Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Алгоритм прогонки вызовов рекурсивных функций, не приводящий к зацикливанию в компиляторе Рефала-5λ

Студент ИУ9-82Б (Группа) _____ (Подпись, дата) М. Д. Апахов (И.О.Фамилия)

Руководитель ВКР _____ (Подпись, дата) А. В. Коновалов (И.О.Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О.Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О.Фамилия)

Нормоконтролер _____ (Подпись, дата) _____ (И.О.Фамилия)

2021 г.

АННОТАЦИЯ

Темой данной работы является «Алгоритм прогонки вызовов рекурсивных функций, не приводящий к зацикливанию в компиляторе Рефала-5λ». Объем работы составляет 55 страниц.

Основной объект исследования – алгоритм ациклической суперкомпиляции при оптимизации прогонки в компиляторе языка Рефала-5λ. Рассматривается построение и использование данного алгоритма. С его помощью алгоритм прогонки становится более гибким и эффективным.

Дипломная работа состоит из трех глав. Первая глава посвящена теоретическим сведениям, необходимым для реализации алгоритма прогонки, результатам исследования алгоритма суперкомпиляции. В главе “Разработка” описываются модификации компилятора языка, необходимые для осуществления изменений существующего алгоритма, а также тонкости его реализации. В главе “Тестирование” рассматриваются результаты применения нового алгоритма оптимизации прогонки к самому компилятору и результаты работы на тестовых файлах.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В разработке программного обеспечения одними из важнейших критериев качества продукта являются производительность программы и сопровождаемость исходного кода. Производительность подразумевает высокую скорость работы программ, которая может обеспечиваться как за счет использования правильного подбора алгоритмов и структур данных для определенных задач, так и за счет использования знаний о внутреннем устройстве компилятора или интерпретатора, которые позволяют за счет использования различных подходов в коде реализации алгоритмов получать различные в производительности решения. Сопровождаемость же подразумевает атрибуты программного обеспечения, относящиеся к усилиям, необходимым для диагностики недостатков или случаев отказов или определения составных частей для модернизации.

Зачастую описанные выше критерии вступают в противоречие так как использование различных тактик для увеличения производительности кода приводит к увеличению сложности и запутанности кода. Именно поэтому во всех продвинутых компиляторах исходный код не просто переводится в машинный код полностью повторяя то, что написал программист, а проводятся различные оптимизации кода, которые позволяют ускорить программу.

Одними из таких оптимизаций в компиляторе Рефала-5λ являются прогонка[11] и встраивание функций, однако реализованный в них подход оказался несколько ограниченным. Он предполагал, что на проходе прогонки в каждой правой части берется очередной вызов, который можно оптимизировать, и для него выполняется прогонка или встраивание, проходы повторяются до неподвижной точки — пока в оптимизируемых функциях не кончатся вызовы, которые можно оптимизировать или до исчерпания предела количества итераций прогонки. Цель данной работы – устранение необходимости в итеративных попытках оптимизировать очередной вызов, вместо этого

реализовать алгоритм позволяющий оптимизировать вызов сразу полностью за одну итерацию прогонки.

В рамках работы требуется решить следующие задачи:

1. Модификация алгоритма оптимизации прогонки и встраивания вызовов функций в компиляторе языка Рефал-5λ.
2. Оптимизация самоприменимого компилятора языка Рефал-5λ с помощью разработанных решений.
3. Оценка результатов оптимизации прогонки и встраивания функций.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор языка Рефал-5λ

1.1.1 Введение

Рефал (Рекурсивных Функций Алгоритмический язык) – функциональный язык программирования, ориентированный на осуществление символьных вычислений.

Язык Рефал-5λ [1] — точное надмножество Рефала-5. Это означает, что корректные программы написанные на Рефале-5 могут быть успешно скомпилированы компилятором Рефала-5λ без изменения семантических свойств программы. Основным отличием Рефала-5λ по сравнению с Рефалом-5 является поддержка анонимных функций, которые и дали название языку. Поддержка анонимных функций была бы бесполезна в компиляторе, если бы Рефал-5λ не поддерживал функции высшего порядка, которые также являются расширением по сравнению с Рефалом-5.

1.1.2 Символы и выражения

Символ в Рефале — базовый минимальный синтаксический элемент, используемый для представления данных в языке. Перечислим виды символов:

1. Идентификатор. Последовательность знаков, которая включает в себя буквенные символы, цифры, дефисы и нижние подчеркивания. Или заключенная в двойные кавычки последовательность знаков, которая может дополнительно включать пробелы и прочие символы с помощью управляющих последовательностей, начинающихся с символа обратной косой черты. Примеры корректных идентификаторов: `"x1", x1, x-y, Y5t66, catch-22, Hit-and-run, "the last stage"`.
2. Макроцифры. Целые числа от 0 до $2^{32} - 1$.
3. Литеры.

4. Указатели на функции. Идентификатор, который является именем функции определенной в области значения, с предшествующим знаком амперсанда. Пример: `&Map`.

Для организации структур данных в языке используются структурные скобки. Структурные скобки могут быть вложенными. Пример выражения со структурными скобками: `(A(B)())(C)D`. Так же существуют именованные структурные скобки, использующиеся для построения абстрактных типов данных. Они представляют из себя выражение в квадратных скобках, первым символом которых является идентификатор, именующий абстрактный тип данных, представляющий из себя имя функции в текущей области видимости. Пример выражения с именованными скобками: `[Refal 'Version' 'Commit']`.

Объектным выражением – именуется последовательность некоторых термов, в которой каждый терм является либо произвольным символом, либо выражением в именованных или структурных скобках. Также объектным выражением является выражение, которое не содержит ни одного символа, оно называется пустым выражением.

1.1.3 Сопоставление с образцом

Еще один тип выражений, представленный в языке — выражения образцы. В отличие от объектных выражений, выражения образцы могут содержать переменные. У переменной есть тип и идентификатор. Язык Рефал-5λ позволяет объявлять переменные трёх типов:

- s-переменные – значением этого типа переменной является один и только один символ;
- t-переменные – значением этого типа переменной является произвольный терм;
- e-переменные – значением этого типа переменной является произвольное объектное выражение;

Переменные имеют следующий синтаксис: type.name, где type – тип переменной, а name – идентификатор или целое положительное число. Выражение-образец также определяет множество объектных выражений языка, которые могут быть подставлены в данный образец при условии, что все значения переменных будут допустимыми. Декомпозицию объектного выражения с присваиванием значений переменным из выражения-образца называют сопоставлением с образцом [2]. Приведем пример декомпозиции.

Пусть имеется выражение образец:

```
s.1 e.2 s.1
```

В этом случае все объектные выражения, которые успешно сопоставятся с данным образцом, состоят из выражений, содержащих одинаковые первый и последний символы. Рассмотрим следующее объектное выражение:

```
A (A B C () (E) A
```

При сопоставлении выражения с образцом переменная `s.1` примет значение `A`, а переменная `e.2` – `(ABC()(E)`. Если невозможно подобрать такие значения, которые могут принять переменные с учетом накладываемых ограничений, то сопоставление оказывается неудачным. Например рассмотрим объектное выражение `X Y` и выражение-образец `s.A s.B s.C`. В данном случае сопоставление очевидно завершится неудачей, так как в множестве объектных выражений, которые могут успешно сопоставиться с данным выражением, присутствуют только выражения, состоящие из трех символов.

1.1.4 Функции в языке Рефал-5λ

Любая программа на Рефале-5λ представляет собой набор *функций* [3]. Точкой входа является функция `Go` (предваренная директивой `$ENTRY`) с пустым аргументом. Определение функции записывается как имя функции, за которым следует блок – тело функции, ограниченное фигурными скобками. Тело функции в общем случае состоит из нескольких *предложений*, разделенных точкой с запятой. Предложение – это правило, определяющее, как построить значение функции на некотором подмножестве её аргументов. Любое

предложение состоит из двух частей – левой части, выражения-образца, описывающей подмножество значений аргумента функции, на котором это предложение применимо, и правой части, *результатного выражения*, описывающей значение функции на этом подмножестве. Левая и правая части разделяются знаком равенства (см. Листинг 1.1).

```
$ENTRY Имя Функции {
  образец1 = результат1;
  образец2 = результат2;
  …
  образецN = результатN;
}
```

Листинг 1.1 Общий вид функции.

Модификатор `$ENTRY` используется для указания области видимости функции. Если у функции он отсутствует, то функция является локальной и может быть использована только в рамках одной единицы трансляции. Если же данный флаг присутствует, функция является глобальной и может быть использована в других местах.

Результатное выражение отличается от выражения-образца тем, что может включать в себя еще и вызовы функций:

```
FunctionCall ::= <FunctionName Arg>
```

Аргументами вызовов функций могут являться символы и переменные, раннее определенные в левой части предложения, а также вызовы функций. Также результатное выражение, может включать в себя набор присваиваний.

Пример функции, написанной на языке Рефал-5λ описан в листинге 1.2.

```
IsPalindrome {
      s.1 e.m s.1 = <IsPalindrome e.m>;
      s.1 = True;
      /* empty */ = True;
      e.Other = False;
}
```

Листинг 1.2 Пример функции в языке Рефал-5λ

В данном примере представлена функция, которая определяет, является ли последовательность символов палиндромом. При этом в языке Рефал-5λ, как и в

почти любом функциональном языке, описание функций очень близко к математическому описанию алгоритмов. В данном случае правила такие:

- Если последовательность имеет префикс и суффикс из одного символа, которые при этом равны, то последовательность палиндром, если подпоследовательность без данных префикса и суффикса является палиндромом.
- Последовательность из одного символа палиндром.
- Пустая последовательность палиндром.
- Любая другая последовательность не палиндром.

1.2 Абстрактная Рефал-машина

1.2.1 Обзор работы

Процесс выполнения программы, описанной на языке Рефал-5λ, можно описать с помощью абстрактной Рефал-машины [4].

Полем зрения называется рабочая память абстрактной Рефал машины. В поле зрения всегда находятся определенные выражения. Определенные выражения, в отличие от объектных, могут содержать в себе вызовы функций с аргументом, который является объектным выражением.

Исполнение программы на Рефале-5λ является конечной последовательностью шагов в этой машине.

Последовательность шагов — последовательность чередующихся фаз анализа и синтеза.

1.2.2 Фаза анализа

Пусть на этом этапе в поле зрения находится определенное выражение вида $E = s_1 \dots s_k < Fa_0 > s_{k+1} \dots s_n$, а функция `F` определяется набором предложений вида:

```
L1 = R1
L2 = R2
   …
Ln = Rn
```

Фаза анализа разбивается на более мелкие шаги:

1. В поле зрения Рефал машиной выбирается активный вызов функции. Вызов функции называется активным, если его закрывающая скобка находится ближе всего к началу выражения. Например в определённом выражении `<A <B>><C>` активным вызовом считается `<B>`. Пусть в определенном выражении *E* активный вызов $< Fa_0 >$.

2. Для каждого из предложений функции последовательно осуществляется сопоставление с образцом. Нужно заметить, что в некоторых случаях решений по сопоставлению с образцом оказывается несколько. При этом e-переменные, для которых существует несколько решений, называются открытыми переменными.

   Рефал-машина однозначно решает, какие значения должны быть присвоены открытым переменным по следующему правилу. Из всех возможных решений выбираются такие, что первая открытая переменная имеет наименьшую длину, из оставшихся выбирается такое, что вторая открытая переменная имеет наименьшую длину. Так продолжается, пока не останется одно единственное решение. Такой выбор решения является однозначным и не приводит к возможности появлению двух (и более) решений, в которых всем соответствующим открытым переменным соответствуют значения с одинаковыми длинами, так как в таком случае решения бы полностью совпадали.

   Если невозможно подобрать решение, то Рефал-машина аварийно завершается с ошибкой “Отождествление невозможно”.

Выходными данными фазы анализа является предложение вида $L_j = R_j$ и набор подстановок ко всем переменным, которые входят в выражение L_j.

1.2.3 Фаза синтеза

Пусть имеется предложение вида $L_j = R_j$ и набор подстановок для всех переменных, содержащихся в L_j, а в поле зрения Рефал-машины находится выражение $E = s_1 \dots s_k < Fa_0 > s_{k+1} \dots s_n$. На данном этапе происходит применение подстановок в правую часть предложения R_j, а затем преобразованное выражение R_c вставляется в поле зрения Рефал-машины

вместо вызова функции F. Поле зрения приобретает вид: $E = s_1 \ldots s_k \; R_c \; s_{k+1} \ldots s_n$.

В начале работы абстрактной Рефал-машины в ее поле зрения находится функция Go, которая является входной точкой в программу, реализованную на языке Рефал-5λ.

Рефал машина продолжает свою работу, пока не возникнет ошибки, либо пока в поле зрения не закончатся вызовы функции, в таком случае определенное выражение находящееся в поле зрения становится результатом работы Рефал-машины.

Пример последовательного выполнения программы, исходный код которой приведен на Листинге 1.3, Рефал-машиной представлен на Листинге 1.4.

```
$ENTRY Go {
      = <IsPalindrome 'ABACBA'>
}

IsPalindrome {
      s.1 e.m s.1 = <IsPalindrome e.m>;
      s.1 = True;
      /* empty */ = True;
      e.Other = False;
}
```

Листинг 1.3 Пример программы

```
Поле зрения: <Go>
Шаг 1. Анализ
Предложение 1, Нет присваиваний
Шаг 2. Синтез
Поле зрения: <IsPalindrome 'ABACBA'>
Шаг 3. Анализ
Предложение 1, Присваивания 'A' ← s.1, 'BACB' ← e.m
Шаг 4. Синтез
Поле зрения: <IsPalindrome 'BACB'>
Шаг 5. Анализ
Предложение 1, Присваивания 'B' ← s.1, 'AC' ← e.m
Шаг 6. Синтез
Поле зрения: <IsPalindrome 'AC'>
Шаг 7. Анализ
Предложение 4, Присваивания 'AC' ← e.Other
Шаг 8. Синтез
Поле зрения: False
```

Листинг 1.4 Последовательное выполнение программы Рефал-машиной

1.2.4 Сущность прогонки

Из пунктов 1.2.1 – 1.2.3 известно, что работу абстрактной Рефал-машины можно представить в качестве следующей цепочки шагов:

$A_1 S_1 A_2 S_2 \dots A_n A_n$

На шаге анализа аргумент вызова функции разделяется на составные части, затем происходит синтез – соединение этих составных частей. На следующем шаге анализа аргумент вызова опять разделяется на составные части, при этом часто на те же самые, что и в предыдущем шаге. Перестройка выражений может занимать значительное время. Также после выполнения каждого шага Рефал-машины вся информация о разборе выражения на составные части теряется. Это приводит к ухудшению производительности программы.

Основной задачей прогонки является композиция нескольких аналогичных шагов Рефал-машины в один [5]. В этом случае процесс работы Рефал-машины можно представить цепочкой $A'S' \dots A_n A_n$, в которой шаг анализа A' является композицией шагов A_1 и A_2, а шаг синтеза S' является композицией шагов S_1 и S_2.

Таким образом, оптимизация прогонки позволяет программе выполняться быстрее, путем уменьшения количества промежуточных вызовов функций.

Встраивание функций – частный случай прогонки, при котором вызов функции в результатном выражении может быть заменен результатом сопоставления с одним из ее предложений без изменения семантических свойств функции. В этом случае шаг анализа A' тождественен шагу A_1.

1.3 Суперкомпиляция и граф конфигураций

1.3.1 Основные принципы суперкомпиляции

Суперкомпилятор — программа с помощью некоторой системы анализа осуществляющая суперкомпиляцию. Суперкомпиляция — это метод анализа и преобразования программ, предложенный В.Ф. Турчиным, в основе которого лежит выполнение следующих действий [6]:

1. Делается попытка выполнить программу в общем виде, без использования произвольных или ограниченных входных данных, символьно. Для этого строится дерево конфигураций также называемое деревом процессов. В узлах дерева оказываются конфигурации — описание множеств состояний вычислительного процесса. В рёбрах же находятся какие-либо проверки, которые приводят программу в состояние, описываемое конфигурацией.
2. Исходная программа может содержать циклы (или рекурсивные вызовы, которые могут быть описаны ими). Тогда дерево может получится бесконечным, ведь программа может не достигать условия выхода из цикла. Чтобы такого не происходило, конфигурации в дереве сравнивают с конфигурациями в родительских вершинах, пытаясь найти в них что-то общее, что позволит свести одну конфигурацию к другой и не раскрывать дерево дальше.
3. Построенное дерево преобразуется в остаточную программу, которая и является результатом работы суперкомпилятора. Название остаточная связано с тем, что после преобразований программа может не содержать некоторую исходную логику, которая оказывается лишней после проведения вычислений суперкомпилятором.

Термины суперкомпилятор и суперкомпиляция происходит от слов компиляция и супер в значении присматривающий[10]. Суперкомпилятор “присматривает” за компилятором, позволяя в некоторых случаях

компилировать более эффективные программы за счет вычислений проводимых на этапе компиляции, а не во время исполнения самой программы.

1.3.2 Построение графа конфигураций

Чтобы продемонстрировать работу суперкомпилятора, рассмотрим пример на простом учебном языке SLL, небольшую программу, которую суперкомпилятор мог бы оптимизировать. Она будет представлять собой программу по вычислению суммы трёх чисел Пеано. Ноль выражается как `Z`, число `1` как `S(Z)`, целое положительное число `n+1` выражается как `S(<выражение для n>)` или `S(S(...S(S(Z))))` где `S` написано `n+1` раз.

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

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

Листинг 1.5 Определение программы сложения трех чисел Пеано

Сначала необходимо построить вершину графа конфигураций, в неё помещается единственное выражение, которое необходимо просуперкомпилировать, в данном случае `add(add(a, b), c)`.

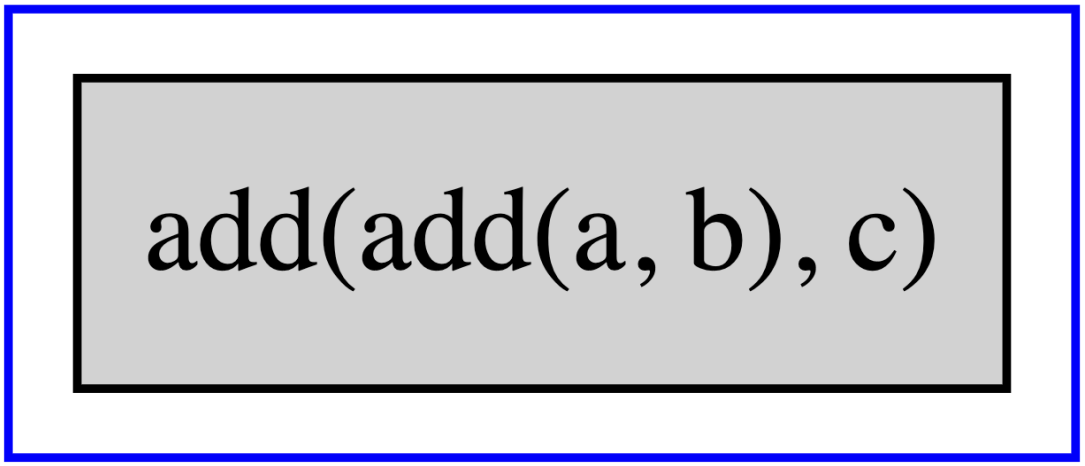

Рисунок 1.1 Начальное состояния графа конфигураций

Производится попытка вычислить символьно внешний вызов `add`, но так как правила в `add` определены для первого аргумента, только по второму аргументу невозможно сделать раскрытие. Переходим к внутреннему вызову

`add`, у функции есть два правила, когда `a=Z` и когда `a=S(a1)`. Рассматриваем оба правила, строим рёбра графа выходящие из начальной вершины, помечаем их подстановками, в вершинах записываем выражения, которые получаются, если эти подстановки были верны `add(b, c)` и `add(S(add(a1, b)), c)` соответственно.

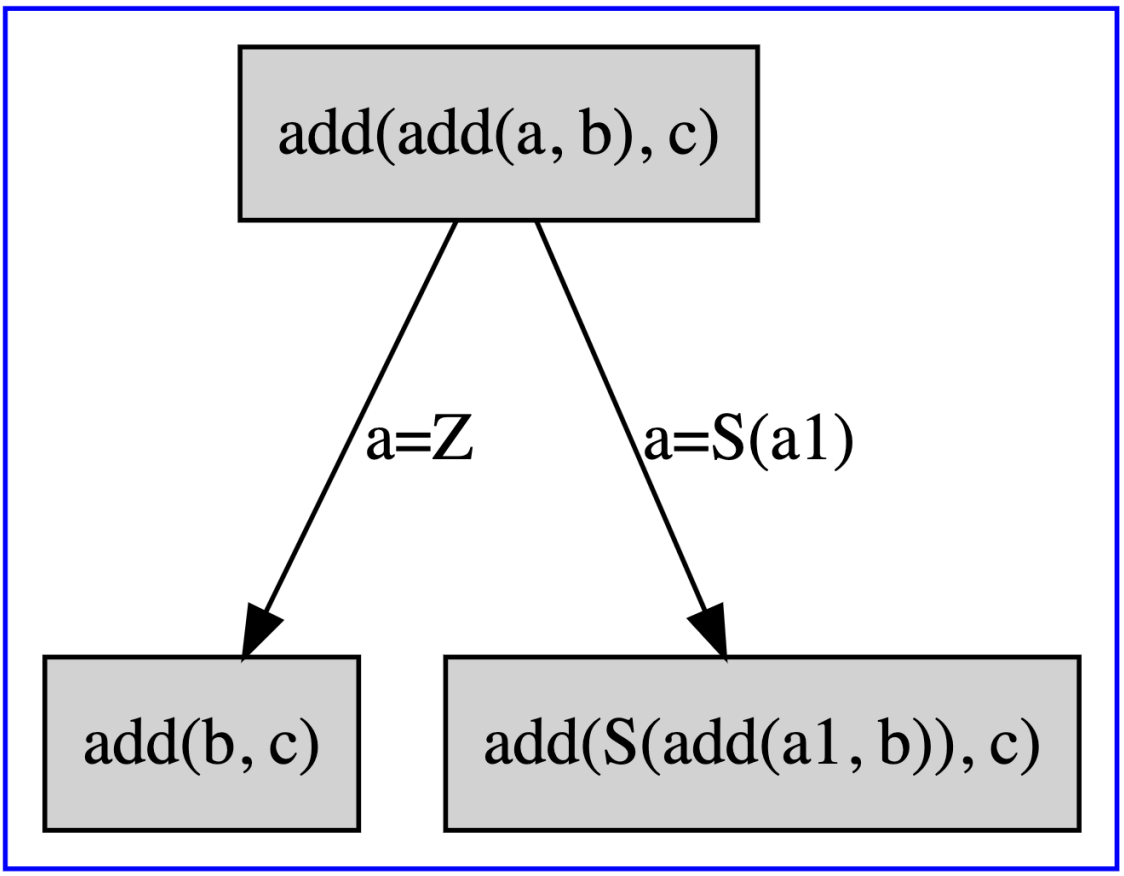

Рисунок 1.2 Состояние графа конфигураций после первого шага суперкомпиляции.

Продолжим разбор с конфигурации `add(S(add(a1, b)), c)`. Так как конструктор `S(..)` вышел наружу внутреннего `add`, появляется возможность раскрыть внешний `add`, при этом подходит только второе правило `add`, поэтому новое ребро можно никак не помечать.

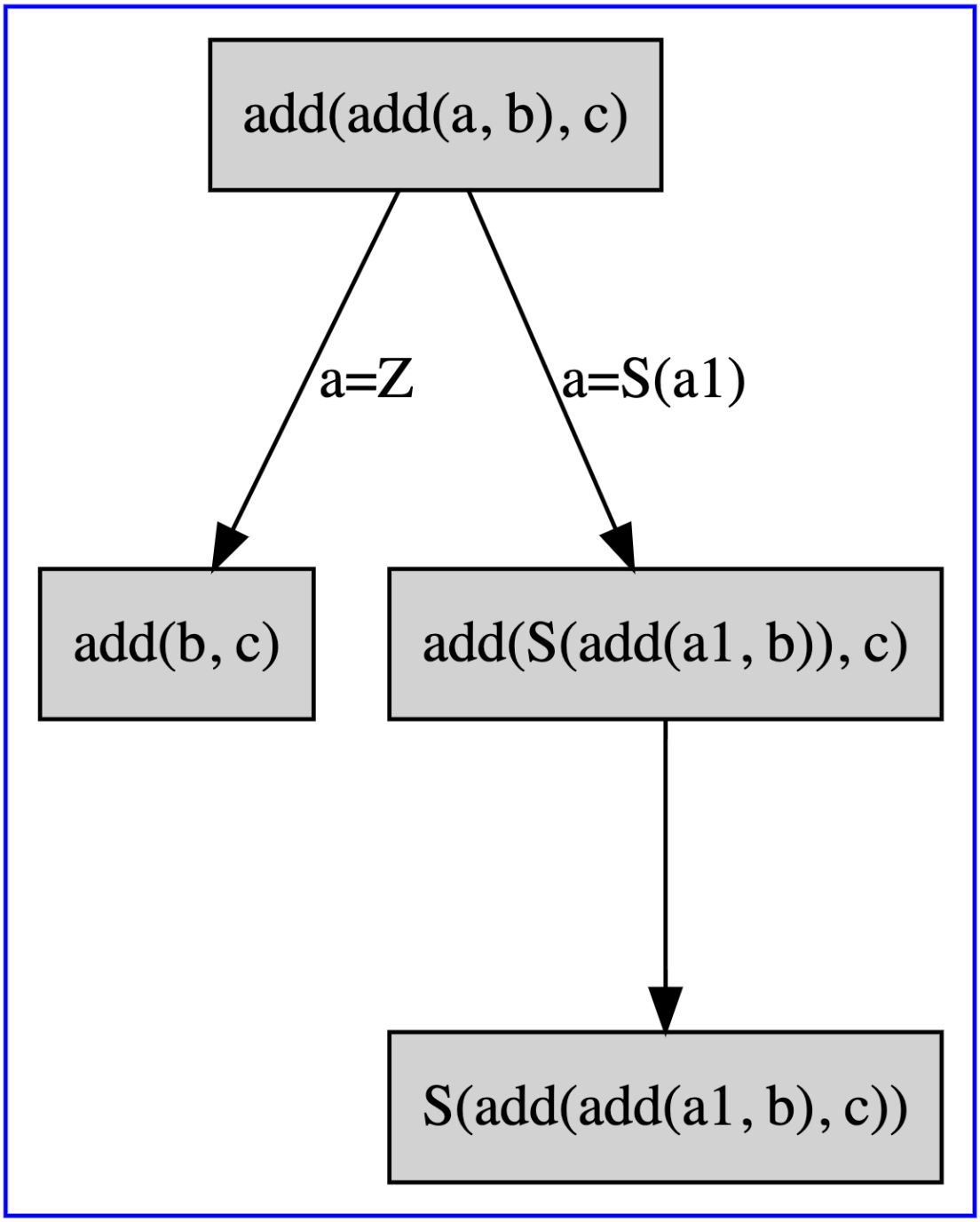

Рисунок 1.3 Состояние графа конфигураций после раскрытия

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

В новом выражении конструктор S оказался снаружи, поэтому можно провести декомпозицию выражения.

```
S(add(add(a1, b), c)) ~> add(add(a1, b), c)
```

В общем случае, если конструктор состоит из N выражений, при декомпозиции получается N выражений

```
C(e1,...,eN) ~> e1,...,eN
```

Таким образом, граф конфигураций принимает вид

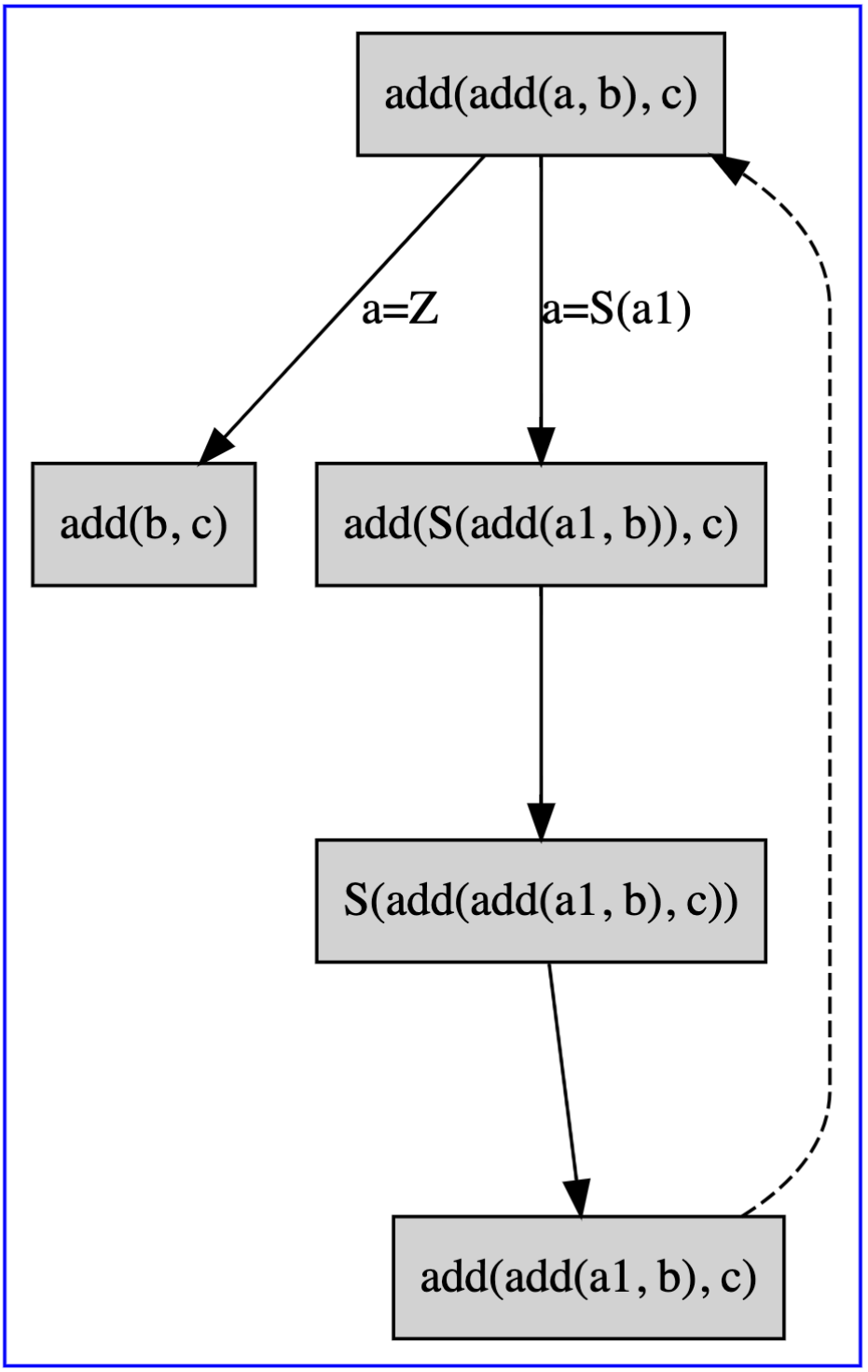

Рисунок 1.4 Состояние графа конфигураций после декомпозиции `S(add(add(a1, b), c))`.

Пунктирная стрелка обозначает отношение “похожести” между конфигурациями и может быть проведена только от потомка к предку. Данное отношение означает, что дальнейшее построение графа бессмысленно ввиду

того, что данная конфигурация уже встречалась и повторное ее рассмотрение приведет к бессмысленному разрастанию дерева.

В данном случае легко увидеть, что конфигурации одинаковы с точностью до имен переменных, а значит повторное раскрытие вызовов будет бесконечно выдавать уже существующие в графе конфигурации. Поэтому раскрытие этой ветки останавливается и алгоритм продолжает работу с еще не раскрытого вызова `add(b, c)`. Данный вызов раскрывается аналогично внутреннему вызову add ранее, проводя все раскрытия получаем конечную конфигурацию (Рисунок 1.5), которая является результатом работы суперкомпилятора.

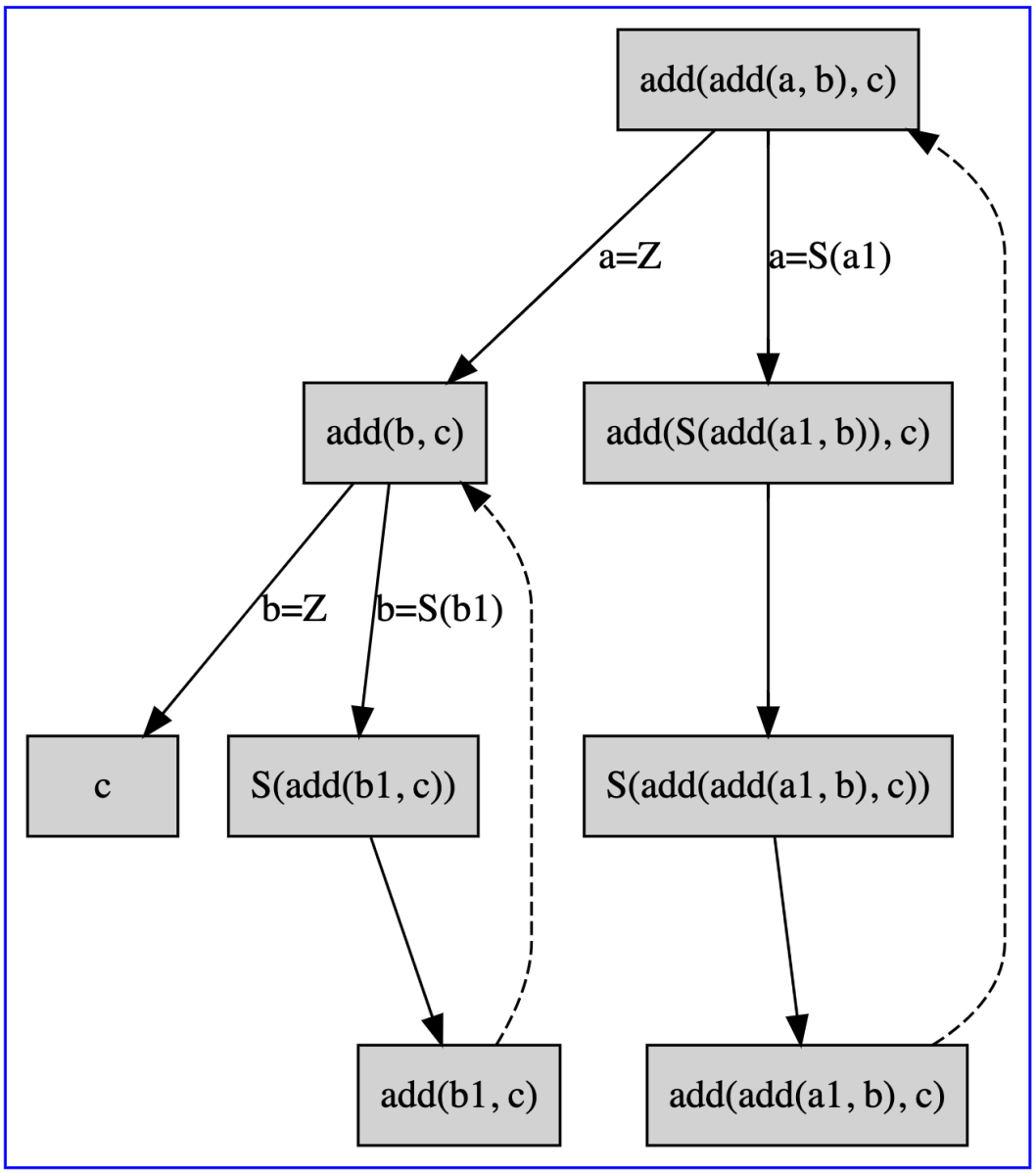

Рисунок 1.5 Конечное состояние графа конфигураций.

1.3.3 Получение остаточной программы

После получения графа конфигураций можно построить остаточную программу, которая на корректных входных данных (данные на которых не возникает ошибки сопоставления или бесконечного зацикливания программы) будет полностью семантически идентична исходной программе. При этом часть вычислений из исходной программы будет оптимизирована.

Продемонстрируем извлечение остаточной программы на примере со сложением трех чисел Пеано. Для каждого узла создаем функцию, аргументами которой будут переменные в конфигурации.

Например узлу `add(add(a, b), c)` будет соответствовать функция `g1(a, b, c)`.

```
add(add(a, b), c)      -> g1(a, b, c)
add(b, c)              -> g2(b, c)
c                      -> g3(c)
S(add(b1, c))          -> g4(b1, c)
add(b1, c)             -> g5(b1, c)
add(S(add(a1, b)), c) -> g6(a1, b, c)
S(add(add(a1, b), c)) -> g7(a1, b, c)
add(add(a1, b), c)     -> g8(a1, b, c)
```

Теперь для каждой конфигурации необходимо посмотреть на выходящие из нее стрелки и по ним создать правила для функций. Например для `add(add(a, b), c)` существуют две стрелки, значит необходимо создать два правила:

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

Затем для каждого правила создаются правые части. Для этого нужно посмотреть на ограничения в стрелке и создать вызов функции соответствующий дочернему узлу. Для `add(add(a, b), c)` получаем

```
g1(S(a1), b, c) = g6(a1, b, c)
```

```
g1(Z, b, c) = g2(b, c)
```

Для остальных функций и правил поступаем аналогично. Для совпавших конфигураций (помеченных пунктирной стрелкой) вызовы функций совпадают, поэтому вызов функции `g8` заменяется на вызов функции `g1`, а вызов функции `g5` на `g2`. Правила для таких функций не строятся.

Создав такую программу и удалив в ней промежуточные функции, получаем остаточную программу

```
g1(a, b, c)
where

g1(S(a1), b, c) = S(g1(a1, b, c));
g1(Z, b, c)     = g2(b, c);
g2(S(b1), c)    = S(g2(b1, c));
g2(Z, c)        = c;
```

Листинг 1.6 Остаточная программа порожденная суперкомпилятором.

Можно заметить, что полученная программа хоть и увеличилась в объёме, но стала более эффективной так как раньше конструктор `S` из аргумента a перемещался в результат за два шага (два вызова `add`), а теперь всего за один (один вызов `g1`).

1.3.4 Сравнение конфигураций на общность

В предыдущем разделе признаком остановки увеличения дерева послужило отношение похожести конфигураций с помощью переименования переменных, однако такое отношение срабатывает не всегда. Рассмотрим пример программы мультипликации двух чисел Пеано

```
mult(a, b)
where

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

Листинг 1.7 Определение программы умножения двух чисел Пеано

Построим граф конфигураций для данного примера.

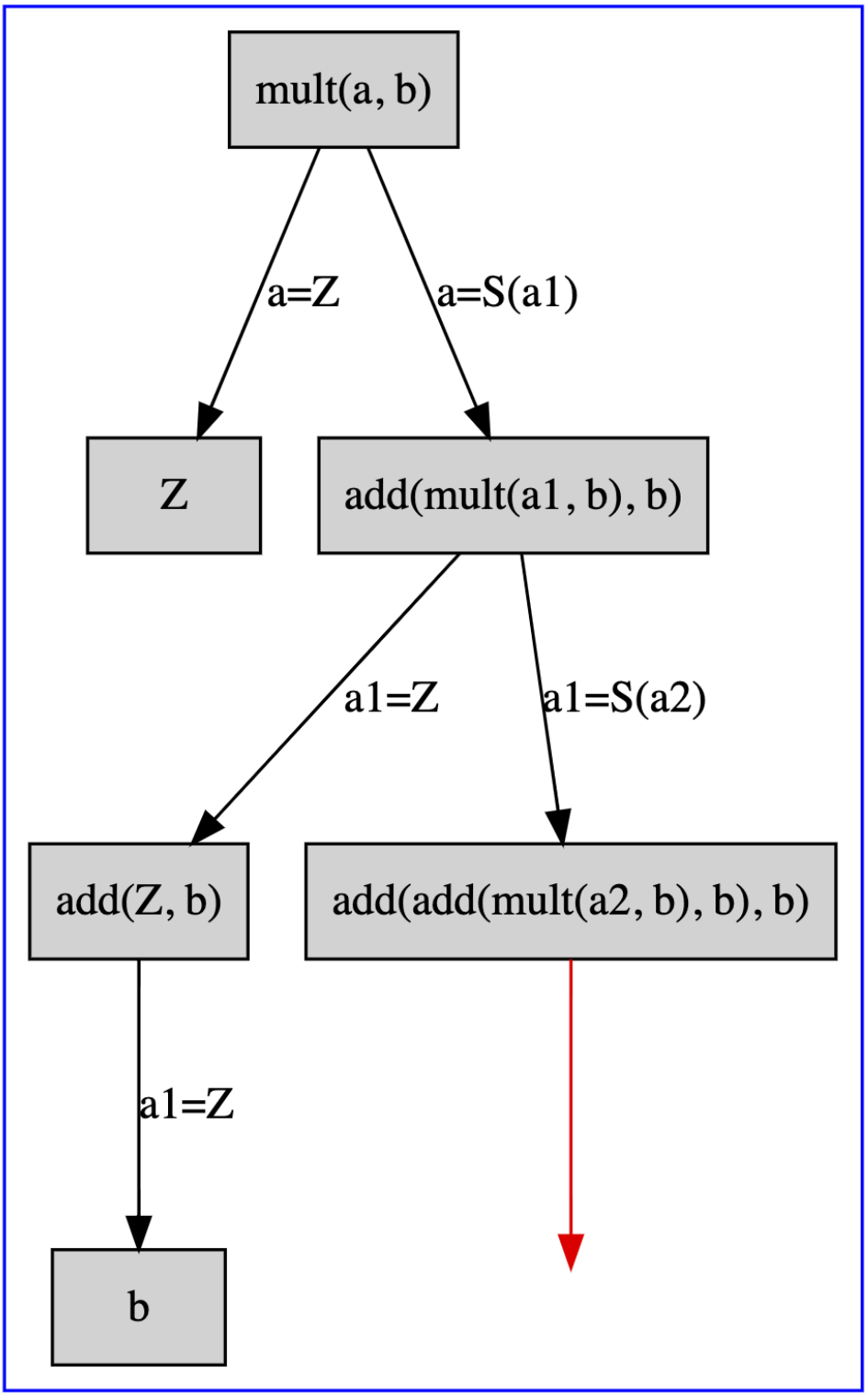

Рисунок 1.6 Неполный граф конфигураций программы умножающей два числа Пеано.

Данный граф получается бесконечным из-за конфигураций `add(mult(a1, b), b)`, `add(add(mult(a1, b), b), b)`,

`add(add(add(mult(a2, b), b), b), b)`, и так далее. На рисунке это продолжение обозначено красной стрелкой.

Возникает проблема, необходимо обобщить полученные конфигурации, при этом из рисунка видно, что в конфигурации

`add(add(mult(a1, b), b), b)`

присутствует член `mult(a1,b)`, который можно обобщить до `mult(a, b)`, но наше существующее отношение не покрывает данный случай.

Рассмотрим две конфигурации X_1 и X_2, если они “похожи”, то можно обобщить X_2 до X_1. Суперкомпиляция — процесс имитации обычных вычислений, но на этапе компиляции программы, а не в момент ее исполнения, поэтому при сравнении конфигураций нужно учитывать, какая из конфигураций появилась раньше, а какая — позже.

Пусть конфигурация X_1 предок (не обязательно непосредственный) конфигурации X_2. Возникает противоречие в логике, с одной стороны, для получения наиболее эффективной программы на выходе, необходимо как можно дольше не обобщать конфигурацию, накапливая различные изменения состояния программы. С другой стороны, чем больше конфигураций, тем больше требуется памяти и времени работы компилятора, а так как компилятор должен работать за разумное время, необходимо уменьшать количество итоговых конфигураций, что скажется на точности суперкомпиляции, если не уменьшит ее эффект до нуля, выдав программу полностью равную исходной.

Таким образом, если не обобщать — дерево конфигураций может получиться бесконечным. А если слишком часто обобщать — дерево получится конечным, но ненужным. Был выбран следующий подход [7]:

1. Сравниваем конфигурации X_1 и X_2. Если они не “похожи”, то не обобщаем X_2 до X_1. Если похожи, переходим к следующему шагу.
2. Если X_2 “меньше” X_1, то смысла обобщать нет, так как при правильном определении данного отношения уменьшающаяся цепочка конфигураций

в конце концов придет к конечной конфигурации без вызовов функций. Если же X_2 “больше” X_1, то переходим к следующему шагу.

3. В этой случае происходит ситуация схожая с рассмотренными ранее конфигурациями вида `add(...add(mult(a1, b), b)..., b)`, которые могут повторятся, вызывая бесконечный рост дерева, поэтому необходимо обобщить X_2 до X_1.

Введем для ситуации, когда конфигурации X_1 и X_2 “похожи” и при этом X_2 “больше” X_1 следующее обозначение

$X_1 \trianglelefteq X_2$

Такое отношение впервые придумали Хигман и Крускал, определяется оно единственным правилом. А именно, пусть $X_1 \trianglelefteq X_2$ если X_2 можно превратить в X_1 с помощью стирания некоторых символов и переименованием переменных. Таким образом покрывается сразу два отношения:

- Если X_2 можно превратить в X_1 стиранием, то они действительно похожи, X_2символьно содержит в себе X_1.
- Если X_2 можно превратить в X_1 стиранием, то оно действительно “больше” в смысле затрат на его хранение в символьном виде. И наоборот X_1 оказывается меньше, что позволяет использовать второй шаг алгоритма определения необходимости обобщения между двумя конфигурациями.

При этом отношение $\trianglelefteq$ называется отношением гомеоморфного вложения или отношением Хигмана-Крускала.

Рассмотрим в качестве примера отношения возникающие при построении графа конфигураций ранее рассмотренной программы вычисляющей произведение чисел Пеано.

Верно ли, что `mult(a, b)` ⊴ `add(mult(a1, b), b)` ? Да, верно, таким образом необходимо ограничить граф уже на этом этапе, провести пунктирную стрелку отношения ранее называемого “похожи”, а теперь гомеоморфной вложенности конфигураций. В итоге получаем следующий конечный граф конфигураций, из которого теперь можно получить остаточную программу.

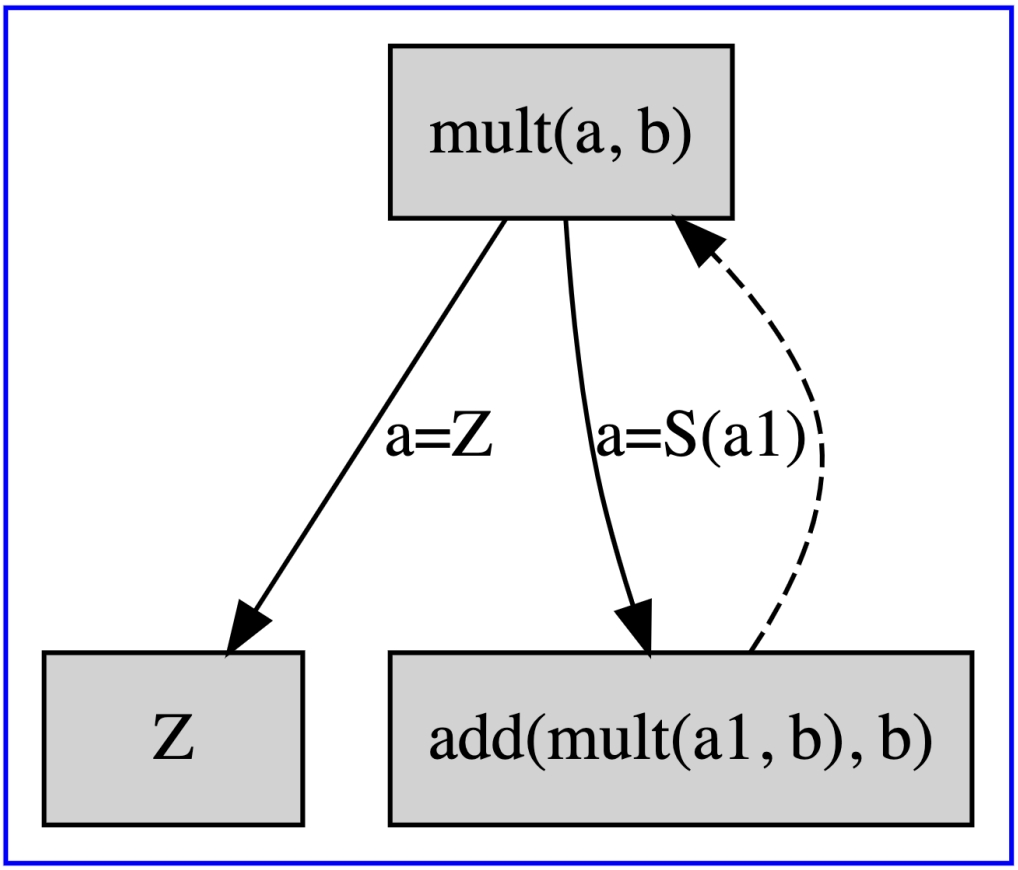

Рисунок 1.7 Конечный граф конфигураций программы умножающей два числа Пеано.

Остаточная программа полученная из этого графа будет мало чем отличаться от исходной программы, однако это не является недостатком суперкомпиляции, так как некоторые программы, особенно такие простые как предложенная, уже являются достаточно оптимизированными. В больших же программах зачастую находятся способы оптимизировать программу.

Отношение Хигман-Крускала можно определить более формально с помощью индуктивного определения через три правила:

1. $x \trianglelefteq y$для любых переменных x и y .
2. $X \trianglelefteq f(Y_1, \dots, Y_n)$, если f имя конструктора или функции и существует i такое, что $X \trianglelefteq Y_i$.
3. $f(X_1, \dots, X_n) \trianglelefteq f(Y_1, \dots, Y_n)$, если f имя конструктора или функции и для любого $i \leq n$ верно, что $X_i \trianglelefteq Y_i$.

Последнее, что нужно определить для отношения $\trianglelefteq$ — является ли оно действительно нужным отношением, которое никогда не приведет к зацикливанию? Оказывается, что да, соответствующую теорему доказал Крускал для случая, когда выражения составлены только из конструкторов. Однако по сути функции тоже являются конструкторами, а переменные можно считать нульарными конструкторами. Таким образом, отношение Хигмана-Крускала гарантирует построение конечного дерева конфигураций для любой заданной программы.

1.4 Архитектура компилятора Рефал-5λ

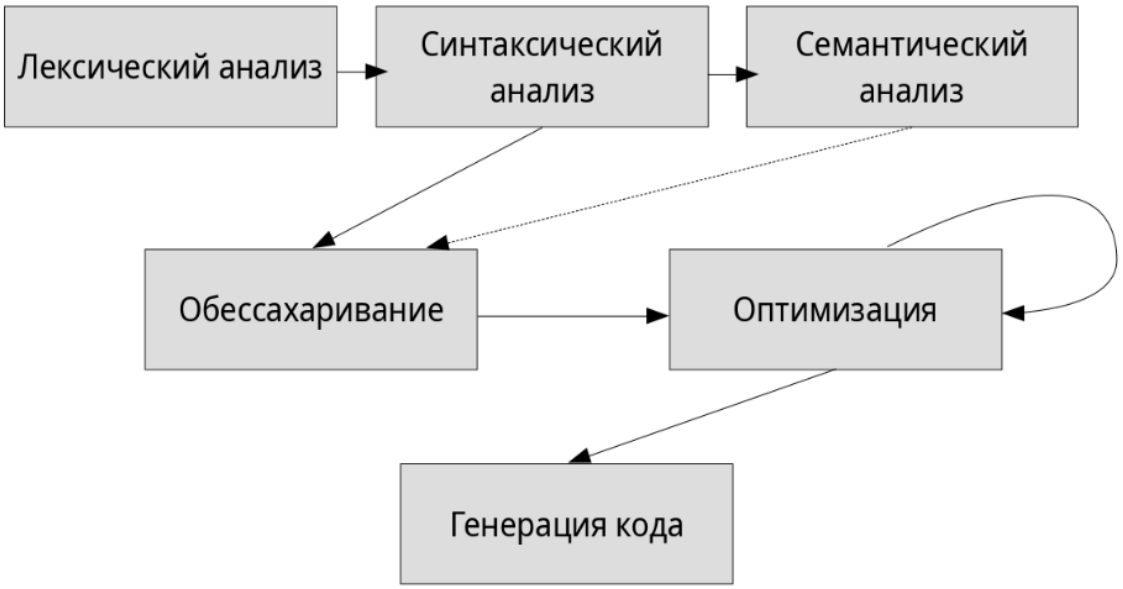

Рисунок 1.8 Архитектура компилятора

Компиляция программы на языке Рефал-5λ состоит из следующих этапов.

1. Лексический анализ. На этом этапе из входного потока символов формируется выходной поток токенов, корректных для языка.
2. Синтаксический анализ. На этом этапе на основе грамматики языка из потока токенов строится абстрактное синтаксическое дерево [8].
3. Семантический анализ [9]. На этом этапе проходит проверка семантики построенного синтаксического дерева. Пример: поиск объявлений функций без их определения и. т. д.
4. Обессахаривание абстрактного синтаксического дерева. Принимает дерево, порожденное на этапе синтаксического анализа, при условии, что семантический анализ успешно завершился. На этом этапе некоторые синтаксические конструкции, являющиеся «синтаксическим сахаром», преобразуются, и результатом этого этапа является упрощенное синтаксическое дерево. Пример: все анонимные функции именуются, их тела выносятся из тел других функций на верхний уровень и заменяются их вызовами.

5. Оптимизация полученного на предыдущих этапах синтаксического дерева. На этом этапе синтаксическое дерево обходится несколько раз, с применением различных алгоритмов оптимизации.
6. Генерация кода состоит из более мелких этапов. На первом этапе на основе синтаксического дерева порождается символический ассемблер. На втором этапе из промежуточного кода порождается двоичный код или код на языке C++.

2. РАЗРАБОТКА

2.1 Отладочные функции, необходимые при разработке

2.1.1 Вывод дерева в отладочный файл

Так как при разработке необходимо было работать с довольно большими деревьями, понадобилась функция, которая бы выводила их в лог, при этом форматируя в соответствии с некоторыми правилами, применение которых позволяло бы получить в выводе удобное представление структуры. Так как в компиляторе общей функции для такого форматирования не было предусмотрено, необходимо было разработать собственную реализацию.

Приведем правила, по которым должно работать форматирование:

1. Так как работа происходит со вложенными структурами, которые формируются с помощью структурных открывающих и закрывающих круглых скобок, необходимо чтобы каждое выражение, начинающееся с круглой скобки, начиналось с новой строки.
2. Каждое выражение, находящееся внутри структурных скобок должно быть на одном уровне (отделено от начала строки одним количеством символов табуляции) и на один уровень выше выражений, находящихся на одном уровне с соответствующими структурными скобками.

Таким образом выражение вида

```
(first ((k3(k5))(k4) second third) (r1) (r2)) (forth)
```

Должно быть преобразовано при печати в

```
( first
  ( ( k3
      ( k5 ) )
    ( k4 )
    second third )
  ( r1 )
  ( r2 ) )
```

```
( forth )
```

Листинг 2.1 Пример вывода выражения в лог.

Реализация данной функции занимает малое количество кода, однако позволила существенно снизить время при отладке. Она использует две вспомогательные функции, одна для определения вложенных скобок и их глубины в структуре, вторая расставляет символы табуляции перед символами.

```
$ENTRY PFormat {
  e.X = <PFormat-rec (0) (False) e.X >;
}

PFormat-rec {
  (s.Depth) (s.NLine) (e.Inner) =
    <PFormat-rec-tabs (s.Depth) (s.NLine)>
    '( ' <PFormat-rec (<+ s.Depth 1>) (False) e.Inner > ') ';
  (s.Depth) (s.NLine) e.Begin (e.Inner) =
    <PFormat-rec (s.Depth) (s.NLine) e.Begin >
    <PFormat-rec-tabs (s.Depth) (True)>
    '( ' <PFormat-rec (<+ s.Depth 1>) (False) e.Inner > ') ';
  (s.Depth) (s.NLine) (e.Inner) e.End =
    <PFormat-rec-tabs (s.Depth) (s.NLine)>
    '( ' <PFormat-rec (<+ s.Depth 1>) (False) e.Inner > ') '
    <PFormat-rec (s.Depth) (True) e.End >;
  (s.Depth) (s.NLine) e.Begin (e.Inner) e.End =
    <PFormat-rec (s.Depth) (s.NLine) e.Begin >
    <PFormat-rec-tabs (s.Depth) (True)>
    '( ' <PFormat-rec (<+ s.Depth 1>) (False) e.Inner > ') '
    <PFormat-rec (s.Depth) (True) e.End >;
  (s.Depth) (s.NLine) e.X =
    <PFormat-rec-tabs (s.Depth) (s.NLine)>
    e.X;
}

PFormat-rec-tabs {
  (s.Depth) (False) = '';
  (0) (True) e.Rest = '\n' e.Rest;
  (s.Depth) (True) e.Rest =
    <PFormat-rec-tabs (<- s.Depth 1>) (True) /*s.Depth*/ ' '
e.Rest>;
}
```

Листинг 2.2 Функция PFormat и вспомогательные функции.

2.1.2 Логирование аргументов и результатов функции

Не менее полезной оказалась функция, которая перед вызовом функции печатала название функции, а после получения результатов функции печатала их, возвращая их своим результатом.

Она позволила эффективно и быстро находить ошибки, так как не нужно было писать отладочную функцию для каждой функции.

```
Log-Call {
  (e.Name) t.Func = {
    e.Args
      , <Log-PutLine 'function ' e.Name ' called with args
***' <PFormat e.Args> '***' > : /* empty */
      , <t.Func e.Args> : e.Res
      , <Log-PutLine 'function ' e.Name ' returned ***'
<PFormat e.Res> '***' > : /* empty */
      = e.Res
  }
}
```

Листинг 2.3 Функция для вывода аргументов и результатов функции.

2.2 Разработка алгоритма прогонки, не приводящего к зацикливанию

2.2.1 Существующая реализация

Алгоритм прогонки в компиляторе Рефала-5λ уже был реализован. Данный подход предполагает, что на проходе прогонки в каждой правой части берется очередной вызов, который можно оптимизировать, и для него выполняется прогонка или встраивание, проходы повторяются до неподвижной точки — пока в оптимизируемых функциях не кончатся вызовы, которые можно оптимизировать.

Однако такой подход имеет несколько недостатков, а именно:

- Для выполнения прогонки нужен отдельный этап разметки — нужно выбрать среди оптимизируемых те функции, которые безопасно прогонять, то есть неподвижная точка в них будет достигнута.
- Проход специализации (другой оптимизации) за один раз выполняет максимум работы. Проход прогонки нужно повторять до неподвижной точки.
- Взаимодействия проходов оптимизации достаточно сложны.
- Для предотвращения риска зацикливания были введены итерации оптимизирующих проходов, что приводит к увеличению запутанности кода и усложнению его логики работы.
- Рекурсивные и взаимно-рекурсивные функции не могут прогоняться, что снижает эффективность оптимизации

2.2.2 Разработка нового алгоритма

Существующая реализация прогонки берёт правую часть `R` предложения `L = R` функции, которую пытается оптимизировать и строит для нее набор решений. Решением является набор сужений и выражение `{E,C={C1,…,CN}}`

(`E` — выражение, `C` — набор сужений) . Будем записывать применение набора сужений `C` к выражению или набору сужений `X` как `X*C`, порядок применения сужений при этом важен и является левоассоциативным. Применяя сужение к сужению получаем сужение, а применяя сужение к выражению получаем выражение. Тогда для каждого набора решений в функции появляется новое предложение `L*C = E` , исходное предложение при этом стирается.

Новый алгоритм строится на идеях суперкомпиляции, описанных в разделе 1.3.

Можно заметить, что выражение R в правой части предложения можно считать выражением (начальной конфигурацией), которое должен оптимизировать суперкомпилятор, при этом выражения, полученные на одном шаге прогонки являются новыми конфигурациями, которые вносятся в граф, а соответствующими сужениями помечаются рёбра графа. Таким образом новая функция прогонки ограниченно суперкомпилирует правую часть предложения Рефала. Результатом ее вызова становится ациклический направленный граф конфигураций. Каждое ребро графа помечается сужением, а вершины содержат результатные выражения, которые будут вставляться в итоговые правые части.

Работа функции прогонки при этом разбивается на три этапа.

На первом этапе строится сам граф конфигураций по следующему алгоритму:

1. Выбирается очередной подлежащий оптимизации вызов функции. Вызов прогоняется, получается набор решений. Каждое решение — это сужения и выражение, которое займет место старой правой части. Если решение получилось без сужений, то дальше прогонять невозможно (всегда будет получаться одно и то же выражение), помечаем вершину как остановочную.
2. Для каждого решения проводится проверка на зацикливание графа с помощью проверки отношения Хигмана-Крускала с каждым из предков вершины. Если оно выполняется с какими-либо предком, необходимо

удалить из графа всех потомков данного предка и пометить вершину как остановочную.

3. Для каждой вершины, еще не помеченной как остановочная, необходимо взять выражение, которое в ней храниться и применить Шаг 1.

В результате те вызовы, прогонка которых приводит к зацикливанию, останутся в листьях дерева. Также в листьях останутся все вызовы, в которых прогонка невозможна, и те, которые прогнать нельзя. При этом для любой бесконечной цепочки выражений, которые получаются друг из друга с помощью набора сужений, найдутся два выражения, входящие в эту цепочку, такие что для них будет выполняться отношение Хигмана-Крускала, что обеспечивает конечность алгоритма.

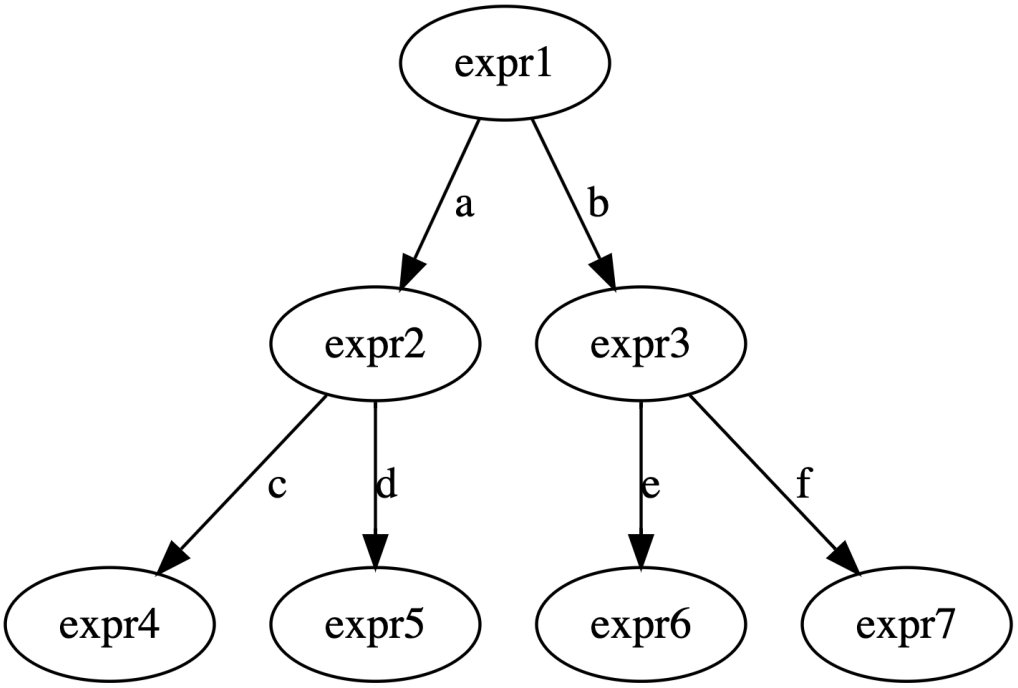

Рисунок 2.1 Схематическое изображение графа конфигураций.

Выражения обозначены как$expr_i$, expr1 — начальное выражение.

Сужения обозначены латинскими буквами a,b,c,d,e,f.

На втором этапе в дереве конфигураций необходимо удалить все внутренние вершины и вычислить композиции соответствующих сужений. Формально для каждого существующего пути из начальной конфигурации A в листовую конфигурацию X_n через через конфигурации X_1,.. , X_{n-1} и сужения

c_1,.., c_n нужно вычислить композицию сужений c_1 *..* $c_n = C$ и построить новую вершину конфигурации со значением X_n непосредственным предком которой является начальная конфигурация, а ребро помечено сужением C.

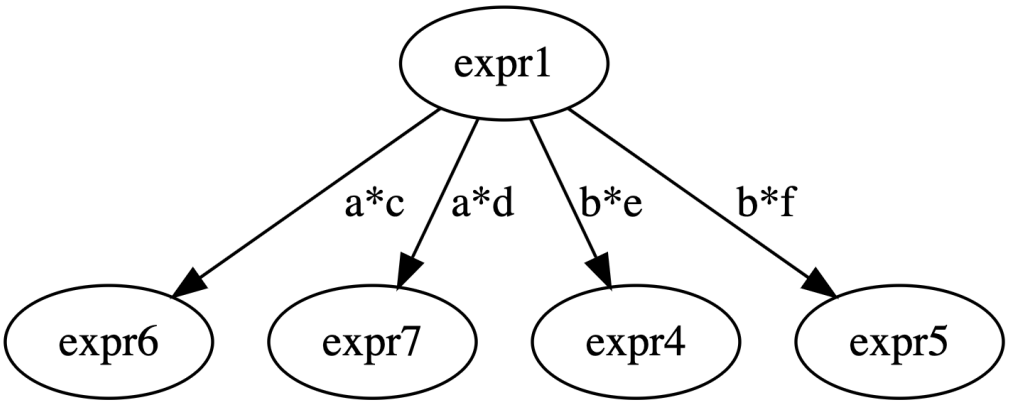

Рисунок 2.2 Схематическое изображение итогового результата прогонки. Выражения обозначены как$expr_i$, expr1 — начальное выражение.

Композиции сужений x и y обозначены x*y.

На третьем этапе результат нового алгоритма приводится в формат результата старого, то есть все выражения-конфигурации из полученного вырожденного графа конфигурации становятся выражениями в решениях, а сужения, которыми помечены соответствующие ребра становятся сужениями в в решениях.

Стоит отметить, что так как итоговые выражения получаются из графа конфигураций, в котором они были в листовых вершинах, дальнейшая их прогонка бесполезна, поэтому все вызовы функций в этих выражениях помечаются как холодные, что значит, что в дальнейшем они оптимизироваться не будут.

2.2.3 Реализация нового алгоритма

Так как новый алгоритм разрабатывался на основе старого, все изменения получились в модуле, в котором старый алгоритм производил один шаг прогонки. При этом сигнатуру функции OptTree-Drive-Expr, входной точки в модуль, удалось сохранить, что должно обеспечить легкость соединения ветки разработки с основной кодовой базой.

```
/**
  <OptTree-Drive-Expr
    (e.UsedVars) (e.WholeVars) t.OptInfo e.Expr>
    == t.OptInfo^ e.Branches (e.NewFunctions)

  e.Branches ::= (e.Contractions (e.DrivenExpr))*
  e.UsedVars, e.WholeVars ::= (s.Mode e.Index)*
*/
$ENTRY OptTree-Drive-Expr {
  (e.UsedVars) (e.WholeVars) t.OptInfo e.Expr
    = <MakeDriveTree (0) (e.UsedVars) (e.WholeVars)
        t.OptInfo e.Expr (0)>
    : ((s._CntNodes) t.OptInfo^ (e.NewFunctions)) e.DriveTree
    = t.OptInfo
      <ColdTree <MapUnBracket <FlatDriveTree e.DriveTree>>>
      (e.NewFunctions);
}
```

Листинг 2.4 Функция `OptTree-Drive-Expr` и описание ее сигнатуры.

В теле данной функции и находится разбиение алгоритма на три этапа, это сделано с помощью функций `MakeDriveTree`, `FlatDriveTree` и `ColdTree`.

Реализация первого этапа содержится в функции `MakeDriveTree` . Данная функция рекурсивно строит дерево конфигураций и проверяет между ними отношение Хигмана-Крускала. Она принимает на вход несколько аргументов (в контексте языка они считаются одним аргументом, но по сути это несколько параметров, которые разбиты структурными скобками):

- Аргумент `s.CntNodes` содержит количество вершин, которые оказались в дереве. Используется для ограничения роста дерева совместно с аргументом `s.TreeDepth`.

- Аргумент `s.TreeDepth` содержит количество вершин, которые оказались в текущей ветке дерева. Используется для ограничения роста дерева совместно с аргументом `s.CntNodes`.
- Аргумент `e.UsedVars` содержит имена использованных переменных, необходим для накопления сведений об этих именах, так как при создании сужений можно использовать только еще не встретившиеся имена
- Аргумент `e.WholeVars` содержит подмножество `e.UsedVars`, имена переменных, которые нельзя заменять при прогонке (в режиме вставки `e.WholeVars` полностью содержит `e.UsedVars`).
- Аргумент `t.OptInfo` содержит информацию необходимую для прогонки такую как: имена и тела функций, использованные функции, таблицу функций и другие.
- Аргумент `e.Expr` содержит выражение, которое нужно оптимизировать, то есть правую часть одного из предложений оптимизируемой функции или одну из получившихся конфигураций.
- Аргумент `e.canoninzedExprs` содержит список родительских конфигураций для текущей вершины, именно по нему проверяется отношение Хигмана-Крускала для остановки процесса увеличения дерева.

Функция `MakeDriveTree` сначала проверяет встречались ли похожие на текущую конфигурации ранее, если встречались, то дальше дерево построено не будет, но вернётся выражение, позволяющие вызвавшей ее функции понять, какое поддерево нужно удалить и начать построение заново. Первый вызов этой функции всегда осуществляется с пустым множеством `e.canoninzedExprs`, что значит, что первое выражение всегда прогонится, что в свою очередь значит, что результатом вызова функции всегда будет дерево состоящие хотя бы из одной вершины.

Затем выполняется проверка превышения значений `s.TreeDepth` и `s.CntNodes`, если значения превышают заданные, возвращается пустое

дерево. Хоть отношение Хигмана-Крускала и гарантирует нахождение двух выражений в цепочке, которые им связаны, оно не дает гарантии, что такие выражения найдутся в начале цепочки[10]. Из-за этого необходимо иметь ограничения на глубину дерева и количество вершин в дереве.

Ограничение количества вершин в дереве не дает цепочке разрастись до размеров, которые программе было бы сложно хранить и обрабатывать. Ограничение глубины дерева дает равный шанс на получение оптимального результата каждой операции построения потомков определенной вершины.

После прохождения всех проверок вызывается функция `GrowDriveTree` со всеми теми же аргументами. Однако задача этой функции построить новое дерево и вызвать для новых вершин `MakeDriveTree` для продолжения построения дерева. Рассмотрим работу данной функции подробнее.

Сначала определяется, есть ли в выражении вызов, который можно оптимизировать. Если нельзя, то возвращается дерево из одной вершины, содержащей данное выражение, то есть появляется листовая вершина.

Если вызов есть, то он прогоняется и для каждого полученного нового выражения делается попытка продолжить построение дерева конфигураций. Если хотя бы один такой вызов возвращает признак зацикливания `Generalize s.Cnt`, то делается проверка, если отношение Хигмана-Крускала сработало не с данным выражением, то текущая вершина и все её потомки уничтожаются, и возвращается так же возвращается признак зацикливания. Если же отношение Хигмана-Крускала сработало с данным выражением, то производится заморозка вызова, который привёл к зацикливанию и повторная попытка вызова `GrowDriveTree` с целью прогнать другие вызовы, которые могли в общем случае не зациклиться.

```
 = <MapAccum
    {
      ((s.CntNodes^) t.OptInfo^ (e.NewFunctionsAccum))
(e.Contractions (e.DrivenRight))
        = <MakeDriveTree
            (s.CntNodes) (e.UsedVars) (e.WholeVars)
```

```
            t.OptInfo e.DrivenRight
            (s.TreeDepth e.canoninzedExprs)
          >
        : {
            ((s.CntNodes^) t.OptInfo^ (e.NewFunctions^))
            e.Branches^
              = ((s.CntNodes) t.OptInfo
                  (e.NewFunctionsAccum e.NewFunctions))
                (e.Contractions (e.DrivenRight)
                 (e.Branches));
            Generalize s.Cnt
              = (Generalize s.Cnt);
          };
      (Generalize s.Cnt) (e.Contractions (e.DrivenRight))
        = (Generalize s.Cnt);
    }
    ((s.CntNodes) ((e.OptFuncNames) e.OptFuncs)
     (e.NewFunctions))
    e.Branches
  >
: {
    (Generalize 1) e._DeletedTree
      = <GrowDriveTree
          (s.CntNodes) (e.UsedVars) (e.WholeVars) t.OptInfo
          <RestoreDrivenCall t.Call e.Expr>
          (s.TreeDepth e.canoninzedExprs)
        >;
    (Generalize s.Cnt) e._DeletedTree
      = Generalize <Sub s.Cnt 1>;
    e.Tree = e.Tree;
  };
```

Листинг 2.5 Фрагмент функции GrowDriveTree, выполняющий попытки увеличить дерево для различных вызовов.

Реализация второго этапа содержится в функции `FlatDriveTree`, которая принимает на вход единственный аргумент — дерево, которое было построено в функции `MakeDriveTree`.

Функция `FlatDriveTree` рекурсивно проходит по всем вершинам графа конфигураций удаляя внутренние вершины и создавая композиции сужений. Удаление вершины тривиальная задача, но на создании композиции сужений следует остановится подробней.

Дело в том, что сужения хранятся в Рефале в виде пар Значение:Результат. Пусть есть два набора сужений C_1 и C_2 . Рассмотрим их содержимое, чтобы получить композицию.

Пусть набор C_1 содержит следующие пары Значение:Результат:

$$V_1^1 : R_1^1$$

$$V_2^1 : R_2^1$$

…

$$V_n^1 : R_n^1$$

А набор C_2 содержит:

$$V_1^2 : R_1^2$$

$$V_2^2 : R_2^2$$

…

$$V_m^2 : R_m^2$$

Тогда обозначим применение одной пары Значение:Результат к выражению X как $X * (V : R)$, тогда результатом композиции двух наборов сужений $C_1 * C_2$ будет набор:

$$V_1^1 : R_1^1 * (V_1^2 : R_1^2) * (V_2^2 : R_2^2) * \ldots * (V_m^2 : R_m^2)$$

$$V_2^1 : R_2^1 * (V_1^2 : R_1^2) * (V_2^2 : R_2^2) * \ldots * (V_m^2 : R_m^2)$$

…

$$V_n^1 : R_n^1 * (V_1^2 : R_1^2) * (V_2^2 : R_2^2) * \ldots * (V_m^2 : R_m^2)$$

$$V_1^2 : R_1^2$$

$$V_2^2 : R_2^2$$

…

$$V_m^2 : R_m^2$$

То есть помимо применений к правым частям пар первого набора сужений второго набора, необходимо добавить весь второй набор в результирующий, так как в нем могут содержаться сужения переменных из предшествующих вызовов.

После удаления из дерева внутренних вершин, и составления композиций сужений, результат почти можно отдавать, единственное, что нужно сделать, пометить все наружные вызовы холодными, чтобы оптимизация в них больше не проводилась.

Это и есть работа третьего этапа, который реализован в функциях `ColdTree` и `ColdAll`, их исходный код представлен на листинге 2.6

```
ColdTree {
  e.Branches
    = <Map
      {
        (e.Contractions (e.DrivenExpr))
          = (e.Contractions (<ColdAll e.DrivenExpr>))
      }
      e.Branches
      >;
}
ColdAll {
  CallBrackets e.Expr
    = ColdCallBrackets e.Expr;
  ColdCallBrackets e.Expr
    = ColdCallBrackets e.Expr;
  e.B (e.Expr) e.E
    = <ColdAll e.B> (<ColdAll e.Expr>) <ColdAll e.E>;
  e.X = e.X;
}
```

Листинг 2.6 Исходный код функций ColdTree и ColdAll.

2.2.4 Руководство пользователя

Изменения вносились в исходный код компилятора Рефала-5λ [5], получить готовый компилятор на системах macOS или Linux с изменениями, и настроить его использование, можно с помощью системы следующих команд:

```
git clone https://github.com/bmstu-iu9/refal-5-lambda.git
cd refal-5-lambda
git checkout apakhov-cycleless-drive
cd src/compiler
RLMAKEFLAGS='-X-ODPRSi' ./makeself-s.sh
RLMAKEFLAGS='-X-ODPRSi' ./makeself.sh
cd -
export PATH=$(pwd)/bin:$PATH
```

Листинг 2.7 Скачивание исходных текстов и сборка компилятора Рефал-5λ

Для Windows необходимо использовать аналогичные команды, более подробно можно посмотреть в репозитории компилятора.

Компилятор может принимать в качестве входных аргументов набор ключей. Для вызова компилятора с флагами оптимизации необходимо передать следующий ключ:

`-O`flags``

где `` `flags` `` - набор флагов оптимизации.

Флаг `I` включает оптимизацию встраивания. Для всех функций с метками `$INLINE` и `$DRIVE` будет осуществлена попытка встраивания.

Флаг `D` включает оптимизацию прогонки. Для всех функций с меткой `$INLINE` будет осуществлена попытка встраивания, а для функций с меткой `$DRIVE` – прогонки.

Стоит отметить, что флаг `D` имеет более высокий приоритет.

3 ТЕСТИРОВАНИЕ

Так как компилятор Рефала-5λ является самоприменимым, хорошим показателем работоспособности алгоритма является проверка сборки его из исходных текстов. Четвертая и пятая команды выполняют данный тест сначала собирая новую версию компилятора стабильной его версией, а затем повторяя сборку, но уже свежесобранными компилятором.

Для выполнения проверки автотестами необходимо выполнить следующие команды.

```
cd autotests
./run.sh
```

Листинг 3.1 Запуск автотестов компилятора Рефал-5λ

Ручная же проверка может быть выполнена с помощью специально созданных файлов и их компиляции.

```
rlc -o main -ODi --opt-tree-cycles=1 --log=comp.log <файл>
./main
```

Листинг 3.2 Сборка и запуск тестовых программ

```
$DRIVE INC;
INC {
    (t.X) = <Add t.X 1>;
}

$ENTRY Go {
    = <Prout <MAP &INC (1) (2) (3)>>
  ;
}

$DRIVE MAP;
MAP {
    t.Closure t.Elem e.Rest =
        <t.Closure t.Elem> <MAP t.Closure e.Rest>
    ;
    t.Closure =
    ;
}
```

Листинг 3.3 Файл map.ref

Результатом работы программы, скомпилированной из файла map.ref является вывод чисел 2 3 4, однако интересен файл comp.log, который появляется после сборки программы. Он содержит полученные после оптимизации функции. После прогонки из функции `Go` за один проход полностью удаляется вызов функции `MAP`, вместо него оказывается подставлен результат вызова.

```
$ENTRY Go {
  /* empty */ = <Prout <MAP &INC (1) (2) (3)>>;
}
.........
$ENTRY Go {
  /* empty */ = <Prout 2 3 4>;
}
```

Листинг 3.4 Часть содержимого файла comp.log после компиляции map.ref

```
$DRIVE ADD;
ADD {
    (Z) (e.X) = e.X
    ;
    ('1' e.N) (e.X) = '1' <ADD (e.N) (e.X)>
    ;
}

$ENTRY Go {
  = <Prout <ADD ('11' Z) ('111' Z)>>
  ;
}
```

Листинг 3.5 Файл add.ref

Аналогично результату компиляции файла map.ref, результат компиляции файла add.ref в файле comp.log показывает, что функция `Go` была полностью оптимизирована за один проход.

```
$ENTRY Go {
  /* empty */ = <Prout <ADD ('11' Z) ('111' Z)>>;
}
........
$ENTRY Go {
  /* empty */ = <Prout '11111' Z>;
}
```

Листинг 3.6 Часть содержимого файла comp.log после компиляции add.ref

```
$DRIVE Apply;
Apply {
      s.Fn e.Argument = <s.Fn e.Argument>;
      (t.Closure e.Bounded) e.Argument
            = <Apply t.Closure e.Bounded e.Argument>;
}


$ENTRY Go {
      = <Prout <Apply &Add  1 1 >>;
}
```

Листинг 3.7 Файл apply.ref

Файл apply.ref использовался для простой проверки работоспособности прогонки на разных этапах разработки. Аналогично предыдущим результатам, результат компиляции файла add.ref в файле comp.log показывает, что функция Go была полностью оптимизирована за один проход.

```
$ENTRY Go {
  /* empty */ = <Prout <Apply &Add 1 1>>;
}
........
$ENTRY Go {
    /* empty */ = <Prout 2>;
}
```

Листинг 3.8 Часть содержимого файла comp.log после компиляции apply.ref

ЗАКЛЮЧЕНИЕ

В процессе выполнения этой работы были выполнены все установленные задачи. А именно, были изучены язык Рефал-5λ, теоретическая основа суперкомпиляции, алгоритмы построения графа конфигураций и возможные отношения, позволяющие останавливать рост графа.

Компилятор языка Рефал-5λ был модифицирован: реализован, отлажен и протестирован новый алгоритм прогонки, который не приводит к зацикливанию на любой функции в языке Рефал-5λ.

Алгоритм оптимизации прогонки был применен к более широкому спектру функций в компиляторе языка. Также алгоритм позже позволит уменьшить количество итераций на этапе древесных оптимизаций в компиляторе, что может дать прирост в скорости компиляции программ на языке Рефал-5λ.

В дальнейшем работа предполагает создание новой универсальной метки `$OPT` в компиляторе, которая придет на замену меткам `$DRIVE`, `$INLINE` и `$SPEC`. Также теоретическую часть работы можно использовать для реализации оптимизаций в языках, основанных на сопоставлении с образцом, и для организации алгоритмов, которые используют суперкомпиляцию.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Руководство по программированию и справочник по языку РЕФАЛ-5 [Электронный ресурс] – Режим доступа: http://www.refal.ru/rf5_frm.htm. Дата обращения: 09.06.2021.
2. Компилятор Рефал-5λ [Электронный ресурс] – Режим доступа: https://github.com/bmstu-iu9/r-5-lambda. Дата обращения: 09.06.2021.
3. V.F.Turchin. REFAL-5 programming guide & reference manual [Электронный ресурс] – Режим доступа: http://refal.botik.ru/book/html. Дата обращения: 09.06.2021.
4. А.П.Немытых. Лекции по языку программирования Рефал. Сборник трудов по функциональному языку программирования Рефал. Том No1 — Переславль-Залесский: Сборник, 2014.
5. С.А.Романенко. Прогонка для программ на РЕФАЛе-4. Препринт No211 — Институт Прикладной Математики АН СССР, 1987.
6. Климов А.В., Романенко С.А. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М.В.Келдыша. 2018. № 111. 36 с. doi:10.20948/prepr-2018-111.
7. Романенко С.А. Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления // Препринты ИПМ им. М.В.Келдыша. 2018. № 209. 32 с. doi:10.20948/prepr-2018-209
8. А. Ахо, М. Лам, Р. Сети, Д. Ульман. Компиляторы: принципы, технологии и инструментарий — 2 изд. — М.: Вильямс, 2008.
9. Steven S. Muchnik. Advanced Compiler Design and Implementation — Morgan Kaufmann, 1997.
10. Ключников И.Г. Суперкомпиляция: идеи и методы // Практика функционального программирования. — 2011. — № 7. — С. 133–165, URL: http://fprog.ru/2011/issue7/practice-fp-7-print.pdf, дата обращения 20.06.2021.

11. Турчин В.Ф. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ. В сб.: Труды симпозиума "Теория языков и методы построения систем программирования". – Киев-Алушта, 1972.

ПРИЛОЖЕНИЕ А

```
/**
  <OptTree-Drive-Expr (e.UsedVars) (e.WholeVars) t.OptInfo
e.Expr>
    == t.OptInfo^ e.Branches (e.NewFunctions)

  e.Branches ::= (e.Contractions (e.DrivenExpr))*
  e.UsedVars, e.WholeVars ::= (s.Mode e.Index)*
*/
$ENTRY OptTree-Drive-Expr {
  (e.UsedVars) (e.WholeVars) t.OptInfo e.Expr
    = <MakeDriveTree (0) (e.UsedVars) (e.WholeVars) t.OptInfo
       e.Expr (0)>
    : ((s._CntNodes) t.OptInfo^ (e.NewFunctions)) e.DriveTree
    = t.OptInfo
      <ColdTree <MapUnBracket <FlatDriveTree e.DriveTree>>>
      (e.NewFunctions);
}

MapUnBracket {
  e.Sequence = <Map &UnBracket e.Sequence>;
}

ColdTree {
  e.Branches
    = <Map
      {
        (e.Contractions (e.DrivenExpr))
          = (e.Contractions (<ColdAll e.DrivenExpr>))
      }
      e.Branches
      >;
}

ColdAll {
  CallBrackets e.Expr
    = ColdCallBrackets e.Expr;
  ColdCallBrackets e.Expr
    = ColdCallBrackets e.Expr;
  e.B (e.Expr) e.E
    = <ColdAll e.B> (<ColdAll e.Expr>) <ColdAll e.E>;
  e.X = e.X;
}

MaxCntNodes { = 100 }
```

```
$DRIVE MaxCntNodes;

MaxTreeDepth { = 20 }

$DRIVE MaxTreeDepth;

IncWithMax {
  s.Cnt s.Max, <Compare s.Max s.Cnt> : '+'
    = <Add s.Cnt 1>;
  s.Cnt s.Max = s.Max;
}

CheckExprsLength {
  0 e._ = TooMany;
  s.Cnt t.Expr e.Exprs = <CheckExprsLength <Sub s.Cnt 1>
e.Exprs>;
  s.Cnt = OK;
}

MakeDriveTree {
  (s.CntNodes) (e.UsedVars) (e.WholeVars) t.OptInfo e.Expr
(s.TreeDepth e.canoninzedExprs)
    = (<OptTree-CanonizeExpr e.Expr>) e.canoninzedExprs
    : e.canoninzedExprs^
    = <IncWithMax s.CntNodes <MaxCntNodes>>
      <IncWithMax s.TreeDepth <MaxTreeDepth>>
    : s.CntNodes^ s.TreeDepth^
    = <CheckExprsStopRelation e.canoninzedExprs>
    : {
        True s.Cnt
          = Generalize s.Cnt;
        s._ s._, <MaxCntNodes> : s.CntNodes
          = ((s.CntNodes) t.OptInfo (/* нет функций */))
            (/* нет сужений */ (e.Expr)(/*пустое дерево*/) );
        s._ s._, <MaxTreeDepth> : s.TreeDepth
          = ((s.CntNodes) t.OptInfo (/* нет функций */))
            (/* нет сужений */ (e.Expr)(/*пустое дерево*/) );
        False s._Cnt
          = <GrowDriveTree
              (s.CntNodes) (e.UsedVars) (e.WholeVars)
              t.OptInfo e.Expr
              (s.TreeDepth e.canoninzedExprs)
            >;
    };
}

GrowDriveTree {
  (s.CntNodes) (e.UsedVars) (e.WholeVars) t.OptInfo e.Expr
```

```
(s.TreeDepth e.canoninzedExprs)
  = t.OptInfo
  : ((e.OptFuncNames) e.OptFuncs)
  = <FindOptimizedCall (e.OptFuncNames) e.Expr>
  : {
      (e.OptFuncNames^ None) e.Expr^
        = ((s.CntNodes) ((e.OptFuncNames) e.OptFuncs)
           (/* нет функций */))
          (/* нет сужений */ (e.Expr)(/*пустое дерево */));

      (e.OptFuncNames^ t.Call) e.Expr^
        = <OptExpr-Aux (e.UsedVars) (e.WholeVars) (e.Expr)
                       t.Call e.OptFuncs>
        : (e.OptFuncs^) (e.NewFunctions) e.Branches
        = e.UsedVars <BranchesVars e.Branches>
        : e.UsedVars^
        = <MapAccum
            {
              ((s.CntNodes^)
               t.OptInfo^ (e.NewFunctionsAccum))
              (e.Contractions (e.DrivenRight))
                = <MakeDriveTree
                    (s.CntNodes) (e.UsedVars) (e.WholeVars)
                    t.OptInfo e.DrivenRight
                    (s.TreeDepth e.canoninzedExprs)
                  >
                : {
                    ((s.CntNodes^) t.OptInfo^
                     (e.NewFunctions^)) e.Branches^
                      = ((s.CntNodes) t.OptInfo
                         (e.NewFunctionsAccum
                          e.NewFunctions))
                        (e.Contractions (e.DrivenRight)
                         (e.Branches));
                    Generalize s.Cnt
                      = (Generalize s.Cnt);
                  };
              (Generalize s.Cnt)
               (e.Contractions (e.DrivenRight))
                = (Generalize s.Cnt);
            }
            ((s.CntNodes) ((e.OptFuncNames) e.OptFuncs)
             (e.NewFunctions))
            e.Branches
          >
        : {
            (Generalize 1) e._DeletedTree
              = <GrowDriveTree
                   (s.CntNodes) (e.UsedVars) (e.WholeVars)
```

```
                    t.OptInfo
                    <RestoreDrivenCall t.Call e.Expr>
                    (s.TreeDepth e.canoninzedExprs)
                  >;
              (Generalize s.Cnt) e._DeletedTree
                = Generalize <Sub s.Cnt 1>;
              e.Tree = e.Tree;
            };
      };
}

RestoreDrivenCall {
  t.Call e.Expr
    = <ApplyContractions e.Expr
       ((<eDRIVEN> ':' <ColdAll t.Call>)) >
}

CheckExprsStopRelation {
  (e.currentExpr) e.historyExprs-B (e.historyExpr)
  e.historyExprs-E
    , <OptTree-CheckExprStopRelation (e.currentExpr)
       e.historyExpr> : True
    = <Lenw e.historyExprs-B> : s.Len e._
    = True <Inc s.Len>;

  (e.currentExpr) e.historyExprs = False 0;
}

BranchesVars {
  e.Branches
    = <Map
        {
          (e.Contractions (e.DrivenExpr))
            = <ContractionVars e.Contractions>
        }
        e.Branches
      >;
}

ContractionVars {
  e.Contractions
    = <Map
        {
          (t.Var ':' e.Expr) = <ExtractVariables-Expr e.Expr>
        }
        e.Contractions
      >;
}
```

```
FlatDriveTree {
  e.Branches
    = <Map
        {
          (e.Branch)
            = (<FlatDriveTreeAux (e.Branch)>)
        }
        e.Branches
      >;
}

FlatDriveTreeAux {
  (e.Contractions (e.DrivenRight) (e.Branches))
    = <FlatDriveTree e.Branches> : e.BrachPacks
    = <FlatDriveTreeSingle (e.Contractions (e.DrivenRight)
(e.BrachPacks))>;
}

FlatDriveTreeSingle {
  (e.Contractions (e.DrivenRight) (/* пустое дерево */))
    = (e.Contractions (e.DrivenRight));

  (e.Contractions (e.DrivenRight) (e.BrachPacks))
    = <Map
        {
          (e.Branches)
            = <Map
                {
                  (e.BrachContractions (e.BranchDrivenRight))
                    = (
                        <ComposeContractions (e.Contractions)
                         (e.BrachContractions)>
                        (e.BranchDrivenRight)
                      )
                }
                e.Branches
              >;
        }
        e.BrachPacks
      >;
}

ComposeContractions {
  (e.A) (e.B)
    = <Map
        {
          ((e.ALeft) ':' e.ARight)
            = (
                (e.ALeft) ':'
```

```
                  <Map
                    {
                      e.ARightPart
                        = <ApplyContractions e.ARightPart
                            (e.B)>;
                    }
                    e.ARight
                  >
                );
          }
          e.A
        > e.B;
}
```