.104;)
F18 {
   +  e.101 -c-> s.105 e.101; e.103 -c-> t.106 e.103;
      { e.101 <-a- e.101; s.102 <-a- s.102; e.103 <-a- e.103; t.104 <-a- t.104; }
       ≪F18 e.101, s.102, e.103, t.104≫
       {{ s.out1 <-a- s.133; t.out2 <-a- t.134; e.out3 <-a- e.135; }};
       { s.105 <-a- s.123; t.106 <-a- t.124; (s.133 t.134) e.135 <-a- e.125; }
   +  e.101 -c-> []; e.103 -c-> [];
       { s.102 <-a- s.123; t.104 <-a- t.124; [] <-a- e.125; }
} (OUTF s.123 <-a- s.out1; t.124 <-a- t.out2; e.125 <-a- e.out3;)
/* Здесь выходная среда имеет местность три. */
```

10.4. Квази-дистрибутивность подзадачи

В процессе решения задачи свёртки метадерева возможных вычислений эта задача может быть разбита на несколько подзадач с указанием связей между этими подзадачами (см. 5.1, 5.3.2, 5.3.3, 6.2). Напомним (см. 3.1), что в поле зрения суперкомпилятора SCP4

```
scp-view-filed ::= + meta-branch (FOREST forest)
forest ::= current-graph roots
current-graph ::= block
block ::= assignments  { branching } output
        | assignments  { branching } inductive-output
```

`current-graph` есть результат преобразований текущей задачи, `roots` — задачи, которые будут решаться после того, как текущая задача будет решена, а результаты решённых к данному моменту задач находятся в структуре `meta-branch`. Параметризованная среда каждой задачи локальна внутри скобок структуры `block`, и общение с «внешним миром» происходит через вход `assignments` и выход `output` (`inductive-output`). Описанный выше глобальный анализ производится не только для компонент факторизации, но и для каждой подзадачи, когда она полностью свёрнута. Здесь мы хотим показать каким образом распространяется информация от задачи к задаче: через связи между ними и через разделяемые ими параметры.

10.4.1. Правая квази-дистрибутивность

Индуктивный выходной формат определяет структуру выходной среды подзадачи. Значение частичной функции, определяемой подзадачей, может подаваться на вход другим задачам, ждущим своего решения. Подстановка параметризованной структуры выходной среды в аргументы тех задач, где она используется, и передача отрицательной информации `pd_` о параметрах, описывающих эту структуру, приводит к сужению области определения задач, и, следовательно, к потенциальной возможности последующих более глубоких преобразований этих задач. Эти же соображения применимы и относительно константных, тождественных и мономиальных функций; их выходная среда строится простой подстановкой их входной среды.

Пример. Изменим входную точку в Примере 1 из раздела 10.3.

```
$ENTRY IncFact1 { e.numb = <Plus (I) <Fact1 (e.numb)>>; }
```

В процессе преобразований суперкомпилятором SCP4 задача, сформулированная во входной точке, будет декомпозирована и разбита на две задачи (см. 5.3.1, 5.3.2):

```
{ e.1 <-a- e.1; []  <-a- e.2; }
{ [<Fact (e.1) (e.2)> <-a- out.3; , ]} { out.3 <-a- e.3; };
    { e.3 <-a- e.3; } { [<Plus (I) e.3> <-a- out.0; , ]}
    { out.0 <-a- e.4; };
```

После полной свёртки первой из них будет обнаружен нетривиальный индуктивный выходной формат (`IND-OUTF (e.119)` $\overset{a}{\leftarrow}$ `e.out;`), подстановкой которого в аргумент функционального вызова

$$\{\ \texttt{e.3} \overset{a}{\leftarrow} \texttt{e.3;}\ \}\ \texttt{<Plus (I) e.3>}$$

получим { `e.119` $\overset{a}{\leftarrow}$ `e.119;` } `<Plus (I) (e.119)>`. После чего определение `Plus` будет содержательно проспециализировано по появившимся структурным скобкам.

Дополнительные подробности могут быть найдены в [51].

10.4.2. Левая квази-дистрибутивность

Левая квази-дистрибутивность отвечает за распространение информации посредством разделяемых задачами параметров. Корневой узел задачи может не принадлежать к множеству базисных узлов. Более того, в результате преобразований задачи может быть построен граф вообще без базисных узлов, т. е. дерево. Предположим, что корневой узел подзадачи оказался полутранзитным, тогда на ребре, выходящем из этого узла, могут оказаться предикаты-контракции (см. 3.2), сужающие области значений параметров, общих для текущей задачи `current-graph` и задач из `roots`, которые ещё предстоит решить. В этом случае вынос ребра за скобки подзадачи уточнит структуру сред тех задач из `roots`, которые разделяют суженные параметры.

Подчеркнём, что сужение разделяемых параметров происходит косвенно; среда в задаче локальна, и разделяемые параметры входят лишь в описание значений подстановки — входной среды `assignments`. Следовательно, вынос ребра за скобки подзадачи требует повторной прогонки среды `assignments`. Суперкомпилятор SCP4 выносит ребро за скобки подзадачи только тогда, когда прогонка входной среды по этому ребру не приводит к ветвлениям параметров (см. главу 4), определяющих значение среды.

Пример.

```
$ENTRY Go { e.input = <H e.input> e.input; }

H {
 s.1 e.x = s.1;
 e.x = <F e.x B>;
}
```

/* Где функция F определена в примере раздела 9.1.1. В момент прогонки граф определения H имеет ветвление. После глобального анализа компоненты факторизации, соответствующей определению F, вторая ветвь графа H будет удалена и входная точка H станет полутранзитным узлом (оставшись опорным, но не базисным). Единственное ребро может быть полностью вынесено за скобки задачи. Мы получаем результат: */

```
* InputFormat: <Go e.41>
$ENTRY Go { s.132 e.41  = s.132 s.132 e.41; }
```

10.5. К вопросу о целях преобразований

Преобразования на основе результатов глобального анализа могут улучшить порядок алгоритма. В случае синтаксических констант, проекций, тождественных функций и мономов конкатенации это очевидно, ибо циклы преобразуются в одношаговые программы. Если ленивая стратегия развития стека (см. 6.2) может привести к упрощению композиционной структуры преобразуемой программы, когда глубина вложенности функциональных вызовов нетривиальна, то преобразование циклов в одношаговые программы является естественным дополнением к этому инструменту, так как, в частности, охватывает и базовый случай (отсутствие вложенности) и работает после упрощений в результате ленивого развития стека. Кроме того, как уже отмечалось выше, достаточное условие синтаксической мономиальности имеет более широкое применение именно в случае программ с хвостовой рекурсией, т. е. в базисном случае композиционной вложенности.

Для понимания общей структуры суперкомпиляции уместно и важно провести сравнение с интерпретацией языка ПРОЛОГ. Оба алгоритма допускают параметры в аргументах вызовов функций и сужают области допустимых значений этих параметров, иногда обрезая ветви недостижимости дерева развития программы, откатываясь на соответствующие точки ветвления для выбора альтернативной (к обрезанной) ветви. Суперкомпилятор SCP4, в отличие от интерпретатора ПРОЛОГа, может автоматически (без специальной ПРОЛОГ-разметки в программе *"CUT"*) обнаружить некоторые бесконечные циклы и считает ветви, выходящие на такие циклы, недостижимыми (см. раздел 10.2). С другой стороны, такие *CUT*-разметки могли бы дать пользователю новые механизмы настройки процесса суперкомпиляции.

Следует также отметить, что синтаксическая тождественность и её обобщения дают потенциальную возможность динамической типизации данных во время суперкомпиляции; пользователь может определить на РЕФАЛе частичную тождественную функцию — фильтр `Filter`, смысл которой состоит в описании своей собственной области определения, совпадающей с множеством её значений. Предположим, что множество значений фильтра `Filter` совпадает с областью определения некоторой функции `G` из преобразуемой программы, тогда, заменив вызовы `<G e.args>` на `<G <Filter e.arg>>`, мы не меняем семантику программы, но даём дополнительную информацию об аргументе `G`, которая может быть использована преобразователем для оптимизации вычисления данного вызова `<G e.args>`. Если компонента факторизации соответствующая `<Filter e.arg>` всё же построится, то она может быть удалена после проверки достаточного условия на синтаксическую тождественность. Удаление синтаксических мономов является частью интересной задачи самоприменения суперкомпилятора (см. 15.3).

Использование результата построения нетривиального выходного формата также может привести к дальнейшему упрощению композиционной структуры программы. Этот механизм работает при любой стратегии развития стека и может привести к понижению временно́й сложности программы.

Пример 1. Рассмотрим вариант примера 2 из раздела 10.3:

```
$ENTRY Go { (e.names s.name) (e.values t.value)
        = <last <Zip (e.names s.name) (e.values t.value)>>;
}

/* Функция Zip определена во Введении. */

last { e.x t.last = t.last; }

/* Нетривиальный выходной формат компоненты, соответствующей Zip
(см. 10.3), позволяет преобразовать программу к одношаговой: */

* InputFormat: <Go e.41>
$ENTRY Go { (e.101 s.102) (e.103 t.104) = (s.102 t.104); }
```

Каждый конкретный инструмент суперкомпилятора SCP4 производит простейшие элементарные преобразования. Результат композиции этих элементарных преобразований часто бывает интересен. Часть этих элементарных преобразований не зависят друг от друга и могут быть сделаны в порядке, отличном от используемого суперкомпилятором SCP4. Эти преобразования, конечно, не коммутируют. Откуда следует, что повторное применение суперкомпилятора к его результату может привести к последующей оптимизации. Например, описанный нами выше алгоритм проверки достаточного условия синтаксической мономиальности в актуальной версии SCP4 реализован не в общем виде (как это показано в разделе 9.2.4). Как следствие, получаем следующие интересные, на наш взгляд, примеры.

Специализация следующего примера обсуждалась в работах А. Р. Ершова [25, 26], Ё. Футамуры, К. Ноги и А. Такано [31]. Мы рассматриваем вариант для унарной системы счисления. Рассмотрим определение функции Аккермана.

```
* <Ack (e.m) (e.n)>
Ack {
 ()      (e.n)    = I e.n;
 (I e.m) ()       = <Ack (e.m) (I)>;
 (I e.m) (I e.n) = <Ack (e.m) (<Ack (I e.m) (e.n)>)>; }
```

Пример 2. Простейший пример — мало интересен. Мы приводим его лишь для полноты. На вход суперкомпилятору подаётся задача на специализацию:

```
$ENTRY Go { e.n = <Ack () (e.n)>; }
```

* Остаточная программа:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41 = I e.41; }
```

Пример 3. На вход суперкомпилятору подаётся задача на специализацию:

```
$ENTRY Go { e.n = <Ack (I) (e.n)>; }
```

* Остаточная программа:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41  = I <F5 e.41>; }
```

```
* InputFormat: <F5 e.41>
F5 {
          = I;
 I e.41   = I <F5 e.41>;
}
```

* Применяем SCP4 к собственному результату:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41   = I I <F9 e.41>; }

* InputFormat: <F9 e.41>
F9 {
          = ;
 I e.41   = I <F9 e.41>;
}
```

* Третья итерация SCP4 даёт идеальный результат:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41 = I I e.41; }
```

Пример 4. На вход суперкомпилятору подаётся задача на специализацию:

```
$ENTRY Go { e.n = <Ack (I I) (e.n)>; }
```

* После трёх итераций нашего преобразователя получаем остаточную программу:

```
* InputFormat: <Go e.41>
$ENTRY Go { e.41   = I I I <F7 e.41>; }

* InputFormat: <F7 e.41>
F7 {
          = ;
 I e.41   = I I <F7 e.41>;
}
```

Пример 5. Проверка префикса выражения:

```
$ENTRY Go {
  (e.pref) (e.exp) = <Prefix (e.pref) e.pref e.exp>;
}

Prefix {
 (s.1 e.pref) s.1 e.exp = <Prefix (e.pref) e.exp>;
 ((e.1) e.pref) (e.2) e.exp
                        = <Prefix (e.1 e.pref) e.2 e.exp>;
 () e.exp = True;
 (s.1 e.pref) e.exp = False;
 ((e.1) e.pref) e.exp = False;
}
```

* Остаточная программа:

```
* InputFormat: <Go e.41>
$ENTRY Go { (e.101) (e.102) = True; }
```

Пример 6. На вход SCP4 подаётся тождественная подстановка в выражение `e.where`:

```
$ENTRY Go {
 e.SYMBOL (e.where) = <Subst (s.SYMBOL s.SYMBOL) e.where>;
}
Subst {
 (s.what e.value) = ;
 (s.what e.value) s.what e.where
                  = e.value <Subst (s.what e.value) e.where>;
 (s.what e.value) s.1 e.where
                  = s.1 <Subst (s.what e.value) e.where>;
 (s.what e.value) (e.1) e.where
                  = (<Subst (s.what e.value) e.1>)
                    <Subst (s.what e.value) e.where>;
}
```

* На выходе SCP4 имеем:

```
* InputFormat: <Go e.41>
$ENTRY Go { s.101 (e.102) = e.102; }
```

В разделе 5.4 мы уже отмечали проблемы, связанные с нелокальностью (по отношению к текущей ветви метадерева) базисных конфигураций (см. 5.2) — их нелокальность является серьёзным препятствием при анализе глобальных свойств компонент факторизации. Теперь мы готовы указать на эту проблему более конкретно. При анализе глобальных свойств компонент факторизации (описанном нами выше в главе 9) для актуальной версии SCP4 принципиальной является самодостаточность компоненты (см. главу 7), выбор же алгоритмом вложения базисной конфигурации, нелокальной по отношению к рассматриваемой компоненте факторизации, нарушает это важное свойство.

Глава 11. Чистка входных, выходных формальных параметров и вызовов функций

Задача уменьшения местности «функций»-определений возникает не только из природы неоптимальности входной программы, но и из свойств алгоритма обобщения (см. 5.4.1), который может быть вынужден, при факторизации некоторого пути в цикл, построить формальный параметр будущей компоненты факторизации, не влияющий на значение функций, определяемых этой компонентой (или несколькими компонентами) и всеми обращениями к ней (её вызовами) из остаточной программы. Таким образом, эта задача существенно глобальна; и мы решаем её отдельно, после завершения основной стадии преобразований всей входной программы.

Ленивая прогонка (позволяющая иногда отказаться от вычисления значения функции, если оно не необходимо в контексте данной композиции)

дополняется поиском в остаточной программе части выходных параметров, вычисление которых также может быть опущено.

Уменьшение местности или ко-местности некоторой функции-определения в остаточной программе может повлечь за собой удаление вызовов функций, вычислявших значения сокращённых формальных параметров, и, следовательно, может привести к улучшению порядка сложности остаточного алгоритма. Временна́я сложность входной программы из данного ниже примера 1 (число шагов РЕФАЛ-машины) порядка $O(2^n)$, где n-длина входной строки. Преобразователь SCP4 выдаёт остаточную программу, время исполнения которой линейно по n. С учётом копирования значения переменной `e.1` входная программа требует для своего исполнения памяти порядка $O(n * 2^n)$. Остаточная программа потребляет память порядка $O(n)$. (В процессе основной стадии суперкомпиляции будет построена компонента с тремя формальными параметрами, соответствующими следующему естественному разбиению образца `(e.0) (e.2) e.1` (см. 9.1.2).)

Пример 1. Входная программа:

```
$ENTRY Go { e.1 = <F (e.1) () e.1>; }

F {
 (e.0) (e.2) A e.1
               = <F (<F (e.0) (e.2 C) e.1>) (e.2 B) e.1>;
 (e.0) (e.2)   = e.2;
}
```

* Остаточная программа:

```
$ENTRY Go { e.1 = <F45 (e.1)>; }

F45 { (A e.1) e.2 = <F45 (e.1) e.2 B>;
        ()    e.2 = e.2;
}
```

Пример 2 показывает, что чистки входных и выходных формальных параметров взаимосвязаны. Основная стадия суперкомпиляции построит компоненту-граф (соответствующий функции F), местность которого равна трём, а ко-местность двум (см. 4.3, 9.1.2). Далее будет обнаружено, что значение второго выходного формального параметра не влияет на результат программы, а следовательно, результат программы не зависит и от третьего входного формального параметра этой компоненты, в которой вычисляется значение второго выходного параметра. Остаточная компонента будет построена местности два и ко-местности один.

Пример 2.

```
$ENTRY Go { e.1 = <Out <F (e.1) () ()>>; }

Out { (e.1) (e.2) = (e.1) (); }

F {
 ()       (e.2) (e.3) = (e.2) (e.3);
 (A e.1) (e.2) (e.3) = <F (e.1) <F (e.1) (e.2 B) (e.3 C)>>;
}
```

Структуру нашего алгоритма анализа-чистки можно грубо описать как «суперкомпиляцию» программы, множество данных которой состоит из двух точек — **используемый** и **неиспользуемый** параметр для построения результата функции, определяемой входной точкой (стартовой конфигурацией). Мы начинаем развёртку метадерева возможных вычислений со стартовой конфигурации со значениями всех входных параметров, равными **неиспользуемый**, и со значением её единственного выходного формального параметра, равным **используемый**. Все формальные параметры, как входных, так и выходных форматов всех функций, *a priori* помечаются как **неиспользуемые**; аналогично помечаются фактические выходные параметры всех вызовов функций (кроме входной точки, см. выше). В процессе развёртки мы анализируем вдоль каждой ветви: понадобится ли значение данного входного параметра p при выборе этой ветви (при отождествлении) во время интерпретации и понадобится ли значение p для построения фактических параметров, соответствующих формальным параметрам помеченных (к моменту анализа) как **используемые**, вызовов функций F_i (расположенных на этой ветви), таких что хотя бы один формальный выходной параметр функции F_i помечен как **используемый**. (Анализ, естественно, — аппроксимационный.) Кроме того, мы анализируем, понадобится ли значение p, если рассматриваемая ветвь будет пройдена, для построения выходных фактических параметров текущей функции (развёртку которой мы анализируем), помеченных к данному моменту как **используемые**. Аналогично анализируются и фактические выходные параметры всех вызовов функции F_i (как на выбор ветви, так и на построение). В результате такого анализа некоторые входные формальные параметры текущей функции и выходные формальные параметры функций F_i «обобщаются» (помечаются) до **используемых**; «обобщаются» и некоторые фактические выходные параметры вызовов функций F_i. Если такое «обобщение» формальных параметров произошло, тогда мы отказываемся от построенной части метадерева с корнем в текущей конфигурации и начинаем развёртку текущей конфигурации (вызова) заново в контексте «обобщённых» форматов. Если «обобщение» форматов не произошло, тогда переходим к развёртке очередного вызова функции F_i, при условии что в построенном к этому моменту метадереве не разворачивалось ни одно обращение к функции F_i. Таким образом просматриваются все ветви метадерева.

Когда всё дерево потенциально-возможных вычислений пройдено, формальные (входные и выходные) параметры, оставшиеся помеченными как **неиспользуемые**, удаляются из определений-функций. Удаляются и соответствующие им фактические параметры вызовов функций, также удаляются из программы те вызовы функций, все фактические выходные параметры которых сохранили метки **неиспользуемые**. В случае, если мы удалили хотя бы один такой вызов, весь процесс анализа начинается заново для откорректированной программы, ибо **используемые** фактические выходные параметры этого вызова удалены из программы и, следовательно, могут появиться новые ненужные формальные параметры. Мы заканчиваем анализ, когда построили неподвижную точку-программу.

Глава 12. ***Чистка повторных определений***

12.1. Глобальность базисных конфигураций внутри задачи и по задачам

В любой данный момент основной стадии преобразований список базисных конфигураций содержит все конфигурации, определяемые теми базисными узлами (см. 5.1), которые достижимы из корня метадерева посредством путей; как непосредственно, так и косвенно — через точки входов в компоненты факторизации. Таким образом, в этом списке нет базисных конфигураций, определяемых базисными узлами, лежащими на отброшенных ветвях (либо в результате обобщения конфигураций; либо как ветви, недостижимые при любых входных данных задачи, определяемой корнем метадерева). Такая глобальность базисных конфигураций [17] позволяет сократить время факторизации метадерева, когда параметризованный стек `st` некоторого данного узла частично или полностью вкладывается в одну из этих базисных конфигураций. С другой стороны, это вложение может стать причиной построения менее эффективного конечного результата преобразований — подзадача, определяемая `st`, не будет проспециализирована (см. также 5.4 и 10.5).

12.2. Повторные определения

В этом случае глобальность может сократить число компонент факторизации в остаточной программе (т. е. число рекурсивных определений) и аппроксимирует понятие построения повторного, дублирующего определения. Следующий пример показывает другую причину появления дублирующих определений в остаточной программе.

Пример 1.

```
$ENTRY Go { e.1 = <Def e.1>; }

Def {
  A e.1 = B <Def e.1>;
  s.1 e.1 = s.1 <Rep_Def e.1>;
          = ;
}

Rep_Def {
  A e.1 = B <Rep_Def e.1>;
  s.1 e.1 = s.1 <Def e.1>;
          = ;
}
```

Данная программа будет полностью воспроизведена (с точностью до имён определений и имён переменных) описанными выше в нашей работе преобразованиями; будут построены два синтаксически разных, но семантически совпадающих определения, ибо формально-синтаксические структуры стеков `<Def e.1>` и `<Rep_Def e.1>` не совпадают (см. 5.1, 5.3).

---

[17] Специализация посредством техники поливариантных частичных вычислений имеет (по логике существенно более простой) аналог глобального списка базисных конфигураций (*"dejavu"*), где «базисная конфигурация» есть *specialization point*. В этой технике вложение основано на полном совпадении статической части конфигураций (см. [39, 63]).

Чистка формальных параметров функций (см. главу 11) также может привести к подобному повторению определений. Следующей стадией (после этой чистки) преобразований из остаточной программы выкидываются повторные, с точностью до имён определений и имён переменных, определения. И затем повторные вычисления, определяемые повторными вызовами вдоль данной ветви, заменяются копированиями значения первого из этих вызовов (посредством переменных).

Пример 2. (Определение `Def` дано в примере 1.)

```
$ENTRY Go {
 A e.1 = <Def e.1> <Def e.1>;
 B e.1 = <Test A <Def e.1>> <Test B <Def e.1>>;
}

Test { s.1 s.2 e.1 = e.1; }
```

После чистки повторных определений результат преобразований выглядит так:

```
* InputFormat: <Go e.41>
$ENTRY Go {
 A e.41 ,<F19 e.41>:e.129 ,<F19 e.41>:e.129 = e.129 e.129;
 s.101 e.41 ,<F19 e.41>:s.143 e.142
            ,<F19 e.41>:s.152 e.151 = e.142 e.151;
}

* InputFormat: <F19 e.41>
F19 {
 A e.41 = B <F19 e.41>;
 s.106 e.41 = s.106 <F19 e.41>;
  = ;
}
```

Отметим, что область определения функции `Go` расширена. В окончательном ответе эта функция выглядит так:

```
* InputFormat: <Go e.41>
$ENTRY Go {
 A e.41 ,<F19 e.41>:e.129 = e.129 e.129;
 s.101 e.41 ,<F19 e.41>:s.157 e.142 = e.142 e.142;
}
```

Глава 13. Неадекватная выразимость результата преобразований средствами РЕФАЛа-5

Любая реализация языка программирования L фиксирует модель вычислений, которая включает в себя некоторые понятия вычислительных ресурсов абстрактной машины этого языка. Нас будет интересовать понятие логиче-

ского времени, которое требуется машине для вычисления данной входной конфигурации. В идеальной ситуации программист (пользователь) опирается именно на это понятие логического времени при разработке программы. По мнению автора, логический уровень языка L определяется, в том числе, и его операционной семантикой; чем более простые понятия вычислительных ресурсов, тем более высокого уровня язык. Если встанем на такую точку зрения, то расширение L синтаксическими средствами, позволяющими более адекватно ориентироваться на физические вычислительные ресурсы компьютера, есть понижение логического уровня этого языка. (Хотя расширенный язык включает в себя расширяемый, первый стимулирует более низкий уровень мышления, чем второй.) С другой стороны, отсутствие в L такого рода синтаксических средств является препятствием для построения автоматических оптимизаторов (преобразователей из L в L), конечная цель которых есть уменьшение физических вычислительных ресурсов, необходимых для решения конкретной задачи. Одно из возможных решений (обычно используемое) данной проблемы — компилятор $L \mapsto L_1$, где L_1 — язык более низкого уровня. Система становится незамкнутой, если операционная семантика языка L_1 находится в руках других разработчиков. Пусть мы имеем следующую ситуацию: программы на языке L компилируются в язык LVM (язык L — «виртуальной» машины), и далее интерпретируются. Обычно LVM разрабатывается (поддерживается) совместно с L. В этом случае LVM является естественным языком для отражения результатов анализа программы, написанной на L. Более того, язык LVM недоступен пользователю, и поэтому может быть свободно расширяем, с целью более адекватной выразимости результатов оптимизации.

Именно по такой схеме и реализованы все известные автору диалекты языка РЕФАЛ, кроме РЕФАЛ+. Соответствующие LVM в РЕФАЛ-е принято называть языками сборки. Язык РЕФАЛ-графов (см. главу 3) представляет собой расширенный вариант одного из языков сборки.

Перечислим некоторые синтаксические конструкции языка РЕФАЛ-графов, при отображении которых в РЕФАЛ-5 (мы говорим о конкретной реализации) может быть потеряна потенциальная возможность проинтерпретировать их более эффективно. Многоместные функции-определения, с одной стороны, позволяют сократить число конструкторов для построения аргументов функциональных вызовов, и, следовательно, упростить образцы в определениях этих вызываемых функций; с другой стороны, временна́я цена конкатенации выражений весьма мала, а время, необходимое на передачу нескольких аргументов (нетривиальной среды), больше времени передачи одного аргумента. В языке РЕФАЛ-графов процесс сопоставления (отождествления) фактических аргументов функции с образцами в её определении, в некотором смысле, происходит со всеми образцами «одновременно». Синтаксически это описывается в виде нетривиального дерева разбора образцов (см. пример в 3.3.2). При переводе программы в РЕФАЛ-5, хотя и можно сохранить структуру этого дерева (компилируя его ветвления в блоки, см. [69]), возникающие дополнительные копирования значений переменных (свойство актуальной реализации) не только сводят на нет данную оптимизацию отождествления, но и существенно ухудшают ситуацию. В дереве ветвлений РЕФАЛ-графа

точки ветвления «функциональны», т. е. подобно функциям-определениям неудача при отождествлении внутри конкретного ветвления приводит к аварийной остановке (соответствующая функция не определена на данной среде). Это свойство ветвлений может позволить значительно сократить число копирований значений переменных (см. главу 14). Нетривиальная ко-местность функций-определений не может быть никак адекватно выражена в синтаксисе РЕФАЛа-5. Мы вынуждены представлять многоместную выходную среду одним РЕФАЛ-выражением и сразу же сопоставлять это представление с самим собой; будучи уверенными, что сопоставление будет удачным (см. пример остаточной программы в Приложении А). В языке РЕФАЛ-графов выразимо возвращение значения блока (подобно значению вызова функции). То есть блок является одновременно и определением, и вызовом этого определения, который может быть размещён в любом месте программы. В то время как в РЕФАЛе-5 блок может быть только хвостовым вызовом. Типизация параметров (см. главу 15) может дать дополнительную информацию о множестве значений входных и выходных формальных параметров остаточной функции, которая может быть использована для более эффективной интерпретации.

Глава 14. **Разметка свойств переменных и компиляция в Си (или в язык сборки)**

Любая компиляция (оптимизация) имеет смысл только в контексте фиксированной модели вычислений, в которой определено понятие единицы измерения логического времени. Это логическое время может иметь (и чаще всего имеет) весьма отдалённое отношение к физическому времени, измеряемому какими бы то ни было часами. Эта фиксированная модель вычислений есть модель вычислений языка, в который переводится исходная программа. (Не может быть суперкомпилятора РЕФАЛа, и даже РЕФАЛа-5; можно лишь говорить о суперкомпиляторе данной реализации РЕФАЛа-5.)

14.1. Уменьшение числа копирований

В модели вычислений рассматриваемой реализации РЕФАЛа-5 значения `e`- и `t`-переменных из среды, которая строится в процессе отождествления, «копируются» (т. е. строятся их «физические» двойники из новых «кирпичиков», а не просто указывается адрес уже существующего в памяти компьютера выражения) в следующих случаях: (1) при построении выражения (синтаксически описанного в правой части предложения из РЕФАЛ-программы) в той части, в которую входит повторное вхождение данной переменной — т. е. $n + 1$, $n + 2$ и т. д. вхождения, где n есть суммарное число вхождений этой переменной во все образцы из левой части рассматриваемого предложения; (2) при построении выражения в конструкции условия (см. [69]) в той его части, в которую входит любое вхождение данной переменной.

Значения n первых вхождений переменной (n экземпляров) в правую часть уже существуют (построены на предыдущих стадиях работы программы) в соответствующих образцах и, при замене этих n вхождений переменной их значениями, лишь указываются адреса построенных ранее значений. Таким образом, время замены первых n вхождений переменной их значениями равномерно ограничено по размеру $size$ входных данных, а замена повторных вхождений e- и t-переменных требует времени порядка $O(size)$. Размер значений e- и t-переменных может быть сколь угодно большим.

Пример 1.

```
F { e.x t.y, <G t.y e.x>:
        { s.1 e.z t.y = t.y <T t.y t.y t.y e.x e.x>;
          t.1 e.z = A;
        };
}
```

Здесь повторными вхождениями переменных являются подчёркнутые вхождения, и только они. При построении выделенного выражения **<G t.y e.x>** в конструкции условия значения переменных будут скопированы.

Рассмотрим причину копирования значений переменных при построении выражения exp в конструкции условия: при интерпретации exp значения аргументов функциональных вызовов (входящих в exp) не являются инвариантами, и могут быть изменены. С другой стороны, эти значения могут ещё понадобиться (в случае неудачного отождествления на текущей ветви) для продолжения интерпретации данного ветвления. Древовидная структура отождествления в языке РЕФАЛ-графов (см. 3.3.2) позволяет минимизировать степень ветвлений в определении-функции и, следовательно, расширяет возможности анализа — необходимо ли значение данной переменной для отождествления вдоль нижележащих (в данном ветвлении) ветвей или для построения их правых частей. Кроме того, некоторые из аргументов функциональных вызовов, входящих в exp, могут оказаться инвариантами интерпретации exp. Например, если мы определим функцию G, которая вызывается в примере 1, нижеследующим образом:

Пример 2.

```
G { t.y  s.1  e.x = <R t.y>;
    t.y (e.z) e.x = <R t.y t.y>;
  }
```

Тогда значение переменной e.x при вычислении вызова **<G t.y e.x>** будет инвариантом, и потому в копировании этого значения нет необходимости. Анализ остаточной программы с целью сокращения числа копирований (соответствующей разметки переменных), с точки зрения автора, является необходимой частью дальнейшего развития суперкомпилятора SCP4 и находится в стадии разработки.

14.2. Хвостовая рекурсия

Другая важная оптимизация, которая, как и предыдущая, может быть сделана на уровне интерпретации языка сборки (или при прямой компиляции языка РЕФАЛ-графов в язык С (например), где такая оптимизация более актуальна) — интерпретация хвостовой рекурсии. На содержательном уровне, под хвостовой рекурсией мы понимаем рекурсию, которую можно откомпилировать в язык С без использования стековых операций (посредством операторов GOTO), либо с использованием минимального количества таких простейших операций. Задача состоит в выявлении таких рекурсий на уровне синтаксиса. Рассмотрим несколько примеров.

Пример 1.

```
F { A e.1 = B <F e.1>;
    s.2 e.1 = s.2 <F e.1>;
            = ;
}
```

Эта рекурсия может быть оформлена в виде цикла L, в котором выделенное **F** можно рассматривать как метку — вход в L, функциональные вызовы — операторы перехода на эту метку. Параметр цикла (выражение-аргумент функции **F**, являющийся глобальной структурой по отношению к L) изменяется внутри L. Выход из цикла L — по исчерпанию этого выражения.

Пример 2.

```
* InputFormat: <F1 s.1 e.2>
F1 { s.1 = s.1;
     s.1 e.2 s.3 = <F1 <Add s.1 s.3> e.2>;
}
```

Здесь в теле цикла вызывается внешняя функция Add, которую цикл должен корректно запустить и принять её результат.

Пример 3.

```
G { B e.1 = <G e.1>;
    A e.1 = <F2 e.1>;
  s.2 e.1 = False;
          = True;
}
```

```
F2 { B e.1 = <G e.1>;
     A e.1 = <F2 e.1>;
   s.2 e.1 = False;
           = False;
}
```

В этом примере существуют две метки **G**, **F2** и перекрёстные переходы на них.

А. П. Конышев написал компилятор, отображающий выход суперкомпилятора SCP4 в язык программирования С [4]. Остаточная программа компилируется в новую встроенную (примитивную) функцию РЕФАЛа-5 и подлинковывается к интерпретатору РЕФАЛа-5. После чего эта функция может быть использована из другого РЕФАЛ-5 модуля как любая стандартная примитивная функция. Компилятор требует дальнейшего развития.

Глава 15. Поднятие параметра (уточнение языка параметров). О синтаксисе входных точек

15.1. Постановка задач на специализацию

Любой метаинструмент T использует представление (т. е. некоторую кодировку μ) преобразуемых или анализируемых им программ, написанных на языке M, в виде данных языка реализации L этого инструмента. Если T преобразователь программ, тогда, чтобы можно было восстановить закодированную программу, μ должна быть инъекцией (по модулю имён функций-определений и имён переменных). Далее под метаинструментом мы будем всегда понимать пару (T, μ) — т. е. кодировка μ не только фиксирована, но и известна пользователю. Саму кодировку μ будем обозначать подчёркиванием (например, Prog есть результат кодировки программы Prog). Чтобы можно было удобно рассуждать о преобразуемой программе (т. е. манипулировать её понятиями посредством T), кодировка μ должна, по возможности, сохранять те синтаксические конструкторы языка M, которые есть в данных языка L, и отображать другие конструкторы языка M в адекватные конструкторы данных языка L. Основной конструктор построения программ на любом языке есть конструктор конкатенации (приписывания) строк. С нашей точки зрения, сохранение конкатенации, с точки зрения удобства рассуждений (как о самом метаинструменте T, так и о преобразуемых им программах), является принципиальным моментом.

Приведём некоторые детали кодировки, используемой нашим преобразователем для представления РЕФАЛ программ (точнее РЕФАЛ-графов) и РЕФАЛ данных *DATA* (как части программ) посредством РЕФАЛ данных.

```
Program = (definition+ )
F { branch;+ } = (F branch;+ )
    ...
expr1 expr2 = expr1 expr2
(expr ) = ('*' expr )
<F expr> = (Call F expr)
e.name = (Var 'e' name)
t.name = (Var 't' name)
s.name = (Var 's' name)
SYMBOL = SYMBOL
```

Пустое выражение кодируется самим собой.

С точки зрения преобразований, интересными и привлекательными являются задачи специализации интерпретаторов по фиксированной программе и неизвестным данным [18] [28, 30, 35, 42, 56, 63, 66, 71],

```
<Int <Go e.data> (Prog)>
```

[18] То есть λe.data•<Int <Go e.data> (Prog)>.

и самоприменения преобразователя [14, 28, 35, 57, 63]. В нашей формулировке первой задачи есть неточность: фактическая область допустимых значений параметра есть образ множества $DATA$ — $\mu(DATA)$ и является собственным подмножеством $DATA$. Это обстоятельство никак не отражено на уровне синтаксиса постановки задачи. Если мы попытаемся вычислить вызов `Int` на данных $d \notin \mu(DATA)$, то результат будет непредсказуемым. Если же попытаемся проспециализировать задачу в нашей формулировке, то, как мы покажем ниже, получим в остаточной программе накладные расходы (неэффективность), причина которых в отсутствии информации об О. Д. З. параметра `e.data`. Следуя за В. Ф. Турчиным [57, 74], мы изобразим эту ситуацию посредством двумерно-позиционного языка [19]:

```
<Int      e.data    (      ) >
    <Go   !      >   Prog
```

В этой схеме закодированные синтаксические структуры сдвинуты на один уровень вниз (и более никак не изменены). Параметр `e.data`, как синтаксические структура, не закодирован, но ему можно придавать лишь закодированные значения; восклицательный знак указывает место, где эти значения должны оказаться после их подстановки. Входная точка `Go` программы `Prog` (вызываемая на нижней строке) полностью определена; в момент обращения (интерпретации) к ней `Int` её аргумент будет фиксированной константой. Определение интерпретируемой программы мы обозначили символом `Prog`, который является метасимволом по отношению к нашей схеме. После включения в эту схему самого специализатора, окончательная постановка задачи выглядит так:

```
<SCP4                                  (          ) >
      <Int      e.data    (     )>   SourceInt
          <Go   !     >   Prog                          (I)
```

Здесь синтаксические структуры на второй сверху строке закодированы один раз, на третьей — два раза. Верхняя строка не содержит параметров, следовательно, аргументы вызова SCP4 полностью определены и этот вызов может быть вычислен интерпретатором РЕФАЛа. На второй строке есть ровно один параметр — это означает, что суперкомпилятор построит программу, определяющую функцию от одного аргумента.

Одна из многочисленных задач самоприменения может быть описана так:

```
<SCP4          ......   ......................(          )>
    <SCP4              e.Prog  (          )> SourceSCP4
        <Int e.data ( !    )>  SourceInt
                                                        (II)
```

Схема **(II)** описывает задачу преобразования суперкомпилятором `SourceSCP4` (вторая строка) языка, семантика которого описана интерпретатором `SourceInt`, в РЕФАЛ-5. Эта постановка задачи может быть уточнена относительно О. Д. З.

[19] Турчин называет его языком «метасистемных переходов».

параметра `e.data` аналогично схеме **(I)**. Заметим, что без «поднятия» параметра (посредством `!`) мы не смогли бы корректно поставить эту задачу, ибо с точки зрения преобразуемого SourceSCP4, `e.Prog` есть *фиксированная* константа, по которой ведётся специализация; а `e.data` — параметр, который изменяется. В то же время `e.data` не является параметром с точки зрения преобразующего SCP4, ибо эта синтаксическая конструкция видна ему как `e.data`, а не как `e.data`[20].

15.2. Подтипы параметров

Описанные выше входные конфигурации суперкомпилятора (схемы **(I)** и **(II)**) показывают необходимость расширения языка параметров, описанного нами в главе 2. Уточним этот язык. Сопоставим каждому из трёх типов параметров (`type ::= s|t|e`) бесконечное множество подтипов, занумерованное натуральными числами. Пусть $D(type.n)$ есть О. Д. З. параметра $type.n$, тогда

$$type_0.n ::= type.n, \quad D(type_i.n) ::= \mu^i(D(type.n)).$$

Отметим свойство нашей кодировки:

$$\mu^{i+1}(D(e.n)) \subset \mu^i(D(e.n)),$$
$$\mu^{i+1}(D(t.n)) \subset \mu^i(D(t.n)),$$
$$\mu^{i+1}(D(s.n)) \equiv \mu^i(D(s.n)),$$

где в первых двух случаях включения строгие. i-й подтип параметра соответствует поднятию этого параметра в схеме на i строчек вверх от места подстановки. Выбранная нами кодировка тождественна на $D(s.n)$. Это замечание и свойство сохранения кодировкой μ конструктора конкатенации позволяют ввести ещё один (часто удобный) подтип $type_*.n$, О. Д. З. которого совпадает с множеством μ-неподвижных точек в О. Д. З. соответствующего типа:

$$D(s_*.n) ::= D(s.n), \quad D(t_*.n) ::= D(s.n), \quad \text{а} \quad D(e_*.n) ::= SYMBOL^*.$$

Расширив язык параметров, мы расширили и язык множеств параметризованных полей зрения и РЕФАЛ-выражений (см. главу 2).

15.2.1. Уточнение прогонки

Прогонка решает уравнения с параметрами

```
Pattern = Parameterized_Expression,        (III)
```

где `Pattern` — строгий образец (см. 4.1). В правую часть этого уравнения могут входить введённые выше параметры. Алгоритм из Утверждения 1 раздела 4.1 легко расширяется на этот случай[21]. Важно отметить, что остаётся

---

[20] Две нижние строки этой схемы могли бы быть сформулированы как `λe.Prog•<SCP4 λe.data•<Int e.data (<μ e.Prog>)> (SourceInt)>`, но, с операционной точки зрения преобразующего SCP4, такая формулировка не эквивалентна данной в тексте, если функция μ будет рассматриваться как любая другая функция, определённая на РЕФАЛе.

[21] При выбранной нами кодировке μ.

верным Следствие 1 из этого Утверждения 1. Напомним его: «Предикаты, которыми помечены ребра грозди прогонки, выразимы в языке элементарных строгих образцов». Листья грозди и её узлы — точки ветвления — могут описываться в расширенном языке параметризованных полей зрения. Мы должны закодировать информацию об О. Д. З. подтипов в виде композиции таких образцов. Решим уравнение **(III)**, не учитывая подтипы параметров[22]. Так как не учтены только ограничения на О. Д. З. подтипов, то построенная таким образом гроздь Ψ может отличаться от искомой грозди Ω только дополнительными предикатами-ограничениями p_j $(1 \leqslant j \leqslant k)$ на ребрах Ω и уточнениями описаний узлов и листьев; ибо множества, заданные параметрами, входящими в эти описания в Ω, могут быть собственными подмножествами соответствующих им множеств в Ψ (предикаты p_j на ребрах «сузили» множества из Ψ). Например, это может выражаться в исчезновении подветвей, на стартовых ребрах которых последовательность предикатов p_j противоречива. (Отбрасывание таких ветвей и есть основная цель введения подтипов.)

Пусть предикат $q(type_i.n, patt)$ (напомним, что $patt$ — строгий образец, см. 4.1) на ребре в гроздн Ψ сужает О. Д. З. параметра $type_i.n$ посредством образца $patt$[23], обозначим его композицию с предикатами p_j в гроздн Ω через p:

$$p(type_i.n, patt') ::= p_1 \circ \ldots \circ p_k \circ q(type_i.n, patt).$$

Так как $D(type_i.n) = \mu^i(D(type.n))$, то все конструкторы в $patt'$ должны быть закодированы (предикаты p_j проверяют наличие этой кодировки). $patt' = \nu^i(patt)$, где ν^i определяется так:

$\nu^0(\ patt\ ) ::= patt$

$\eta^0(\ patt\ ) ::= patt$

$\nu^*(\ (patt)\ )$ — есть предикат-противоречие.

$\nu^*(\ t_*.k\ ) ::= s_*.m$ — выполнить замену параметра $t_*.k$ на $s_*.m$.

$\nu^*(\ s_*.k\ ) ::= s_*.k$

$\nu^*(\ e_*.k\ ) ::= e_*.k$

$\nu^{i+1}(\ (patt)\ ) ::= (\eta^{i+1}(\nu^{i+1}(patt)))$

$\eta^{i+1}(\ '*'\ \ patt\ ) ::= '*'\ \ \eta^i(\ patt\ )$

$\eta^{i+1}(\ s_j.k\ \ patt\ ) ::= '*'\ \ \eta^i(\ patt\ )$ — выполнить замену параметра $s_j.k$ на $'*'$.

$\eta^{i+1}(\ t_j.k\ \ patt\ ) ::= '*'\ \ \eta^i(\ patt\ )$ — выполнить замену параметра $t_j.k$ на $'*'$.

$\eta^{i+1}(\ e_j.k\ \ patt\ ) ::= '*'\ \ \eta^i(\ e_j.m\ \ patt\ )$ — выполнить замену параметра $e_j.k$ на $'*'\ \ e_j.m$ и продолжить с $e_j.m\ \ patt$, при других значениях аргумента

$\eta^{i+1}(patt)$ есть предикат-противоречие для оставшихся случаев $patt$.

$\nu^i(SYMBOL) ::= \mu^i(SYMBOL)$

$\nu^i() ::= \mu^i()$

---

[22] То есть считая их, как и точку, формальным разделителем и дублируя их вместе с этой точкой (см. примеры, данные ниже).

[23] Все параметры, входящие в $patt$, являются параметрами подтипа i.

$\nu^{i+1}(\ type_{i+1}.k\ ) ::= type_{i+1}.k$

$\nu^{i}(patt_1\ \ patt_2) ::= \nu^{i}(patt_1)\ \ \nu^{i}(patt_2)$

Здесь *(,)* — метаскобки, а (,) — структурные скобки РЕФАЛа (конструктор), m — новое имя, которого нет ни в *patt*, ни в построенной части *patt'*. i и j — натуральные числа или $*$. Повсюду, где встречается $i+1$, там i — натуральное число.

Пример 1.

Уравнение: (s.x) e.y = [e.1 (A) e.2 <F e.1> , e.2 ≠ []]
Ответ:
Если e.1 $\xrightarrow{c}$ [], то [{s.x = A, e.y = e.2 <F >} , e.2 ≠ []]
иначе, если e.1 $\xrightarrow{c}$ (s.11) e.12, то
[{ s.x = s.11, e.y = e.12 A e.2 <F (s.11) e.12> }, e.2 ≠ []]
иначе корней нет.

Уравнение: (s.x) e.y = [e_1.1 (A) e_2.2 <F e_1.1>, e_2.2 ≠ []]
Ответ:
Если e_1.1 $\xrightarrow{c}$ [], то [s.x = A, e.y = e_2.2 <F > , e_2.2 ≠ []]
иначе, если e_1.1 $\xrightarrow{c}$ ('*') e_1.12, то
[{ s.x = '*', e.y = e_1.12 A e_2.2 <F ('*') e_1.12> }, e_2.2 ≠ []]
иначе корней нет.

Пример 2.

Уравнение: (e.x) e.y = [e.1 e.3 t.2 <F e.1> ,]
Ответ:
Если e.1 $\xrightarrow{c}$ [], то если e.3 $\xrightarrow{c}$ [], то если t.2 $\xrightarrow{c}$ (e.21), то
[{e.x = e.21, e.y = <F >} ,]
иначе корней нет;
иначе, если e.3 $\xrightarrow{c}$ (e.31) e.32, то
[{ e.x = e.31, e.y = e.32 t.2 <F > },]
иначе корней нет;
иначе, если e.1 $\xrightarrow{c}$ (e.11) e.12, то
[{ e.x = e.11, e.y = e.12 e.3 t.2 <F (e.11) e.12> },]
иначе корней нет.

Уравнение: (e.x) e.y = [e_1.1 e_2.3 t_3.2 <F e_1.1> ,]
Ответ:
Если e_1.1 $\xrightarrow{c}$ [], то если e_2.3 $\xrightarrow{c}$ [], то если
t_3.2 $\xrightarrow{c}$ ('***' e_3.21), то
[{e.x = '***' e_3.21, e.y = <F >} ,]
иначе корней нет;
иначе, если e_2.3 $\xrightarrow{c}$ ('**' e_2.31) e_2.32, то
[{ e.x = '**' e_2.31, e.y = e_2.32 t_3.2 <F > },]
иначе корней нет;
иначе, если e_1.1 $\xrightarrow{c}$ ('*' e_1.11) e_1.12, то

⟦ { e.x = '*' e_1.11, e.y = e_1.12 e_2.3 e_3.2 <F ('*' e_1.11) e_1.12> }, ⟧
иначе корней нет.

15.2.2. Уточнение свёртки

Подстановки параметров, сводящие вычисления одних конфигураций к вычислению других, в момент свёртки метадерева потенциальных вычислений (см. 5.1) должны быть согласованы с выбранной нами подтипизацией. При обобщении параметризованных выражений (см. 5.3.3) подтипы обобщённых термов есть минимумы (как числа, и подтип * соответствует бесконечности) подтипов обобщаемых выражений. Выходные и входные форматы компонент факторизации (см. 4.3, 9.1.2) уточняются согласно расширенному языку параметров.

15.3. Синтаксические мономы в задаче самоприменения

Рассмотрим на простейшей задаче самоприменения каким образом работает описанная нами подтипизация в паре с механизмом распознавания синтаксических мономов. Пример, который мы собираемся рассмотреть здесь, показывает, что без использования этих двух инструментов невозможно ожидать приемлемых по оптимальности свойств остаточных программ в столь притягательных задачах самоприменения [14, 35, 56, 57, 63].

Сформулируем нашу задачу на языке MST-схем:

```
<SCP4  ......    .......................     (              )>
     <IntLC     e.data     e.data1     (      )>  SourceInt
              <F  !       <G   !       > >  Prog
```

Мы рассматриваем специализацию интерпретатора Int по фиксированной программе Prog и интересуемся одной из её промежуточных конфигураций: функция IntLC просматривает свой первый аргумент (в свою очередь являющийся интерпретируемой конфигурацией), для того чтобы найти самый левый из самых внутренних вызовов функций, — с целью объявления его текущим активным вызовом для продолжения вычислений. И, как мы видим из схемы, вызов функции IntLC не обладает явно выраженной информацией о том, что в первом аргументе-конфигурации композиция состоит ровно из двух вызовов, и что второй, самый внутренний вызов G стоит на последней позиции в аргументе вызова F. Эта информация дана неявно, и должна быть обнаружена при просмотре композиции. Таким образом, необходим проход по всем конструкторам, составляющим значения параметров e_1.data и e_1.data1; т. е. цикл на всю глубину по структурным скобкам РЕФАЛа и по всей длине выражения на каждом скобочном уровне. И эти циклы по значениям e_1.data и e_1.data1 реализуют частичные тождественные функции, ибо они должны оставить безфункциональную [24] часть своих данных неизменной. Мы рассмотрели интерпретацию двух нижних строк схемы, — рассмот-

[24] Без вызовов функций.

рим их специализацию. С точки зрения SCP4 (т. е. в момент специализации), параметры e_1.data и e_1.data1 представляют собой *фиксированные, но неизвестные* данные, размер которых также неизвестен. Следовательно, оба указанных выше цикла останутся в соответствующих компонентах факторизации (см. главы 5, 7). С другой стороны, в О. Д. З. этих параметров не входят выражения вида (Call ...), которые кодируют синтаксические конструкторы вызовов функций, и, следовательно, ветви вычисления функции IntLC, обнаруживающие (отождествляющие) вызовы функций, будут отброшены прогонкой.

Определение функции IntLC:

```
IntLC {
...
... (Call ...) ... = ...;
/* После прогонки этого предложения не будет. */
...
}
```

По определению функции IntLC, только эти ветви передают управление другим функциям интерпретатора. Таким образом, вспоминая, что второй аргумент ((Prog)) функции IntLC есть константа, и, предполагая, что эта константа не будет обобщена свёрткой, мы получаем, что из определения IntLC будет построена компонента факторизации, являющаяся частичной тождественной функцией; более того, синтаксически тождественной функцией (см. раздел 9.2.1 и [52]). Следовательно, цикл, просматривающий значение параметра e_1.data, будет убран, оставив в месте своего вызова e_1.data. Аналогично исчезнет и цикл по e_1.data1. Наконец, отметим, что рассмотренная нами задача может оказаться и стартовой конфигурацией, если интерпретатор допускает произвольную композицию на входе, как, например, это делает суперкомпилятор. Любопытный читатель может сам построить ситуацию (или посмотреть в [52]), когда после прогонки останется синтаксический моном (см. 9.2.2) более общего вида.

15.4. Язык MST-схем

Язык MST-схем, введение в который мы дали на примерах в предыдущих разделах текущей главы, был предложен (и использовался) В. Ф. Турчиным как неформальный язык публикаций [57, 71–74], помогающий рассуждать о преобразованиях функциональных программ на разных метауровнях и взаимодействиях между этими метауровнями, по возможности, не вдаваясь в технические детали преобразуемых и преобразующих программ. В частности, синтаксический конструктор <...> вызова функции понимается как абстрактный конструктор аппликации, т. е. его конкретная интерпретация зависит от его уровня в MST-схеме (от высоты строки, где этот вызов находится, относительно самой нижней строки схемы). Описанная нами подтипизация параметров была реализована в суперкомпиляторе SCP3 [57, 74], как элемент внутреннего языка преобразований. В суперкомпиляторе SCP4 сделана первая попытка построить на базе идей Турчина формальный язык постановки задач

на специализацию и реализовать компилятор этого языка во внутренний язык конфигураций. Прежде чем описать текущее (реализованное) состояние языка MST-схем, мы напомним, что он является двумерным позиционным языком (т. е. символы пробела и переноса строки и т. п. являются значимыми — уточнение см. ниже), и поясним понятие, которое мы использовали в примерах без комментариев — это идентификаторы, обозначающие исходные тексты программ (например, `SourceInt`); эти идентификаторы есть формальные имена исходных текстов («макросы») — в реализованном языке они предваряются префиксом `%.` (в нашем примере `%.SourceInt`). По определению, все самые верхние строки задач на суперкомпиляцию совпадают с соответствующими строками из MST-схем **(I)** и **(II)**, и потому всегда опускаются, дабы не утруждать программиста.

Общая структура программы в языке MST-схем:

```
program ::= declaration* mst-scheme declaration* program | empty

mst-scheme ::= mst ; | multi-level-mst
declaration ::= basics | preparatory | user-gener
              | user-reduce | define


basics ::= $BASIC calls call;
calls ::= t.call,*
call ::= <function-name format>

preparatory ::= $PREPARATORY function-list;
function-list ::= function-names function-name
function-names ::= function-name,*

user-gener ::= $USER_GENER user-list;
user-reduce ::= $USER_REDUCE user-list;
user-list ::= user-fun-names user-fun-name
user-fun-names ::= user-function-name,*
user-function-name ::= function-name | EveryFunction__

define ::= $DEFINE macros-name definition ;
definition ::= { string } | <function-name string>

format ::= (expr) format | (expr)
expr   ::= (expr) expr | call expr | symbol expr
         | sequence-of-chars expr | variable expr | empty


sequence-of-chars ::= 'chars'
chars ::= char*
symbol ::= compound-symbol | char | number
compound-symbol ::= "chars" | indentifier

mst ::= mst-term*
```

```
mst-term ::= (mst) | <function-name mst>
                     /* Здесь function-name не совпадает ни
                        с одним из следующих имён:
                          Const__ ,UnConst__ ,Appl__ .
                     */
           | <Const__ obj-expr> | <UnConst__ obj-expr>
           | <Appl__ obj-expr> | symbol
           | sequence-of-chars | variable

/*****************************************************
    В нижеследующих определениях символы пробела blank и символы
переноса строки new-line являются значимыми символами.
*****************************************************/

multi-level-mst ::= end-mst-block | mst-block multi-level-mst
mst-block ::= mst-bl new-line | empty
end-mst-block ::= mst-bl ; end-line

mst-bl ::= up-mst-string up-mst-string* down-mst-string
up-mst-string ::= break -- end-line
down-mst-string ::= break end-line

end-line ::= comment new-line

break ::= (break | <break | break) | break> | ! break
        | symbol break | variable break | blank break | empty
```

/* Если мы формально припишем `mst-block` и `end-mst-block` (один после другого в указанном порядке), стирая знаки: `--` , `;` и `end-line`, то мы должны получить:

1) `mst` на каждой строке, где допускаются символы восклицательного знака;
2) непосредственно над каждым символом восклицательного знака должен быть:
   a) или другой символ восклицательного знака;
   b) или точка, разделяющая тип и имя переменной;
3) непосредственно над каждым символом, отличным от `blank` и восклицательного знака, должен быть `blank`, если этот символ не на самой верхней строке схемы `multi-level-mst`.

Горизонтальная табуляция представляет собой один невидимый символ, т. е. отлична (например) от восьми подряд идущих пробелов. */

```
/*****************************************************
Конец определений, где символы пробела и переноса строки являются значимыми символами.
*****************************************************/
```

```
obj-expr ::= obj-term*

variable ::= e-variable | s-variable | t-variable
t-variable ::= t.index
e-variable ::= e.index
s-variable ::= s.index

/* Нижеследующий тип переменных используется для макросов. Для
каждого макроса должно существовать определение ($DEFINE), которым
макрос и заменяется. */

%-variable ::= %.index
index ::= number | identifier
```

Внешние базисные понятия:

```
number ::= INTEGER
identifier ::= IDENTIFIER
char ::= CHARACTER | \escape-char | \xhh
escape-char ::= \ | " | ' | ( | ) | < | > | t | n | r
h ::= HEXITAL-DIGIT
function-name ::= IDENTIFIER
empty ::=          /* пустая строка - ничто */
new-line ::= NEW-LINE-ASCII-CHARACTER
blank ::= space | tab | dot
space ::= SPACE
tab   ::= HORIZONTAL-TABULATION
dot   ::= .
comment ::= STRING
file-name ::= PATH
macros-name ::= index
string ::= STRING
```

Фактически, `program` (см. выше) позволяет поставить несколько задач на специализацию. В этом случае задачи решаются последовательно в общем контексте базисных конфигураций (см. 5.1, 5.2), т. е. множество базисных конфигураций, построенное при решении предыдущих задач, рассматривается как стартовое значение множества базисных конфигураций текущей задачи. Параметры локальны по задачам. Практическая высота схем ограничена некоторой константой; если подтип-поднятие параметра больше этой константы, то он отождествляется с подтипом `*`.

Во второй альтернативе определения

```
definition ::= { string } | <function-name string>
```

внешняя, известная MST-компилятору, функция `function-name` исполняется до подстановки определяемого макроса и реально используется для кодировки своего аргумента.

Безусловно, язык MST-схем требует своего развития. Например, введение подтипа параметров, О. Д. З. которого совпадает с образом кодировки программ (а не данных), приведёт к улучшению эффективности остаточных программ.

Глава 16. **Несколько примеров преобразований**

Цель данной главы — показать дополнительные, к уже отмеченным выше, результаты преобразований программ, полученные автоматически актуальной версией суперкомпилятора SCP4 [58] (могут выглядеть слегка по-другому в других версиях SCP4). Мы начнём с простейших примеров и закончим специализацией самоописания подмножества РЕФАЛа.

16.1. Простейшие примеры

Все примеры этого раздела будут преобразовываться при ключе суперкомпиляции, который, грубо говоря, позволяет сохранять более точными образцы последних предложений в каждой функции [25] и задаётся псевдокомментарием (см. главу 18):

```
*$MATCHING ForRepeatedSpecialization;
```

Пример 1. Во входном цикле есть инвариант-предикат, который порождает накладные расходы.

```
$ENTRY Go { e.input = <F e.input>; }
F { A A A e.1 = A <F A e.1> ;
    A = ;
}
```

Выход: инвариантный предикат вынесен за тело цикла (см. разделы 3.3.2 и 4.3).

```
$ENTRY Go { A e.input = <F7 e.input>; }
F7 { A A e.1 = A <F7 e.1> ;
           = ;
}
```

Пример 2. Кроме выноса инварианта за тело цикла, показывает увеличение местности входной среды цикла (см. разделы 4.3 и 5.4.1).

```
$ENTRY Go { e.input = <F e.input>; }
F { s.x s.x e.1 = s.x  <F e.1> ;
    s.x         = ;
}
```

Выход: входной формат построенной функции F7 имеет местность, равную двум, что при более тонком механизме интерпретации может повысить эффективность (см. главу 13).

```
$ENTRY Go { s.x e.input = <F7 s.x e.input>; }
* InputFormat: <F7 s.x e.1>
F7 { s.x s.x s.y e.1 = s.x <F7 s.y e.1>;
     s.x             = ;
}
```

[25] И, следовательно, более точно описывается область определения преобразованной программы относительно преобразуемой программы.

Пример 3. Показывает распространение информации в метадереве через кратные вхождения параметров.

```
$ENTRY Go { e.input = <F e.input> e.input; }
F { A A e.1 = A <F e.1> ;
    A       = ;
}
```

Выход: если построенная компонента связности является частью другой подзадачи, то уточнение структуры в правой части за телом цикла будет использовано на последующих стадиях специализации (см. 10.4).

```
$ENTRY Go { A e.input = <F9 e.input> A e.input; }
F9 { A A e.1 = A <F9 e.1> ;
           = ;
}
```

Пример 4. Итерация. Во входном цикле есть инвариант-конструктор.

```
$ENTRY Go { e.input = <LispStyleReverse e.input A ()>; }
LispStyleReverse {
 s.x e.1 (e.res) = <LispStyleReverse e.1 (s.x e.res)>;
         (e.res) = e.res;
}
```

Выход: инвариант-конструктор вынесен за тело цикла и может быть использован при последующей специализации данной компоненты; например, в контексте композиции вызовов функций (см. 9.1.2 и 10.4).

```
$ENTRY Go { e.input  = A <F12 e.input ()>; }
* InputFormat: <F12 e.1 (e.2)>
F12 { s.x e.1 (e.2) = <F12 e.1 (s.x e.2)>;
              (e.2) =  e.2; }
```

Пример 5. Рекурсия. Во входном цикле есть инвариант-конструктор.

```
$ENTRY Go { s.y e.input = <RefalStyleReverse s.y e.input>; }
RefalStyleReverse {
 s.x e.1 = <RefalStyleReverse e.1> s.x;
         = ;
}
```

Выход: показывает вынос параметра за тело цикла. См. пояснения к предыдущему примеру.

```
$ENTRY Go { s.y e.input  = <F10 e.input> s.y; }
F10 {
 s.x e.1 = <F10 e.1> s.x ;
         = ;
}
```

Пример 6. Показывает вычисления «более чем ленивые» во время суперкомпиляции.

```
$ENTRY Fa { e.x = <Fba <Fab e.x>>; }
Fab {
A e.S = B <Fab e.S>;
      = ;
}
Fba {
B e.S = A <Fba e.S>;
      = ;
}
```

Выход: тривиален, область определения функции расширена.

```
$ENTRY Fa { e.41  = e.41 ; }
```

Пример 7. Представляет определение предиката pal — входная строка есть палиндром. Функция RefalStyleReverse определена в примере 5.

```
$ENTRY Go { e.ls = <pal e.ls <RefalStyleReverse e.ls>>; }
pal {
     = True;
 s.x = True;
 s.x e.ls s.x = <pal e.ls>;
 s.x e.ls s.y = False;
}
```

Выход: специализируя по контексту вызова предиката pal, получаем тавтологию на области определения. То есть доказано простое утверждение об аргументе pal. Заметим, что в построенной программе нет цикла, вычисляющего константу, который присутствует во входной программе (см. раздел 9.1.3).

```
* InputFormat: <Go e.41>
$ENTRY Go {
 s.101 e.41  = True;
             = True;
}
```

Пример 8 (популярный в работах по специализации программ). Демонстрирует преобразование наивного поиска данной подстроки в строке в алгоритм Кнута—Морриса—Пратта (далее КМП) [19, 30, 44].

```
$ENTRY Go { e.string = <Search  ('abcabcacab') e.string>; }
Search {
(s.a e.1) s.a e.2
                = <Look (s.a e.1) s.a e.2 ((s.a e.1) e.2)>;
(s.a e.1) s.b e.2 = <Search (s.a e.1) e.2>;
(s.a e.1)         = False ;
}

Look {
(s.a e.1) s.a e.2 (e.3) = <Look (e.1) e.2 (e.3)>;
(s.a e.1) s.b e.2 (e.3) = <Search e.3>;
```

```
(s.a e.1)          (e.3) = <Search e.3>;
(       )      e.2 (e.3) = True ;
}
```

Выход: первая стадия алгоритма КМП отработала во время специализации, таблица переходов по строке в построенной программе представлена в виде функций. И в этом смысле остаточный алгоритм даже более эффективен, чем КМП. С другой стороны, остаточная программа, представленная на РЕФАЛе, не отражает структуру дерева разбора искомой подстроки. Это дерево не потеряно в языке РЕФАЛ-графов (см. разделы 3.3.2 и 13). В построении данного результата важную роль играет работа с «отрицательной» информацией при описании параметрических множеств данных (см. разделы 2.1, 4, 5.3.4).

```
$ENTRY Go { e.1 = <F6 e.1>; }
F6 {
 'a'   e.1 = <F12 e.1>;
 s.101 e.1 = <F6 e.1>;
           = False;
}

F12 {
 'bcab' e.1 = <F32 e.1>;
 'bcaa' e.1 = <F12 e.1>;
 'bca' s.105 e.1 = <F6 e.1>;
 'bca'  = False;
 'bc' s.104 e.1 = <F6 e.1>;
 'bc'  = False;
 'ba' e.1 = <F12 e.1>;
 'b' s.103 e.1 = <F6 e.1>;
 'b' = False;
 'a' e.1 = <F12 e.1>;
 s.102 e.1 = <F6 e.1>;
   = False ;
}

F32 {
 'cacab' e.1 = True;
 'cacaa' e.1 = <F12 e.1>;
 'caca' s.110 e.1 = <F6 e.1>;
 'caca' = False;
 'cac' s.109 e.1 = <F6 e.1>;
 'cac' = False;
 'cab' e.1 = <F32 e.1>;
 'caa' e.1 = <F12 e.1>;
 'ca' s.108 e.1 = <F6 e.1>;
 'ca' = False;
 'c' s.107 e.1 = <F6 e.1>;
 'c' = False;
```

```
 'a' e.1 = <F12 e.1>;
 s.106 e.1 = <F6 e.1>;
  = False;
}
```

Пример 9. Квадрат унарного натурального числа. Функция `Times` определена в разделе 5.3.1.1, пример 1. Рекурсия.

```
$ENTRY Sq { e.numb = <Times (e.numb) (e.numb)>; }
```

Выход: ленивое вычисление во время суперкомпиляции (6.2) позволяет преобразовать рекурсию в хвостовую рекурсию (итерацию).

```
* InputFormat: <Sq e.numb>
$ENTRY Sq {
 e.numb  = <F32 (e.numb) e.numb> ;
}
* InputFormat: <F51 (e.112) (e.113) e.114>
F51 {
 (e.112) (e.113)  = <F32 (e.112) e.113>;
 (e.112) (e.113) I e.114 = I <F51 (e.112) (e.113) e.114>;
}
* InputFormat: <F32 (e.108) e.109>
F32 {
 (e.108)  = ;
 (e.108) I e.109 = <F51 (e.108) (e.109) e.108>;
}
```

Пример 10 (А. В. Корлюков [7, 8, 45]). Представляет классическую попытку использования специализатора в качестве компилятора из некоторого языка L в объектный язык специализатора, в нашем случае РЕФАЛ (см. раздел 15.1 и [26, 28, 30, 35, 45, 56]). Пусть L будет языком машины Тьюринга (MT). Память машины MT, называемая лентой, есть бесконечная в обе стороны цепочка ячеек. Программа для MT — конечная последовательность инструкций вида:

`CurrState CurrSymb NextSymb NextState Movement`

Согласно программе, MT передвигает головку из текущей ячейки в одну из двух её ближайших соседок, содержимое текущей ячейки и состояние MT изменяются. Выделены начальное состояние `start` и конечное `stop`. Входом MT является конечная последовательность символов на ленте (по одному символу в ячейке), оставшаяся часть ленты заполнена пробелами (мы будем обозначать их буквами **B**).

Рассмотрим пример программы для MT. Программа `DoublePQ` заменяет данную последовательность символов **P** последовательностью символов **Q**, в два раза более длинной, чем входная. Например, если в начале работы MT на ленте написаны 10 подряд идущих символов **P** и головка указывает на первый из них, тогда после остановки MT на ленте будут написаны 20 подряд идущих символов **Q**.

Программа DoublePQ

CurrState	start	start	start	moveleft	moveleft
CurrSymb	B	Q	P	Q	B
NextSymb	B	Q	Q	Q	Q
NextState	stop	start	moveleft	moveleft	start
Movement	right	right	left	left	right

Проспециализируем нижеследующий интерпретатор машины Тьюринга Turing по программе DoublePQ, оставляя ленту MT в общем положении. Лента разделена ячейкой, указанной головкой, на левую и правую части, машина начинает работу в стартовом состоянии.

Interpreter of the Turing Machine written in Refal

```
*$MATCHING ForRepeatedSpecialization;
* Call for a concrete Turing machine (a program).
$ENTRY Go {
 (e.LeftTape) (s.CurrSymb) (e.RightTape) =
   <Turing (
    (start      B  B   stop       right)
    (start      Q  Q   start      right)
    (start      P  Q   moveleft   left )
    (moveleft   Q  Q   moveleft   left )
    (moveleft   B  Q   start      right) )
    (start)(e.LeftTape)(s.CurrSymb)(e.RightTape)>;
 }

* The interpreter itself.
* <Turing (e.Program) (s.CurrState)(e.LeftPartOfTape)
*                     (s.CurrSymb)(e.RightPartOfTape)>
Turing {
 (e.instr)(stop)(e.left)(s.symbol)(e.right)
       = (e.left)(s.symbol)(e.right);

 (e.instr)(s.q)(e.left)(s.symbol)(e.right)
       = <Turing (e.instr) <Turing1
                            <Search (s.q s.symbol)(e.instr)>
                             (e.left)(s.symbol)(e.right)> >;
 }
Turing1 {
 (s.c s.r left)(e.left s.a)(s.symbol)(e.right)
                 = (s.r)(e.left)(s.a)(s.c e.right);
```

```
 (s.c s.r right)(e.left)(s.symbol)(s.a e.right)
                  = (s.r)(e.left s.c)(s.a)(e.right);
 }
Search {
 (s.key1 s.key2)((s.key1 s.key2 e.value) e.table) = (e.value);
 (s.key1 s.key2)((e.row) e.table)
                         = <Search (s.key1 s.key2)(e.table)>;
 }
```

Таким образом, входная точка `Go` определяет функцию `DoublePQ` в терминах РЕФАЛа и с большими накладными расходами интерпретации.

В результате суперкомпиляции получаем:

The residual program `DoublePQ` (in Refal)

```
* InputFormat:  <Go (e.LeftTape)(s.CurrSymb)(e.RightTape)>
$ENTRY Go {
 (e.Left) (s.Symbol) (e.Right)  = <F6 (e.Left) s.Symbol e.Right>;
}

F27 {
(e.1 s.2) (e.4)       Q = <F27 (e.1) (Q e.4) s.2>;
(e.1)       (s.3 e.4) B = <F6 (e.1 Q Q) s.3 e.4>;
}

F6 {
(e.1)      B s.3 e.4 = (e.1 B) (s.3) (e.4);
(e.1)      Q s.3 e.4 = <F6 (e.1 Q) s.3 e.4>;
(e.1 s.2)  P     e.4 = <F27 (e.1) (e.4) s.2>;
}
```

Функция `F6` соответствует состоянию MT `start`, функция `F27` — состоянию `moveleft`. Каждое предложение соответствует одной инструкции машины MT. Число шагов РЕФАЛ-машины равно числу шагов MT в исходном определении. Остаточная программа не содержит синтаксических терминов, описывающих программу в языке MT (`start`, `moveleft`, `stop`, `right`, `left`). Термины **P**, **Q**, **B** представляют данные. Следовательно, произошла полная (эффективная) компиляция программы `DoublePQ` из языка MT в РЕФАЛ. Построенная программа итерационна (хвостовая рекурсия): в данном случае это отражает тот факт, что длина стека функций исходной программы равномерно ограничена по входным данным (не больше трёх). Накладные расходы на поиск данной инструкции в программе убраны благодаря условию упрощающего отношения. (См. разделы 5.3.1.1, 5.3.1.2, 5.3.6.)

Пример 11. Аналогично предыдущему примеру здесь мы будем специализировать интерпретатор `IntRegStart` регистровых машин (PM) по конкретным программам. Каждая PM [9] имеет фиксированное число регистров для хранения любых натуральных чисел. Арифметическое устройство машины умеет складывать, умножать и вычислять характеристическую функцию равенства χ содержимых двух регистров (χ`(x,y) = (x==y)?1:0`). Программа

PM является последовательностью команд-операторов, перенумерованных в порядке возрастания. Каждая команда представляет собой: команду присваивания любому регистру результата работы арифметического устройства над любыми двумя регистрами; команду перехода GoTo к команде с номерами `address-of-reg-then` и `address-of-reg-else` в зависимости от того, равно или нет единице содержимое её регистра-аргумента; команду остановки Stop. Результат работы машины находится в первом регистре, а исходные данные в остальных регистрах. Перед началом работы машины в первом регистре указан номер последнего из регистров, содержащих входные данные. Оставшиеся регистры содержат нули. Каждая команда GoTo, встречающаяся в программе, содержит только номера команд, имеющиеся в этой программе. Ниже, во входных точках интерпретатора IntRegStart, определены функции факториала и умножения на языке *PM*. Определим интерпретатор IntRegStart:

```
/*
operator ::= add | mult | equal | goto | stop
add ::= Add (reg-arg1) (reg-arg2) (reg-result)
mult ::= Mult (reg-arg1) (reg-arg2) (reg-result)
equal ::= Eq (reg-arg1) (reg-arg2) (reg-result)
goto ::= GoTo (reg-if) (address-of-reg-then) (address-of-reg-else)
stop ::= Stop
*/
*$MATCHING ForRepeatedSpecialization;
$EXTERN Add, Mul;
/*
 <IntRegStart
   (Prog (P1 e.operator1) ... (PM e.operatorM))
         (Registers (A1 s.K-last-input-address)(A2 e.input2)...
                    (AK e.inputK)(AK1 e.zero)...(AN e.zero)
         )
 > ==> (Registers (A1 e.result) (A2 e.trash) ... (AN e.trashN));
*/
IntRegStart {
 t.Prog t.Regs = <Result <IntReg t.Prog t.Regs t.Prog>>;
}
Result { (Registers (A1 e.value) e.regs) = e.value; }

IntReg {
(Prog (s.label Stop) e.ops) t.Regs t.Prog = t.Regs;
(Prog (s.label GoTo t.if t.then t.else) e.ops) t.Regs t.Prog
   = <IntReg (Prog <GoTo (GoTo t.if t.then t.else) t.Regs t.Prog>)
             t.Regs t.Prog
     >;
(Prog t.op e.ops) t.Regs t.Prog
   = <IntReg (Prog e.ops) (Regs <Operator t.op t.Regs>) t.Prog>;
}
```

```
GoTo {
 (GoTo t.if t.then t.else) t.Regs t.Prog
           = <If (<LookFor t.if t.Regs>) t.then t.else t.Prog>;
}
If {
 (1)   t.then t.else t.Prog = <Search t.then t.Prog>;
 (e.0) t.then t.else t.Prog = <Search t.else t.Prog>;
}
Search {
 (s.label) (Prog (s.label e.op) e.prog)
                   = (s.label e.op) e.prog;
 (s.label) (Prog (s.label1 e.op) e.prog)
                   = <Search (s.label) (Prog e.prog)>;
}
LookFor {
 (s.address) (Registers (s.address e.value) e.regs) = e.value;
 (s.address) (Registers (s.address1 e.value) e.regs)
                   = <LookFor (s.address) (Registers e.regs)>;
}
Operator {
 (s.label s.name t.arg1 t.arg2 t.res) t.Regs
   = <PutIn (<Op s.name (<LookFor t.arg1 t.Regs>)
                        (<LookFor t.arg2 t.Regs>)>) t.res t.Regs>;
}
PutIn {
 (e.what) (s.where) (Registers (s.where e.value) e.regs)
      = (s.where e.what) e.regs;
 (e.what) (s.where) (Registers (s.address e.value) e.regs)
      = (s.address e.value)
        <PutIn (e.what) (s.where) (Registers e.regs)>;
}
Op {
 Add  (e.arg1) (e.arg2) = <Add (e.arg1) e.arg2>;
 Mult (e.arg1) (e.arg2) = <Mul (e.arg1) e.arg2>;
 Eq   (e.arg1) (e.arg2) = <Eq (e.arg1) (e.arg2)>;
}
Eq {
 (s.x) (s.x) = 1;
 (e.x) (e.y) = 0;
}
```

Задача А. Проспециализируем наш интерпретатор PM по программе, вычисляющей факториал. Соответствующие ограничения на входную точку интерпретатора PM даны ниже. Суперкомпиляция будет происходить при аппликативной стратегии развития стека функций (см. 6.2), что указано псевдокомментарием (см. главу 18).

```
*$STRATEGY Applicative;
$ENTRY Factorial {
 s.n
  = <IntRegStart
      (Prog (P1 Eq (A1)(A1) (A1))   /* A1=1 */
            (P2 Eq (A1)(A1) (A3))   /* A3=1 */
            (P3 Eq (A1)(A1) (A5))   /* A5=1 */
            (P4 Mult (A3)(A1) (A1)) /* A1 *= A3 */
            (P5 Eq   (A2)(A3) (A4)) /* A4 = (A2==A3)?1:0 */
            (P6 Add  (A3)(A5) (A3)) /* A3 ++ */
            (P7 GoTo (A4)(P8) (P4))
            (P8 Stop)
      )
      (Registers (A1 A2) (A2 s.n) (A3 0) (A4 0) (A5 0))
    >;
}
```

Выход: суперкомпилятор SCP4 в данном случае отработал как очень эффективный компилятор из языка PM в РЕФАЛ.

```
$EXTERN Add, Mul;

* InputFormat: <Factorial s.n>
$ENTRY Factorial { s.n = <F244 s.n (1) 1>; }

* InputFormat: <F244 s.111 (e.112) e.113>
F244 {
 s.111 (s.111) e.113, <Add (s.111) 1>:e.117 = e.113;
 s.111 (e.112) e.113, <Add (e.112) 1>:e.118
           = <F244 s.111 (e.118) <Mul (e.118) e.113>>;
}
```

Единственная погрешность в построенной программе: лишний вызов функции Add при выходе из рекурсии. Значение вызова функции Add во втором предложении функции F244 используется дважды, т. е. он не перевычисляется. Функция Factorial определена посредством итерации.

Задача Б. Специализация интерпретатора PM по программе, определяющей умножение через сложение. Входная точка интерпретатора PM дана ниже. Аппликативная стратегия развития стека функций:

```
*$STRATEGY Applicative;
$ENTRY Multiplication {
 s.n s.m
   = <IntRegStart
       (Prog (P1 Eq (A1)(A1) (A1))  /* A1=1 */
             (P2 Eq (A1)(A1) (A4))  /* A4=1 */
             (P3 Eq (A1)(A1) (A5))  /* A5=1 */
             (P4 Add (A1)(A1) (A1)) /* A1=2 */
```

```
            (P5 Eq  (A1)(A4) (A1)) /* A1=0 */
            (P6 Add (A3)(A1) (A1)) /* A1 += A3 */
            (P7 Eq  (A2)(A4) (A6)) /* A6 = (A2==A4)?1:0 */
            (P8 Add (A4)(A5) (A4)) /* A4 ++ */
            (P9 GoTo (A6) (P10)(P6))
            (P10 Stop)
      )
     (Registers (A1 A3)(A2 s.n)(A3 s.m)(A4 0)(A5 0)(A6 0))
    >; }
```

Выход: итеративное определение умножения в чисто рефальских терминах.

```
$EXTERN Add;

* InputFormat: <Multiplication e.nm>
$ENTRY Multiplication {
  s.n s.m  = <F299 s.n (1) (s.m) s.m>;
}

* InputFormat: <F299 s.114 (e.115) (e.116) s.118>
F299 {
 s.114 (s.114) (e.116) s.118,
       <Add (s.114) 1>:e.121 = e.116;
 s.114 (e.115) (e.116) s.118
       = <F299 s.114 (<Add (e.115) 1>)
                     (<Add (s.118) e.116>) s.118
         >;
}
```

Как и в примере с машиной Тьюринга, длина стека функций интерпретатора `IntRegStart` равномерно ограничена по входным данным (в данном случае не больше пяти), если не учитывать стека развёртки внешних функций `Add` и `Mult`. Как следствие, — на выходе имеем хвостовую рекурсию по модулю обращений к внешней функции `Add`. Функции поиска `Search`, `LookFor`, `PutIn` полностью вычислены во время специализации, так как выбор веток в них определяется исключительно интерпретируемой программой, но не входными данными этой программы, и условие упрощающего отношения обеспечило нужное число развёрток (см. 5.3.1.2, 5.3.6). В процессе преобразований произошло ровно два обобщения без перестройки стека функций: входная точка `<F299 s.n (1) (s.m) s.m>`[26] в цикл первым обобщением была обобщена до формата `<F299 s.1 (s.2) (e.3) s.4>`, и вторым обобщением до `<F299 s.114 (e.115) (e.116) s.118>` (см. 5.3.3). Ветвление в остаточной программе полностью соответствует ветвлению функции `Eq`, которая и тестирует данные интерпретируемой программы.

[26] Реально здесь стоит композиция вызовов функций исходной программы. Суперкомпилятор переименовал эту композицию в `F299`.

16.2. Специализация самоописания РЕФАЛа

Рассмотрим семантическое[27] самоописание следующего подмножества РЕФАЛа:

```
Program ::= $ENTRY definition+
definition ::= function-name { sentence;+ }
sentence ::= pattern = expr
expr ::= empty | term expr| | function-call expr|
function-call ::= <function-name expr>
pattern ::= empty | term pattern|
term ::= SYMBOL | var | (expr)
var ::= e.name | t.name | s.name
empty ::=
```

с двумя дополнительными ограничениями: (1) первое есть естественное условие, исходящее из полного РЕФАЛа, — множество переменных правой части любого предложения является подмножеством переменных левой части этого предложения; (2) каждый образец содержит не более одной e-переменной на каждом скобочном уровне (например, образец `(e.1) e.2 (e.3)` допускается, а образец `(e.1 A e.2) e.3` нет).

В нашем интерпретаторе мы используем для представления программ в виде данных кодировку, описанную в главе 15, уточнив её на случай предложения:

pattern = expr; = ((pattern) '='(expr)).

```
*$STRATEGY Applicative;
*$MATCHING ForRepeatedSpecialization;

$ENTRY GoInt {
  t.Program e.data = <Interpreter (Call Go e.data) t.Program>;
}
Interpreter {
  (Call s.F e.d) t.P = <Eval <EvalCall s.F (e.d) t.P> t.P>;
}

* <Eval (e.env) (e.expr) t.Program> => e.data
Eval {
 (e.env) ((Call s.F e.expr1) e.expr) t.P
    = <Eval <EvalCall s.F (<Eval (e.env) (e.expr1) t.P>) t.P> t.P>
      <Eval (e.env) (e.expr) t.P>;

 (e.env) ((Var e.var) e.expr) t.P
    = <Subst (e.env) (Var e.var)> <Eval (e.env) (e.expr) t.P>;
```

[27] То есть операционное.

```
 (e.env) (('*' e.expr1) e.expr) t.P
    = ('*' <Eval (e.env) (e.expr1) t.P>)
      <Eval (e.env) (e.expr) t.P>;

 (e.env) (s.x e.expr) t.P = s.x <Eval (e.env) (e.expr) t.P>;
 (e.env) () t.P = ;
}
EvalCall {
s.F (e.d) t.P
 = <Matching False (((False)'='(False)) <LookFor s.F t.P>) (e.d)>;
}
* <Matching t.boolen (e.pattern)'='(e.data)> => (e.env) (e.expr)
* , where t.boolen ::= (e.env) | False
Matching {
 False (t.sent ((e.p)'='(e.expr)) e.def) (e.d) =
                  = <Matching <RigitMatch (e.p)'='(e.d) ()>
                              (((e.p)'='(e.expr)) e.def) (e.d)
                    >;
 (e.env) (((e.p)'='(e.expr)) e.def) (e.d) = (e.env) (e.expr);
}

* <RigitMatch (e.patt)'='(e.data) (e.env)> => (e.env1) | False
RigitMatch {
 ((Var 'e' s.n))'='(e.s) (e.env)
      = <RigitMatch ()'='() <PutVar ((Var 'e' s.n) e.s) (e.env)>>;
 ((Var 's' s.n) e.p)'='(s.1 e.s) (e.env)
      = <RigitMatch (e.p)'='(e.s)
                    <PutVar ((Var 's' s.n) s.1) (e.env)>>;
 ((Var 't' s.n) e.p)'='(t.1 e.s) (e.env)
      = <RigitMatch (e.p)'='(e.s)
                    <PutVar ((Var 't' s.n) t.1) (e.env)>>;
 (('*' e.p1) e.p)'='(('*' e.1) e.s) (e.env)
      = <RigitMatch (e.p)'='(e.s)
                    <RigitMatch (e.p1)'='(e.1) (e.env)>>;
 (s.1 e.p)'='(s.1 e.s) (e.env)
      = <RigitMatch (e.p)'='(e.s) (e.env)>;

 ((Var 'e' s.n) e.p (Var 's' s.n1))'='(e.s s.1) (e.env)
      = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s)
                   <PutVar ((Var 's' s.n1) s.1) (e.env)>>;
 ((Var 'e' s.n) e.p (Var 't' s.n1))'='(e.s t.1) (e.env)
      = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s)
                   <PutVar ((Var 't' s.n1) t.1) (e.env)>>;
 ((Var 'e' s.n) e.p ('*' e.p1))'='(e.s ('*' e.1)) (e.env)
      = <RigitMatch (e.p1)'='(e.1)
                <RigitMatch ((Var 'e' s.n) e.p)'='(e.s) (e.env)>>;
```

```
 ((Var 'e' s.n) e.p s.1)'='(e.s s.1) (e.env)
      = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s) (e.env)>;

 ()'='() (e.env) = (e.env);
 (e.p)'='(e.s) e.False = False;
}

* <PutVar (e.assignment) (e.env) > => t.boolean
PutVar { t.a (e.env) = <CheckPut <PutVar1 t.a (e.env)>>; }
PutVar1 {
((Var s.t s.n) e.val) (((Var s.t s.n) e.val1) e.env)
             = ((Var s.t s.n) e.val1) e.env <Eq (e.val) (e.val1)>;
t.assign (t.assign1 e.env) = t.assign1 <PutVar1 t.assign (e.env)>;
t.assign () = t.assign True;
}

CheckPut {
 e.env True = (e.env);
 e.trash False = False;
}

* <Eq (e.expression1) (e.expression2)> => s.boolean
* , where s.boolean ::= True | False
Eq {
 (s.1 e.xpr1) (s.1 e.xpr2) = <Eq (e.xpr1) (e.xpr2)>;
 (('*' e.1) e.xpr1) (('*' e.2) e.xpr2)
                          = <Eq (e.1 e.xpr1) (e.2 e.xpr2)>;
 () () = True;
 (e.xpr1) (e.xpr2) = False;
}

* <LookFor s.Function-name (e.Program)> => e.def
LookFor {
 s.F ((s.F e.def) e.P) = e.def;
 s.F ((s.F1 e.def) e.P) = <LookFor s.F (e.P)>; }

* <Subst (e.environment) t.variable> => e.data
Subst {
 (((Var s.t s.n) e.val) e.env) (Var s.t s.n) = e.val;
 (t.assign e.env) t.var = <Subst (e.env) t.var>;
}
```

Будем специализировать `Interpreter` по конкретным программам, уже рассмотренным выше в нашей работе. Если не оговорено обратное, наши примеры суперкомпилируется в условиях аппликативной стратегии развития стека функций, что мы указали псевдокомментарием в первой строке программы (см. 6.2 и 18). Второй псевдокомментарий полезен, так как мы намерены

повторно суперкомпилировать некоторые остаточные программы (см. главу 18). К объектным программам мы будем относиться как к определениям $DEFINE, давая им имена %.name (см. язык MST-схем 15.4). Постановки задач будут достаточно простыми и, там где это возможно, мы будем для краткости кодировку обозначать подчёркиванием, а не переносом на отдельную строку. Фактически, большинство наших задач будут более общими, чем при использовании специализатора как компилятора[28]. В разделе 15.1 мы подчёркивали интерес к подобным задачам; специализацию самоописания подмножества РЕФАЛа можно рассматривать как шаг к самоприменению нашего суперкомпилятора (в том или ином виде), в книге Нила Джонса [42] рассматриваются другие аспекты специализации самоописаний.

Пример 1. SCP4 на входе получает параметризованную конфигурацию <GoInt %.Go e.data>, где Go определена в примере 1 раздела 16.1.

Выход: SCP4 построил ту же программу[29], как и при прямой суперкомпиляции функции Go. Таким образом, не только полностью убраны расходы на интерпретацию, но и инвариантный предикат вынесен за тело цикла: остаточная программа эффективней интерпретируемой.

```
$ENTRY GoInt {
 A e.data = <F30 e.data>;
}
* InputFormat: <F30 e.41>
F30 {
 A A e.41 = A <F30 e.41>;
          = ;
}
```

Пример 2. Мы ставим задачу <GoInt %.Go e.data> для SCP4, где Go определена в примере 2 раздела 16.1.

Выход: чист от накладных интерпретационных расходов, но дополнительных преобразований не произошло (см. пример 2 раздела 16.1).

```
$ENTRY GoInt {
 e.data = <F27 e.data>;
}
* InputFormat: <F27 e.41>
F27 {
 s.107 s.107 e.41 = s.107 <F27 e.41>;
 s.104 = ;
}
```

Пример 3. Задача <GoInt %.Go e.data>, где Go определена в примере 3 раздела 16.1.

---

[28] В данном случае из РЕФАЛа в РЕФАЛ, т. е. оптимизатора.

[29] С точностью до имён функций и переменных, которые мы, впрочем, иногда изменяем для лучшей читаемости.

Выход: аналогичен предыдущему примеру. Повторная суперкомпиляция обоих этих примеров, конечно, даёт преобразования, получаемые прямой суперкомпиляцией функций Go.

```
$ENTRY GoInt {
 e.data = <F27 e.data> e.data;
}
* InputFormat: <F27 e.41>
F27 {
 A A e.41 = A <F27 e.41>;
 A = ;
}
```

Пример 4. Задача <GoInt %.Go e.data>, где Go определена в примере 4 раздела 16.1.

Выход: совпадает, с точностью до перестановки форматных переменных, с остаточной программой после прямой суперкомпиляции функции Go.

```
$ENTRY GoInt {
 e.data  = A <F81 () e.data>;
}
* InputFormat: <F81 (e.113) e.114>
F81 {
 (e.113) s.118 e.114  = <F81 (s.118 e.113) e.114>;
 (e.113) = e.113;
}
```

Пример 5. Задача <GoInt %.Go e.data>, где Go определена в примере 5 раздела 16.1.

Выход: комментарии совпадают с данными к предыдущему примеру.

```
$ENTRY GoInt {
 s.y e.data = <F60 e.data> s.y;
}
* InputFormat: <F60 e.41>
F60 {
 s.109 e.41 = <F60 e.41> s.109;
  = ;
}
```

Пример 6. Задача <GoInt %.Fa e.data>, где Fa определена в примере 6 раздела 16.1 и есть композиция двух рекурсий, оформленная в функциональном стиле (внутренний цикл не является итерацией, хотя и является хвостовой рекурсией).

Выход при *аппликативной стратегии развития стека функций*: мы не можем ожидать преобразования данной в примере композиции в один цикл, так как сам `Interpreter` также реализует аппликативную операционную семантику [30].

[30] При ленивой операционной семантике интерпретатора естественно ожидать подобных преобразований (см., например, работы [56, 57]).

```
$ENTRY GoInt {
 e.data = <F50 (e.data) ()>;
}
* InputFormat: <F87 (e.109) e.112>
F87 {
 (B e.109) e.112 = A <F87 () e.109 e.112>;
 () B e.112 = A <F87 () e.112>;
 () = ;
}
* InputFormat: <F50 (e.105) (e.109) e.112>
F50 {
 (A e.105) (e.109) e.112 = <F50 (e.105) (e.109 B) e.112>;
 () (e.109) e.112 = <F87 (e.109) e.112>;
}
```

Внутренний цикл оформился в итерационном стиле[31]. Такое оформление внутреннего цикла есть следствие того, что стек функций полностью развёрнут[32] (при рассматриваемой стратегии) и внешний вызов функции стартовой композиции Fbc не является абсолютным (неизменяемым) контекстом[33] вычислений-развёрток вызова Fab (во время суперкомпиляции, при рассматриваемой стратегии), а аккумулирует результат развёрток вызова Fab. И эти свойства вычислений наследует композиция в первом предложении интерпретирующей функции Eval, отображающая стек функций интерпретируемой программы в стек функций интерпретирующей программы, и далее эти свойства наследуются остаточной программой[34]. В построенной программе видны следы интерпретационных накладных расходов (лишняя форматная переменная e.112), хотя терминология Interpreter-а отсутствует. Повторная суперкомпиляция устраняет этот недостаток:

```
$ENTRY GoInt {
 A A e.data = A A <F18 (e.data)>;
 A = A;
  = ;
}
* InputFormat: <F32 e.103>
F32 {
 B e.103 = A <F32 e.103>;
  = ;
}
```

---

[31] То есть в выражениях — в правых частях предложений нет вложенных вызовов функций (синтаксических композиций) и нет конструкторов с аргументами — вызовами функций (см. также раздел 14.2).

[32] В аргументах вызовов функций синтаксически нет других вызовов функций, но есть их результаты (семантические значения) (см. раздел 6.2).

[33] См. раздел 5.3.1.1.

[34] О наследовании свойств интерпретаторов остаточными программами при специализации этих интерпретаторов по интерпретируемым программам смотри статью Т. Могенсена [50].

```
* InputFormat: <F18 (e.102) e.103>
F18 {
 (A e.102) e.103 = <F18 (e.102) e.103 B>;
 () e.103 = <F32 e.103>;
}
```

Выход при *ленивой стратегии развития стека функций*: также не идеален. Произошло только воспроизведение интерпретируемой программы — результатов ленивых вычислений в ней мы не наблюдаем. Структура остаточной программы композиционно функциональна. Функции S1 и S2 являются технической деталью транслятора из языка РЕФАЛ-графов в РЕФАЛ: их нет на уровне РЕФАЛ-графов — это только ветвления, построенные посредством дополнительной развёртки, следы которой мы видим и в случае аппликативной стратегии.

```
$ENTRY GoInt {
 e.data = <S1 e.data>;
}
* InputFormat: <F204 e.152>
F204 {
 B e.152 = A <F204 e.152>;
  = ;
}
* InputFormat: <F130 e.41>
F130 {
 A e.41 = B <F130 e.41>;
  = ;
}
* InputFormat: <S1 e.41>
S1 {
 A e.41 = A <S2 <F130 e.41>>;
  = ;
}
* InputFormat: <S2 e.152>
S2 {
 B e.152 = A <F204 e.152>;
 e.152 = ;
}
```

В данном случае стек функций в конфигурациях развёрнут только частично — часть вызовов не участвовали в вычислениях, и потому являются синтаксическими аргументами других вызовов функций. Обобщение одного из таких аргументов (см. 5.3.3)

```
<Eval ((e.S e.data)) (<Fab e.S>) %.Fa>
```

до выражения `e.generalized`, а не выделение контекста (см. 5.3.1.1), привело к интерпретации вызова `<Fab e.data>` в автономном цикле, без передачи его частично вычисленных значений последующей рекурсии, и построению композиционной синтаксической структуры в остаточной программе.

Повторная суперкомпиляция, естественно, даёт ожидаемый результат:

```
$ENTRY GoInt {
 A e.data = A e.data;
  = ;
}
```

Пример 7. Задача <GoInt %.Go e.data>, где Go определена в примере 8 раздела 16.1.

Выход: терминология Interpreter-а отсутствует в остаточной программе. Число функций в построенной программе больше, чем в результате прямого преобразования функции Go (см. пример 8 раздела 16.1), и они имеют нетривиальные входные форматы: для достижения результата прямого преобразования произошло недостаточное число развёрток — часть ветвлений оформилась в виде вызовов функций.

```
$ENTRY GoInt {
 e.data = <F50 e.data>;
}
* InputFormat: <F372 (e.41) s.135>
F372 {
 ('cacab' e.41) 'b' = True;
 ('caca' s.179 e.41) 'b' = <F180 (e.41) s.179>;
 ('caca') 'b' = False;
 ('cac' s.169 e.41) 'b' = <F55 (e.41) s.169>;
 ('cac') 'b' = False;
 ('ca' s.160 e.41) 'b' = <F372 (e.41) s.160>;
 ('ca') 'b' = False;
 ('c' s.151 e.41) 'b' = <F308 (e.41) s.151>;
 ('c') 'b' = False;
 (s.143 e.41) 'b' = <F244 (e.41) s.143>;
 () 'b' = False;
 (e.41) s.135 = <F180 (e.41) s.135>;
}
* InputFormat: <F308 (e.41) s.127>
F308 {
 (s.135 e.41) 'a' = <F372 (e.41) s.135>;
 () 'a'  = False;
 (e.41) s.127 = <F55 (e.41) s.127>;
}
* InputFormat: <F244 (e.41) s.120>
F244 {
 (s.127 e.41) 'c' = <F308 (e.41) s.127>;
 () 'c' = False;
 (e.41) s.120 = <F55 (e.41) s.120>;
}
* InputFormat: <F180 (e.41) s.113>
F180 {
```

```
 (s.120 e.41) 'b' = <F244 (e.41) s.120>;
 () 'b' = False;
 (e.41) s.113 = <F55 (e.41) s.113>;
}
* InputFormat: <F55 (e.41) s.104>
F55 {
 (s.113 e.41 ) 'a' = <F180 (e.41) s.113>;
 () 'a' = False;
 (e.41) s.104 = <F50 e.41>;
}
* InputFormat: <F50 e.41>
F50 {
 s.104 e.41 = <F55 (e.41) s.104>;
  = False; }
```

Пример 8. Задача `<GoInt %.Sq e.data>`, где Sq определена в примере 9 раздела 16.1.
Выход: не только не содержит терминологии интерпретатора, но и представляет собой проспециализированную версию унарного квадрата Sq, хотя и отличается от результата прямой специализации, данного в разделе 16.1. В данном случае остаточная программа оформлена в виде хвостовой рекурсии, а не итерации.

```
$ENTRY GoInt {
 e.data = <F98 (e.data) e.data>;
}
* InputFormat: <F98 (e.106) e.107>
F98 {
 (e.106) = ;
 (e.106) I e.107 = e.106 <F98 (e.106) e.107>;
}
```

Итерация в остаточной программе, данной в 16.1, есть следствие ленивых вычислений во время суперкомпиляции; прямая специализация программы Sq при аппликативной стратегии развития стека функций даёт результат, совпадающий с данным выше. При ленивой стратегии развития стека функций результат специализации задачи `<GoInt %.Sq e.data>` содержит терминологию `Interpreter`-а, после повторной суперкомпиляции результат снова совпадает с данным выше.

Пример 9. Задача `<GoInt %.Go e.data>`, где Go определена в примере 10 раздела 16.1.
Выход: накладные расходы двойной интерпретации полностью убраны.

```
$ENTRY GoInt {
 ('*' e.Left) ('*' s.Symbol) ('*' e.Right)
                      = <F215 (e.Left) s.Symbol e.Right>;
}
```

```
* InputFormat: <F1347 (e.101) (e.114) s.167>
F1347 {
 (e.101 s.186) (e.114) Q = <F1347 (e.101) (Q e.114) s.186>;
 (e.101) (s.219 e.114) B = <F215 (e.101 Q Q) s.219 e.114>;}
* InputFormat: <F215 (e.101) s.105 e.114>
F215 {
 (e.101) B s.129 e.114
             = ('*' e.101 B) ('*' s.129) ('*' e.114);
 (e.101) Q s.152 e.114 = <F215 (e.101 Q) s.152 e.114>;
 (e.101 s.167) P e.114 = <F1347 (e.101) (e.114) s.167>;
}
```

Остаточная программа совпадает с результатом прямой специализации, данным в примере 10 раздела 16.1, с точностью до кодировки `'*'` структурных скобок во входных данных (см. определение кодировки в главе 15).

Пример 10. Параметризованной входной точкой будет

```
<GoInt %.Fact1 e.data>,
```

где определение функции факториала `Fact1` в унарной системе счисления дано в примере 1 раздела 10.3 (функция `Go`). В этом примере определение `Fact1` существенно рекурсивно: длина стека не является равномерно ограниченной по размеру входных данных. Кроме того, интерпретируемая функция `Fact1` имеет нетривиальный выходной формат.

Выход: не содержит терминологии `Interpreter`-а. В остаточной программе нет цикла, соответствующего функции `Plus` (см. 9.2).

```
$ENTRY GoInt {
 e.data  = ('*' <F31 e.data>);
}
* InputFormat: <F547 (e.41) (e.144) e.145>
F547 {
 (e.41) () = ;
 (e.41) () I e.145 = I e.41 <F547 (e.41) () e.145>;
}
* InputFormat: <F252 (e.41) e.114>
F252 {
 (e.41) = ;
 (e.41) I e.114 = I I e.41 <F252 (e.41) e.114>; }
* InputFormat: <F125 e.41>
F125 {
 I e.41 = <F252 (e.41) <F125 e.41>>;
 e.41 = <F547 (e.41) () <F31 e.41>>;
}
* InputFormat: <F31 e.41>
F31 {
  = I;
```

```
 I e.41 = <F125 e.41>;
}
```

Представляет интерес остаточная функция F125. Композиция в первом предложении интерпретирующей функции Eval отобразила стек функций интерпретируемой программы в стек функций интерпретирующей программы, и далее эти свойства наследуются компонентой факторизации F125. Выход из рекурсии оформлен странно и требует двух дополнительных шагов РЕФАЛ-машины. В программе есть «мусор» в виде пустых структурных скобок в функции F547. Повторная суперкомпиляция чистит функцию F547 до вида

```
* InputFormat: <F33 (e.41) e.110>
F33 {
 (e.41) = ;
 (e.41) I e.110 = I e.41 <F33 (e.41) e.110>;
}
```

и только переименовывает остальные функции и переменные. Выход из основной рекурсии остаётся таким же странным.

Пример 11. Ещё более сложный вид рекурсии представлен в примере I раздела 5.3.6. Подадим на вход суперкомпилятору нижеследующую параметризованную конфигурацию

```
<GoInt %.The_task e.data>.
```

Выход: не содержит терминологии Interpreter-а и фактически есть повторение программы The_task, где выполнен транзитный шаг (см. 4.4) определения BPred, упрощены входной и выходной форматы функции BSub и построены две дополнительные развёртки S1 и S2. В отображении рекурсивной структуры из The_task в остаточную программу через интерпретирующую функцию Eval основную роль сыграло обнинское условие выделения цикла (см. 5.3.1.1).

```
$ENTRY GoInt {
 ('*' e.data)  = ('*' <S1 (e.data) e.data>);
}
* InputFormat: <F450 e.140>
F450 {
 '1' = ;
 '1' e.140 = '0' e.140;
 '0' e.140 = '1' <F450 e.140>;
  = ;
}
* InputFormat: <F392 (e.140) e.141>
F392 {
 (e.140) = <F450 e.140>;
 (e.140) e.141 = <F450 <F450 <F392 (e.140) <F450 e.141>>>>;
}
* InputFormat: <S1 (e.140) e.141>
```

```
S1 {
 (e.140) = e.140;
 (e.140) e.141 = <S2 (e.140) e.141>;
}
* InputFormat: <S2 (e.140) e.141>
S2 {
 (e.140) '1' e.141 = <F392 (e.140) e.141>;
 (e.140) '0' e.141 = <F450 <F392 (e.140) <F450 e.141>>>;
 (e.140) = <F450 e.140>;
}
```

Пример 12. В этом и следующем примере мы рассмотрим специализацию нашего интерпретатора по рекурсивному обходу РЕФАЛ-деревьев. На входе у SCP4 параметризованная конфигурация

```
<GoInt %.Go e.data>,
```

где Go определена в примере 6 раздела 10.5.

Выход: комментарии излишни (см. раздел 9.2.2).

```
$ENTRY GoInt {
 s.SYMBOL ('*' e.where) = e.where;
}
```

Пример 13. Рассмотрим предыдущий пример в общем положении, переопределив функцию Go так:

```
Go {
 s.SYMBOL1 s.SYMBOL2 (e.where)
                  = <Subst (s.SYMBOL1 s.SYMBOL2) e.where>;
}
```

Выход: не содержит терминологии Interpreter-а и воспроизводит интерпретируемую программу по модулю кодировки данных.

```
$ENTRY GoInt {
 s.SYMBOL1 s.SYMBOL2 ('*' e.where)
                   = <F70 s.SYMBOL1 s.SYMBOL2 e.where>;
}
* InputFormat: <F70 s.102 s.105 e.108>
F70 {
 s.102 s.105 = ;
 s.121 s.105 s.121 e.108 = s.105 <F70 s.121 s.105 e.108>;
 s.102 s.105 s.121 e.108 = s.121 <F70 s.102 s.105 e.108>;
 s.102 s.105 ('*' e.142) e.108
   = ('*' <F70 s.102 s.105 e.142>) <F70 s.102 s.105 e.108>;
}
```

Пример 14. Задача <GoInt %.Go e.data>, где Go определена в примере 1 главы 11.

Выход: остаточная программа не только не содержит терминологии Interpreter-а, но и временна́я сложность интерпретируемой программы понижена с $O(2^n)$ до $O(n)$.

```
$ENTRY GoInt {
 e.41 = <F100 () e.41>;
}
* InputFormat: <F100 (e.108) e.109>
F100 {
 (e.108) A e.109 = <F100 (e.108 B) e.109>;
 (e.108) = e.108;
}
```

Приведём таблицу времён работы суперкомпилятора SCP4 при специализации рассмотренных выше примеров самоописания фиксированного нами подмножества РЕФАЛа. Эти времена включают полную сессию работы SCP4: трансляцию входной программы в язык РЕФАЛ-графов (см. главу 3), основную стадию суперкомпиляции, чистку остаточной программы (см. главы 11, 12), трансляцию результата преобразований из языка РЕФАЛ-графов в РЕФАЛ. Сессия работы SCP4 оформлена в виде пяти последовательных процессов исполнения РЕФАЛ-5 интерпретатора, времена загрузки которых, как и времена загрузки rsl-исходных модулей суперкомпилятора, также включены в показанные в таблице времена. Тестирование производилось на конфигурации:

Microsoft Windows-XP 5.1.2600, Processor: x86/6/8/10 Intel, 995 Mhz,
Total Physical Memory: 256 MB, Available Physical Memory: 144 MB.

Малые времена достаточно условны, так как кроме отмеченных выше условий сильно зависят и от конфигурации оперативной памяти вычислительной машины в момент исполнения тестов.

Времена специализации самоописания подмножества РЕФАЛа

Пример	1	2	3	4	5	6/Appl.	6/Lazy
Время (сек.)	0,85	0,93	0,83	1,29	0,91	1,61	4,04

Пример	7	8	9	10	11	12	13	14
Время (сек.)	118,84	2,27	129,25	16,32	45,31	3,00	3,76	1,80

16.3. Другие эксперименты

Большое число интересных экспериментов со специализатором «суперкомпилятор SCP4» было проведено А. В. Корлюковым [5–8, 45], безвременно ушедшим от нас. Автору посчастливилось работать с Александром.

Часть из этих экспериментов построена на совершенно оригинальных идеях А. В. Корлюкова не только в контексте технологии суперкомпиляции, но и в более широком контексте специализации функциональных программ. Например, в [5, 6] он показывает, как можно использовать специализатор для получения правил арифметики в поле комплексных рациональных чисел и, в более общей постановке задачи, для получения правил арифметики в расширении поля F, когда известна арифметика в F. Он же впервые начал проводить эксперименты с суперкомпилятором, как некоторым «аналогом» интерпретатора ПРОЛОГа [7, 8]. С этой точки зрения, Александр в работе [7] исследовал известную головоломку о «миссионерах и людоедах» и решил средствами SCP4 «головоломку Эйнштейна». В статье [45] описываются сложные примеры специализации интерпретатора языка преобразования XML интернет-документов XSLT. Комментарии к некоторым экспериментам А. В. Корлюкова можно найти также в [54].

Много примеров читатель может найти в дистрибутиве суперкомпилятора SCP4 [58].

Отметим также удачные эксперименты, инициированные А. П. Лисицей, по верификации cache coherence протоколов [10, 11, 49].

Глава 17. ***О соотношении сложности***

17.1. Анализ двух примеров

Здесь мы оценим ускорение (в достаточно грубой модели вычислений РЕФАЛ-программы), получаемое в результате суперкомпиляции рассмотренного нами интерпретатора машины Тьюринга `Turing` (раздел 16.1, пример 10), и сравним нашу оценку с результатами экспериментов. Напомним, что мы представляем программы для машины Тьюринга в виде данных РЕФАЛа:

```
Prog ::= (instruction+)
instruction ::= (CurrState CurrSymb NextSymb NextState Movement)
CurrState ::= SYMBOL
NextState ::= SYMBOL
CurrSymb ::= SYMBOL
NextSymb ::= SYMBOL
Movement ::= left | right
```

Важным для нас будет следующее

Утверждение 1. *Для любой программы* `Prog` *на языке* MT *результат специализации суперкомпилятором SCP4 интерпретатора* `Turing` *по* `Prog` *не содержит терминологии программы* `Prog`. *То есть включает в себя только синтаксические единицы Prog, описывающие данные на ленте: текущие символы в ячейках и записываемые в ячейки символы. Следовательно, происходит «эффективная» компиляция программы* `Prog` *из языка* MT *в РЕФАЛ.*

Это утверждение непосредственно следует из замечаний, данных нами в комментариях к примеру 10 (16.1): длина стека функций программы Turing равномерно ограничена по входным данным, число состояний в программе Prog конечно, отношение похожести конфигураций включает в себя условие упрощающего отношения (см. 5.3.1.1, 5.3.1.2, 5.3.6). Пусть $\{\texttt{step}_n(\underline{\texttt{<F data>}})\}$ есть последовательность полей зрения РЕФАЛ-машины (2.2, [69]), вычисляющая значение РЕФАЛ-программы F на данных data. Из вышеприведённых соображений и утверждения 1 следует, что существует последовательность натуральных чисел $\{n_k\}$ и частичная функция ψ из множества полей зрения РЕФАЛ-машины во множество полей зрения РЕФАЛ-машины, такие что для любой tape верно следующее равенство:

$$\texttt{step}_k(\underline{\texttt{<Turing Prog, tape>}}) = \texttt{step}_{n_k}(\psi(\underline{\texttt{<Turing}_{Prog}\ \texttt{tape>}})),$$

где $\texttt{Turing}_{Prog}$ есть результат специализации Turing по Prog. Таким образом, ψ и $\{n_k\}$ не зависят от ленты.

Определим функцию размера $|\bullet|$ на множестве данных РЕФАЛа по индукции:

```
| | = 0;
|SYMBOL data| = 1+|data|;
|(data1) data2| = 2+|data1 data2|;
```

Значение функции |data| есть число элементарных (с точки зрения РЕФАЛа) элементов памяти вычислительной машины, необходимое для отображения (построения) data в оперативной памяти [35] в модели вычислений конкретной реализации РЕФАЛа-5 [69, 75].

Мы оценим ускорение, получаемое в результате специализации интерпретатора MT, при нижеследующих достаточно грубых предположениях:

(1) время исполнения каждого шага РЕФАЛ-машины равно const, в том случае, когда оно не зависит от входных данных, и будем считать эту константу единицей, так как нас интересует лишь ускорение;

(2) времена построений (копирований) в поле зрения одного символа, левой структурной скобки, правой структурной скобки равны друг другу и равны времени переноса РЕФАЛ-выражения из одного поля зрения в другое;

(3) инструкция MT, следующая для исполнения после данной инструкции, всегда находится на последнем месте в списке инструкций;

(4) качество отображения РЕФАЛ-графа в РЕФАЛ (см. главу 13) приемлемо.

Пусть $ProgMT(L, N)$ есть множество всех программ Prog для MT, таких что |Prog| = L, каждая из которых содержит N инструкций. Обозначим $CntOfCells(InputTape)$ длину пути, пройденного головкой MT

[35] Мы не рассматриваем встроенные функции ввода, для которых данное утверждение неверно.

до остановки конкретной программы Prog $\in ProgMT(L, N)$. Тогда количество шагов РЕФАЛ-машины, затраченное на интерпретацию Prog в худшем случае [36] равно

$$(N + 1 + 1) * CntOfCells$$

(где слагаемые в скобках есть вклады функций Search, Turing1, Turing соответственно). Времена исполнения всех РЕФАЛ-шагов, за исключением шагов, соответствующих выбору второго предложения функции Turing, равномерно ограничены по размерам L входных программ (3.3.1, [2, 15, 22, 69]). Время построения второй копии программы Prog шагами, соответствующими второму предложению функции Turing, линейно по L. Таким образом, время интерпретации Prog, в худшем случае, в первом приближении порядка

$$(N + 2 + (1 - \epsilon) * L) * CntOfCell, \qquad (**)$$

где $0 < \epsilon < 1$ зависит лишь от интерпретатора РЕФАЛа-5 [69, 75] и отражает тот факт, что копирование одного символа быстрее шага РЕФАЛ-машины.

В нашем эксперименте (пример 10, раздел 16.1) суперкомпилятор SCP4 построил остаточную программу, в которой отсутствует терминология программы DoublePQ, по которой мы специализировали интерпретатор (и, как мы заметили выше, таким свойством будет обладать каждый результат специализации Turing по конкретной программе [37]). Следовательно, мы должны учесть время копирования и построения терминов DoublePQ (исключая описание данных) текущего состояния MT на каждом шаге оригинальной РЕФАЛ-программы (смотри исходный текст интерпретатора): второе предложение функции Turing даёт вклад

$$(1 - \epsilon) * (1 + 2 + 4) * (N + 2) * CntOfCells$$

(перенос значения в первое вхождение `e.instr`, вклад копирования значения второго вхождения мы уже учли, построение значений `s.q s.symbol`, и построение соответствующих форматных скобок), функция Turing1 даёт вклад

$$(1 - \epsilon) * (1 + 2) * (N + 2) * CntOfCells$$

(построение значения `(s.r)`), второе предложение функции Search даёт вклад

$$(1 - \epsilon) * (1 + 1 + 4) * (N + 2) * CntOfCells$$

(построение значения `s.key1`, перенос значения `e.table`, и построение соответствующих форматных скобок). То есть в сумме получаем

$$(1 - \epsilon) * 18 * (N + 2) * CntOfCells.$$

Учитывая $(**)$, получаем время

$$\big((1 - \epsilon) * (L + 16 * (N + 2)) + N + 2\big) * CntOfCells.$$

---

[36] То есть в том случае, когда после обработки очередной ячейки MT всегда переходит к выполнению последней инструкции в Prog.

[37] Следовательно, наши рассуждения не зависят от этой программы, но мы предпочитаем говорить о DoublePQ, чтобы дать читателю возможность проследить исходные и остаточные программы.

Алгоритм движения головки по ленте в результате специализации не изменился. В остаточной программе все РЕФАЛ-шаги равномерно ограничены по размеру входных данных, и в данном случае качество отображения РЕФАЛ-графа в РЕФАЛ (см. главу 13) достаточно приемлемо. Таким образом, можно ожидать ускорения:

$$StepSpeedup \approx N + 2,$$

$$TimeSpeedup \approx \big((1 - \epsilon) * (L + 16 * (N + 2)) + N + 2\big).$$

В нашем эксперименте (`DoublePQ`) $N = 5$, а длина L, в терминах представления РЕФАЛ-выражений в интерпретаторе РЕФАЛа-5, равна $7 * N + 2$. Ожидаемое

$$StepSpeedup \approx N + 2 = 7,$$

$$TimeSpeedup \approx N + 2 + (23 * N + 34) * (1 - \epsilon),$$

$$\frac{TimeSpeedup}{StepSpeedup} \approx 1 + 23 * (1 - \epsilon) - \frac{12 * (1 - \epsilon)}{(N + 2)} \approx 1 + 21{,}29 * (1 - \epsilon).$$

Мы провели эксперименты в конфигурации тестирования времён суперкомпиляции (16.2). Эксперимент `DoublePQ` с 2^{12} символами **P** даёт такие числа:

$$StepSpeedup \approx 5{,}00, \qquad \frac{TimeSpeedup}{StepSpeedup} \approx 1{,}44, \qquad (1 - \epsilon) \approx 0{,}02.$$

Мы также проспециализировали интерпретатор `Turing` по программе унарного умножения `Multiplication`, которая вместе с результатом специализации дана нами в Приложении А. В данном случае $N = 35$ и наши оценки выглядят так:

$$StepSpeedup \approx 37, \qquad \frac{TimeSpeedup}{StepSpeedup} \approx 1 + 22{,}68 * (1 - \epsilon).$$

Тестирование прямой интерпретации `Multiplication` и соответствующей остаточной программы при умножении 40 на 30 даёт числа:

$$StepSpeedup \approx 25{,}99, \qquad \frac{TimeSpeedup}{StepSpeedup} \approx 1{,}26, \qquad (1 - \epsilon) \approx 0{,}01.$$

При больших $N > 100$ следует ожидать (в нашей модели[38)]) ускорение по времени, значительно большее ускорения по шагам. Кроме того, наши эксперименты ещё раз показывают, что при специализации можно получить сколь угодно большое ускорение, которое зависит от размера входных данных оригинальной программы, по которым ведётся специализация.

Рассмотрим специализацию самоописания подмножества РЕФАЛа по интерпретатору MT, который, в свою очередь, исполняет фиксированную

[38)] Измерив практическое значение $(1 - \epsilon)$ на конкретной вычислительной машине, можно оценить, насколько адекватна построенная нами модель.

программу[39] `DoublePQ` (пример 9, раздел 16.2). Пусть $s(\underline{\texttt{<Prog data>}})$ есть число шагов РЕФАЛ-машины при исполнении РЕФАЛ-программы `Prog` на входных данных `data`. Пусть `Prog := Go`, где `Go` определена в примере 10 раздела 16.1. Тогда

$$s(\underline{\texttt{<Go data>}}) = O(|\texttt{data}|^2) = O(m^2),$$

где m — число символов **P** на ленте перед началом работы MT. Далее, для любой РЕФАЛ-программы `Prog` $\exists\, C \in \mathbb{R}$, такая что для любых РЕФАЛ-данных `data` выполняется неравенство

$$s(\underline{\texttt{<IntGo Prog, data>}}) \leqslant C * s(\underline{\texttt{<Prog data>}}).$$

Таким образом, $s(\underline{\texttt{<IntGo Go, tape>}}) = O(m^2)$. Тот же порядок имеет и время исполнения результата нашей специализации (см. пример 9, раздел 16.2). Времена исполнения шагов РЕФАЛ-машины при исполнении остаточной программы равномерно ограничены по входным данным. Следовательно, и время исполнения остаточной программы имеет порядок $O(m^2)$. Иначе обстоит дело с оригинальной программой `IntGo`. В модели вычислений РЕФАЛа-5 копирование данных интерпретируемой программы `Prog` происходит при исполнении шагов РЕФАЛ-машины, соответствующих функциям `Eval` (первое, второе и третье предложения, переменная `e.env`), `Matching` (первое предложение, переменная `e.d`), `PutVar1` (первое предложение, переменная `e.val1`). Число вызовов этих функций и выбора соответствующих предложений, при интерпретации любого РЕФАЛ-шага `Prog` посредством `IntGo`, не более чем $|\underline{\texttt{Prog}}|$. То есть получаем

$$time(\underline{\texttt{<IntGo Go, tape>}}) = O(m^3).$$

Суперкомпилятор SCP4 улучшил сложность специализируемой программы по времени.

Уточним результат наших рассуждений:

$$time(\underline{\texttt{<IntGo Go, tape>}}) = k * m^3 + C * m^2,$$

где постоянная k много меньше, чем постоянная C.

Проанализируем, как соотносятся размеры памяти, потребляемой при исполнении вызова `<IntGo Go, tape>`, и результата специализации этого вызова при одних и тех же входных данных[40]. Длина стека РЕФАЛ-функций при непосредственном вычислении вызова `<Go tape>` всегда не больше 3. При его исполнении посредством `<IntGo Go, tape>` эта длина уже зависит от m: каждый интерпретируемый РЕФАЛ-шаг программы `Go` создаёт в памяти (в поле зрения, далее П3) РЕФАЛ-машины вызов

$$\texttt{CallEval}_0 \texttt{ := <Eval (env}_0\texttt{) (expr}_0\texttt{) Go>}$$

---

[39] Идея специализации двойной интерпретации впервые была использована в работах А. В. Корлюкова [7, 45].

[40] С точностью до кодировки.

(первое предложение функции Eval), который исполняется только после полного завершения вычисления вызова функции из Go, соответствующего интерпретируемому РЕФАЛ-шагу. При интерпретации вызова <Go tape> число вызовов $\texttt{CallEval}_0$ пропорционально числу РЕФАЛ-шагов вычисления <Go tape>, т.е. порядка m^2. Размер каждого вызова

$$|\texttt{CallEval}_0| = O(m).$$

Следовательно, порядок |ПЗ|, необходимый для интерпретации

<IntGo Go, tape>,

не меньше чем $O(m^3)$. Во всех других случаях для увеличения δ — размера ПЗ (посредством копирования данных) при исполнении вызова

<IntGo Go, tape>

верно неравенство $\delta \leqslant |\texttt{Go}| * m$, и это увеличение происходит лишь на время интерпретации конкретного шага программы Go; после завершения этого шага δ вычитается из размера текущего ПЗ. Размер ПЗ, необходимого для исполнения результата специализации (в нашем примере), линеен по m.

Сложность по потребляемой при исполнении памяти сократилась на два порядка с $O(m^3)$ до $O(m)$.

Отношение мультипликативных констант, практически значимых, в полученных оценках также велико в пользу остаточной программы. Это видно, например, из того, что в критическом первом предложении функции Eval создаётся три копии интерпретируемой программы Go. Сложность потребляемой интерпретатором IntGo памяти по размеру Go, при фиксированных входных данных, в нашем случае оценивается снизу произведением $c_1 * |\texttt{Go}|^3$.

Насыщение по памяти (swapping) на нашей конфигурации вычислительных ресурсов происходит очень быстро, когда ещё первое слагаемое оценки по времени, данной выше, мало по сравнению со вторым, а время исполнения результата специализации близко к нулю и существенно зависит от текущего состояния оперативной памяти компьютера. При $m = 34$ (последнее значение перед swapping-ом) мы наблюдаем экспериментальное значение постоянной C оценки по времени равное 822,14.

17.2. Общие замечания

В данной работе мы неоднократно обращались к вопросу временно́й эффективности результата суперкомпиляции, как в специальных подразделах «К вопросу о целях преобразований» (далее КВоЦП), так и в других главах (см., например, 3.2, 3.3, 13). Мы показали, что порядок временной сложности может уменьшиться в результате наших преобразований (см., например, 10.5 и 17.1), и это отличает SCP4 от большинства существующих специализаторов[41] [40, 41] (все они, как и SCP4, являются, конечно, экспериментальными). В КВоЦП как приводились утверждения в пользу сокращения

[41] Примером метода специализации, который также может улучшать временной порядок программы, является технология, разрабатываемая лабораторией Ёшико Футамуры [29, 30].

времени исполнения результатов преобразований конкретными инструментами SCP4, так и давались примеры, когда физическое время интерпретации той или иной синтаксической конструкции оригинальной программы меньше времени интерпретации её образа при преобразованиях этими *локальными* инструментами. Точный анализ соотношения временной сложности остаточных программ, которые строятся SCP4, и временной сложности соответствующих им входных, преобразуемых программ не только выходит за рамки нашей работы, но и не возможен в её рамках, так как мы не ставили задачи *точного* определения всех инструментов SCP4. Ниже мы дадим простейшую модель суперкомпиляции и проанализируем её. Если иметь в виду физическое время, то утверждение об ускорении остаточной программы по отношению к входной программе неверно[42]; можно говорить только об *ожидании* такого ускорения. На практике это ожидание часто оправдывается, когда входная программа содержит неоптимальные структуры достаточно простого вида.

В случае «общего положения» преобразуемой программы методом специализации, весьма неудачно (по нашему мнению) названным суперкомпиляцией, следует ожидать только уменьшения мультипликативной постоянной оценки времени исполнения остаточной программы по отношению к входной программе. Естественно, возникает вопрос о смысле цели специализации, ибо мультипликативная постоянная определяется не только «железом», но и тем компилятором, который транслирует базовый исполнитель объектного языка (в нашем случае компилятором языка С), как и тем интерпретатором, который исполняет результат этой трансляции. Оба эти инструмента принадлежат к внешнему миру относительно данного специализатора: исполняющая система не является физически замкнутой.

При фиксированных допустимых ресурсах вычислений, практическая значимость мультипликативной постоянной в оценке необходимых ресурсов не вызывает сомнений. Особенно эта постоянная значима, когда ресурсом является размер памяти, необходимой для решения задачи, так как в случае времени мы может и подождать разумное время, а переполнение по памяти не оставляет шансов на получение результата. В алгоритмически полных моделях вычислений, более приближенных к практике программирования, чем классические модели, мультипликативная постоянная теоретически значима при линейном, почти линейном (см. [34]) и полиномиальном времени по размеру n входных данных задачи. То есть, например, $\exists\, c \in \mathbb{R}$ такая, что $\forall\, b \in \mathbb{R}\ (b \geqslant 1)\ \exists$ множество A, принадлежность к которому можно распознать за время $b*c*n$, но нельзя распознать за время $b*n$ (см. [42]). Отметим, что в соответствующих моделях вычислений размер памяти, потенциально доступной для преобразований входных данных, неограничен, а ограничения накладываются на «стиль программирования». Ниже мы сформулируем аналог этих ограничений в терминах РЕФАЛа. Интересная задача анализа соответствующей модели РЕФАЛ-вычислений выходит за рамки нашего текста.

[42] Остаточная программы примера 2 раздела 16.1 в модели вычислений РЕФАЛа-5 будет работать медленнее, чем оригинальная программа.

Принципиальным здесь является вопрос о соотношении логического и физического[43] времени исполнения программы. Напомним, что в главе 1 в качестве входного языка мы ограничили себя рассмотрением подмножества РЕФАЛа, допускающего U-пошаговую реализацию, в которой программа представляет собой динамическую систему с дискретным временем (логическим временем исполнения шага РЕФАЛ-машины). Далее, в разделе 3.3 мы приблизили понятие этого логического времени к понятию физического времени, объявив, что будем рассматривать только «ограниченный» РЕФАЛ-5: отождествление стало исполняться за физическое время, равномерно ограниченное по размеру входных данных программы, но осталось неверным аналогичное утверждение для физического времени построения правых частей РЕФАЛ-предложений. В данной работе мы предполагаем, что время поиска определения вызываемой функции пренебрежимо мало по сравнению с другими действиями интерпретации. Ниже в данной главе мы рассматриваем только «ограниченный» РЕФАЛ.

17.2.1. Простейшая модель суперкомпиляции

Опишем простейшую модель суперкомпилятора `SpeedupRefal`, который будет аналогом преобразователя программ языка машины Тьюринга, строящегося при доказательстве классической теоремы ускорения[44].

Стратегия развития стека функций будет аппликативной (см. 6.2) и каждая задача на преобразование, описанная параметризованным стеком, который содержит не менее двух функциональных вызовов, будет разбиваться на две подзадачи — соответствующую вершине этого стека и оставшейся части стека (см. 5.1, 5.3.2). Входная и выходная среды каждой подзадачи (не только получаемые при указанном разбиении) обобщаются до тривиальных: обе среды унарны и описаны e-переменными. То есть, например, алгоритм построения выходных форматов не работает (см. 9.1.2). Не работают и все другие инструменты глобального анализа (см. главу 9), экранируемости ветвей (см. главу 8), а также стадии, следующие за основной частью алгоритма суперкомпиляции (см. главы 11, 12). Стартовые узлы из грозди прогонки не удаляются. Опорными узлами в метадереве возможных вычислений объявляются все стартовые узлы (не листья) грозди прогонки, и только они (см. главу 4). Свёртка рассматривает только опорные узлы с нечётными порядковыми номерами на непрофакторизованной к данному моменту части пути от корня метадерева возможных вычислений до текущего опорного узла (т. е. первый непрофакторизованный узел есть узел номер один), текущие опорные узлы с чётными номерами также пропускаются (см. главу 5). Обобщение параметризованных выражений (см. 5.3.3) тривиально: сохраняя функциональную структуру, параметризованные данные всегда считаем

[43] Здесь под «физическим временем» мы имеем ввиду «физическое» время по модулю компилятора K языка C (Си), транслирующего исходные тексты интерпретатора РЕФАЛа-5, и интерпретатора, исполняющего результат трансляции K.

[44] См. любой учебник по теории сложности рекурсивных функций или [42].

«похожими» и обобщаем их до e-параметров, «отрицательная» информация (см. 5.3.4) всегда оставляется пустой.

Преобразователь `SpeedupRefal` строит остаточные программы, каждый шаг *step*(`data`) которых, нестрого говоря, представляет два шага $step_2(step_1(\texttt{data}))$ преобразуемых программ. И, следовательно, мы получаем ускорение по шагам (логического времени исполнения), равное 2. `SpeedupRefal` не изменяет структуру построения поля зрения. Следовательно, число копирований элементарных элементов памяти (см. 17.1), затрачиваемых *step*(`data`) при изменении поля зрения (ПЗ), не больше числа элементарных копирований, затрачиваемых $step_2(step_1(\texttt{data}))$ на изменение ПЗ. Число этих копирований может значительно сократиться, как показывает простой

Пример 1. Входная программа.

```
$ENTRY Go { e.x = <F e.x (e.x)>; }
F { e.y (e.z) = e.y; }
```

Общее число элементарных изменений ПЗ не может увеличиться по той же причине.

Хуже обстоит дело с отождествлением. Образец `patt`, с которым отождествилось `data` на шаге *step*(`data`), есть проспециализированная синтаксическая композиция образцов $\texttt{patt}_1$ и $\texttt{patt}_2$ удачных отождествлений `data` на шаге $step_1(\texttt{data})$ и данных, передаваемых шагом $step_1(\texttt{data})$ шагу $step_2$, на шаге $step_2$. То есть `patt` получается из $\texttt{patt}_1$ некоторой подстановкой переменных и последующей специализацией. Пусть F_s, F_{s_1} есть «функции», соответствующие шагам *step*, $step_1$. Число предложений в построенной функции F_s равно[45] $\sum_{i=1}^{l} k_i$, где l есть число листьев грозди прогонки `cl` вызова входной функции F_{s_1} с корнем в опорном узле, ставшим базисным узлом для F_s; а k_i есть число листьев грозди прогонки с корнем в i-ом листе грозди `cl`, т.е. результата прогонки конфигурации, описанной в этом листе. В общем положении (в худшем случае), эта сумма равна сумме количеств предложений функций входной программы, которые вызываются активными вызовами всех предложений функции F_s.

Пример 2. Входная программа.

```
$ENTRY Go {
 A e.x = <F C e.x>;
 B e.x E = <G e.x>;
}
F {
  s.x e.1 (e.y) = e.y;
  s.x e.1  t.y  = s.x;
```

[45] Для простоты, мы не рассматриваем здесь граничные эффекты.

```
      e.1       = e.1;
}
G {
  D   (e.y) e.1 = e.y;
  s.x (e.y) e.1 = e.y;
  s.x  t.y e.1  = s.x;
      e.1       = e.1;
}
```

Остаточная программа:

```
$ENTRY Go {
 A e.2 (e.1) = e.1;
 A e.2 t.1 = C;
 A e.2 = C e.2;
 B D   (e.3) e.2 E = e.3;
 B s.1 (e.3) e.2 E = e.3;
 B s.1  t.3  e.2 E = s.1;
 B e.2 E = e.2;
}
```

Если образец `patt` удачного отождествления на шаге $step(\texttt{data})$ соответствует j-ому предложению шага $step_1(\texttt{data})$, то произойдёт

$$K = \sum_{i=1}^{j-1} k_i + \delta$$

неудачных попыток отождествления шага `step`, где δ — число неудачных попыток шага $\texttt{step}_2$. Оригинальная программа при исполнении $step_2(step_1(\texttt{data}))$ затрачивает $j - 1 + \delta$ неудачных попыток, что, в общем положении, меньше чем K. Как показывает пример 2, порядок элементарных действий сопоставления данных с конкретным образцом (см. [69] и самоописание в 16.2) в остаточной программе может не совпадать с соответствующим порядком оригинальной программы. Что может привести к увеличению числа этих действий при неудачных попытках отождествления, т. е. к дополнительным проходам по данным.

Итак, физическое время исполнения на `data` входной программы P может быть как меньше, так и больше физического времени остаточной $P_{SpeedupRefal}$ программы на `data`.

17.2.2. Ограничения на стиль программирования

Выше в данной главе мы отметили, что существуют модели вычислений, в которых мультипликативные постоянные нетривиально определяют шкалу разрешимых задач. В терминах РЕФАЛа-5 аналогами ограничений, накладываемых этими моделями, будут следующие: (1) будем рассматривать только алгоритмически полное подмножество ограниченного РЕФАЛа, в котором запрещены кратные вхождения `e-` и `t-`переменных в правую часть

РЕФАЛ-предложения; (2) длины функций-определений (а не всей программы) ограничены сверху константой, не зависящей от программы. В такой РЕФАЛ-модели физическое время исполнения шага РЕФАЛ-машины равномерно ограничено не только по входным данным, но и по программам [46].

Глава 18. ***Разметка входной программы***

Здесь мы рассмотрим простой язык, посредством которого программист может объявить суперкомпилятору SCP4 стратегии преобразований данной программы или подсказать ему правильный выбор конкретных локальных действий. Мы будем называть понятия этого языка псевдокомментариями и псевдофункциями, так как с точки зрения компилятора РЕФАЛа-5 они являются либо комментариями, либо тождественными функциями РЕФАЛа. Читатель уже знаком с некоторыми конструкциями этого языка (см. 16.2).

18.1. Псевдокомментарии

Псевдокомментарии являются объявлениями для суперкомпилятора SCP4 в преобразуемой программе. Они представляют собой однострочные комментарии РЕФАЛа-5 вида:

```
*$pseudo_type switches;
```

Здесь `pseudo_type` есть тип псевдокомментария, а `switches` — аргументы псевдокомментария. Знак `$` должен стоять в строке на второй позиции, тип всегда начинается с третьей позиции в строке псевдокомментария.

```
pseudo-comment :=
    *$EXECUTABLE function-names; | *$STRATEGY strategy;
  | *$PUT_IN_HISTORY function-names; | *$TRANSIENT answer;
  | *$GO_AHEAD function-names; | *$SIMPLIFY answer;
  | *$CALL_DEPTH number; | *$DEPTH number; | *$LENGTH number;
  | *$MST_FROM_ENTRY; | *$VERSION version;
  | *$MATCHING ForRepeatedSpecialization;

function-names := IDENTIFIER+
strategy := Lazy | Applicative
answer   := Yes  | No
version  := CHARACTER*
number := INTEGER
```

Псевдокомментарий `*$EXECUTABLE` позволяет объявить любую функцию, внешнюю по отношению к данному РЕФАЛ модулю, исполняемой во время суперкомпиляции. По умолчанию, исполняемыми на этапе суперкомпиляции функциями являются встроенные функции РЕФАЛа-5 без

[46] По модулю времени поиска определения функции.

побочного эффекта. Программист может объявить исполняемыми, например, процедуры ввода-вывода, но в этом случае ответственность за корректность преобразований ложится на этого программиста. Тем не менее такие объявления иногда имеют смысл: например, чтение из файла сразу всех данных (т. е. содержимое файла рассматривается как неделимый объект).

К данному моменту реализованы две стратегии развития стека функций во время суперкомпиляции: аппликативная (вызов по значению) и ленивая (вызов по необходимости) (см. 6.2). Стратегия развития стека может быть объявлена псевдокомментарием `*$STRATEGY`, по умолчанию SCP4 выбирает ленивую стратегию. Логика аппликативной стратегии более проста.

Объявление `*$PUT_IN_HISTORY` приводит к тому, что из грозди прогонки не будут удаляться те полутранзитные узлы, на вершине функционального стека которых находится вызов одной из функций, указанных в аргументах этого объявления (см. 4.1). Более того, эти не удалённые полутранзитные узлы объявляются опорными узлами (см. 4.3). Данное объявление позволяет исключить бесконечные циклы процесса суперкомпиляции, которые порождаются развёртками конфигураций, находящихся в указанных выше полутранзитных узлах.

Объявление `*$GO_AHEAD` указывает на функции, появление обращений к которым на вершине функционального стека является, при условии что это не противоречит `*$PUT_IN_HISTORY` — псевдокомментарию, запретом для SCP4 объявлять узел, содержащий такое состояние стека, опорным узлом. То есть из правила описанного в разделе 4.3, делаются исключения. Данное объявление приводит к безусловным развёркам всех вызовов функций, указанных в аргументе объявления.

Если аргумент псевдокомментария `*$TRANSIENT` есть `Yes`, то прогонка будет сохранять в грозди все полутранзитные узлы и объявлять их опорными узлами, иначе (как и по умолчанию) удаление полутранзитных узлов определяется правилом, описанным в разделе 4.1 и псевдокомментарием `*$PUT_IN_HISTORY`. При таком объявлении бесконечные циклы суперкомпиляции исключены, но качество преобразований существенно снижается. При других режимах работы SCP4 возможны бесконечные циклы работы SCP4; одним из таких режимов является режим «по умолчанию». Покажем две простейшие причины таких бесконечных циклов:

- В исходной программе существует функция `F` и существуют РЕФАЛ-данные `data`, такие что вызов `<F data>` исполняется бесконечное время. Пример:

```
$ENTRY Go { e.x = <Fab e.x>; }
Fab {
  'a' e.1 = 'b' <Fab e.1>;
  'b' e.1 = <Fab 'b' e.1>;
      e.1 = e.1;
}
```

- В исходной программе существует семантически недостижимое предложение, которое имеет формальный синтаксический цикл.

Пример:

```
$ENTRY Go { e.x = <Start e.x ()>; }
Start {
    t.1 e.x (e.2) = <Start e.x (e.2 'a')>;
          (e.2) = <Fab e.2 ()>;
}
Fab {
    'a' e.1 (e.2) = <Fab e.1 (e.2 'b')>;
    'b' e.1 (e.2) = <Fab 'b' e.1 (e.2)>;
          (e.2) = e.2;
}
```

Если аргумент псевдокомментария `$SIMPLIFY` есть `No`, то условие упрощающего отношения (см. 5.3.1.2) будет заменено на тривиальное. То есть в момент проверки этого условия (а не в какой-то другой момент) все параметрические описания сред считаются «похожими». Если аргумент этого псевдокомментария `$SIMPLIFY` есть `Yes`, то SCP4 сравнивает параметрические описания сред, как и по умолчанию, по правилу, описанному в разделе 5.3.1.2.

Длиной $ln(pexp)$ параметризованного РЕФАЛ-выражения $pexp$ (см. 2.2) назовём суммарное число РЕФАЛ-символов, структурных скобок, вызовов функций, `s-` и `t`-параметров на верхнем структурно-скобочном уровне этого выражения $pexp$. Глубиной $dp(pexp)$ параметризованного РЕФАЛ-выражения $pexp$ назовём максимальную глубину вложенности его структурных скобок. Например, если

$pexp$ `:= (e.2) (A) (B ((<F ((e.1))>))) t.3 e.5 e.6 s.4 C <F e.1 (<G e.2>)>`,

то $ln(pexp) = 7$, $dp(pexp) = 5$. Пусть дано параметризованное выражение pexp. Объявление `$LENGTH number;` запрещает обобщать пустое выражение `[]` с $pexp$ (см. 5.3.3), если $ln(pexp) < number$. Объявление `$DEPTH number;` запрещает обобщать `[]` с $pexp$, если $dp(pexp) < number$.

Псевдокомментарий `$MST_FROM_ENTRY` объявляет задачами на преобразования список тривиальных MST-схем — вызовов функций, объявленных в исходном РЕФАЛ-программном модуле доступными для других модулей (т. е. `$ENTRY`); с аргументом в общем положении. Например, если функция F есть `$ENTRY` функция, то задачей, соответствующей этой функции, будет MST-схема `<F e.1>`. Данный комментарий является альтернативой к описанию MST-схем в отдельном (от преобразуемой программы) MST-модуле, но позволяет формулировать только простейшие задачи (см. 15.4).

Результат суперкомпиляции, конечно, может зависеть от конкретной версии нашего суперкомпилятора SCP4. С этой точки зрения, иногда бывает полезно явно проставлять версию SCP4 в исходном тесте преобразуемой программы: SCP4, в таком случае, предупредит о том, что остаточная программа, возможно, будет отличаться от той, которую пользователь получал при использовании другой версии SCP4. Версия объявляется псевдокомментарием `$VERSION`. Название используемой программистом версии SCP4 можно узнать либо в исходных текстах модуля `KEY.REF`, либо вызовом `<Scp4Version>` в программе, преобразуемой суперкомпилятором. Во втором

случае функция `Scp4Version` должна быть объявлена внешней `$EXTERN` и исполняемой `$EXECUTABLE`.

18.2. Псевдофункции

Назовём функцию, определённую на РЕФАЛе и реализующую тождественное отображение на множестве данных РЕФАЛа, псевдофункцией, если её имя есть выделенное имя для суперкомпилятора SCP4. Мы опишем семантику (по отношению к стадии преобразований посредством SCP4) следующих выделенных имён: `Const__`, `UnConst__`, `Appl__`, `Cut__`. При использовании конкретной псевдофункции программист должен определить её (как тождественное отображение) в соответствующем РЕФАЛ-модуле. Вызовы псевдофункций служат для подсказок SCP4 в выборе конкретных локальных действий и исполняются во время суперкомпиляции.

Вызов псевдофункции `Const__` отрабатывает в соответствии с выбранными стратегиями процесса суперкомпиляции. `Const__` помечает в своём аргументе каждый константный символ РЕФАЛа (т. е. данное отождествляющееся с s-переменной) как символ, обобщать который запрещено. Например,

`<Const__ A (12) e.1 s.2 <F 'g'> > ⇒ A (12) e.1 s.2 <F 'g'>`,

далее в процессе суперкомпиляции константы **A**, **12**, **'g'** обобщаться не будут, пока их защита от обобщения явно не снята вызовом псевдофункции `UnConst__`. Например,

`<UnConst__ <G 12 t.4> (A)> ⇒ <G 12 t.4> (A)`.

Заметим, что ленивая стратегия развития стека функций может привести к неожиданным, на первый взгляд, защитам от обобщения.

Вызов псевдофункции `Appl__` приводит к полной декомпозиции функциональной структуры аргумента этого `Appl__` независимо от выбранной стратегии развития стека функций (см. 6.2).

Пример.

Стек `[ <Appl__ <H e.1 A <G B <F e.1>>>> ⇐a out.1; , ]`
будет преобразован к виду
`[ <F e.1> ⇐a out.3; <G B out.3> ⇐a out.4;`
`<H e.1 A out.4> ⇐a out.2; , ]`.

Глава 19. ***О свойствах модели вычислений***

Здесь мы намерены ещё раз (см. 3.3.1, 17) заострить внимание читателя на том, что «суперкомпиляция», как метод специализации программ, не может рассматриваться только в рамках объектного языка программирования: нужно зафиксировать конкретную реализацию этого языка, т. е. модель вычислений.

Отметим некоторые свойства рассматриваемой нами реализации РЕФАЛа-5 [69, 75], которые являются принципиальными с точки зрения простоты построения суперкомпилятора.

Понятия сложности вычислений одного шага РЕФАЛ-машины в модели вычислений ограниченного РЕФАЛа-5 могут быть определены локально в терминах этого шага, т. е. в терминах синтаксиса конкретного РЕФАЛ-предложения, соответствующего этому шагу; точнее говоря — в терминах синтаксиса конкретной РЕФАЛ-функции F (синтаксисом только предложений функции F, а не других функций, которые F вызывает). Именно синтаксисом, а не семантикой[47].

Более того, если мы рассмотрим F на полном РЕФАЛе-5 (а не в его подмножестве), то понятия сложности построения всех объектов, описанных синтаксисом функции F (а не семантикой), и понятия сложности отождествления как входных данных, так и данных, которые будут построены в результате исполнения вызовов в левых частях предложений функции F, с соответствующими образцами из описания F, полностью определяются локальным синтаксисом F. Это замечание верно и для сложности времени исполнения программы, и для сложности размера оперативной памяти, необходимой для этого исполнения. Нижеследующий пример поясняет данное замечание.

```
F {
 s.x e.y, <G e.y e.y>: e.z (e.z) = e.y <H e.z e.y> e.z;
 t.x e.y = e.y;
}
```

Здесь время построения вызова `<G e.y e.y>` и его аргументов порядка $2 * m$, где m размер входных данных функции F; время полного вычисления вызова функции G определяется семантикой функции G; время отождествления результата вызова G с образцом `e.z (e.z)` в худшем случае порядка $n/2$, где n размер этого результата; время построения правой части первого предложения порядка m; время полного вычисления вызова H определяется семантикой функции H. Размер памяти, необходимой для построения вызова `<G e.y e.y>`, порядка $2 * m$; размер памяти, необходимой для построения правой части первого предложения, порядка m; размер памяти, необходимой для вычисления вызовов функций G и H, определяется их семантикой.

Таким образом, специализатор способен принимать принципиальные решения на основании свойств локального синтаксиса: в реализации РЕФАЛа-5 нет ни сборки мусора, ни счётчика ссылок, оба эти понятия предполагают перенос каких-то необходимых действий из конкретного шага РЕФАЛ-машины на другие шаги машины, выбор которых существенно зависит от семантики. Простота вычислительной модели РЕФАЛа-5 делает задачу построения суперкомпилятора обозримой.

---

[47] Хотя такое разграничение достаточно условно, здесь мы имеем ввиду тот факт, что понятия сложности можно описать не используя дополнительных развёрток функции F.

Заключение

Природа не предполагает для себя никаких целей... и все конечные причины составляют только человеческие вымыслы.

Спиноза

Заключим перечислением некоторых задач, решение которых привело бы не только к существенному развитию преобразователя SCP4, но и к важным шагам в понимании природы специализации программ.

Любая автоматическая специализация программ является некоторым приближением к цели специализации программ как таковой. Ибо, если мы не ограничиваемся простейшими преобразованиями, то эта задача алгоритмически неразрешима. Алгоритмическая неразрешимость проявляется в бесконечных циклах суперкомпилятора SCP4, когда мы выбираем содержательную стратегию преобразований. С практической точки зрения, специализатор может ограничивать время своей работы, выдавая, по достижении этого времени, некоторой промежуточный вариант специализации (например, исходную программу) за результат этой работы.

Другой проблемой является квалификация пользователя специализатора: правильный выбор аргумента специализации (по которому нужно специализировать), описание этого аргумента в языке параметров, указание стратегий специализации[48] требуют как знания природы программы, которую он намерен специализировать, так и свойств самого специализатора. С. А. Романенко [13] предложил решать обе описанные выше проблемы посредством работы специализатора «на фоне» конкретной операционной системы. То есть нужно реализовать метапрограмму (по отношению к специализатору и пользователю), которая бы работала в операционной системе постоянно, следя за эволюцией программ на РЕФАЛе, собирая статистику использования этих программ (например, как часто используются, с какими аргументами, как часто изменяются и т. д.). На основании таких наблюдений эта метапрограмма должна автоматически формулировать задачи на специализацию, запускать специализатор на решение таких задач, отслеживать как его время работы, так и результат специализации. После чего, при попытке исполнения пользователем конкретной программы на РЕФАЛе, исполнять не исходную версию этой программы, а результат её специализации, если данное обращение к программе есть частный случай соответствующей задачи на специализацию.

В широко известной работе 1971 года [28] Ё. Футамура ставит важный вопрос «What kind of semantic metalanguage shall we use to describe an interpreter in order to achieve efficient partial evaluation of the interpreter?» Мне неизвестно (на 2007 год) вполне удовлетворительного ответа на этот вопрос. Хотя шаги в направлении понимания принципов построения такого языка и предпринимались как в контексте Лиспа [35, 42, 59], так и в контексте РЕФАЛа [66, 69, 71, 74], многими глубина проблемы не понимается — такой

[48] То есть правильной постановки задачи на специализацию.

язык воспринимается скорее как язык публикаций, а не как язык программирования. Мне неоднократно приходилось слышать подобное следующему: «Описания рекурсивных свойств параметров достаточно производить обычными средствами программирования» (т. е. через рекурсивную программу с последующим обращением к ней, как фильтру данного параметра). Автор смеет надеяться, что данная работа (см. главу 15) убеждает в обратном. Первоначальная идея В. Ф. Турчина при построении РЕФАЛа («Метаязык для описания алгоритмических языков» [17]) была направлена на то, что РЕФАЛ и должен играть роль такого метаязыка. Однако современные диалекты РЕФАЛа далеки в этом отношении от удовлетворительности. Некоторые начальные понятия метапрограммирования существуют в РЕФАЛе-5 (см. [69], раздел "Freezer"). Здесь уместно отметить работу Д. Хатклифа и Р. Глюка [35], которые предлагают (в терминах Лиспа) опробовать идеи В. Ф. Турчина [66, 69, 71, 74] на некотором интерпретаторе модельного языка. Развитие средств метапрограммирования (работа с параметрами) в РЕФАЛе-5, реализация текстового редактора и компилятора, поддерживающих эти средства [49], были бы важным вкладом в дальнейшее развитие суперкомпилятора SCP4, в особенности в контексте решения задач самоприменения.

РЕФАЛ как претендент на семантический метаязык обладает преимуществами в сравнении с потомками Лиспа: структура моноида на данных РЕФАЛа по операции приписывания расширяет набор средств для обработки текстов, позволяет в более простых терминах формулировать свойства программ и алгоритмы их преобразования (см., например, раздел 9.2 данной работы). С другой стороны, сами программы, написанные на РЕФАЛе, становится более сложными для анализа. Появляются содержательные соотношения между РЕФАЛ-выражениями, как следствия ассоциативности приписывания. Другие соотношения могут возникать и как свойства примитивов (встроенных функций) — например, коммутативность сложения, и как свойства исходного объекта специализации, данные *a priori* пользователем. Было бы интересно реализовать средства для вывода следствий из этих соотношений. Здесь привлекательным является алгоритм Кнута—Бендикса [43], позволяющий иногда добиваться поставленных целей намного проще, чем прямое использование инструментов суперкомпиляции [37].

Важной задачей является задача оптимизации РЕФАЛ-5 определения-«функции» — преобразования в рамках синтаксиса этого определения, т. е. без развёрток вызовов функций, к которым данное определение обращается. Точнее говоря, здесь правильно говорить о компиляторе из выходного подмножества суперкомпилятора языка РЕФАЛ-графов в язык сборки, производящего такую оптимизацию. Разработка такого компилятора ведётся [50]. В разделе 14.1 мы описали один из его инструментов. Функция F1935 (см. Приложение А), построенная суперкомпилятором SCP4, показывает

[49] Необходимо отметить, что некоторые эксперименты в этом направлении делались А. П. Коныщевым и В. А. Пинчук.

[50] Стартовую версию компилятора реализовал Гао Кси, студент университета города Ухань, Китай [33].

другую неэффективность, которую необходимо устранять: в процессе преобразований был построен излишне детальный входной формат этой функции

```
<F1935 s.489 (e.490) s.492 s.493 e.494>.
```

Каждый шаг этой функции сопоставляет значения трём последним форматным переменным, «расщепляя» конкатенацию этих значений, после чего при построении правых частей восстанавливает эту конкатенацию, и только для этого использует значения переменных `s.492`, `s.493`, `e.494`. Этот входной формат должен быть преобразован к формату

```
<F1935 s.489 (e.490) e.495>.
```

Заметим, что, с точки зрения промежуточных преобразований SCP4, необходимо строить именно наиболее точные форматы. Излишняя детальность проявляется лишь на этапе исполнения остаточной программы.

Необходимо реализовать в суперкомпиляторе SCP4 параллельную стратегию развития стека функций в процессе прогонки (см. раздел 4.2). То есть при существовании в текущей конфигурации нескольких параллельных вызовов функций, позволяющих исполнить соответствующие им шаги РЕФАЛ-машины транзитивно или полутранзитивно, все эти шаги должны быть выполнены перед ветвлением параметров в каком бы то ни было другом функциональном вызове из данной конфигурации. К сожалению, отсутствие такой возможности существенно ограничивает применение SCP4 в экспериментах по автоматическому доказательству простейших тождеств, например даже для конфигураций вида `<Equal (<F e.1>) (<F e.1>)>`.

Н. Джонс [42] предлагает определять оптимальность специализатора *Spec* по отношению к фиксированному интерпретатору *Int*, исходя из того, может ли *Spec* убрать все накладные расходы интерпретации любой программы P_0, когда мы специализируем *Int* по P_0. (Мы предполагаем, для простоты, что *Spec*, *Int*, P_0 написаны на одном и том же языке.) И формулирует это свойство синтаксически: результат специализации P_1 должен совпадать с P_0 с точностью до переименовок имён переменных и имён функций. Такое определение, продиктованное слабостью специализаторов, построенных в русле «частичных вычислений», абсолютно неприемлемо для суперкомпиляции. Оно накладывает ограничения на преобразования, например, запрещая перестановку и обрезание ветвей в P_1, или поглощение одних ветвей другими (см. главу 8). Интересно дать более адекватное определение оптимальности специализатора по выходу. Здесь должна действовать не группа переименовок, а скорее какая-то подгруппа простых преобразований. Заметим, что от зависимости определения Джонса от конкретного интерпретатора избавиться достаточно просто. Нужно брать тот интерпретатор *Int*, расширение которого всегда присутствует в исходных текстах любого содержательного специализатора *Spec*. Возможно, правильным определением будет следующее.

Назовём специализатор *Spec* оптимальным по выходу по отношению к интерпретатору *Int*, если для любой программы P_0 программа P_1, результат специализации *Int* по P_0, совпадает, с точностью до указанных выше переименовок, с результатом «специализации» самóй P_0, когда все её аргументы находятся в общем положении.

Таким образом, оптимальный по выходу специализатор должен «не замечать» накладных расходов интерпретации, а не только убирать их. Для каждого интерпретатора *Int*, очевидно, существует специализатор, оптимальный по отношению к *Int*. Существует ли хотя бы один специализатор, оптимальный по выходу, если мы избавимся от произвола выбора интерпретатора? Автор не знает ответа на этот вопрос.

Необходимо построить упрощённые модели суперкомпиляции и произвести сложностной анализ как по остаточным программам, так и самих алгоритмов преобразований. Реализация таких моделей может являться замечательным материалом для экспериментов с существующим суперкомпилятором SCP4.

Нам осталось отметить, что результаты экспериментов, приведённые в данной работе, соответствуют конкретной версии SCP4 и могут выглядеть слегка по-другому в других версиях. Кроме того, они представлены во «внешнем виде» — реальный выход SCP4 есть РЕФАЛ-граф, свойства которого часто не отражены во «внешнем» РЕФАЛ-виде. Список литературы не претендует на полноту.

Благодарности

Без Валентина Фёдоровича Турчина работа над суперкомпилятором SCP4 не только не могла бы состояться, но и не могла бы начаться. Я благодарен ему за поддержку. Я признателен Александру П. Конышеву за поддержку РЕФАЛа-5 и его вклад в реализацию системы SCP4, отмеченный уже мною выше. Блистательные идеи Александра В. Корлюкова находили своё воплощение в его экспериментах с суперкомпилятором SCP4 и стимулировали развитие SCP4. Я был счастлив работать вместе с этими людьми и буду всегда вспоминать наше дружеское общение.

Литература

1. *Абрамов С. М.* Метавычисления и их применение. М.: Наука; Физматлит, 1995.
2. Базисный РЕФАЛ и его реализация на вычислительных машинах. М.: ЦНИПИАСС, 1977.
3. *Гурин Р. Ф., Романенко С. А.* Язык программирования РЕФАЛ Плюс. М.: ИНТЕРТЕХ, 1991.
4. *Конышев А. П.* Компилятор из языка РЕФАЛ-графов в язык С: исходные тексты и демонстрация, 2000 [Электрон. ресурс] // www.botik.ru/pub/local/scp/refal5/.
5. *Корлюков А. В.* Пособие по суперкомпилятору SCP4, 1999 [Электрон. ресурс] // www.refal.net/supercom.htm.

6. *Корлюков А. В.* Получение формул в математике. Тезисы докладов VIII Белорусской математической конференции, 2000. С. 178 [Электрон. ресурс] // www.bsu.by/Converences/konf8/Sections/Abstr14/Korlukov/Korlukov.htm.
7. *Корлюков А. В.* Несколько примеров преобразований программ суперкомпилятором SCP4, 2001 [Электрон. ресурс] // www.refal.net/ korlukov/pearls/.
8. *Корлюков А. В.* Суперкомпилируем суперагентов // Альфа. Т. 1. Гродно: Гродненский государственный ун-т, 2001. С. 89–98.
9. *Косовский Н. К.* Основы теории элементарных алгоритмов. Л.: Изд-во ЛГУ, 1987.
10. *Лисица А. П., Немытых А. П.* Работа над ошибками: Программные системы: теория и приложения. Т. 1. М.: Физматлит, 2006. С. 195–232 (доступна на ftp://www.botik.ru/pub/local/scp/refal5/psta_errors2006.pdf).
11. *Лисица А. П., Немытых А. П.* Верификация как параметризованное тестирование (эксперименты с суперкомпилятором SCP4) // Программирование. М., 2007. Т. 33(1). С. 22–34.
12. *Магницкий Л. Ф.* Арифметика (Арифметика, сиречь наука числительная с разных диалектов на славенский язык переведеная и во едино собрана, и на две книги разделена). На славянском языке. М.: Государева типография, 1703.
13. *Романенко С. А.* О выборе аргументов специализации, 2000. Частное сообщение.
14. *Романенко С. А.* Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру / ИПМ им. М. В. Келдыша АН СССР. М., 1987. Препринт № 26.
15. *Романенко С. А.* Реализация РЕФАЛа-2 / ИПМ им. М. В. Келдыша АН СССР. М., 1987.
16. *Романенко С. А.* РЕФАЛ-4 — расширение РЕФАЛа-2, обеспечивающее выразимость прогонки / ИПМ им. М. В. Келдыша АН СССР. М., 1987. Препринт № 147.
17. *Турчин В. Ф.* Метаязык для формального описания алгоритмических языков (Цифровая вычислительная техника и программирование). М.: Советское радио, 1966. С. 116–119.
18. *Филд А., Харрисон П.* Функциональное программирование. М.: Мир, 1993.
19. *Шень А.* Программирование: теоремы и задачи. М.: МЦНМО, 1995.
20. *Эйлер Леонард.* Дифференциальное исчисление. М.; Л.: Государственное изд-во технико-теоретической литературы, 1949.
21. *Burstall R. M. and Darlington J.* A transformation system for developing recursive programs // Journal of the ACM. 1977. January. Vol. 24(1). ACM Press. P. 44–67.
22. *Chmutov S. V., Gaydar E. A., Ignatovich I. M., Kozadoy V. F., Nemytykh A. P. and Pinchuk V. A.* Implementation of the symbol analytic transformation language FLAC // LNCS. Proc. of DISCO'90. Vol. 429, Springer-Verlag, 1990. P. 276.
23. *Chmutov S. V., Nemytykh A. P., Gaydar E. A., Ignatovich I. M., Kozadoy V. F. and Pinchuk V. A.* The symbol analytic transformation language FLAC: sources, executable modules, 1991 (ftp://www.botik.ru/pub/local/scp/flac/flac386.zip).
24. *Dershovitz N. and Jouannaud J. P.* Rewrite Systems, Handbook of theoretical computer science / Ed. by J. van Leeuwen. Elsevier Science Publishers B. V., 1990. P. 241–320.
25. *Ershov A. P.* Mixed computation in the class of recursive program schema // Acta Cybernet. 1978. Vol. 4(1).
26. *Ershov A. P.* Mixed computation: potential applications and problems for study // Theoretical Computer Science. 1978. Vol. 18. P. 41–67.
27. *Field A. J. and Harrison P. G.* Functional Programming. Addison-Wesley Publishing Company, Inc., 1988.

28. *Futamura Y.* Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler // Systems. Computers. Controls. 1971. Vol. 2(5). P. 45–50 (An updated version of the paper was republished in J. Higher-Order and Symbolic Computation. 1999. Vol. 12. P. 381–391).
29. *Futamura Y.* Partial computation of programs // LNCS. Proc. of RIMS Symposium on Software Science and Engineering. Vol. 147. Springer-Verlag, 1983. P. 1–35.
30. *Futamura Y. and Nogi K.* Generalized partial computation // Proc. of the IFIP TC2 Workshop. Amsterdam: North-Holland Publishing Co., 1988. P. 133–151.
31. *Futamura Y., Nogi K. and Takano A.* Essence of generalized partial computation // Theoretical Computer Science. Vol. 90. Amsterdam: North-Holland Publishing Co., 1991. P. 61–79.
32. *Futamura Y., Konishi Z. and Glück R.* Program Transformation System Based on Generalized Partial Computation // New Generation Computing. Vol. 20. Ohmsha Ltd. and Springer-Verlag, 2002. P. 75–99.
33. *Gao Xi.* An optimizing Refal-5 compiler written in Refal-5, 2004 (www.botik.ru/pub/local/scp/refal5/).
34. *Gurevich Yu. and Shelah S.* Nearly Linear Time // LNCS. Proc. of Logic at Botik'89. Vol. 363. Springer-Verlag, 1989. P. 108–118.
35. *Hatcliff J. and Glück R.* Reasoning about Hierarchies of Online Program Specialization Systems // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 161–182.
36. *Higman G.* Ordering by divisibility in abstract algebras // Proc. London Math. Soc. 1952. 2. Vol. 2(7). P. 326–336.
37. *Huet G. and Oppen D.* Equations and rewrite rules: A survey // Formal languages: perspectives and open problems. New York: Pergamon Press, 1980. P. 349–405 (Рус. пер.: *Юэ Ж., Оппен Д.* Равенства и правила переписывания: Обзор // Математическая логика в программировании. М.: Мир, 1991. С. 176–232).
38. *Hughes J.* Type Specialization for the Lambda-Calculus; or a new Paradigm for Partial Evaluation Based on Type Inference // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 183–215.
39. *Jones N. D., Gomard C. K., and Sestoft P.* Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993.
40. *Jones N. D.* MIX ten years after // Proc. of the ACM SIGPLAN PEPM'95. ACM Press, 1995. P. 24–38.
41. *Jones N. D.* What not to do when writing an interpreter for specialization // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 216–237.
42. *Jones N. D.* Computability and Complexity from a Programming Perspective. The MIT Press, 1997.
43. *Knuth D. E. and Bendix B. P.* Simple Word Problem in Universal Algebras // Proc. of Computational Problems in Abstract Algebra. Pergamon Press, 1970. P. 263–297 (Proc. Conf., Oxford, 1967).
44. *Knuth D. E., Morris J. H. and Pratt V. R.* Fast Pattern Matching in strings // SIAM J. Comput. 1977. Vol. 6(2). P. 323–350.
45. *Korlyukov A. V. and Nemytykh A. P.* Supercompilation of Double Interpretation. (How One Hour of the Machine's Time Can Be Turned to One Second). Vol. 1, 2004. P. 123–150. Вестник национального технического университета «Харьковского политехнического института», Харьков [Электрон. ресурс] // www.refal.net/ korlukov/scp2int/Karliukou_Nemytykh.pdf; исходные тексты и демонстрация: www.refal.net/ korlukov/demo_scp4xslt.zip.

46. *Kruskal J. B.* Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture // Trans. Amer. Math. Society. 1960. March. Vol. 95. P. 210–225.
47. *Lawall J. L.* Faster Fourier Transforms via Automatic Program Specialization // Proc. the Lecture notes of the DIKU International Summer School'98 on Partial Evaluation: Practice and Theory. Copenhagen, Denmark, 1998 (www.diku.dk/users/julia/fft.ps.gz).
48. *Leuschel M.* Homeomorphic Embedding for Online Termination // LNCS. Proc. of 8th Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR). Vol. 1559, Springer-Verlag. P. 199–218.
49. *Lisitsa A., and Nemytykh A. P.* Verification of parameterized systems using supercompilation. A case study // Proc. of the Third Workshop on Applied Semantics (APPSEM05) / M. Hofmann, H. W. Loidl (Eds.). Fraunchiemsee, Germany. Ludwig Maximillians Universitat Munchen, September 2005 (ftp://www.botik.ru/pub/local/scp/refal5/appsem_verification2005.ps).
50. *Mogensen T.* Evolution of Partial Evaluators: Removing Inherited Limits // LNCS. Proc. of Partial Evaluation, International Seminar / O. Danvy, R. Glueck, P. Thiemann (Eds.). Vol. 1110. Springer-Verlag, 1996. P. 303–321.
51. *Nemytykh A. P.* Supercompiler SCP4: Use of quasi-distributive laws in program transformation // Wuhan University Journal of Natural Sciences. Proc. International Software Engineering Symposium. Vol. 95(1–2). Wuhan, China, March 2001. P. 375–382.
52. *Nemytykh A. P.* A Note on Elimination of Simplest Recursions // Proc. of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation. ACM Press. Aizu, Japan, 2002. P. 138–146.
53. *Nemytykh A. P.* The Supercompiler SCP4: General Structure (extended abstract) // LNCS. Proc. of the Perspectives of System Informatics. Vol. 2890. Springer-Verlag, 2003. P. 162–170 (ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_PSI03.ps.gz).
54. *Nemytykh A. P.* Playing on REFAL // Proc. of the International Workshop on Program Understanding. A. P. Ershov Institute of Informatics Systems. Siberian Branch of Russian Academy of Sciences. Novosibirsk. Altai Mountains. July 2003. P. 29–39 (www.botik.ru/pub/local/scp/refal5/korlyukov.html).
55. *Nemytykh A. P.* The Supercompiler SCP4: General Structure. Vol. 1 // Программные системы: теория и приложения. М.: Физматлит, 2004. С. 448–485 (ftp://ftp.botik.ru/pub/local/scp/refal5/GenStruct.ps.gz).
56. *Nemytykh A. P. and Pinchuk V. A.* Program Transformation with Metasystem Transitions: Experiments with a Supercompiler // LNCS. Proc. of the Perspectives of System Informatics. Vol. 1181. Springer-Verlag. 1996. P. 249–260 (ftp://ftp.botik.ru/pub/local/APP/meta-trans.ps.gz).
57. *Nemytykh A. P., Pinchuk V. A. and Turchin V. F.* A Self-Applicable Supercompiler // LNCS. Proc. of Partial Evaluation, International Seminar / O. Danvy, R. Glueck, P. Thiemann (Eds.). Vol. 1110. Springer-Verlag, 1996. P. 322–337 (ftp://ftp.botik.ru/pub/local/APP/self-appl.ps.gz).
58. *Nemytykh A. P. and Turchin V. F.* The Supercompiler SCP4: sources, on-line demonstration, 2000 (http://www.botik.ru/pub/local/scp/refal5/).
59. *Nielson F. and Nielson H. R.* Multi-Level Lamda-Calculi: An Algebraic Description // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 338–354.
60. *Pettorossi A. and Proietti M.* Transformation of logic programs: Foundations and techniques // Journal of Logic Programming. 1994. Vol. 19, 20. P. 261–320.
61. *Pettorossi A. and Proietti M.* A Comparative Revisitation of Some Program Transformation Techniques // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 355–385.

62. *Romanenko S. A.* Arity raiser and its use in program specialization // LNCS. Proc. the ESOP'90. Vol. 432. Springer-Verlag, 1990. P. 341–360.
63. *Sestoft P.* The structure of a self-applicable partial evaluator // LNCS. Proc. the Programs as Data Objects. Vol. 217. Springer-Verlag, 1986. P. 236–256.
64. *Sørensen M. H. and Glück R.* An algorithm of generalization in positive supercompilation // Proc. Logic Programming: Proceedings of the 1995 International Symposium. MIT Press, 1995. P. 486–479.
65. *Thibault S., Bercot L., Consel C., Marlet R. Muller G. and Lawall J.* Experiments in Program Compilation by Interpreter Specialization // IRISA Research Report 1212. 1998. December (ftp://ftp.irisa.fr/techreports/1998/PI–1212.ps.gz).
66. *Turchin V. F.* The Language Refal, the Theory of Compilation and Metasystem Analysis // Courant Computer Science Report #20. New York University, 1980.
67. *Turchin V. F.* The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. Vol. 8. ACM Press, 1986. P. 292–325.
68. *Turchin V. F.* The algorithm of generalization in the supercompiler // Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation. North-Holland Publishing Co., 1988. P. 531–549.
69. *Turchin V. F.* Refal-5, Programming Guide and Reference Manual. Holyoke, Massachusetts: New England Publishing Co., 1989 (www.botik.ru/pub/local/scp/refal5/, 2000).
70. *Turchin V. F.* Metacomputation in the language Refal. 1990 (unpublished, private communication).
71. *Turchin V. F.* Program Transformation with Metasystem Transitions // J. of Functional Programming. 1993. Vol. 3(3). P. 283–313.
72. *Turchin V. F.* Metacomputation: Metasystem transition plus supercompilation // LNCS. Proc. of the PEPM'96. Vol. 1110. Springer-Verlag, 1996. P. 481–509.
73. *Turchin V. F.* Supercompilation: Techniques and results // LNCS. Proc. of Perspectives of System Informatics. Vol. 1181. Springer-Verlag, 1996. P. 227–248.
74. *Turchin V. F. and Nemytykh A. P.* Metavariables: Their implementation and use in Program Transformation: Technical Report CSc. TR 95–012. City College of the City University of New York, 1995.
75. *Turchin V. F., Turchin D. V., Konyshev A. P. and Nemytykh A. P.* Refal-5: sources, executable modules, 2000 (www.botik.ru/pub/local/scp/refal5/).
76. Turing Machine Multiplication-Code, 1990 (www.ams.org/new-in-math/cover/turing_multiply_code.html).
77. *Wadler P.* Deforestation: Transforming programs to eliminate tree // Theoretical Computer Science. 1990. Vol. 73. P. 231–238.

Приложение А. Специализация интерпретатора МТ по программе умножения натуральных чисел

Ниже приведена программа умножения двух натуральных чисел, написанная на языке L Тьюринга (см. пример 10 раздела 16.1). Программа взята из [76]. Натуральное число n представлено на ленте строкой из $(n+1)$-й единицы '1'. В начальном состоянии множители даны на ленте в виде $\_n_1\_n_2\_$,

где _ есть пробел **B** (см. 16.1), а головка указывает на самую левую ячейку с единицей. После окончания работы алгоритма результат записан справа от второго множителя и отделён от него пробелом, головка снова указывает на начало первого множителя:

$$\underset{\uparrow}{\_n_1\_n_2\_} \Rightarrow \underset{\uparrow}{\_n_1\_n_2\_result\_}$$

```
 Multiplication ::=
(
                        /* пометить первый бит 1-го аргумента */
 (start        '1'  W  move1right  right)
                        /* движение вправо за 1-й аргумент    */
 (move1right   '1' '1' move1right  right)
               /* найдена ячейка между 1-м и 2-м аргументами */
 (move1right    B   B  mark2start  right)
                        /* пометить первый бит 2-го аргумента */
 (mark2start   '1'  Y  move2right  right)
                           /* движение вправо за 2-й аргумент */
 (move2right   '1' '1' move2right  right)
         /* найдена ячейка между 2-м аргументом и результатом*/
 (move2right    B   B  initialize  right)
                           /* лидирующая 1-ца результата       */
 (initialize    B  '1' backup      left)
         /* возвращение к первому слева неиспользованному
                                              биту 1-го аргумента */
 (backup        B   B  backup      left)
 (backup       '1' '1' backup      left)
 (backup        Z   Z  backup      left)
 (backup        Y   Y  backup      left)
                           /* готовы начать следующий проход   */
 (backup        X   X  nextpass    right)
 (backup        W   W  nextpass    right)
                           /* если пробел, то результат готов, */
 (nextpass      B   B  finishup    right)
                           /* иначе пометить бит               */
 (nextpass     '1'  X  findarg2    right)
                           /* движение за 1-й аргумент         */
 (findarg2     '1' '1' findarg2    right)
               /* найдена ячейка между 1-м и 2-м аргументами */
 (findarg2      B   B  findarg2    right)
                           /* начало 2-го арг., копируем его   */
 (findarg2      Y   Y  testarg2    right)
               /* если пробел, то результат этого цикла готов, */
 (testarg2      B   B  cleanup2    left)
/* иначе пометить бит, найти результат и приписать к нему 1-цу */
```

```
(testarg2    '1'  Z  findans     right)
                        /* движение по 2-ому аргументу      */
(findans     '1' '1' findans     right)
       /* найдена ячейка между 2-м аргументом и результатом*/
(findans      B   B  atans       right)
                        /* движение по результату           */
(atans       '1' '1' atans       right)
           /* приписать 1-цу к результату и вернуться назад */
(atans        B  '1' backarg2    left)
      /* возвращение к первому слева неиспользованному
                                        биту 2-го аргумента */
(backarg2    '1' '1' backarg2    left)
(backarg2     B   B  backarg2    left)
                        /* проверить 2-й аргумент           */
(backarg2     Z   Z  testarg2    right)
(backarg2     Y   Y  testarg2    right)
                        /* движение влево по результату     */
(cleanup2    '1' '1' cleanup2    left)
       /* найдена ячейка между 2-м аргументом и результатом*/
(cleanup2     B   B  cleanup2    left)
                       /* восстановить биты 2-го аргумента */
(cleanup2     Z  '1' cleanup2    left)
/* биты восстановлены, вернуться для начала следующего прохода */
(cleanup2     Y   Y  backup      left)
                        /* восстановить 1-й бит 2-го арг. */
(finishup     Y  '1' finishup    left)
    /* 2-й аргумент восстановлен, перейти к 1-ому аргументу */
(finishup     B   B  finishup    left)
                        /* восстановить биты 1-го аргумента */
(finishup     X  '1' finishup    left)
                     /* восстановить 1-й бит 1-го аргумента */
(finishup     W  '1' almostdone  left)
(almostdone   B   B  stop        right)
);
```

В результате специализации MT по Multiplication терминология программы языка L исчезла, т. е. произошла компиляция в Реафал-5. Однако результат показывает и недостатки этой компиляции. О некоторых из них мы уже указали в главе 20. Сразу после остаточной программы показаны упрощённые, построенные руками, варианты двух её функций; естественное пожелание к SCP4, чтобы он сам производил подобные преобразования.

```
* InputFormat: <Go e.41>
$ENTRY Go {
 (e.101) ('1') (s.105 e.104) = <F23 s.105 W (e.101) e.104>;
}

* InputFormat: <F2232 s.560 s.561 s.562 s.563 (e.564) e.566>
```

```
F2232 {
 '1' s.561 s.562 s.563 (e.564) s.568 e.566
           = <F2232 s.568 '1' s.561 s.562 (e.564 s.563) e.566>;
B s.561 s.562 s.563 (e.564) e.566
           = <F1672 s.561 (e.564 s.563 s.562) '1' e.566>;
}

* InputFormat: <F2146 s.543 s.544 s.545 (e.546) e.548>
F2146 {
 '1' s.544 s.545 (e.546) s.550 e.548
                 = <F2146 s.550 '1' s.544 (e.546 s.545) e.548>;
 B s.544 s.545 (e.546) s.557 e.548
                 = <F2232 s.557 B s.544 s.545 (e.546) e.548>;
}

* InputFormat: <F1935 s.489 (e.490) s.492 s.493 e.494>
F1935 {
 '1' (e.490 s.495) s.492 s.493 e.494
                 = <F1935 s.495 (e.490) '1' s.492 s.493 e.494>;
 B (e.490 s.501) s.492 s.493 e.494
                 = <F1935 s.501 (e.490) B s.492 s.493 e.494>;
 Z (e.490 s.512) s.492 s.493 e.494
                 = <F1935 s.512 (e.490) '1' s.492 s.493 e.494>;
 Y (e.490 s.528) s.492 s.493 e.494
                 = <F200 s.528 (e.490) Y s.492 s.493 e.494>;
}

* InputFormat: <F1672 s.456 (e.457) s.459 e.460>
F1672 {
 '1' (e.457 s.461) s.459 e.460
                     = <F1672 s.461 (e.457) '1' s.459 e.460>;
  B  (e.457 s.466) s.459 e.460
                     = <F1672 s.466 (e.457) B s.459 e.460>;
  Z  (e.457 s.476) B     e.460
                     = <F1935 s.476 (e.457) '1' B e.460>;
  Z  (e.457)   '1' s.533 e.460 = <F2146 s.533 Z Z (e.457) e.460>;
  Y  (e.457 s.589) B     e.460 = <F200 s.589 (e.457) Y B e.460>;
  Y  (e.457)   '1' s.594 e.460 = <F2146 s.594 Z Y (e.457) e.460>;
}

* InputFormat:
*      <F1383 s.406 s.407 s.408 s.409 s.410 s.411 (e.412) e.414>
F1383 {
'1' s.407 s.408 s.409 s.410 s.411 (e.412) s.415 e.414
 = <F1383 s.415 '1' s.407 s.408 s.409 s.410 (e.412 s.411) e.414>;
B s.407 s.408 s.409 s.410 s.411 (e.412) e.414
```

```
 = <F1672 s.407 (e.412 s.411 s.410 s.409 s.408) '1' e.414>;
}

* InputFormat:
*       <F1297 s.386 s.387 s.388 s.389 s.390 (e.391) e.393>
F1297 {
 '1' s.387 s.388 s.389 s.390 (e.391) s.396 e.393
      = <F1297 s.396 '1' s.387 s.388 s.389 (e.391 s.390) e.393>;
  B s.387 s.388 s.389 s.390 (e.391) s.404 e.393
      = <F1383 s.404 B s.387 s.388 s.389 s.390 (e.391) e.393>;
}

* InputFormat: <F1095 s.355 s.356 s.357 (e.358) e.360>
F1095 {
 '1' s.356 s.357 (e.358) s.361 e.360
                 = <F1095 s.361 '1' s.356 (e.358 s.357) e.360>;
 B s.356 s.357 (e.358) s.367 e.360
                 = <F1095 s.367 B s.356 (e.358 s.357) e.360>;
 Y s.356 s.357 (e.358) B e.360
                 = <F200 s.356 (e.358 s.357) Y B e.360>;
 Y s.356 s.357 (e.358) '1' s.384 e.360
                 = <F1297 s.384 Z Y s.356 s.357 (e.358) e.360>;
}

* InputFormat: <F589 s.240 (e.241) s.243 s.244 s.245 e.246>
F589 {
 Y (e.241 s.248) s.243 s.244 s.245 e.246,
         <F589 s.248 (e.241) '1' s.243 s.244
               s.245 e.246>:e.657 s.658 s.659 s.660 (e.661)
                           = e.657 s.658 s.659 s.660 (e.661);
 B (e.241 s.255) s.243 s.244 s.245 e.246,
         <F589 s.255 (e.241) B s.243 s.244
               s.245 e.246>:e.662 s.663 s.664 s.665 (e.666)
                           = e.662 s.663 s.664 s.665 (e.666);
 X (e.241 s.268) s.243 s.244 s.245 e.246,
       <F589 s.268 (e.241) '1' s.243
             s.244 s.245 e.246>:e.667 s.668 s.669 s.670 (e.671)
                           = e.667 s.668 s.669 s.670 (e.671);
W (e.241 B) s.243 s.244 s.245 e.246
                           = e.241 s.243 s.244 s.245 (e.246);
}

* InputFormat: <F200 s.160 (e.161) s.163 s.164 e.165>
F200 {
 B (e.161 s.166) s.163 s.164 e.165
                  = <F200 s.166 (e.161) B s.163 s.164 e.165>;
```

```
'1' (e.161 s.172) s.163 s.164 e.165
                = <F200 s.172 (e.161) '1' s.163 s.164 e.165>;
Z (e.161 s.183) s.163 s.164 e.165
                = <F200 s.183 (e.161) Z s.163 s.164 e.165>;
Y (e.161 s.199) s.163 s.164 e.165
                = <F200 s.199 (e.161) Y s.163 s.164 e.165>;
X (e.161 s.220) B Y e.165,
        <F589 s.220 (e.161) '1' B
              '1' e.165>:e.309 s.310 s.311 s.312 (e.313)
                = (e.309 B) ('1') (s.310 s.311 s.312 e.313);
X (e.161 s.317) B B e.165,
        <F589 s.317 (e.161) '1' B
              B e.165>:e.318 s.319 s.320 s.321 (e.322)
                = (e.318 B) ('1') (s.319 s.320 s.321 e.322);
X (e.161 s.329) B X e.165,
        <F589 s.329 (e.161) '1' B
              '1' e.165>:e.330 s.331 s.332 s.333 (e.334)
                = (e.330 B) ('1') (s.331 s.332 s.333 e.334);
X (e.161) B W e.165 = (e.161 X B) ('1') (e.165);
X (e.161) '1' s.164 e.165 = <F1095 s.164 X X (e.161) e.165>;
W (e.161 B) B Y e.165 = (e.161 B) ('1') (B '1' e.165);
W (e.161 B) B B e.165 = (e.161 B) ('1') (B B e.165);
W (e.161 B) B X e.165 = (e.161 B) ('1') (B '1' e.165);
W (e.161) B W e.165 = (e.161 W B) ('1') (e.165);
W (e.161) '1' s.164 e.165 = <F1095 s.164 X W (e.161) e.165>;
}

* InputFormat: <F76 s.120 s.121 s.122 s.123 (e.124) e.126>
F76 {
 '1' s.121 s.122 s.123 (e.124) s.129 e.126
             = <F76 s.129 '1' s.121 s.122 (e.124 s.123) e.126>;
B s.121 s.122 s.123 (e.124) B e.126
             = <F200 s.121 (e.124 s.123 s.122) B '1' e.126>;
}

* InputFormat: <F23 s.107 s.108 (e.109) e.111>
F23 {
 '1' s.108 (e.109) s.112 e.111
                 = <F23 s.112 '1' (e.109 s.108) e.111>;
  B  s.108 (e.109) '1' s.118 e.111
                 = <F76 s.118 Y B s.108 (e.109) e.111>;
}
************************ The End ******************************
```

Функции F589 и F200 могут быть упрощены до:

```
* InputFormat: <F589 s.240 (e.241) s.243 s.244 s.245 e.246>
```

```
F589 {
 Y (e.241 s.248) s.243 s.244 s.245 e.246
             = <F589 s.248 (e.241) '1' s.243 s.244 s.245 e.246>;
 B (e.241 s.255) s.243 s.244 s.245 e.246
             = <F589 s.255 (e.241) B s.243 s.244 s.245 e.246>;
 X (e.241 s.268) s.243 s.244 s.245 e.246
             = <F589 s.268 (e.241) '1' s.243 s.244 s.245 e.246>;
 W (e.241 B) s.243 s.244 s.245 e.246
             = e.241 s.243 s.244 s.245 (e.246);
}

* InputFormat: <F200 s.160 (e.161) s.163 s.164 e.165>
F200 {
 B (e.161 s.166) s.163 s.164 e.165
             = <F200 s.166 (e.161) B s.163 s.164 e.165>;
 '1' (e.161 s.172) s.163 s.164 e.165
             = <F200 s.172 (e.161) '1' s.163 s.164 e.165>;
 Z (e.161 s.183) s.163 s.164 e.165
             = <F200 s.183 (e.161) Z s.163 s.164 e.165>;
 Y (e.161 s.199) s.163 s.164 e.165
             = <F200 s.199 (e.161) Y s.163 s.164 e.165>;
 X (e.161 s.220) B Y e.165
             = <SS <F589 s.220 (e.161) '1' B '1' e.165>>;
 X (e.161 s.317) B B e.165
             = <SS <F589 s.317 (e.161) '1' B B e.165>>;
 X (e.161 s.329) B X e.165
             = <SS <F589 s.329 (e.161) '1' B '1' e.165>>;
 X (e.161) B W e.165 = (e.161 X B) ('1') (e.165);
 X (e.161) '1' s.164 e.165 = <F1095 s.164 X X (e.161) e.165>;
 W (e.161 B) B Y e.165 = (e.161 B) ('1') (B '1' e.165);
 W (e.161 B) B B e.165 = (e.161 B) ('1') (B B e.165);
 W (e.161 B) B X e.165 = (e.161 B) ('1') (B '1' e.165);
 W (e.161) B W e.165 = (e.161 W B) ('1') (e.165);
 W (e.161) '1' s.164 e.165 = <F1095 s.164 X W (e.161) e.165>;
}

SS {
 e.309 s.310 s.311 s.312 (e.313)
             = (e.309 B) ('1') (s.310 s.311 s.312 e.313);
}
```