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:

«Использование отношения Хигмана-Крускала для прерывания рекурсивной специализации функций»

Студент ИУ9-82(Б) (Группа)	_____ (Подпись, дата)	А. А. Кошелев (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)	С. Ю. Скоробогатов (И.О.Фамилия)

2020 г.

АННОТАЦИЯ

Темой данной работы является «Использование отношения Хигмана-Крускала для прерывания рекурсивной специализации функций». Объём данной дипломной работы составляет 50 страниц. Для её написания было использовано 11 источников. В работе содержатся 15 листингов, 10 рисунков и 1 таблица.

Объектом исследований данной ВКР является проблема зацикливания, возникающая при рекурсивной специализации функций.

Цель работы — устранение зацикливания компилятора языка Рефал-5λ, возникающего при специализации функций, с помощью использования отношения Хигмана-Крускала.

Дипломная работа состоит из пяти глав. В первой главе рассматриваются основные теоретические сведения о специализации функций, проблеме рекурсивного зацикливания специализации, отношении Хигмана-Крускала, обобщении конфигураций, применении методов суперкомпиляции для решения проблемы зацикливания специализации, языке Рефал-5λ и реализации специализации функций в его компиляторе. Во второй главе обсуждаются алгоритмы применения отношения Хигмана-Крускала и обобщения сигнатур. В третьей главе рассматриваются детали реализации выбранных алгоритмов. В четвёртой главе осуществляются проверка корректности работы реализованных методов и измерение времени выполнения компиляции. В последней главе изложено руководство пользователя по применению компилятора Рефала-5

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Многие современные компиляторы используют методы, позволяющие увеличить скорость выполнения программы или уменьшить объём требуемой для выполнения памяти за счёт преобразования кода программы. При этом результат работы программы меняться не должен. Такие методы называют методами оптимизации программ [1], а часть из них — методами суперкомпиляции [2].

Одним из таких методов является специализация, заключающаяся в построении новой версии программы, в которой часть аргументов принимает конкретные значения. Использование специализации открывает возможность применения других оптимизационных методов, но она может использоваться и как самостоятельный метод оптимизации.

Применение специализации в ряде случаев может оказаться затруднительным. Так, её использование может привести к зацикливанию компилятора, из-за чего в лучшем случае компилятор сгенерирует большое количество ненужного кода, а в худшем компиляция не сможет завершиться вообще. Рассмотрение проблемы зацикливания представляет интерес в связи с небольшим количеством работ, посвящённых обсуждению различных решений данной проблемы и применению их на практике.

Подобная проблема на момент начала написания работы присутствовала в компиляторе языка Рефал-5λ [3], в который в 2019 году была добавлена специализация функций [4].

Целью данной работы является устранение зацикливания компилятора языка Рефал-5λ, возникающего при специализации функций, с помощью использования отношения Хигмана-Крускала.

Для выполнения этой цели были поставлены следующие задачи:

1. Выполнить обзор предметной области, включающий изучение процесса возникновения зацикливания специализации, способа его обнаружения при помощи отношения Хигмана-Крускала, метода обобщения конфигураций, языка Рефал-5λ и его компилятора.

2. Разработать алгоритм применения отношения Хигмана-Крускала и алгоритм обобщения сигнатур.

3. Реализовать разработанные алгоритмы в компиляторе языка Рефал-5λ.

4. Протестировать корректность работы реализованных методов и изменение времени выполнения компиляции.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Специализация функций и зацикливание

Сначала рассмотрим более подробно понятие специализации функций. Пусть имеется функция $F(X, Y)$, принимающая на вход параметры X, Y, каждый из которых может принимать некоторое множество значение. Допустим, что одним из значений параметра Y является значение A. Построим новую версию функции F, для которой параметр Y всегда принимает значение A, и обозначим её $F_A(X)$. Результат работы функции $F_A(X)$ можно выразить так:

$$F_A(X) = F(X, A) \tag{1}$$

Тогда функция $F_A(X)$ является специализацией функции $F(X, Y)$ по параметру Y. При этом параметры, по котором производится специализация, называют статическими, а остальные — динамическими.

Диапазон значений, которые могут принимать статические параметры, можно определить по-разному. Например, можно ограничить их до числовых и строковых констант. В рамках данной работы рассматривается случай, когда значениями статических параметров могут являться выражения, включающие в себя переменные тех или иных типов и вызовы функций.

Специализация осуществляется в предположении, что новая версия функции позволит увеличить скорость выполнения программы или понизить требуемый для её работы объём памяти. Кроме того, она может использоваться для анализа поведения функции при различных значениях её аргументов.

Рассмотрим общий алгоритм, по которому выполняется специализация функций в программе. Его можно представить в виде следующей последовательности шагов:

1. Происходит поиск вызова специализируемой функции.

2. Определяются значения статических параметров.
3. Если значения новые, строится экземпляр функции, соответствующий данным значениям. В ином случае указывается вызов уже построенного экземпляра.
4. Запоминается сигнатура вызова функции. Её можно представить в виде кортежа, состоящего из имени экземпляра и значений статических переменных.

Данная процедура повторяется, пока не останется необработанных вызовов специализируемых функций. В ряде случаев использование подобного алгоритма может привести к зацикливанию, причина которого состоит в том, что построение новых экземпляров специализируемых функций способно породить новые вызовы специализируемых функций (см. Рисунок 1).

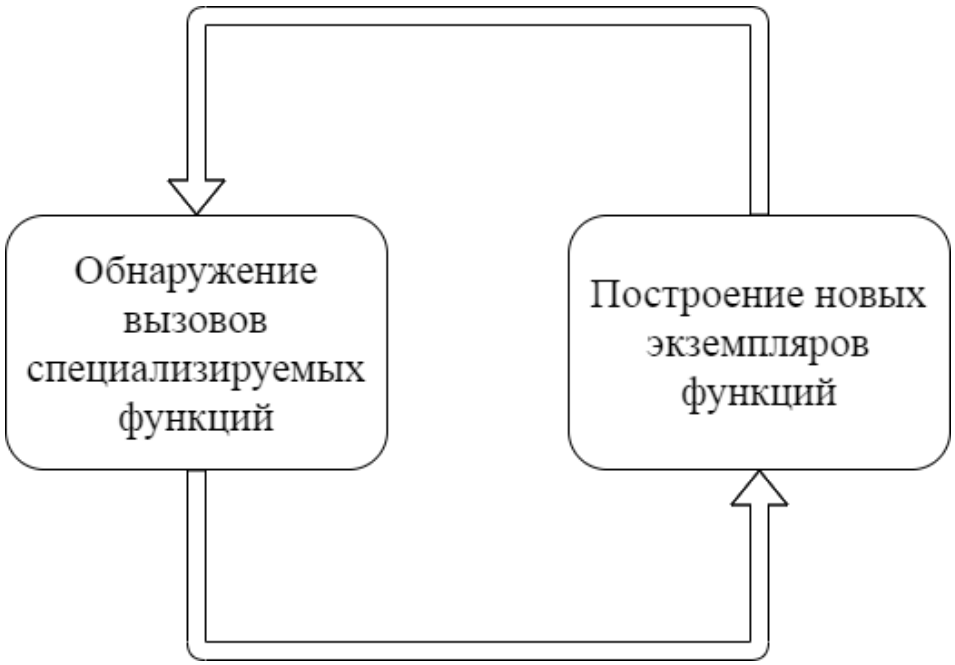

Рисунок 1 — Схема зацикливания алгоритма специализации функций

Особенно значимой данная проблема является для функциональных языков программирования, в которых зачастую используются рекурсивные вызовы. Вместо присвоения переменным программы новых значений в подобных языках нередко выполняется рекурсия с новыми значениями аргументов. При выполне-

нии специализации функций это может стать причиной построения новых экземпляров функций, также содержащих рекурсивные вызовы с, возможно, новыми значениями статических параметров. Чтобы более наглядно продемонстрировать процесс зацикливания, процедуру специализации функций в программе можно представить в виде ориентированного графа, в узлах которого находятся экземпляры, характеризующиеся сигнатурой, а рёбра графа означают, что один экземпляр порождает другой. Пример графа, иллюстрирующего одну из возможных ситуаций зацикливания, приведён на рисунке 2.

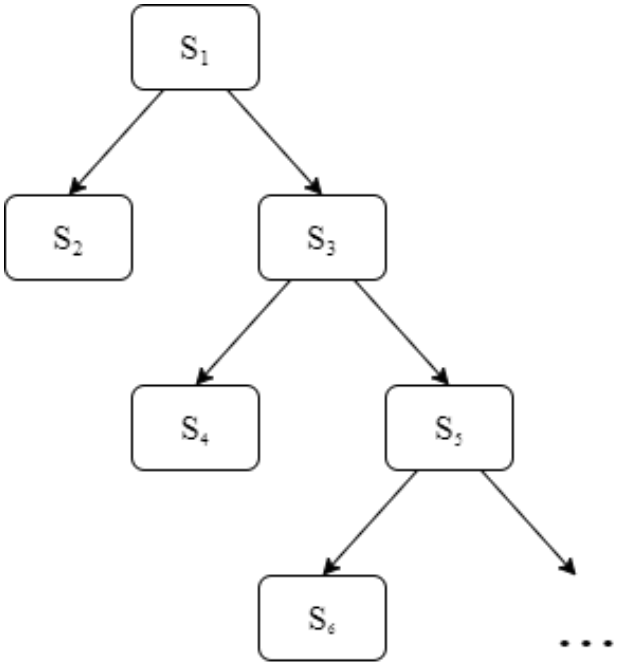

Рисунок 2 — Пример графа сигнатур, иллюстрирующего случай зацикливание специализации функций

Итак, необходимо найти способ решения проблемы зацикливания специализации. В данной работе для этого предлагается рассмотреть схожую проблему, возникающую при использовании метода преобразования программ, называемого суперкомпиляцией, и её решение, которое можно адаптировать под случай специализации функций.

Термин суперкомпиляции впервые был введён советским и американским физиком и кибернетиком В. Ф. Турчиным. При этом суперкомпиляцию можно

рассматривать в виде некоторого процесса действий как над программой, описанной в общих чертах, так и над программой, записанной на конкретном языке программирования.

Идею метода суперкомпиляции [5] можно сформулировать так. Поданной на вход программе сопоставляется машина, моделирующая поведение программы в общем виде. Данная машина описывается различными обобщёнными состояниями процесса вычислений, называемыми конфигурациями, и переходами между данными состояниями. Конфигурации представляются выражениями с параметрами. Помимо параметров, они могут содержать, например, последовательности символов, числа и вызовы функций. При подстановке вместо параметров конфигурации конкретных значений получают конкретное состояние машины.

Сопоставленная исходной программе машина представляется в виде графа, вершины которого помечены конфигурациями, а рёбра указывают ограничения на параметры конфигурации, при которых исходная конфигурация превращается в дочернюю. Такие переходы осуществляются с помощью методов прогонки и обобщения. Под прогонкой понимается выполнение спекулятивного шага вычислительной системы. При обобщении происходит переход от конфигурации C к конфигурации $let\ v' = C'\ in\ C''$, после чего в качестве дочерних рассматриваются конфигурации C' и C''. Таким образом выполняется переход от более частной конфигурации к более общей, которая сводится к первой некоторой подстановкой. Если при прогонке или обобщении возникают дочерние конфигурации, совпадающие с имеющимися в графе с точностью до переименования параметров, новые узлы не строятся. Вместо этого добавляются обратные рёбра в уже имеющиеся вершины.

Цель суперкомпиляции — построить конечный самодостаточный граф, являющийся моделью вычислений стартовой конфигурации. По нему можно будет построить остаточную программу, которая будет эквивалентна вычислению стартовой конфигурации с определениями функций из исходной программы. Предполагается, что такая программа может работать быстрее.

В качестве примера рассмотрим функцию $BracketsWrapper(x, y)$, принимающую на вход параметры x и y, каждый из которых является либо выражением, состоящим из некоторого количества скобок, внутри которых находится символ a (например, $((a))$), либо символом a. Действие данной функции заключается в следующем. Если параметр x равен символу a, функция возвращает значение параметра y. Если же x является выражением в скобках, функция "снимает" скобки с параметра x и оборачивает в них параметр y, после чего выполняет рекурсивный вызов. Во втором случае задаётся не конкретное значение параметра x, а множество таких значений.

Попробуем просуперкомпилировать данную функцию. Применив один раз метод прогонки, получим граф конфигураций, представленый на рисунке 3.

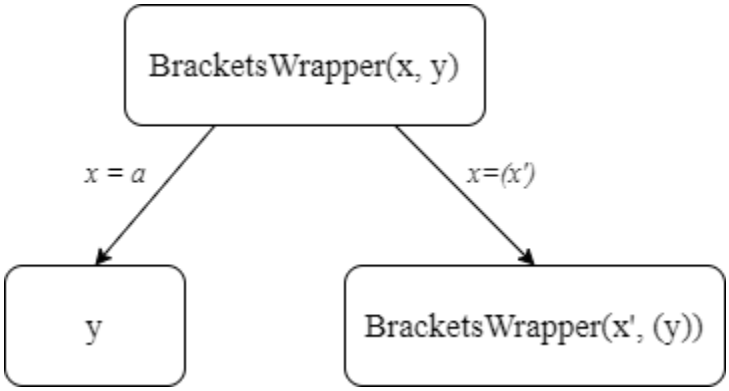

Рисунок 3 — Граф конфигураций функции $BracketsWrapper(x, y)$ после одного шага суперкомпиляции

Попытка продолжить суперкомпиляцию функции $BracketsWrapper(x, y)$ приведёт к получению графа, изображённого на рисунке 4.

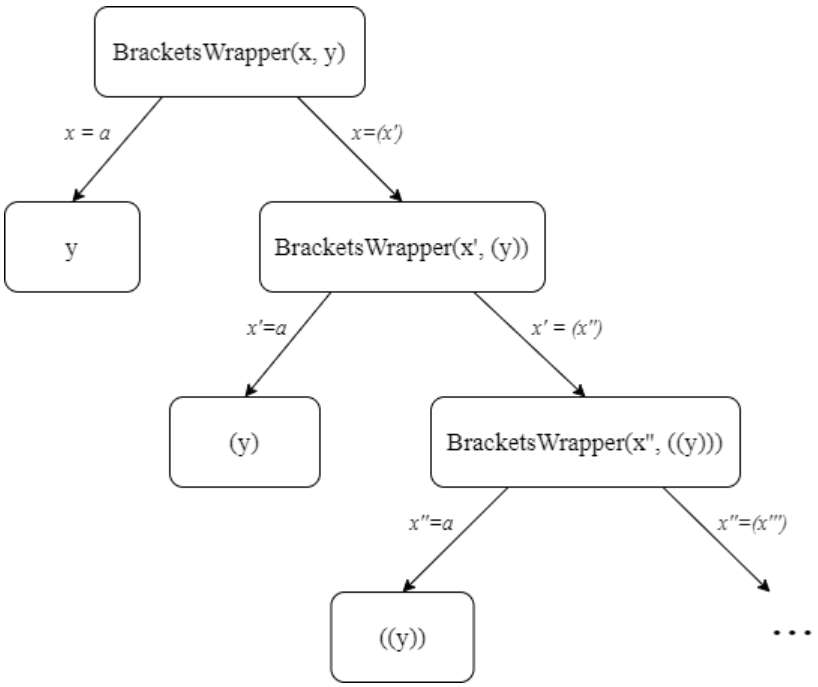

Рисунок 4 — Часть графа конфигураций для функции $BracketsWrapper$

Как видно из данного рисунка, без введения дополнительных действий процесс суперкомпиляции функции $BracketsWrapper(x, y)$ зацикливается, в результате чего может продолжаться бесконечно. Это связано с тем, что при использовании прогонки возникает конфигурация с функцией $BracketsWrapper(x, y)$, в которой второй параметр отличается от всех предыдущих. Последовательность значений, принимаемых вторым параметром выглядит так: y, (y), $((y))$, $\left(((y))\right)$ и т. д.

Как уже было замечено ранее, основная задача суперкомпиляция — построить конечный граф. Поэтому возникает потребность в создании мер, позволяющих обнаружить и устранить зацикливание.

1.2 Прерывание зацикливания суперкомпиляции

Для решения рассматриваемой проблемы возникает идея ввести некоторой способ определения схожести конфигураций, который позволил бы своевре-

менно обнаружить зацикливание. В качестве такого способа может быть использовано отношение Хигмана-Крускала [6], также именуемое гомеоморфным вложением. Оно основано на следующем наблюдении. Во время зацикливания процесса суперкомпиляции возникают конфигурации, обладающие следующим свойством. Путём удаления некоторых элементов (символов, параметров, функций и т. д.) из выражений более новых конфигураций можно получить одну из предыдущих конфигураций. Например, в случае функции $BracketsWrapper(x, y)$ при удалении скобок в выражении (y) будет получено выражение для предыдущей конфигурации: $(y) \rightarrow y$. Данное свойство сохраняется и при рассмотрении более сложных функций.

Отношение Хигмана-Крускала обозначается символом $\trianglelefteq$. Если конфигурацию C_1 можно получить путём удаления из конфигурации C_2 некоторых элементов, то пишут $C_1 \trianglelefteq C_2$, подразумевая, что C_1 "вкладывается" в C_2. Формальное определение отношения Хигмана-Крускала можно задать индуктивно с помощью следующих правил:

1. $x \trianglelefteq y$ для любых параметров x и y.

2. $X \trianglelefteq f(Y_1, Y_2, \dots, Y_n)$, если f — функция и $\exists\, i: X \trianglelefteq Y_i$.

3. $f(X_1, X_2, \dots, X_n) \trianglelefteq f(Y_1, Y_2, \dots, Y_n)$, если f — функция и $\forall\, i = 1, \dots, n\ X_i \leq Y_i$.

Можно привести следующие примеры выполнения отношения Хигмана-Крускала: $x \trianglelefteq (y)$, $(y) \trianglelefteq \big(((y))\big)$, $x\ y \trianglelefteq f\big(x, g(y)\big)$.

Итак, отношение Хигмана-Крускала позволяет определить похожие конфигурации. Однако гарантирует ли его использование обнаружение зацикливания процесса суперкомпиляции? Ответ на этот вопрос следует из теоремы Хигмана-Крускала [6][7][8], которая формулируется следующим образом: отношение Хигмана-Крускала является отношением хорошего предпорядка для выражений в конечном алфавите. Применительно к рассматриваемой проблеме это означает следующее: если множество символов, из которых составляются пара-

метры в конфигурациях, является конечным, то для любой бесконечной последовательности конфигураций $\{C_i\}_{i\in\mathbb{N}}$ найдутся такие индексы i и k, $i < k$, что будет выполняться $C_i \trianglelefteq C_k$. Значит, если в ходе суперкомпиляции появится зацикливание, то в некоторый момент его обязательно можно будет обнаружить с помощью отношения Хигмана-Крускала.

Итак, отношение Хигмана-Крускала позволяет обнаружить зацикливание, из-за которого задача построения конечного графа конфигураций может оказаться невыполнимой. Однако обнаружение зацикливания само по себе не решает проблему полностью, так как необходимо определить дальнейшие действия, совершаемые с похожими конфигурациями.

1.3 Обобщение конфигураций

После обнаружения похожих по отношению Хигмана-Крускала конфигураций в суперкомпиляции применяется метод обобщения. Пусть на некотором шаге проведения суперкомпиляции было обнаружено, что для конфигураций C_i и C_k выполняется $C_i \trianglelefteq C_k$. В таком случае имеем одну из следующих ситуаций:

1. Родительская конфигурация имеет вид $f(x)$, а дочерняя — $f\big(g(x)\big)$. В таком случае дочернюю конфигурацию обобщают в виде $let\ y = g(x)\ in\ f(y)$ (см. Рисунок 5). Тогда конфигурация $f(y)$ зацикливается в родительскую, то есть добавляется ребро из конфигурации $f(y)$ в конфигурацию $f(x)$. Конфигурация $g(x)$ подлежит дальнейшему рассмотрению.
2. Родительская конфигурация представлена выражением $f(x)$, а дочерняя — выражением $g\big(f(x)\big)$. В такой ситуации дочернюю конфигурацию обобщают в виде $let\ y = f(x)\ in\ g(y)$ (см. Рисунок 6), конфигурация $f(x)$ зацикливается в верхнюю конфигурацию, а $g(y)$ суперкомпилируется дальше.

3. Родительская конфигурация имеет вид $f\big(g(x)\big)$, а дочерняя — $f\Big(h\big(g(x)\big)\Big)$. В данном случае возможно применение одного из двух подходов, именуемых обобщением сверху (см. Рисунок 7) и обобщением снизу (см. Рисунок 8). При обобщении сверху весь граф, построенный от родительской вершины, отбрасывается, после чего родительская вершина обобщается до выражения $let\ y = g(x)\ in\ f(y)$, дочерние конфигурации $g(x)$ и $f(y)$ суперкомпилируются отдельно. При обобщении снизу дочерняя вершина обобщается до вида $let\ y = h\big(g(x)\big)\ in\ f(y)$, дальше суперкомпилируются конфигурации $h\big(g(x)\big)$ и $f(y)$.

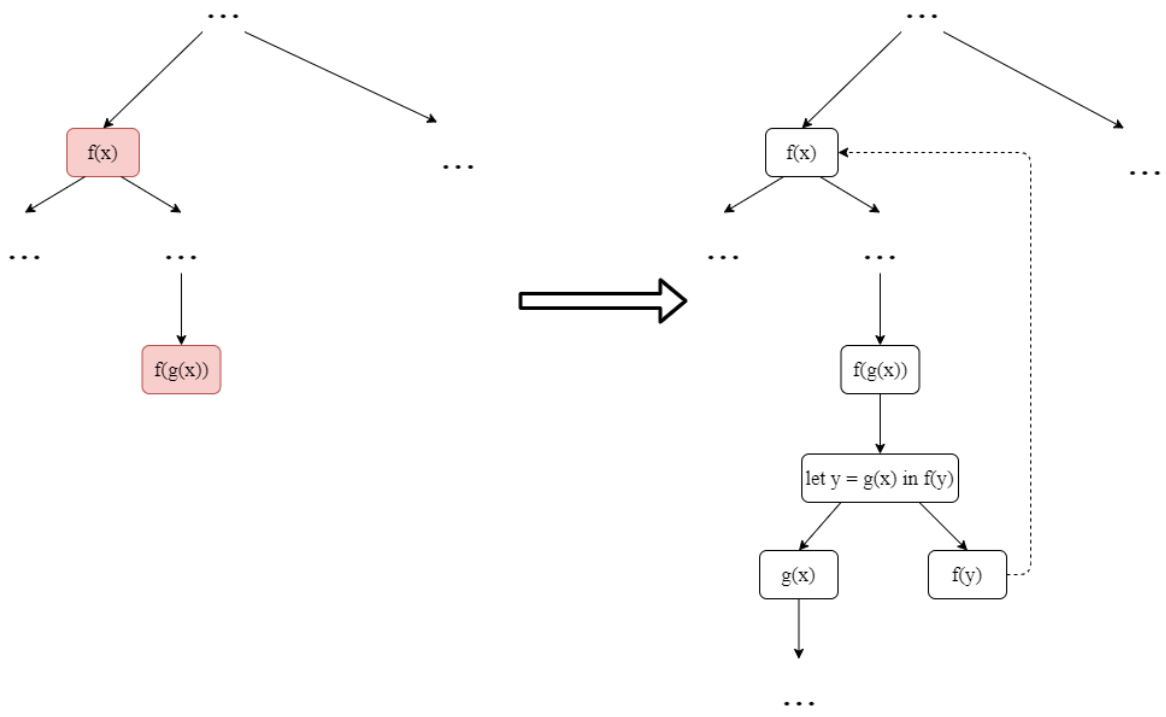

Рисунок 5 — Первый случай обобщения конфигураций

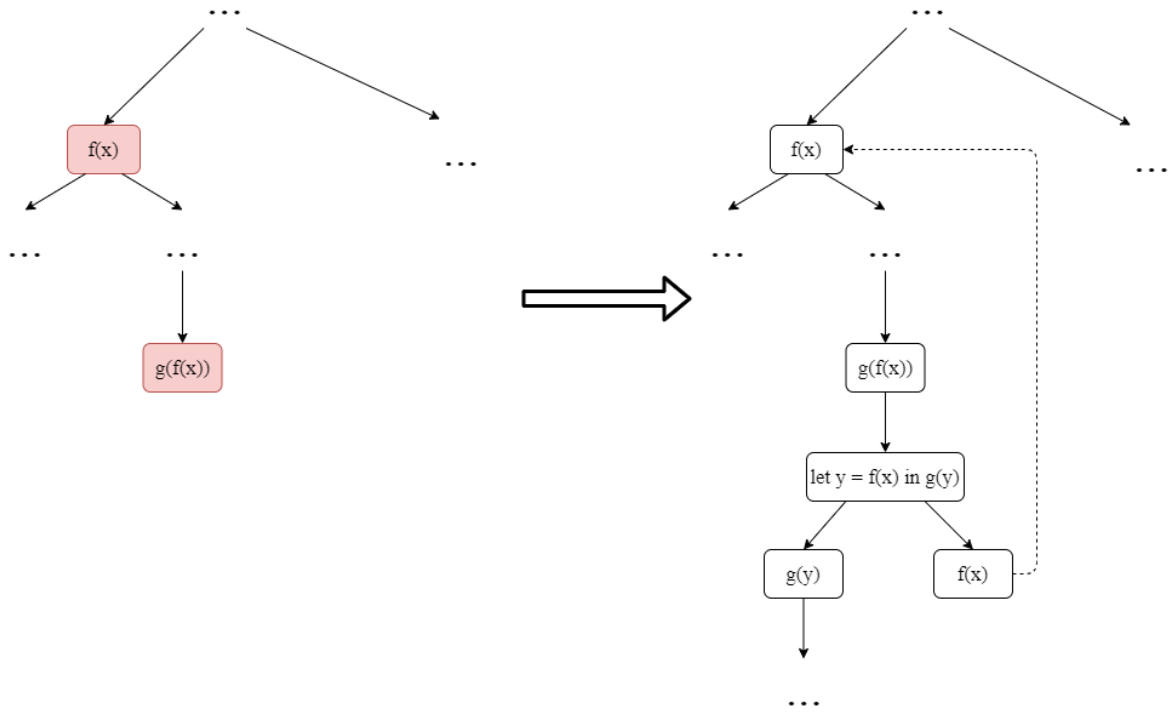

Рисунок 6 — Второй случай обобщения конфигураций

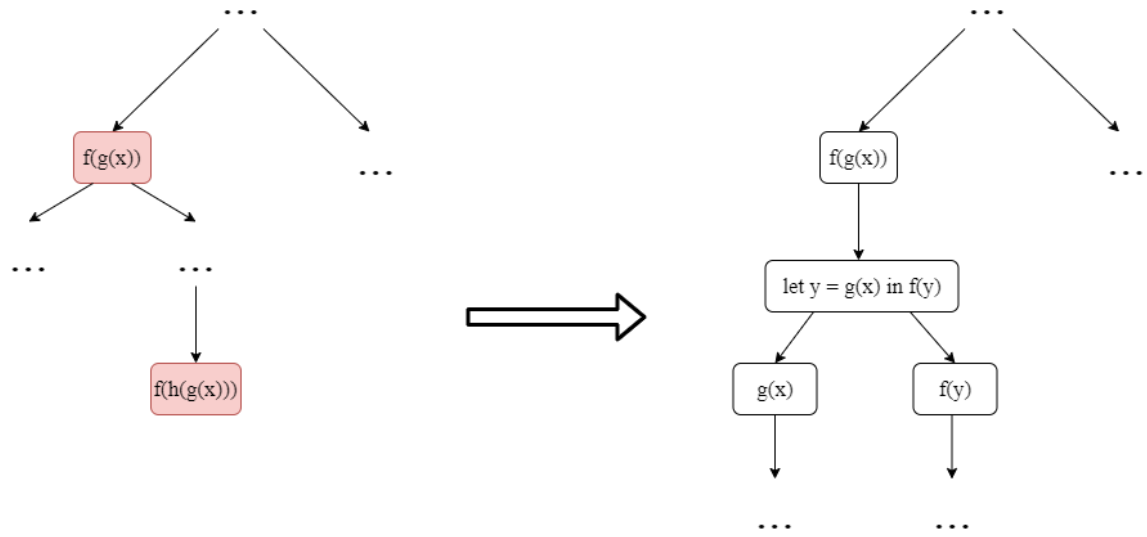

Рисунок 7 — Метод обобщения конфигураций сверху

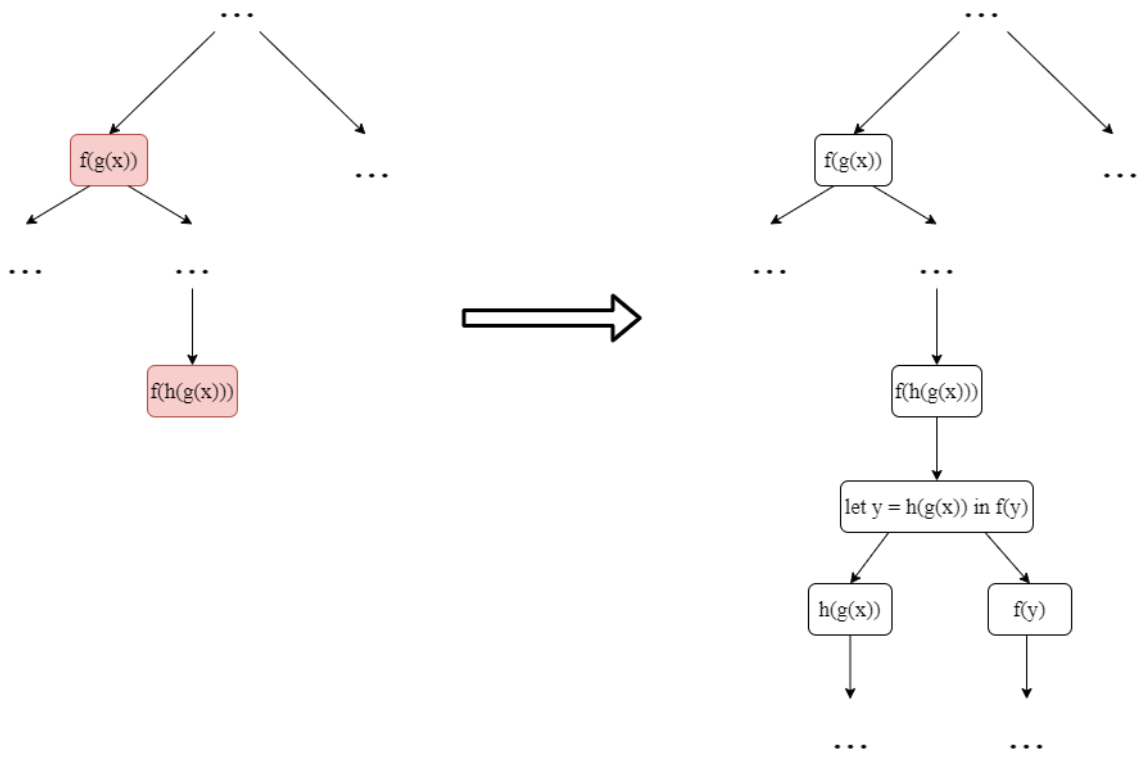

Рисунок 8 — Метод обобщения конфигураций снизу

Итак, были рассмотрены возможные варианты действий при обнаружении конфигураций, для которых выполняется отношение Хигмана-Крускала.

1.4 Обнаружение и прерывание зацикливания специализации функций

Теперь обсудим применение описанных выше методов для решения проблемы зацикливания специализации функций. Сначала необходимо отметить, что отношение Хигмана-Крускала применимо и на множестве сигнатур, так как в начале данной главы было определено, что рассматривается случай специализации функций, когда значения статических параметров могут включать в себя вызовы функций и переменные различных типов. Кроме того, имеется возможность построить граф сигнатур, отображающий процесс специализации функций в программе и являющийся некоторой упрощённой аналогией графа конфигураций.

Чтобы обеспечить выполнение теоремы Хигмана-Крускала при рассмотрении одноимённого отношения на множестве сигнатур, достаточно при сравнении переменных в конфигурации учитывать только их тип без имени. Так как количество типов переменных ограничено, рассматриваемое условие будет удовлетворено. Значит, при данных ограничениях отношение Хигмана-Крускала может быть использовано для обнаружения зацикливания специализации функций.

Как следует поступить при обнаружении сигнатур, для которых сработало отношение Хигмана-Крускала? Конечно, существует простой вариант дальнейших действий: прекратить специализацию функции, для некоторых экземпляров которой сработало отношение Хигмана-Крускала. Однако, как правило, специализация применяется в сочетании с другими методами оптимизации программы и расширяет возможности их применения благодаря рассмотрению различных частных случаев работы функций. Значит, с точки зрения увеличения скорости генерируемой программы более выгодно рассмотреть как можно большее количество частных случаев. В случае прекращения специализации останавливается и процесс рассмотрения частных случаев. Если же применить некоторый аналог метода обобщения конфигураций, можно добиться рассмотрения большего количества частных случаев.

Опираясь на метод обобщения, используемый в суперкомпиляции, опишем способ обобщения сигнатур. Предположим, что на некотором проходе специализации функций были обнаружены сигнатуры S_i и S_k, для которых выполняется $S_i \trianglelefteq S_k$. Тогда строим сигнатуру S_{gen}, для которой должны существовать подстановки, позволяющие преобразовать её в сигнатуры S_i и S_k.

В качестве примера можно рассмотреть сигнатуры $S_1 = (\,)\ x\ y$, $S_2 = (1)\ x\ y$. Очевидно, что выполняется $S_1 \trianglelefteq S_2$. Их обобщением является сигнатура $S_{gen} = (z)\ x\ y$, так как, применяя к ней подстановки $z \to \varepsilon$ и $z \to 1$ можно получить сигнатуры S_1 и S_2 соответственно.

Сигнатура S_{gen} может совпасть с одной из уже построенных сигнатур S_j с точностью до переименования переменных. Тогда в граф сигнатур необходимо добавить обратное ребро из нижней сигнатуры S_k в сигнатуру S_j. Добавление

такого ребра соответствует указанию вызова экземпляра S_j в экземпляре S_k. В ином случае нужно использовать адаптированную версию метода обобщения сверху или метода обобщения снизу, которые можно определить следующим образом. При обобщении сверху (см. Рисунок 9) необходимо удалить подграф, выросший из верхней сигнатуры S_i, и добавить ребро из вершины S_i в сигнатуру обобщения S_{gen}.

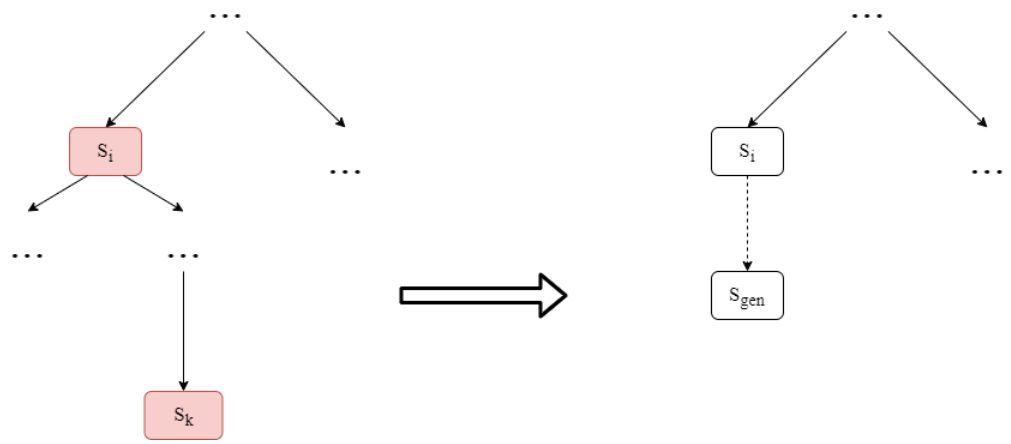

Рисунок 9 — Использование метода обобщения сверху на графе сигнатур

При использовании обобщения снизу к нижней сигнатуре S_k необходимо добавить ребро в сигнатуру обобщения S_{gen} (см. Рисунок 10).

Таким образом, методы, применяемые в суперкомпиляции, могут быть использованы для решения проблемы зацикливания специализации функций.

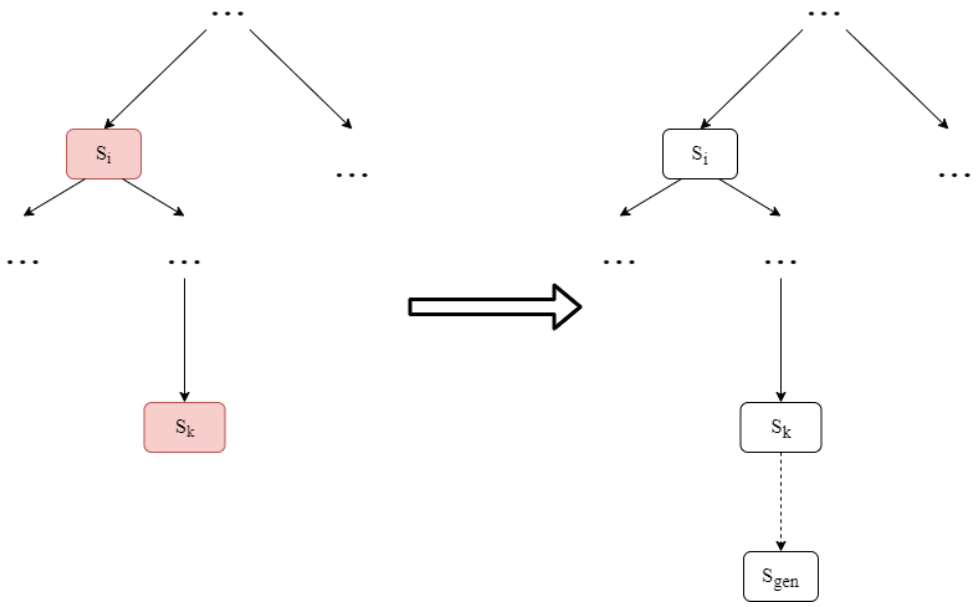

Рисунок 10 — Использование метода обобщения снизу на графе сигнатур

1.5 Язык Рефал-5λ

Компилятор языка Рефал-5λ является самоприменимым. Это означает, что код компилятора также написан на языке Рефал-5λ. В связи с этим для реализации в нём алгоритмов устранения проблемы зацикливания необходимо изучить основы его синтаксиса и особенности работы компилятора.

Рефал-5λ [9] является диалектом языка Рефал (Рекурсивных Функций Алгоритмический язык), первая версия которого была разработана В. Ф. Турчиным. Рефал является функциональным языком программирования, ориентирован на выполнение символьных вычислений. Со временем появилось множество различных диалектов Рефала, отличающихся как синтаксисом, так и набором возможностей. Основной особенностью диалекта Рефал-5λ является возможность использования безымянных функций, также называемых λ-функциями. Кроме того, поддерживается возможность передачи функций в качестве аргументов других функций. Данное отличие позволяет эффективно использовать такие механизмы обработки данных, как `Map`, `Reduce`, `MapAccum`, `Filter` и прочие. На данный момент подобные механизмы являются неотъемлемой частью многих

современных языков программирования. Также важно подчеркнуть, что Рефал-5λ является точным надмножеством языка Рефал-5. В силу этого компилятор Рефала-5λ может работать и с программами, написанными на языке Рефал-5.

Синтаксис языка Рефал-5λ можно описать следующим образом. Программа состоит из набора функций. Каждая из них, в свою очередь, состоит из предложений. Посредством символа равенства предложения разделяются на две части: образец и выражение, задающее результат работы функции. При этом каждая функция принимает на вход один аргумент, который последовательно сопоставляется с образцами функции, в результате чего выполняется результатное выражение одного из предложений функции.

Образцы чаще всего состоят из идентификаторов, строк, чисел, переменных и скобочных символов. Круглые скобки именуют структурными, так как они, как правило, используются для задания структур данных. Квадратные скобки называют именованными. Всего выделяют 3 типа переменных: s-, t- и e-переменные. Для определения диапазона значений переменной каждого типа необходимо ввести понятия символа, терма и объектного выражения.

Символ является минимальным элементом описания данных в языке Рефал-5λ. Выделяют следующие виды символов:

- неотрицательные целые числа, называемые макроцифрами;
- печатные знаки, заключённые в одинарные кавычки;
- идентификаторы, представленные последовательностями буквенных символов, цифр, нижних подчёркиваний и дефисов;
- указатели на функции, состоящие из символа `'&'` и следующим за ним идентификатором функции.

Термом называют либо произвольный символ, либо выражение в структурных или именованных скобках. Объектным выражением называют последовательность из нуля или более термов.

Переменные типа s могут быть сопоставлены с символами, переменные типа t — с термами, переменные типа e — с объектными выражениями. Переменные в коде программы записываются в виде типа переменных, за которым следует знак точки, после которой располагается имя переменной.

В качестве примера приведём образец

```
(s.Index) t.FirstTerm e.OtherTerm
```
(2)

С данным образцом может быть сопоставлено, например, выражение

```
(1) ((symbol)) (string) ()
```
(3)

В таком случае переменная `s.Index` примет значение `1`, переменная `t.FirstTerm` — значение `((symbol))`, а переменная `e.OtherTerms` — значение `(string) ()`.

Определение функций осуществляется в соответствии со следующей грамматикой:

```
Function ::= $ENTRY? Name { Sentences }
```
(4)

Ключевое слово `$ENTRY` указывает, является ли функция глобальной. Для вызова функций используются угловые скобки. При выполнении программы сначала запускается функция с именем `Go` или `GO`, помеченная ключевым словом `$ENTRY`. Пример программы на языке Рефал-5λ приведён на листинге 1.

```
$ENTRY Go {
  /* пусто */
    = <Prout <Sum (0) 1 2 3 4 5> >;
}

Sum {
  (s.Sum) s.Number e.Numbers
    = <Sum (<Add s.Sum s.Number>) e.Numbers>;
  (s.Sum) /* пусто */ = s.Sum;
}
```

Листинг 1 — Пример программы на языке Рефал-5λ, считающей сумму чисел

1.6 Специализация функций в компиляторе языка Рефал-5λ

Далее рассмотрим процесс выполнения специализации функций в компиляторе языка Рефал-5λ. Текущее состояние программы в компиляторе представляется в виде абстрактного синтаксического дерева. Проход специализации обходит всё дерево и для каждого вызова специализирумой функции в программе либо строит экземпляр, либо вызывает имеющийся, если сигнатура уже встречалась. Следующий проход специализации обходит новые экземпляры, анализирую вызовы в них. Проходы повторяются до неподвижной точки, то есть пока не перестанут строиться новые экземпляры, или пока счётчик проходов не исчерпается. Зацикливание специализатора происходит, если на каждом проходе будет строиться экземпляр, который на следующем проходе породит новый экземпляр.

В качестве примера зацикливания компилятора рассмотрим реализацию рассмотренной ранее функции $BracketsWrapper(x, y)$ на языке Рефал-5λ. Она приведена на листинге 2.

```
BracketsWrapper {
  (t.X) t.Y = <BracketsWrapper t.X (t.Y)>;
  a t.Y = t.Y;
}
```

Листинг 2 — Реализация функции $BracketsWrapper$ на языке Рефал-5λ

Скомпилируем данную функцию, указав при этом компилятору выполнять специализацию функции `BracketsWrapper` по второму параметру и ограничив количество оптимизационных проходов до пяти. Код программы, полученный после применения специализации, приведён на листинге 3.

```
BracketsWrapper {
  (t.X#1) t.Y#1 = <BracketsWrapper@1 t.X#1 t.Y#1>;

  a t.Y#1 = t.Y#1;
}


BracketsWrapper@1 {
  (t.X#1) t.Y0#1 = <BracketsWrapper@2 t.X#1 t.Y0#1>;
```

```
  a t.Y0#1 = (t.Y0#1);
}

BracketsWrapper@2 {
  (t.X#1) t.Y0#1 = <BracketsWrapper@3 t.X#1 t.Y0#1>;

  a t.Y0#1 = ((t.Y0#1));
}

BracketsWrapper@3 {
  (t.X#1) t.Y0#1 = <BracketsWrapper@4 t.X#1 t.Y0#1>;

  a t.Y0#1 = (((t.Y0#1)));
}

BracketsWrapper@4 {
  (t.X#1) t.Y0#1 = <BracketsWrapper@5 t.X#1 t.Y0#1>;

  a t.Y0#1 = ((((t.Y0#1))));
}

BracketsWrapper@5 {
  (t.X#1) t.Y0#1 = <BracketsWrapper t.X#1 ((((((t.Y0#1))))))>;

  a t.Y0#1 = (((((t.Y0#1)))));
}
```

Листинг 3 — Результат специализации функции $BracketsWrapper$

Как видно из данного листинга, было добавлено пять экземпляров функции `BracketsWrapper`. Их добавление значительно увеличивает занимаемый программой объём памяти. При отсутствии ограничения на количество оптимизационных проходов компиляция программы не смогла бы завершиться в силу зацикливания специализации.

2 РАЗРАБОТКА

2.1 Алгоритм применения отношения Хигмана-Крускала

Рассмотрим возможные алгоритмы проверки выполнения отношения Хигмана-Крускала на сигнатурах. Для применения отношения Хигмана-Крускала можно воспользоваться непосредственно его определением или же следствиями из него [10]. Опишем оба способа и сравним их между собой.

Напомним интуитивное определение отношения Хигмана-Крускала: для выражений E_1 и E_2 выполняется $E_1 \trianglelefteq E_2$, если путём удаления из E_2 некоторых элементов можно получить E_1. Следовательно, для применения отношения необходимо хранить сигнатуры функции и при получении новой по очереди сравнивать её с предыдущими. Сравнение может осуществляться путём удаления из новой сигнатуры некоторой комбинации элементов и последующей проверке на равенство с предыдущей сигнатурой. Таким образом выполняется полный перебор возможных комбинаций элементов в новой сигнатуре. Такой способ будет работать, однако он является неэффективным в общем случае, так как для сигнатуры из n элементов в худшем случае необходимо будет перебрать 2^n комбинаций элементов.

Теперь рассмотрим способ, использующий следствия из определения отношения Хигмана-Крускала. Сначала сформулируем сами следствия. Пусть имеются выражения E_1 и E_2, причём выражение E_2 можно представить в виде конкатенации $E_2'E_2''$. Тогда, если $E_1 \trianglelefteq E_2$, то верно одно из трёх:

1. $E_1 \trianglelefteq E_2'$.

2. $E_1 \trianglelefteq E_2''$.

3. E_1 можно представить в виде конкатенации $E_1'E_1''$, где $|E_1'| > 0$ и $|E_1''| > 0$, для которой выполняются условия: $E_1' \trianglelefteq E_2'$ и $E_1'' \trianglelefteq E_2''$.

Алгоритм проверки выполнения отношения Хигмана-Крускала, использующий данные следствия, следует непосредственно из их формулировок. Его можно представить в виде выполнения следующих действий:

1. Пусть рассматриваются сигнатуры S_1 и S_2, требуется проверить, выполняется ли $S_1 \trianglelefteq S_2$. Разделим сигнатуру S_2 на две части: S_2' и S_2''.

2. Проверяем, содержится ли сигнатура S_1 в сигнатуре S_2'. Если содержится, отношение выполняется. Завершаем алгоритм.

3. Проверяем, содержится ли S_1 в S_2''. Если содержится, отношение выполняется. Завершаем алгоритм.

4. Разделяем сигнатуру S_1 на две части: S_1' и S_1''. Для пар S_1' и S_2', S_1'' и S_2'' по отдельности запускаем алгоритм с первого шага. Если оба вызова возвращают положительный результат, отношение выполняется. В ином случае отношение не выполняется. Завершаем алгоритм.

Вместо перебора всех возможных комбинаций элементов сигнатуры в данном алгоритме используется поиск подвыражений и разделение выражений на части, что влечёт бóльшую скорость работы по сравнению с предыдущим методом.

После рассмотрения приведённых выше методов проверки выполнения отношения Хигмана-Крускала было решено использовать способ, основанный на применении следствий из определения, так как он превосходит алгоритм, работающий на основе определения отношения Хигмана-Крускала, по скорости выполнения.

2.2 Алгоритм обобщения сигнатур

Перейдём к обсуждению алгоритмов обобщения сигнатур. Было рассмотрено два подобных алгоритма: обобщение сверху и обобщение снизу. Сравним данные алгоритмы между собой.

С точки зрения сложности реализации и скорости выполнения алгоритм обобщения снизу является более приоритетным. Это связано с тем, что при использовании обобщения сверху необходимо выполнить откат текущего состояния программы до некоторого предыдущего состояния. Это означает, что в ходе применения оптимизационных методов компилятор должен хранить совершённые изменения и уметь совершать выполнение их отмены, что является трудоёмкой задачей само по себе. Кроме того, удаление части построенного дерева, совершаемое при обобщении сверху, может привести к потере ценной информации о поведении функции. В случае же обобщения снизу построенное поддерево не удаляется, и по завершению оптимизации оно может быть использовано другими алгоритмами оптимизации. Поэтому и в данном аспекте обобщение снизу является более выгодным. Среди недостатков обобщения снизу можно выделить большее количество вершин в итоговом графе сигнатур. Однако в некоторых случаях (например, при анализе поведения функции) данное свойство алгоритма обобщения снизу может оказаться скорее положительным, нежели негативным.

По итогам сравнения было решено использовать алгоритм обобщения снизу. Его основными преимуществами по сравнению с алгоритмом обобщения сверху являются скорость работы, меньшая сложность реализации и сохранение построенной части графа сигнатур. Далее рассмотрим более подробно алгоритм применения обобщения снизу.

Прежде всего необходимо построить сигнатуру, являющуюся обобщением для сигнатур, для которых выполнилось отношение Хигмана-Крускала. При этом обобщение двух сигнатур в общем случае определено неоднозначно, ведь им может являться несколько выражений.

В качестве примера рассмотрим следующие два выражения языка Рефал-5λ: `() t.1 t.2` и `('a') t.1 t.2`. Обобщением для них является как выражение `(e.1) t.1 t.2`, так и выражение `e.1`. Однако при специализации важно сохранять максимальное количество информации о сигнатуре, поэтому более предпочтительным в данном случае является первый вариант обобщения.

Чтобы получить из данного обобщения первое и второе выражение, нужно применить подстановки `e.1` → ε и `e.1` → `'a'` соответственно.

Для однозначного определения обобщения сигнатур можно воспользоваться реализованным в компиляторе языка Рефал-5λ алгоритмом глобального сложнейшего обобщения [11], который позволяет для двух образцовых выражений построить их обобщение в виде так называемого жёсткого выражения. Жёсткое выражение не содержит более одной *e*-переменной на одном скобочном уровне и каких-либо повторных переменных. Данный алгоритм при построении обобщения сохраняет максимум информации об обоих выражениях.

После построения обобщения нужно проверить, совпадает ли оно с одной из сигнатур, для которой уже был построен экземпляр функции. Если такую сигнатуру удалось найти, достаточно указать вызов соответствующего экземпляра. В ином случае необходимо построить новый экземпляр, после чего указать его вызов.

3 РЕАЛИЗАЦИЯ

В данной главе описаны детали реализации алгоритма проверки выполнения отношения Хигмана-Крускала на сигнатурах и алгоритма обобщения сигнатур снизу. Данные алгоритмы были реализованы на языке Рефал-5λ, так как компилятор данного языка является самоприменимым. Все изменения находятся в файле компилятора $OptTree - Spec.ref$, отвечающего за выполнение специализации функций. Реализация рассматриваемых алгоритмов приведена в Приложении А.

3.1 Отслеживание истории сигнатур

Напомним, что сигнатурами именуются значения статических переменных в сочетании с именами функций. Чтобы была возможность обнаружения зацикливания, необходимо запоминать для специализируемых функций аргументы их вызова в виде сигнатур. Для этого были внесены изменения в узел абстрактного синтаксического дерева, обозначаемый записью `(SpecInfo e.SpecInfo)` и используемый во время компиляции для хранения информации о специализации функций программы. Описание содержимого данного узла до и после внесения изменений представлено на листингах 4 и 5 соответственно.

```
e.SpecInfo ::= (e.SpecFuncNames) e.SpecInfo-Specific
e.SpecFuncNames ::= (e.FuncName)*
e.SpecInfo-Specific ::= t.FunctionInfo*
t.FunctionInfo ::= ((e.Name) (e.Pattern) (e.Body) s.NextCounter
t.Signature*)
t.Signature ::= ((e.InstanceName) t.StaticVarVals*)
t.StaticVarVals ::= (e.Expression)
e.InstanceName ::= e.Name
```

Листинг 4 — Описание узла `e.SpecInfo` до внесения изменений

```
e.SpecInfo ::= (e.SpecFuncNames) e.SpecInfo-Specific
e.SpecFuncNames ::= (e.FuncName)*
e.SpecInfo-Specific ::= t.FunctionInfo* (e.Histories)
t.FunctionInfo ::= ((e.Name) (e.Pattern) (e.Body) s.NextCounter
t.Signature*)
t.Signature ::= ((e.InstanceName) t.StaticVarVals*)
t.StaticVarVals ::= (e.Expression)
e.InstanceName ::= e.Name
e.Histories ::= ((e.InstanceName) e.History)*
e.History ::= ((e.FuncName) t.StaticVarVals*)*
```

Листинг 5 — Описание узла `e.SpecInfo` после внесения изменений

В качестве истории сигнатур понимается последовательность специализаций, принцип заполнения которых рассмотрим на следующем примере. В начале выполнения специализации история сигнатур отсутствует. Далее допустим, что был обнаружен вызов некоторой специализируемой функции F. Для этого вызова был построен новый экземпляр функции, который обозначим $F@1$, после чего изначальный вызов был заменён на вызов экземпляра $F@1$ с некоторым аргументом $ARG1$. Тогда сигнатура $(F@1, ARG1)$, состоящая из имени экземпляра $F@1$ и аргумента $ARG1$, запоминается компилятором в качестве истории сигнатур для экземпляра $F@1$. Допустим, далее в функции $F@1$ был обнаружен вызов некоторой другой функции G, которая также является специализируемой. Тогда после построения для неё нового экземпляра $G@1$ и добавления нового вызова $G@1$ с аргументом $ARG2$ будет запомнена история сигнатур для функции $G@1$, состоящая из сигнатуры $(F@1, ARG1)$ и новой сигнатуры $(G@1, ARG2)$. Таким образом накапливается история сигнатур для различных экземпляров функций. Предположим, что далее в функции $G@1$ был обнаружен вызов функции F. В функцию обработки этого вызова передаётся история сигнатур для функции $G@1$, состоящая в данном случае из двух сигнатур. Тогда сигнатура текущего вызова функции F будет сравнена с сигнатурой $(F@1, ARG1)$. Если для текущей сигнатуры и сигнатуры $(F@1, ARG1)$ будет выполнено отношение Хигмана-Крускала, будет применён алгоритм обобщения снизу, в результате работы которого либо вызов F будет заменен на вызов функции $F@1$, либо сначала будет

построен экземпляр для обобщения сигнатур, после чего будет добавлен вызов нового экземпляра.

Сигнатуры из истории сигнатур можно просматривать как от более старых к более новым, так и наоборот. Теорема Хигмана-Крускала гарантирует обнаружение возможного зацикливания в любом случае. В данной работе было решено сначала рассматривать более новые сигнатуры, так как скорость работы алгоритма при таком подходе оказалась выше по результатам тестирования. Кроме того, такой подход является более распространённым в суперкомпиляции.

3.2 Реализация проверки отношения Хигмана-Крускала

Алгоритм проверки выполнения отношения Хигмана-Крускала при реализации был разбит на несколько взаимодействующих функций. Основная функция принимает на вход имя специализируемой функции, текущую сигнатуру, в которой у переменных убраны имена, и историю сигнатур экземпляра, в котором был обнаружен рассматриваемый вызов, после чего по очереди сравнивает текущую сигнатуру с сигнатурами из истории, начиная с наиболее новой. Для сравнения сигнатур вызывается функция, работающая в соответствии с описанным ранее алгоритмом проверки отношения Хигмана-Крускала на основе следствий из его определения. Перед этим у переменных сигнатуры из истории убираются имена.

Для корректной работы основных функций были реализованы вспомогательные функции, позволяющие:

- преобразовать сигнатуру, убрав из неё имена переменных;
- проверить вложение одного выражения в другое;
- проверить частичное вложение одного выражения в другое и вернуть оставшуюся часть выражения в случае выполнения частичного вложения.

3.3 Реализация алгоритма обобщения снизу

Функция, выполняющая обобщение снизу, вызывается в случае, если были обнаружены сигнатуры, для которых выполняется отношение Хигмана-Крускала. Алгоритм работы данной функции можно представить следующим образом:

1. С помощью реализованного в компиляторе алгоритма глобального сложнейшего обобщения строятся обобщения выражений для каждой статической переменной рассматриваемой функции в отдельности. После этого полученные обобщения объединяются в одно выражение.

2. Производится поиск имён переменных, имеющихся в аргументе рассматриваемого вызова.

3. Переменным полученного обобщения присваиваются имена, отличные от имён, найденных в аргументе вызова. Для этого используется реализованная в компиляторе функция `NewVarName`. Этот шаг важен, так как иначе выполнение подстановок в дальнейших шагах данного алгоритма может быть выполнено неверно.

4. Подвыражения обобщённой сигнатуры сопоставляются статическим переменным рассматриваемой функции.

5. Формируется подстановка, переводящая обобщённую сигнатуру в изначальную.

6. Полученная на предыдущем шаге подстановка проверяется на тривиальность. Тривиальной называется подстановка, которая при применении к выражению оставляет его тем же с точностью до переименования переменных. Данная подстановка оказывается тривиальной, если в результате обобщения сигнатур была получена текущая сигнатура. Результат проверки на тривиальность передаётся в функцию, вызываемую на следующем шаге.

7. Вызывается функция, возвращающая вызов экземпляра, соответствующего обобщённой сигнатуре. Эта функция проверяет, был ли уже построен экземпляр функции для полученной сигнатуры. В случае положительного ответа

она формирует вызов данного экземпляра с переданными значениями. В ином случае она сначала строит новый экземпляр функции, после чего возвращает сам новый экземпляр и его вызов.

8. Полученный на предыдущем шаге вызов экземпляра преобразуется путём применения к нему подстановки, полученной на шаге 5, после чего возвращается в качестве результата работы функции.

Для удобства применения добавленных функций были описаны форматы их использования. Примеры такого описания приведены на листингах 6 и 7.

```
<CheckSignaturesPairForRelation (e.CurrentSignature) e.History-
Signature>
  == True
  == False

e.CurrentSignature, e.HistorySignature ::= t.StaticVarVals*
t.StaticVarVals ::= (t.Term*)
t.Term ::=
    (Symbol s.SymType e.SymInfo)
  | (TkVariable 't' e.Index)
  | (TkVariable 's' e.Index)
  | (TkVariable 'e' e.Index)
  | (ADT-Brackets t.Name t.Term*)
  | (Brackets t.Term*)
  | (ClosureBrackets t.Term*)
  | (CallBrackets t.Name t.Term*)
```

Листинг 6 — Описание формата использования функции, проверяющей выполнение отношения Хигмана-Крускала для двух сигнатур

```
<PartiallyContainsSignature (e.Expr) e.Signature>
  == True e.SignaturePart
  == False

e.Expr ::= t.Term*
e.Signature ::= t.Term*
t.Term ::=
    (Symbol s.SymType e.SymInfo)
  | (TkVariable 't' e.Index)
  | (TkVariable 's' e.Index)
  | (TkVariable 'e' e.Index)
  | (ADT-Brackets t.Name t.Term*)
  | (Brackets t.Term*)
  | (ClosureBrackets t.Term*)
  | (CallBrackets t.Name t.Term*)
```

Листинг 7 — Описание формата использования функции, проверяющей частичное вложение выражения в сигнатуру

4 ТЕСТИРОВАНИЕ

4.1 Корректность работы реализованных алгоритмов

Для тестирования корректности работы реализованных алгоритмов была использована встроенная в компилятор возможность составления лога состояния программы на момент различных проходов компилятора. С помощью лога можно отследить, каким образом изменялся код программы после специализации функций.

В качестве примера рассмотрим функцию $BracketsWrapper(x, y)$, приведённую в разделе 1.1. Реализация данной функции на языке Рефал-5λ была приведена в разделе 1.6. Для удобства продублируем её реализацию с указанием специализации по второму параметру на листинге 8.

```
$SPEC BracketsWrapper t.x t.Y;

BracketsWrapper {
  (t.X) t.Y = <BracketsWrapper t.X (t.Y)>;

  a t.Y = t.Y;
}
```

Листинг 8 — Реализация функции $BracketsWrapper(x, y)$ на языке Рефал-5λ с указанием специализации по второму параметру

Рассмотрим содержимое лога, составляемого при компиляции данной функции. Основная информация о состоянии программы после применения специализаций функции `BracketsWrapper` приведена на листинге 9.

```
BracketsWrapper {
  (t.X#1) t.Y#1 = <BracketsWrapper t.X#1 (t.Y#1)>;

  a t.Y#1 = t.Y#1;
}

...

BracketsWrapper {
  (t.X#1) t.Y#1 = <BracketsWrapper@1 t.X#1 t.Y#1>;
```

```
  a t.Y#1 = t.Y#1;
}

BracketsWrapper@1 {
  (t.X#1) t.Y0#1 = <BracketsWrapper t.X#1 ((t.Y0#1))>;

  a t.Y0#1 = (t.Y0#1);
}

...

BracketsWrapper {
  (t.X#1) t.Y#1 = <BracketsWrapper@1 t.X#1 t.Y#1>;

  a t.Y#1 = t.Y#1;
}

BracketsWrapper@1 {
  (t.X#1) t.Y0#1 = <BracketsWrapper@1 t.X#1 (t.Y0#1)>;

  a t.Y0#1 = (t.Y0#1);
}
```

Листинг 9 — Часть лога компиляции функции $BracketsWrapper$

Как видно из данного листинга, сначала был обнаружен рекурсивный вызов функции `BracketsWrapper`, для которого был построен новый экземпляр, обозначенный в логе `BracketsWrapper@1`. После этого в теле нового экземпляра также был обнаружен вызов функции `BracketsWrapper`, причём значение второго параметра данного вызова отличается от предыдущего. После сравнения сигнатуры нового вызова с сигнатурой предыдущего вызова было обнаружено выполнение для них отношения Хигмана-Крускала. В результате этого было построено обобщение данных сигнатур, которое оказалось равным первой сигнатуре, поэтому вызов функции `BracketsWrapper` был заменён на вызов экземпляра `BracketsWrapper@1`. Отсутствие проверки отношения Хигмана-Крускала привело бы к зацикливанию специализации данной функции.

Для избежания части возможных ошибок при дальнейшей работе с реализованными алгоритмами в репозиторий компилятора был добавлен ряд тестов, запуск которых может способствовать обнаружению ошибок в изменениях кода. Код одного из таких тестов приведён на листинге 10.

```
* TREE

$ENTRY Go {
  /* empty */
    = <Test ('b') 'za'> : 'bzb'
    = /* empty */;
}

$SPEC Test (e.ACC) e.input;

Test {
  (e.Acc) 'a' e.Rest = <Test (e.Acc 'b') e.Rest>;
  (e.Acc) s.Next e.Rest = <Test (e.Acc s.Next) e.Rest>;
  (e.Acc) /* empty */ = e.Acc;
}
```

Листинг 10 — Код тестирования функции замены символов

Действие функции, приведённой на данном листинге, заключается в замене каждого символа `'a'`, присутствующего в поданной на вход строке, на символ `'b'`. Для хранения уже обработанной части строки используется первый аргумент-аккумулятор. По данному аргументу компилятору указано выполнять специализацию. Лог компиляции данной функции позволяет убедиться в правильности применения алгоритмов проверки отношения Хигмана-Крускала и построения обобщения снизу. Часть лога, соответствующая финальной версии кода программы, представлена на листинге 11.

```
Go=1 {
  'bzb' = /* empty */;
}

$ENTRY Go {
  /* empty */ = <Go=1 <Test@1 'za'>>;
}

Test {
  (e.Acc#1) 'a' e.Rest#1 = <Test@2 (e.Acc#1) e.Rest#1>;

  (e.Acc#1) s.Next#1 e.Rest#1 = <Test@3 (e.Acc#1) s.Next#1
e.Rest#1>;

  (e.Acc#1) = e.Acc#1;
}

Test@1 {
```

```
  'a' e.Rest#1 = <Test@2 ('b') e.Rest#1>;

  s.Next#1 e.Rest#1 = <Test@4 (s.Next#1) e.Rest#1>;

  /* empty */ = 'b';
}

Test@2 {
  (e.Acc0#1) 'a' e.Rest#1 = <Test@2 (e.Acc0#1 'b') e.Rest#1>;

  (e.Acc0#1) s.Next#1 e.Rest#1 = <Test@3 (e.Acc0#1 'b') s.Next#1
e.Rest#1>;

  (e.Acc0#1) = e.Acc0#1 'b';
}

Test@3 {
  (e.Acc0#1) s.Next0#1 'a' e.Rest#1
    = <Test@3 (e.Acc0#1 s.Next0#1) 'b' e.Rest#1>;

  (e.Acc0#1) s.Next0#1 s.Next#1 e.Rest#1
    = <Test@3 (e.Acc0#1 s.Next0#1) s.Next#1 e.Rest#1>;

  (e.Acc0#1) s.Next0#1 = e.Acc0#1 s.Next0#1;
}

Test@4 {
  (e.X#0) 'a' e.Rest#1 = <Test@4 (e.X#0 'b') e.Rest#1>;

  (e.X#0) s.Next#1 e.Rest#1 = <Test@4 (e.X#0 s.Next#1) e.Rest#1>;

  (e.X#0) = 'b' e.X#0;
}
```

Листинг 11 — Код программы, приведённой в листинге 10, после завершения специализации

Как видно из данного лога, всего было создано 4 новых экземпляра тестируемой функции. Экземпляр `Test@1` соответствует вызову функции, располагающейся в функции Go. Экземпляры `Test@2` и `Test@3` соответствуют случаям, когда к аккумулятору был добавлен символ `'b'` или s-переменная соответственно. И, наконец, экземпляр `Test@4` используется, когда аккумулятор изначально был пустой, но затем к нему добавилась s-переменная. Отсутствие реализованных алгоритмов в коде компилятора привело бы зацикливанию специализации данной функции ввиду изменения содержимого аккумулятора.

Итак, тестирование показало корректность работы реализованных алгоритмов.

4.2 Изменение времени выполнения компиляции

Необходимо проверить, какой вклад во время работы компилятора вносит выполнение реализованных функций. Для этого было решено измерить скорость компиляции исходников последней версии (редакция программы `fdde67c`) сначала с помощью версии компилятора, в которой отсутствуют все внесённые в него в ходе работы изменения (редакция программы `26e90e5`), а затем с помощью последней версии. В обоих случаях компиляция исходников выполнялась 13 раз. Для тестирования использовался компьютер со следующими характеристиками:

- частота процесса — 2.2 ГГц;
- объём оперативной памяти — 4 Гб;
- операционная система — 64-битная версия Linux Ubuntu 16.04;
- компилятор языка C++ — g++;

Результаты тестирования приведены в таблице 1.

Версия компилятора	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции, III квартиль, с
Без внесённых изменений	36.59	36.44	37.17
С внесёнными изменениями	36.97	36.83	37.09

Таблица 1 — Результаты тестирования

По данным таблицы 1 видно, что доверительные интервалы замеров пересекаются. Кроме того, медиана второго замера попадает в доверительный интервал первого замера. Это значит, что имеется разница времени выполнения компиляции на уровне статистической погрешности. Медианное время компиляции

увеличилось на 0.38 секунд или на 1%. Исходя из данных результатов, можно сказать, что в типичной программе, написанной в рекомендованном стиле программирования Рефала-5λ, логика проверки зацикливания не является узким местом, а значит, нет необходимости заниматься оптимизацией алгоритма.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Установка компилятора языка Рефал-5λ

Для использования компилятора Рефала-5λ необходимо сначала загрузить исходники из репозитория проекта, при наличии утилиты `git` это можно сделать с помощью команды представленной на листинге 12.

```
git clone https://github.com/bmstu-iu9/refal-5-lambda
```

Листинг 12 — Использование утилиты `git` для загрузки исходников компилятора

После этого нужно убедиться в наличии любого компилятора C++ и собрать компилятор, выполнив в корневой директории одну из команд, приведённых на листинге 13, в зависимости от используемой операционной системы.

```
bootstrap.bat  --no-tests
./bootstrap.sh  --no-tests
```

Листинг 13 — Команды сборки компилятора из исходников

Далее необходимо добавить каталог bin в переменную среды PATH. Использование компилятора осуществляется с помощью команд `rlc` и `rlmake`.

5.2 Специализация функций и использование лога компиляции

Компилятор Рефала-5λ позволяет через специальную форму записи, начинающуюся с ключевого слова `$SPEC`, указать функции, для которых необходимо выполнить специализацию, и параметры, по которым специализация будет проводиться. После ключевого слова должно следовать имя функции, после которого располагается образец, в котором должны отсутствовать повторные переменные. Также в образце не должно встречаться более одной *e*-переменной на одном скобочном уровне. После образца необходимо указать точку с запятой.

Специализация будет выполняться по переменным образца, имя которых начинается с заглавной латинской буквы. Примеры использования данной формы записи приведены в листинге 14.

```
$SPEC Sum s.Sum (e.numbers);

$SPEC F t.x (t.Y);

$SPEC G s.x t.Y t.Z;
```

Листинг 14 — Примеры использования ключевого слова `$SPEC`

Чтобы компилятор учёл записи подобного вида, нужно при компиляции указать ключ `-OS`.

Другой полезной функцией компилятора является возможность составления лога, включающего в себя различную информацию о компилируемой программе на момент совершения различных проходов. Для создания лога необходимо перед компиляцией указать ключ `--log`, за которым после знака равенства должно следовать имя файла. Для компиляции программы с созданием лога и учётом специализации функций можно воспользоваться командой, приведённой на листинге 15.

```
rlc program.ref  -OS --log=logname.txt
```

Листинг 15 — Команда компиляции программы с составлением лога и выполнением специализаций функций

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы удалось выполнить все поставленные задачи. Были рассмотрены возможные способы реализации отношения Хигмана-Крускала. Предпочтение было отдано алгоритму, использующему следствия из определения данного отношения. Также были сравнены между собой два подхода к выполнению обобщения сигнатур: обобщение сверху и обобщение снизу. Было выявлено, что метод обобщения снизу является более приоритетным при реализации в компиляторе языка Рефал-5λ. Выбранные методы были реализованы, при этом детали реализации были подробно описаны. Тестирование реализованных методов позволило подтвердить корректность их работы.

Для выполнения поставленных задач был изучен диалект Рефал-5λ и компилятор данного языка. В компилятор был добавлен набор вспомогательных функций, которые могут быть использованы другими разработчиками при внесении изменений в рассматриваемый диалект.

Итак, использование отношения Хигмана-Крускала позволило устранить проблему рекурсивного зацикливания специализации функций в компиляторе языка Рефал-5λ. Благодаря этому открылась возможность применения оптимизации специализации функций на большем классе функций. Кроме того, внесённые изменения позволили безопасно использовать при работе компилятора такой оптимизационнный метод, как автоматическая разметка специализируемых функций программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. — М.: Вильямс, 2008, 1184 с.

2. Климов А. В., Романенко С. А. Суперкомпиляция: основные принципы и базовые понятия — Препринты ИПМ им. М. В. Келдыша, № 111, 2018, 36 с.

3. Репозиторий компилятора языка Рефал-5λ [Электронный ресурс] — Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda. Дата обращения: 24.03.2020.

4. Сухомлинова Д. П. ВКР по теме Специализация функций в Рефале-5λ [Электронный доступ] — Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/РПЗ_Сухомлинова_Специализация_функций_2019.pdf. Дата обращения: 26.03.2020.

5. Ключников И. Суперкомпиляция: идеи и методы — Практика функционального программирования, №7, 2011, 157 с.

6. Романенко С. А. Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления — Препринты ИПМ им. М. В. Келдыша, №209, 2019, 32 с.

7. Higman G. Ordering by divisibility in abstract algebras — Proceedings of the London Mathematical Society, Vol. 2(7), 1952, 326 с.

8. Kruskal J. B. Well-quasi-ordering, the tree theorem, and Vazsonyi’s conjecture — Transactions of the American Mathematical Society, Vol. 95, 1960, 210 с.

9. Руководство по программированию и справочник по языку Рефал-5 [Электронный ресурс] — Режим доступа: http://www.refal.ru/rf5_frm.htm. Дата обращения: 24.03.2020.

10. Немытых А. П. Суперкомпилятор SCP4: Общая структура — М.: Издательство ЛКИ, 2007, 152 с.

11. Савельев П. А. ВКР по теме Улучшенная оптимизация совместного сопоставления с образцом в компиляторе Рефала-5-Лямбда [Электронный доступ]

— Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/Савельев_Записка_2018.pdf. Дата обращения: 10.04.2020.

ПРИЛОЖЕНИЕ А

```
HasHigmanKruskalRelation {
  (e.Name) (e.Signature) e.History
  ((e.InstanceName) e.HistorySignature)
    = e.InstanceName
    : {
        e.Name SUF e.InstanceName-Rest
          , <RemoveSignatureVarsNames e.HistorySignature>
          : e.HistorySignatureWithoutNames
          , <CheckSignaturesPairForRelation
              (e.Signature) e.HistorySignatureWithoutNames
            >
          : {
              True = True e.HistorySignature;
              False = <HasHigmanKruskalRelation
                        (e.Name) (e.Signature) e.History
                      >;
            };

        e.OtherName
          = <HasHigmanKruskalRelation
              (e.Name) (e.Signature) e.History
            >;
      };

  (e.Name) (e.Signature) /* пустая история */ = False;
}

CheckSignaturesPairForRelation {
  (/* пусто */) /* пусто */ = True;

  (e.CurrentSignature) e.HistorySignature

    , e.CurrentSignature
    : (e.CurrentSignature-CurVar) e.CurrentSignature-Rest

    , e.HistorySignature
    : (e.HistorySignature-CurVar) e.HistorySignature-Rest

    , e.CurrentSignature-CurVar : t.Term e.OtherPart

    , <DoCheckSignaturesPairForRelation
        (t.Term) (e.OtherPart) e.HistorySignature-CurVar
      >
      : True

    = <CheckSignaturesPairForRelation
        (e.CurrentSignature-Rest) e.HistorySignature-Rest
      >;
```

```
  (e.CurrentSignature) e.HistorySignature = False;
}

DoCheckSignaturesPairForRelation {
  (e.CurSig1) (e.CurSig2) e.HistorySignature

    , <ContainsSignature (e.CurSig1) e.HistorySignature> : True

    = True;

  (e.CurSig1) (e.CurSig2) e.HistorySignature

    , <ContainsSignature (e.CurSig2) e.HistorySignature> : True

    = True;

  (e.CurSig1) (e.CurSig2) e.HistorySignature

    , <PartiallyContainsSignature (e.CurSig1) e.HistorySignature>
    : True e.HistorySignature-Rest

    , <ContainsSignature (e.CurSig2) e.HistorySignature-Rest>
    : True

    = True;

  (e.CurSig1) (e.CurSig2) e.HistorySignature = False;
}

ContainsSignature {
  (e.Expr-B e.Signature e.Expr-E) e.Signature
    = True;

  ((s.TermType e.Inner) e.RestExpr)
  (s.TermType e.OtherInner) e.OtherRestExpr
    , e.Inner : t.Inner-FirstTerm e.Inner-Rest
    , e.OtherInner : t.OtherInner-FirstTerm e.OtherInner-Rest

    , s.TermType :
      {
        Brackets
          = True e.Inner;

        ClosureBrackets
          = True e.Inner;

        ADT-Brackets
          , e.Inner
          : (e.Name) e.ADT-Brackets-Inner-Rest
          = True e.ADT-Brackets-Inner-Rest;

        CallBrackets
```

```
          , e.Inner
          : (e.SymbolName) e.Argument
          = True e.Argument;

        e.OtherTermType
          = False;
      }
    : True e.Inner^

    , e.Inner :
      {
        t.Inner-FirstTerm^ e.Inner-Rest^
          = <DoCheckSignaturesPairForRelation
              (t.Inner-FirstTerm) (e.Inner-Rest) e.OtherInner
            >;

        e.Expr
          = <DoCheckSignaturesPairForRelation
              (e.Expr) (/* пусто */) e.OtherInner
            >;
      }
    : s.InnerCheckResult

    , s.InnerCheckResult
    : True

    , e.RestExpr :
      {
        t.RestExpr-Term e.RestExpr-Rest
          = <DoCheckSignaturesPairForRelation
              (t.RestExpr-Term) (e.RestExpr-Rest) e.OtherRestExpr
            >;

        e.Expr
          = <DoCheckSignaturesPairForRelation
              (e.Expr) (/* пусто */) e.OtherRestExpr
            >;
      }
    : s.RestExprCheckResult

    , s.RestExprCheckResult
    : True

    = True;

  ((s.TermType e.Inner) e.RestExpr) e.Signature
    , e.Inner : t.Inner-FirstTerm e.Inner-Rest
    = s.TermType
    :  {
        Brackets
          = <DoCheckSignaturesPairForRelation
              (t.Inner-FirstTerm)
              (e.Inner-Rest e.RestExpr) e.Signature
```

```
            >;

        ADT-Brackets
          , e.Inner
          : (e.Name) t.ADT-Brackets-Inner-FirstTerm
            e.ADT-Brackets-Inner-Rest
          = <DoCheckSignaturesPairForRelation
              (t.ADT-Brackets-Inner-FirstTerm)
              (e.ADT-Brackets-Inner-Rest e.RestExpr) e.Signature
            >;

        CallBrackets
          , e.Inner : (e.SymbolName) e.Argument
          , e.Argument : t.Argument-FirstTerm e.Argument-RestExpr
          = <DoCheckSignaturesPairForRelation
              (t.Argument-FirstTerm)
              (e.Argument-RestExpr e.RestExpr) e.Signature
            >;

        ClosureBrackets
          = <DoCheckSignaturesPairForRelation
              (t.Inner-FirstTerm)
              (e.Inner-Rest e.RestExpr) e.Signature
            >;

        TkVariable
          = <DoCheckSignaturesPairForRelation
              (/* пусто */) (e.RestExpr) e.Signature
            >;

        Symbol
          = <DoCheckSignaturesPairForRelation
              (/* пусто */) (e.RestExpr) e.Signature
            >;

        s.OtherTermType
          = False;
      };

  (e.Expr) e.Signature = False;
}

PartiallyContainsSignature {
  (e.Expr) e.Signature
    , e.Signature : /*пусто*/
    = False;

  (e.Expr) e.Signature

    , e.Signature : e.Signature-Part t.LastTerm

    , e.Signature-Part : t.FirstTerm e.OtherPart
```

```
    , <ContainsSignature (e.Expr) e.Signature-Part>
    : True

    = True t.LastTerm;

  (e.Expr) e.Signature-Part t.LastTerm
    = <PartiallyContainsSignature
        (e.Expr) e.Signature-Part
      > t.LastTerm;
}

RemoveSignatureVarsNames {
  e.Signature
    = <Map
        {
          (e.SignaturePart)
            = <RemoveSignatureVarsNames-Expr e.SignaturePart>
            : e.SignaturePart^
            = (e.SignaturePart);
        }
        e.Signature
      >
    : e.Signature^
    = e.Signature;
}

RemoveSignatureVarsNames-Expr {
  e.Expr = <Map &RemoveSignatureVarsNames-Term e.Expr>;
}

RemoveSignatureVarsNames-Term {
  (TkVariable s.Mode e.Index)
    = (TkVariable s.Mode);

  (s.BracketsTag e.Expr)
    , <OneOf s.BracketsTag CallBrackets ClosureBrackets Brackets>
    : True
    = <RemoveSignatureVarsNames-Expr e.Expr> : e.Expr^
    = (s.BracketsTag e.Expr);

  (ADT-Brackets (e.Name) e.Expr)
    = <RemoveSignatureVarsNames-Expr e.Expr> : e.Expr^
    = (ADT-Brackets (e.Name) e.Expr);

  t.OtherTerm = t.OtherTerm;
}

MakeGeneralization {
  (e.Signature) (e.HistorySignature) (e.Argument)
  (e.SpecPattern) (e.StaticMatches) (e.Name)
  (e.Body) s.NextNumber (e.Signatures) (e.DynamicMatches)
  (e.History)
```

```
    /* Получаем обобщение двух сигнатур */
    = <GetSignaturesGeneralization
        (e.Signature) e.HistorySignature
      >
    : e.GenSignature

    = <ExtractVariables-Expr e.Argument> : e.ArgumentVars

    = <NameSignatureVars (e.ArgumentVars) e.GenSignature>
    : e.GenStaticMatches

    = <ExtractVariables-Expr e.SpecPattern> : e.SpecPatternVars

    = <GetSignaturesGeneralizationMatches
        (e.SpecPatternVars) e.GenStaticMatches
      >
    : e.GenStaticMatches^

    = <GetRealSubst (e.StaticMatches) (e.GenStaticMatches)>
    : e.RealSubst

    = <IsTrivialSubstitutions e.RealSubst>
    : {
        True
          = False;

        False
          = True;
      }
    : e.NeedRelationCheck

    = <SpecCall-Matches
        (e.Name) (e.SpecPattern)
        (e.Body) s.NextNumber e.Signatures
        (e.Argument) (e.GenStaticMatches)
        (e.DynamicMatches) (e.History) (e.NeedRelationCheck)
      >
    : t.Info
      t.NewCall
      e.NewFunc
      (e.NewHistory)

    = <ApplySubst-Expr (e.RealSubst) t.NewCall> : t.NewCall^

    = t.Info
      t.NewCall
      e.NewFunc
      (e.NewHistory);
}

NameSignatureVars {
  (e.KnownVars) e.Signature
    = <MapAccum
```

```
        {
          (e.KnownVars^) (e.SignaturePart)
            = <NameSignatureVars-Expr
                (e.KnownVars) e.SignaturePart
              >
            : (e.KnownVars^) e.SignaturePart^
            = (e.KnownVars) (e.SignaturePart);
        }
        (e.KnownVars)
        e.Signature
      >
    : (e.KnownVars^) e.Signature^
    = e.Signature;
}

NameSignatureVars-Expr {
  t.State e.Expr
    = <MapAccum &NameSignatureVars-Term t.State e.Expr>;
}

NameSignatureVars-Term {
  (e.KnownVars) (TkVariable s.Mode)
    = <NewVarName (e.KnownVars) s.Mode 'X' 0>
    : (e.KnownVars^) e.Index
    = (e.KnownVars) (TkVariable e.Index);

  t.State (s.BracketsTag e.Expr)
    , <OneOf s.BracketsTag CallBrackets ClosureBrackets Brackets>
    : True
    = <NameSignatureVars-Expr t.State e.Expr> : t.State^ e.Expr^
    = t.State (s.BracketsTag e.Expr);

  t.State (ADT-Brackets (e.Name) e.Expr)
    = <NameSignatureVars-Expr t.State e.Expr> : t.State^ e.Expr^
    = t.State (ADT-Brackets (e.Name) e.Expr);

  t.State t.OtherTerm = t.State t.OtherTerm;
}

GetSignaturesGeneralization {
  (t.Sig1-StaticVarPart e.Sig1-Rest)
  t.Sig2-StaticVarPart e.Sig2-Rest
    = (<GlobalGen t.Sig1-StaticVarPart t.Sig2-StaticVarPart>)
      <GetSignaturesGeneralization (e.Sig1-Rest) e.Sig2-Rest>;

  (/* пусто */) /* пусто */
    = /* пусто */;
}

GetSignaturesGeneralizationMatches {
  (t.Var e.Vars) (e.GenSignature-First) e.GenSignature-E
    = <IsSpecStaticVar t.Var>
    : {
```

```
        True
          , t.Var : (e.VarInner)
          = (e.GenSignature-First ':' (TkVariable e.VarInner))
            <GetSignaturesGeneralizationMatches
              (e.Vars) e.GenSignature-E
            >;

        False
          = <GetSignaturesGeneralizationMatches
              (e.Vars) (e.GenSignature-First) e.GenSignature-E
            >;
      };

  e.OtherExpr
    = /* пусто */;
}

GetRealSubst {
  (e.StaticMatches-B
    (e.Val1 ':' (TkVariable e.Var))
  e.StaticMatches-E)
  (e.GenStaticMatches-B
    (e.Val2 ':' (TkVariable e.Var))
  e.StaticMatches-E)
    = <GenericMatch (e.Val1) (e.Val2)>
    : {
        Clear e.Expr
          = e.Expr
            <GetRealSubst
              (e.StaticMatches-B e.StaticMatches-E)
              (e.GenStaticMatches-B e.StaticMatches-E)
            >;

        e.Other
          = <GetRealSubst
              (e.StaticMatches-B e.StaticMatches-E)
              (e.GenStaticMatches-B e.StaticMatches-E)
            >;
      };

  e.OtherExpr
    = /* пусто */;
}
```

Листинг А.1 — Основные реализованные функции