



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:  
«Специализация функций в Рефале-5λ»**

Студент ИУ9-82	_____	Д. П. Сухомлинова
	(Подпись, дата)	
Руководитель дипломной работы	_____	А. В. Коновалов
	(Подпись, дата)	
Консультант	_____	_____
	(Подпись, дата)	(ФИО)
Консультант	_____	_____
	(Подпись, дата)	(ФИО)
Заведующий кафедрой	_____	И. П. Иванов
	(Подпись, дата)	

2019 г.

## АННОТАЦИЯ

Объем данной дипломной работы составляет 47 страниц. Для ее написания было использовано 12 источников. В работе содержатся 13 листингов и 2 таблицы.

В дипломную работу входят шесть глав. В первой главе описывается постановка задачи, во второй главе содержится описание теоретической базы по специализации, три главы посвящены разработке, реализации и тестированию программы и в последней главе собрано руководство пользователя.

Объектом исследований данной ВКР являются алгоритмы специализации функций.

Цель работы — расширение оптимизационных возможностей компилятора языка Рефала-5 $\lambda$  путём реализации в нём алгоритма специализации функций. Поставленная цель достигается за счет разработки формата объявления специализации и реализации алгоритма специализации в компиляторе Рефала-5 $\lambda$ .

В работе показано, что специализация как отдельно взятая оптимизация на примере раскрутки компилятора не дала особых преимуществ, но открыла больше возможностей для проведения других оптимизаций.

# СОДЕРЖАНИЕ

АННОТАЦИЯ .....	2
ВВЕДЕНИЕ .....	5
1 ПОСТАНОВКА ЗАДАЧИ .....	7
2 СПЕЦИАЛИЗАЦИЯ ПРОГРАММ И ФУНКЦИЙ .....	8
2.1.1 Offline специализация .....	9
2.1.2 Online специализация .....	10
2.1.3 Резюме .....	11
3 АНАЛИЗ ПОСТАНОВКИ ЗАДАЧИ .....	12
3.1 Пример специализации. Специализация шаблонов в языке C++ .....	12
3.1.1 Явная специализация шаблонов .....	12
3.1.2 Специализация шаблонов времени компиляции .....	13
3.2 Анализ возможностей специализации в языке Рефал-5λ .....	14
3.2.1 Сравнение со специализацией шаблонов в C++ .....	14
3.2.2 Подробный анализ .....	15
3.3 Необходимые ограничения и обоснования ограничений .....	16
3.3.1 Количество аргументов .....	17
3.3.2 Статические и динамические параметры образца специализации .....	17
3.3.3 Соответствие шаблона специализации и объявления специализируемой функции .....	18
3.3.4 Прочие замечания по формату шаблона .....	19
3.4 Синтаксис и семантика объявления специализации .....	19
3.5 О прогонке и специализации .....	20

4	АЛГОРИТМ .....	24
4.1	Соответствие шаблона специализации, образцовых выражений и фактических аргументов вызова специализируемой функции .....	24
4.2	Подготовительный этап специализации .....	26
4.3	Общая схема специализации .....	26
4.3.1	Модельный язык .....	27
4.3.2	Специализация функций на объявленном модельном языке .....	28
5	ТЕСТИРОВАНИЕ .....	33
5.1	Специализация функции Replace .....	33
5.2	Проведение эксперимента на самоприменимом компиляторе .....	38
6	РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ .....	40
6.1	Установка и раскрутка самоприменимого компилятор Рефала-5λ .....	40
6.2	Объявление специализации функции .....	41
6.3	Несоответствие формата объявления специализации .....	42
6.4	Компиляция программы с оптимизацией специализации и логирование компиляции .....	43
	ЗАКЛЮЧЕНИЕ .....	44
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	<b>ОШИБКА!</b>
	<b>ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА.</b>	
	ПРИЛОЖЕНИЕ А .....	47

## ВВЕДЕНИЕ

Ключевым преимуществом Рефала-5λ по сравнению с Рефалом-5 является введение функций высших порядков, которые диалект перенял у своего прародителя Рефала-7. Наиболее часто используемые из них – функции Map, MapAccum, Reduce из стандартной библиотеки. Они предоставляют простой интерфейс для обработки последовательности элементов, по сути, заменяя циклы по длине массива. На них основана почти вся обработка программы в самоприменимом компиляторе. Эти функции принимают в качестве параметра последовательность термов, функцию обработки отдельного терма и, возможно, аккумулятор; в процессе работы они для каждого терма из последовательности вызывают функцию обработки и возвращают новую последовательность из результатов вызова и, возможно, преобразованный аккумулятор. При этом вызов функции-параметр происходит при помощи другой функции Apply, приспособленной для вызова замыканий, что приводит к дополнительным накладным расходам на каждой итерации.

И многое множество других функций высших порядков, которые безусловно упрощают жизнь разработчикам, но, как любая абстракция, требуют больших затрат на исполнение готовой программы.

При этом, в подавляющем большинстве на момент компиляции функция, передающаяся параметром, известна, а значит, функции высшего порядка можно оптимизировать, по крайней мере на уровне специализации по аргументу-функции.

Рассмотрим оптимизацию специализации программы. В общем случае, специализация программ — это построение специализированной программы по универсальной со множеством параметров, если значения части аргументов известны и фиксированы.

Исследования в области построения методов специализации программ по отношению к фиксированным свойствам их аргументов были начаты в 1970-х гг.

японским учёным Ё. Футамурой («generalized partial computation») [5], российскими учёными А. П. Ершовым («смешанные вычисления») [4], В. Ф. Турчиным («суперкомпиляция») [3][9][10]. Перехватили эстафету в 80-е гг. группы учёных под руководством Н. Д. Джоунса [6][7] и С. А. Романенко [2][12]. В настоящее время исследования и эксперименты в этой области продолжаются [11] и увлекают всё большее количество умов.

В рамках данной работы предлагается реализовать частный случай оптимизации специализации функций. Мы будем рассматривать специализацию отдельно взятого определения функции для случаев её вызова в отдельных точках программы. Считается, что отдельная функция на Рефале имеет несколько аргументов, часть из которых известны и фиксированы.

За счет переноса вычислений аргументов с процесса исполнения программы на стадию компиляции ожидается увеличение скорости работы программы при увеличении времени компиляции. Отметим, что проспециализированный компилятор будет быстрее компилировать программу без её специализации, а первая раскрутка самоприменимого компилятора со специализацией выполняться медленнее, чем без неё. Но будет ли раскрутка самоприменимого проспециализированного компилятора со специализацией выполняться быстрее, чем обычная раскрутка без специализации? По этому направлению будет проведен соответствующий практический эксперимент.

## 1 Постановка задачи

Конечной целью данной работы является расширение возможностей оптимизаций компилятора Рефала-5λ добавлением специализаций функций и оценка выгоды оптимизации специализации для ускорения работы наиболее употребительных библиотечных функций языка на примере самоприменимого компилятора языка.

Для этого необходимо:

- определить правила объявления специализации функции по шаблону и ключи компиляции для включения данной оптимизации;
- реализовать алгоритм специализации;
- оценить влияние оптимизации специализации функций Map, MapAccum и Reduce из стандартной библиотеки в самоприменимом компиляторе Рефала-5λ.

## 2 Специализация программ и функций

Специализация программ — это порождение по универсальной программе с множеством параметров специализированной программы, когда значения части параметров известны и фиксированы. При специализации известная информация распространяется по тексту программы и используется для её оптимизации и поднятия эффективности во много раз. [1] [2]

То есть, если программа  $f$  принимает два аргумента  $x, y$  и можно предположить, что первый аргумент  $x$  принимает значение  $A$ , то программа может быть специализирована по этому аргументу (1).

$$f(x, y) \rightarrow f_A(y) = f(A, y) \quad (1)$$

где  $f$  – исходная программа;

$x, y$  – аргументы исходной программы;

$A$  – предполагаемое фактическое значение аргумента  $x$  программы  $f$ ;

$f_A$  – специализированная программа, результат специализации программы  $f$  по первому аргументу  $x$ , равному  $A$ .

Под специализатором будем понимать метапрограмму  $SPEC$ , принимающую на вход исходную программу  $f$  и значение первого аргумента  $A$  и порождающую соответствующую специализированную программу (2).

$$SPEC(f, A) = f_A \quad (2)$$

Тогда использование специализатора можно задать формулой (3).

$$f(x, y) = SPEC(f, A)(y) \quad (3)$$

Заметим, что в общем смысле алгоритмы специализации могут подразумевать работу даже для аргументов в общей позиции, для раскрытия неоптимальностей внутри самой программы, которые могут быть устранимы и вычислимы на стадии компиляции. Подходы к специализации: *offline* и *online* специализация

В университете Копенгагена (Н. Д. Джонс, П. Сестофт, Х. Сондергаад) в 1983 году удалось решить аппроксимирующие задачи самоприменения Копенгагенского нетривиального специализатора. [6][7][8] Анализируя остаточную программу, копенгагенская группа предложила понятия «*online*» и «*offline*» методов специализации [6].

### 2.1.1 Offline специализация

*Offline специализация* разделяет в отдельные стадии-проходы анализ исходной программы и метаинтерпретацию локальных шагов этой программы, которые могут быть выполнены без знания конкретных значений неизвестной части аргументов этих шагов.

Для специализации аргументы программы размечаются на *статические* и *динамические*. Полагается, что статические аргументы будут известны на второй стадии преобразований.

В первую стадию, названную *анализом времени связыванием* (*Binding time analysis, BTA*), размечаются аргументы каждого шага и анализируется поток «статической» информации по программе. Под «статической» информацией понимается те данные и переменные, которые включаются в *статические* аргументы программы.

На выходе у ВТА размеченная программа, в которой каждое элементарное действие помечено как статическое, если его можно однозначно проинтерпретировать без знания конкретных значений динамической части входов этих действий-шагов.

Задача, поставленная перед ВТА как таковая, очевидно, алгоритмически неразрешима. Поэтому далее при отсутствии явных указаний на обратное под ВТА будем понимать некоторую аппроксимацию сформулированной выше ВТА-задачи.

На вход второй стадии, *метаинтерпретацией*, которая собственно и называется при этом подходе «специализацией», подаётся результат ВТА и значения её статических аргументов. «Специализация» (вторая стадия) логически проста и алгоритмизируема: все содержательные проблемы перенесены в ВТА.

Группа Н. Д. Джонса, как и С. А. Романенко, решила не оригинальную классическую задачу самоприменения, а задачу, которая существенно проще классической: в обоих исследованиях специализаторы производят лишь вторую стадию преобразований, не занимаясь связыванием по времени. Позже эксперименты *offline* самоприменения Джонса-Романенко повторялись и уточнялись в разных направлениях рядом авторов.

Здесь существенно, что входные данные (как статические, так и динамические) каждого шага программы просматриваются только один раз на этапе «специализации».

### 2.1.2 Online специализация

Online специализация производит метавычисления шагов преобразуемой программы в процессе анализа свойств этой программы. В общем случае, алгоритм не ограничен ни в ресурсах, ни в количестве проходов по исходной программе или по частям этой программы. При этом каждый проход даёт цикл, ко-

торый в общем случае алгоритмически нераспознаваем, даже если он только производит анализ без преобразований. И в общем случае этот цикл будет присутствовать в остаточной программе, ухудшая её временную сложность. При попытке решения задач самоприменения эти циклы существенно осложняют его логику специализатора по сравнению с offline подходом.

### 2.1.3 Резюме

Разработка методов online специализации сложнее разработки методов offline специализации.

По определению, online специализация менее ограничена в методах, чем offline специализация в используемых методах, и, потому, потенциально значительно более сильная.

При offline специализации происходят преобразования только исходной программы; а при online специализации могут происходить также преобразования подпрограмм, построенных самим специализатором, а не только подпрограмм исходной программы.

При преобразованиях online специализации в результирующей программе на каждые проходы специализатора возможно появление алгоритмически нераспознаваемого цикла.

## 3 Анализ постановки задачи

### 3.1 Пример специализации. Специализация шаблонов в языке C++

Прежде, чем углубляться дальше в теорию, рассмотрим пример специализации – специализацию шаблонов в языке C++. Хотя бы постольку, поскольку среда выполнения самоприменимого компилятора Рефала-5λ – библиотека, написанная на C++.

Шаблоны в языке C++ – это мощный инструмент для порождение полиморфного на уровне обрабатываемых типов кода. При этом возможно определение шаблонов классов и шаблонов функций. Сопоставлением типов – фактических и необходимых по определению шаблона – и порождением специализированных по типу шаблонов занимается сам компилятор языка. Вот о последнем стоит упомянуть в данной работе поподробнее.

Специализация — это повторное определение шаблона с конкретным типом либо классом типов. Для каждого использования компилятор сам подбирает наиболее подходящий результат, если разработчик явно описал специализированную версию, или порождает сам в процессе компиляции.

#### 3.1.1 Явная специализация шаблонов

Не особенно интересный для нас вариант. Здесь разработчик сам объявляет специализированную версию шаблона и вручную переопределяет методы и поля для частного случая. Такая специализация понятна разработчику, пишущему код на C++, и проста в реализации. Для большинства задач этого вполне достаточно. Но наша задача автоматизировать этот процесс. Так что перейдём к неявной специализации, происходящей внутри компилятора.

### 3.1.2 Специализация шаблонов времени компиляции

Как уже говорилось, шаблоны в C++ являются средствами метапрограммирования и реализуют полиморфизм времени компиляции: полиморфное поведение, которое разработчик закладывает в код в процессе его написания, разрешается на уровне компилятора, а полученный в результате бинарный код будет иметь постоянное поведение.

Давайте объявим шаблон функции `for_each` и рассмотрим на нём специализацию компилятором языка.

```
template <typename T, typename F>
void for_each(T items[], size_t n, F f) {
    for (size_t i = 0; i < n; ++i) { f(items[i]); }
}

void print(int i) {
    printf("%d\n", i);
}

struct Print {
    void operator() (int i) {
        printf("%i\n", i);
    }
};

int main() {
    int a[] = {1, 2, 3, 4};
    for_each(a, 4, print);
    for_each(a, 4, Print());
    for_each(a, 4, [](int i) { printf("%d\n", i); });
    return 0;
}
```

*Листинг 1 Определение шаблона на языке C++*

Поскольку везде используем один и тот же массив целых чисел типа `int`, то для всех специализированных версий тип `T` будет заменён `int`.

Для первого вызова, куда передаётся глобальная функция `print`, тип `F` компилятор заменит на тип указателя на функцию. Для второго вызова, где мы передаем объект класса `Print` с перегруженными скобками, значение аргумента `f` будет передано как статический параметр. Для третьего вызова, безымянная функция будет так же передана, по указателю.

## 3.2 Анализ возможностей специализации в языке Рефал-5λ

### 3.2.1 Сравнение со специализацией шаблонов в C++

Программист пишет функцию вроде `Map` или пользуется уже готовой функцией `Map`. Выполнение функции занимает достаточно большое количество времени, при этом часть аргумента функции известно на стадии компиляции, то есть для любого вызова можно написать эквивалентную специализированную функцию.

Явная специализация приводит к отказу от библиотечных переиспользуемых функций и написанию большого количества кода вручную. Неудобно. Хочется, чтобы был инструмент, который позволяет автоматически строить специализированные варианты для каждого вызова. В общем, нам хочется того же, что есть в `Ci++` при инстанцировании шаблонов функций.

Наши варианты:

- сделать функции вроде `Map` встроенными в язык и обрабатывать их компилятором особым образом. Выглядит как костыль, не расширяемо;
- использовать какую-нибудь суперкомпиляцию или дефорестацию. Слишком сложно;
- сделать специализацию функций — механизм, позволяющий строить специализированные функции.

Как именно? Все функции пытаться специализировать? Накладно и не нужно. Значит нужна пометка тех функций, которые нужно специализировать. Далее, если мы будем специализировать каждый вызов по его аргументу, то это в сочетании с встраиванием/прогонкой может приводить к закливаниям. Поэтому приходим к сигнатуре со специализируемыми и не специализируемыми параметрами.

Иначе говоря, делаем проще. Мы хотим в функции заменять некоторые фиксированные значения на константы, характерные для каждого вызова. Поэтому мы вводим местность функции, как в Рефале Плюс: вводим формат аргумента. Чтобы можно было сделать замену статически известных значений в теле функции, специализируемые параметры во всех предложениях должны быть простыми переменными. После чего мы для конкретного вызова мы определяем статические параметры, соответствующие им переменные в каждом предложении и подставляем.

### 3.2.2 Подробный анализ

Для Рефала-5λ как языка с функциональной парадигмой допустимо частное определение специализации, где вместо программы рассматривается определение функции для случаев её вызова в отдельных точках программы (читай, в определениях других функций, в том числе в функции Go, входной точке входа программы). При этом, поскольку определение функции может содержать набор предложений со своими образцовыми выражениями, то шаблон специализации необходимо проверять на соответствие каждому образцовому выражению в теле определения функции. И если выявлено несоответствие хотя бы в одном предложении, то объявление специализации следует считать ошибочным.

Что касается способа реализации. Сравнивая подходы online и offline специализации, мы подчеркнули простоту реализации offline подхода против гибко-

сти online специализации. С учётом того, что это лишь первая проба специализации в компиляторе Рефала-5λ, проект жестко ограничен по времени, а ожидаемая практическая выгода в виде ускорения раскрутки компилятора в данный момент в большем приоритете, чем научное исследование, был выбран offline подход с некоторыми модификациями для большего упрощения.

Процесс специализации разбит на три основные стадии:

- подготовительный этап, в процессе которого за некоторое константное количество проходов собирается информация о специализируемых функциях;
- основной этап специализации, в процессе которого выполняется проход за проходом по программе, находятся вызовы специализируемых функций, анализируются фактические значения параметров и по возможности находится существующая или порождается новая специализированная функция; проходы по программе продолжаются до неподвижной точки или пока не будет достигнуто ограничение по количеству циклов оптимизаций дерева;
- завершающий этап, в котором очищаются используемые структуры данных и возвращается преобразованное синтаксическое дерево.

### 3.3 Необходимые ограничения и обоснования ограничений

Специализация предполагает, что в общем случае – программа, а в нашем случае – функция, имеет несколько аргументов, часть из которых известны и фиксированы. Для Рефала-5λ это совсем не так и по количеству аргументов, и по тому, что считать известным, в связи с самим определением языка и тем результатом, который мы ожидаем по факту специализации.

О том, как стоит решать эти проблемы будет рассказано дальше.

### 3.3.1 Количество аргументов

Функции на Рефале-5λ принимают ровно один аргумент. Выполнять специализация по одному аргументу не имеет смысла: она тривиальна и не даёт никаких выгод специализации как оптимизации. В такой постановке стоит решать задачу прогонки, замены вызова на результат в процессе компиляции.

Первое очевидно решение, перейти на другой диалект Рефала, в функции принимают несколько аргументов. Во-первых, это слишком просто. Во-вторых, основная задача, которая перед нами стоит, именно оптимизация самоприменимого компилятора Рефала-5λ, а не академическое изучение специализации. Следовательно, вводим ограничение на применение специализации: для специализируемых функций должен быть явно определен формат аргумента. Будем называть его *шаблоном специализации*.

### 3.3.2 Статические и динамические параметры образца специализации

Нужно уметь отличать параметры функции, по которым имеет смысл делать оптимизацию, и параметры, по которым оптимизацию не имеет смысл делать. Для практического применения это может быть избыточно и опасно с точки зрения заикливания алгоритма оптимизации.

От обратного, что если делать специализацию по всем параметрам, указанным в шаблоне специализации.

Избыточность. Рассмотрим специализацию функции Map. Мы упоминали её выше, как одну из мотиваций – её специализации по передаваемому замыканию. Кроме замыкания она принимает список термом. Выполняя специализацию по этим двум аргументам, мы получим линейное, зависящее от длины списка термом, разрастание кода и, соответственно, увеличение времени на один проход оптимизатора. И это только для одного вызова. Если производить такие преоб-

разования на компиляторе, где вызовом функции Map десятки и за каждый обрабатывает практически весь код компилятора (как потока лексем на первых стадиях или термов в дереве на последующих), то требуемую после оптимизаций память и общее время компиляции со ста по умолчанию итерациями оптимизатора сложно оценить. Как и количество итераций оптимизатора, требуемых для достижения неподвижной точки.

Зацикливание. Рассмотрим следующий пример специализации функции Add1 сложение чисел в унарной системе счисления.

```
Add1 {  
  (e.X 1) (e.Y) = <Add1 (e.X) (1 e.Y)>;  
  (/ * ноль */) (e.Y) = e.Y;  
}
```

*Листинг 2 Определение функции Add1 сложения чисел в унарной системе счисления*

Если специализировать функцию Add1 по обоим параметрам то, за счёт изменений значения второго параметра при рекурсивном вызове специализатор зациклится, если не будет точно определён первый параметр (на его месте может быть переменная) или установлено ограничение по количеству проходов.

### 3.3.3 Соответствие шаблона специализации и объявления специализируемой функции

Дополнительно вводим ограничение: в подстановках, переводящих шаблон специализации в образцовое выражение предложения функции, статические параметры должны заменяться на переменные соответствующего им типа; динамические параметры в подстановках могут заменяться на всё, что угодно (в соответствии с их типом, по определению уточнения).

Отметим также, что в списке переменных, соответствующих статическим параметрам не должно быть повторных переменных. Это тоже упрощение, от

которого можно будет отказаться. Так, если в фактическое значение этих параметров различается, то потребуется дополнительный анализ: поскольку, предложение, очевидно, будет отброшено на стадии выполнения, его можно также отбрасывать в процессе специализации.

### 3.3.4 Прочие замечания по формату шаблона

Дополнительно вводим временное необходимое ограничение, что в шаблоне специализации не должно быть повторных переменных.

Данное ограничение необходимо исключительно для упрощения реализуемых алгоритмов, в частности обобщённого сопоставления с образцом и при формировании определения специализированной функции, в дальнейшем это ограничение можно будет убрать.

## 3.4 Синтаксис и семантика объявления специализации

Зафиксируем все вышесказанные замечания и опишем синтаксис и семантику определения специализации.

Для специализации функции необходимо явное определение формата аргумента функции. Оно задаётся в выражении, которое далее будем называть *объявлением специализации*.

Определение специализации состоит из кодового слова `$SPEC`, имени функции и собственно шаблона специализации, описывающего формат аргумента специализируемого вызова. Формат задаётся жёстким выражением без повторных переменных. Имена переменных не играют значения, кроме первого символа их индекса, который указывает на то, статический это параметр (заглавная латинская буква) или динамический (строчная латинская буква, цифра, дефис или нижнее подчёркивание). Подразумевается, что все вызовы, которые будут удовлетворять объявленному формату и в которых статическим параметрам

будут отвечать нетривиальные выражения (то есть не переменная того же типа) будут проспециализированны по этому шаблону.

Для выполнения дальнейших проверок необходимо, чтобы в той же единице трансляции было доступно само определение функции, то есть функция должна быть либо определена в том же файле, что и объявление её специализации, либо включена через `$INCLUDE`.

Чтобы результат специализации был однозначно определён, потребуем, чтобы на каждую функцию было не более одного объявления специализации.

Все образцовые выражения специализируемой функции должны быть уточнениями указанного шаблона специализации. В подстановках, переводящих шаблон специализации в образцовое выражение предложения функции, статические параметры должны заменяться на переменные соответствующего им типа, различным статическим параметрам удовлетворяют различные же переменные; динамические параметры в подстановках могут заменяться на всё, что угодно (в соответствии с их типом, по определению уточнения).

### 3.5 О прогонке и специализации

В случае, когда путь выполнения программы зависит от динамических параметров, встраивания в сочетании со специализацией недостаточно для полной оптимизации — необходима прогонка.

*Прогонкой* вызова функции в правой части называется попытка выполнения шага рефал-машины на стадии компиляции. При прогонке берётся вызов функции в правой части предложения и заменяется на стадии компиляции на результат своей работы. При этом возможно расщепление предложения на несколько, имеющих более точные образцы.

Интересный пример на взаимодействие этих оптимизаций – их применение на интерпретатор языка Forth. В результате его специализации с прогонкой ожидается получение первой проекции Футамуры [5], то есть компилятор языка Forth в язык Рефал-5λ. Но это уже за пределами данной работы.

Пример более близкий – оптимизация функции Map. Внутри неё происходит вызов функции Apply, которую так раз напрашивается на встраивание с прогонкой (см. Листинг 3).

```

/**
  <Map t.Closure t.Item*> == e.ItemRes*

  <Apply t.Closure t.Item> == e.ItemRes
*/
Map {
  t.Fn t.Next e.Tail = <Apply t.Fn t.Next> <Map t.Fn e.Tail>;

  t.Fn = ;
}

$SPEC Map t.FUNC e.items;

/**
  <Apply t.Closure e.Arg> == e.Res

  t.Closure ::=
    s.FUNCTION
    | (t.Closure e.Bounded)
  e.Arg, e.Res, e.Bounded ::= e.AnyExpr
*/
Apply {
  s.Fn e.Argument = <s.Fn e.Argument>;

  (t.Closure e.Bounded) e.Argument
    = <Apply t.Closure e.Bounded e.Argument>;
}

$INLINE Apply;

```

*Листинг 3 Определение библиотечных функций Map и Apply*

К тому же, прогонка нормально работает только для ограниченного Рефала — Рефала, в образцах которого недопустимы открытые e-переменные и повторные t- и e-переменные. Но нам хотелось бы оптимизировать функции вида

Replace (см. Листинг 10). Специализация по всему аргументу потребует сопоставления аргумента функции с образцом, где есть и открытые, и повторные переменные. Но если мы будем накладывать маску как в шаблоне специализации, то никаких проблем не будет.

## 4 Алгоритм

Встраивание специализации потребовало изменений в следующих проходах компилятора:

- лексический анализ;
- синтаксический анализ;
- проверка контекстных зависимостей;
- редуктор до подмножества, или обессахариватель;
- высокоуровневая оптимизация.

В данном разделе точнее опишем технические тонкости основных моментов реализации:

- сопоставление образцов и выражений;
- проверка корректности объявления;
- подготовительный этап оптимизации;
- непосредственно специализация;
- завершающий этап оптимизации.

О проверке корректности объявления уже и так много сказано выше: какие ограничения введены и почему, что нужно проверять обсудили в более, чем полной, реализации этих проверок – чистая техника. В завершающем этапе ещё меньше интересного, даже описывать нечего. Поэтому опустим их и уделим больше внимания на саму специализацию.

### 4.1 Соответствие шаблона специализации, образцовых выражений и фактических аргументов вызова специализируемой функции

При разработке специализатора будем иметь ввиду, что в компиляторе определена функция обобщенного сопоставления произвольного выражения жёсткому. Опишем, что мы от неё ждём.

Пусть в образцовом выражении определения специализируемой функции мы имеем формальный параметр  $v.X : v \in \{s, t, e\}$ , который в некотором вызове описывается выражением  $Act$ . Будем обозначать это так:

$$Act \rightarrow v.X$$

В шаблоне специализации этот параметр  $v.X$  описывается жёстким выражением  $He$ :

$$v.X \rightarrow He$$

Промежуточная задача специализатора – выполнить обобщённое сопоставление  $Act : He$ . Пусть для этого будет определена вспомогательная функция, принимающая два образцовых выражения, одно из которых должно быть L-выражением с неповторными t-переменными, и возвращает один из следующих результатов:

- Успешно безусловно. Это означает, что существует единственная подстановка, позволяющая получить исходный образец. В результате перечисляются присваивания переменных из жёсткого образца.
- Неуспешно. Значит, невозможно подобрать значения переменных жёсткого выражения таким образом, чтобы получиться исходное образцовое выражение, образцы считаются несопоставимы, данное предложение неприменимо.
- Неопределённо или условно успешно (с ограничениями на переменные в  $Act$ ). Значит, что данных в фактическом параметре недостаточно для специализации, сопоставление можно выполнить только в процессе в процессе исполнения кода.

Для своей работы специализатор будет полагать в качестве успешного сопоставления с шаблоном специализации только успешное безусловное сопоставление фактического результата или образцового выражения с шаблоном специализации.

#### 4.2 Подготовительный этап специализации

Из-за того, что объявления специализации в общем случае могут располагаться свободно относительно определения функции, этап выполняется в два прохода. Сначала собираем все объявления специализаций. Здесь нам нужно имя функции и шаблон специализации. Формируем из этого список, объявления вырезаем из синтаксического дерева. Потом дополняем каждое объявление телом специализируемой функции, инициализированным счётчиком специализированных функций, пустым списком сигнатур и тел для порождённых версий.

Отметим, что этот этап, как и завершающий, выполняется независимо от того, был ли установлен флаг специализации, по крайней мере для того, чтобы убрать из дерева объявления специализаций.

#### 4.3 Общая схема специализации

Вот и мы добрались до самого интересного и содержательного в этой работе. Для объяснений некоторых элементов нам могут понадобиться наглядные примеры.

Давайте объявим простой модельный язык, на котором алгоритм будет особенно просто для восприятия. На нём мы пошагово рассмотрим алгоритм специализации.

Дальше приведём и опишем листинги исходных данных и результатов специализации в компиляторе Рефала-5λ. Для получения результатов, использовалось логирование между проходами компилятора (см. раздел 6.4).

### 4.3.1 Модельный язык

В нашем языке в качестве атомарных элементов будем иметь переменные, функции и конструкторы. В качестве имён можно использовать последовательность латинских букв и цифр, тире и нижнее подчёркивание, первый символ строго латинская буква.

Программа будет состоят из набора определений функций.

Определение функции содержит её имя, перечисление формальных аргументов, при этом в начале списка аргументов стоят аргументы, по которым допустима специализация. Это статические параметры в нашей терминологии. Они отделяются от динамических точкой с запятой. Между собой аргументы разделены запятой. Представленный таким образом список формальных аргументов образует в нашем понимании шаблон специализации – определён формат аргумента функции и задана разметка на статический и динамические параметры. Тело функции состоит из списка выражений, которые могут содержать условие, вызов функции, значение (константу или переменную), объявление переменной в формате `let`.

Все вызовы функций должны быть согласованы с их определениями — количество статических и динамических параметров в вызовах и определениях должно совпадать.

Конструкторы имеют фиксированную местность, конструкторы с местностью ноль суть константы.

Для наглядности – Листинг 4. Структурные элементы выделены жирным шрифтом.

```
var ∈ VARS, f,g ∈ FUNC, C ∈ CONSTR
prog ::= def*
def ::= f(params; params) = expr
params ::= var*
expr ::= if expr then expr else expr
        | g(args; args)
        | value
        | let var := expr in expr
args ::= value*
value ::= var | C(args)
```

*Листинг 4 Объявление модельного языка для объяснения алгоритма специализации*

#### 4.3.2 Специализация функций на объявленном модельном языке

Пусть дана программа на объявленном модельном языке (Листинг 4). Задача специализации: для каждого вызова этой функции определить вызов специализированной функции, содержащий значения динамических параметров и переменные из значений статических параметров, и саму эту функцию, где в теле вместо специализируемых аргументов подставлены соответствующие выражения из вызова.

Поскольку при замены статических аргументов переменными из соответствующих им выражений могут возникать конфликты имён, то будем переименовывать эти переменные последовательностью `_v1, _v2, _v3, ...`. Эти имена гарантировано не вызовут конфликтов, поскольку для пользователя запрещено использовать в начале имени переменной отличный от латинской буквы символ.

Объявим исходную функцию `replace`, пример вызова и опишем алгоритм специализации.

Функция `replace` имеет два статических аргумента – искомый элемент списка и значение, которым элемент будет заменён, и один динамический аргумент – список, по которому нужно провести замены. Предлагается выполнить специализацию для вызова, меняющего местами аргументы `x`, `y` в конструкторе `C` (см. Листинг 5).

```
replace(x, y; list) =
  if is_nil(list) then
    Nil
  else
    let head := car(list) in
    let tail := cdr(list) in
    let new_tail := replace(x, y; tail)
    let new_head := (if head == x then y else head) in
    Cons(new_head, new_tail)

replace(C(x, y), C(y, x); x y z C(x, y) B(x, z) A(z, y))
```

*Листинг 5 Исходная функция на модельном языке*

Пусть мы встретили вызов функции `replace` и хотим его проспециализировать. Для этого:

1. в исходном определении функции заменяем точку запятой на запятую; это значит, что в результате получаем не специализируемую больше функцию;
2. определим сигнатуру вызова функции как перечисление фактических значений статических аргументов через запятую, переменные в них переименуем по формату, описанным выше (см. Листинг 6):
  - если сигнатура тривиальна, то заменяем точку с запятой на запятую в вызове и определении функции и завершаем специализацию;

- если сигнатура совпадает с теми, что уже встречались раньше, выбираем подходящую уже созданную специализированную версию и преобразуем только вызов;

```
replace(C(x, y), C(y, x); x y z C(x, y) B(x, z) A(z, y))

=> C(_v1, _v2), C(_v2, _v1)
```

*Листинг 6 Получение сигнатуры вызова*

3. из нового вызова и нового определения убираем аргументы до точки с запятой, заменяем их на переименованные переменные из соответствующих им фактических выражений (см. Листинг 7);

```
replace(C(x, y), C(y, x); x y z C(x, y) B(x, z) A(z, y))

=> replace@1(x, y, x y z C(x, y) B(x, z) A(z, y))
```

*Листинг 7 Формирование вызова специализированной функции*

4. заменим точку с запятой в вызове и определении специализированной версии функции на запятую;
5. в теле специализированной функции вхождение специализируемых аргументов заменяем на соответствующие им фактические выражения, динамические аргументы остаются без изменений (см. Листинг 8);

```
replace@1 (_v1, _v2, list) =
  if list is nil then
    Nil
  then
    let head := car(list) in
    let tail := cdr(list) in
    let new_tail :=
      replace(C(_v1, _v2), C(_v2, _v1); tail)
    let new_head :=
      (if head == C(_v1, _v2) then C(_v2, _v1) else head) in
      Cons(new_head, new_tail)
```

*Листинг 8 Формирование определения специализированной функции, промежуточный результат*

6. повторяем шаги 4-5, пока не дойдём до неподвижной точки (см. Листинг 9) или не превысим максимально допустимое количество проходов.

```
replace@1 (_v1, _v2, list) =  
  if list is nil then  
    Nil  
  else  
    let head := car(list) in  
    let tail := cdr(list) in  
    let new_tail := replace@1(_v1, _v2; tail)  
    let new_head :=  
      (if head == C(_v1, _v2) then C(_v2, _v1) else head) in  
    Cons(new_head, new_tail)
```

*Листинг 9 Формирование определения специализированной функции, конечный вид*

## 5 Тестирование

Для тестирования работоспособности разработанного алгоритма был составлен набор модульных тестов, включённых в сам комплятор:

- один большой тест на то, что минимально необходимый для специализации внутри комплятора набор библиотечных функций специализируется как положено; это функции Map, MapAccum, Reduce со всеми своими вспомогательными для нескольких вариантов вызовов;
- несколько тестов на проверку, что компиляция со специализацией, имеющей некорректное объявление, будет остановлена после стадии синтаксического анализа и выдаст заданный набор ошибок.

В дальнейшем эти модульные тесты можно (и следует) запускать при добавлении изменений в код специализатора для быстрой проверки стабильности внесённых изменений.

В процессе же написания кода специализатора были использованы некоторые очень искусственные примеры, на которых хорошо видно изменение абстрактного синтаксического дерева прохождениями специализатора.

### 5.1 Специализация функции Replace

Одной из таких наглядных для специализации функций является функция Replace. Она принимает три параметра: искомое выражение, конечное выражение для замены и список элементов для поиска и замены. Возвращает она новый список элементов, в котором в указанном списке искомое выражение и заменено на конечное выражение. При этом входные выражения могут быть любого формата, что удобно для рассмотрения сложных фактических выражений статических параметров, а внутри своего определения функция вызывается рекурсивно, что продемонстрирует необходимость нескольких проходов и проверку тривиальности сигнатуры.

Специализацию функции `Replace` мы будем проводить по первым двум параметрам – искомому и конечному выражениям.

Исходный код функции `replace`, объявление специализации для неё и тестовый пример вызова можно увидеть на Листинг 10.

```
$SPEC Replace (e.FROM) (e.TO) e.items;

Replace {
  (e.f) (e.t) e.f e.Tail = e.t <Replace (e.f) (e.t) e.Tail>;

  (e.f) (e.t) t.X e.Tail = t.X <Replace (e.f) (e.t) e.Tail>;

  (e.f) (e.t) /* empty */ = /* empty */;
}

F {
  s.1 s.2 e.X = <Replace (s.1 s.2) (s.2 s.1) e.X>;
}

$ENTRY Go {
  = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
```

*Листинг 10 Исходный код функции `replace`*

После первых проходов компилятора, получаем следующее абстрактное синтаксическое дерево (см. Листинг 11). Именно такое синтаксическое дерево передаётся на вход оптимизатору. Поскольку у нас включена только специализация, будем называть его дальше специализатором.

```
$SPEC Replace (e.FROM#0) (e.TO#0) e.items#0;

Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1
    = e.t#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;

  (e.f#1) (e.t#1) t.X#1 e.Tail#1
    = t.X#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;

  (e.f#1) (e.t#1) = /* empty */;
}

F {
  s.1#1 s.2#1 e.X#1 = <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.X#1>;
}

$ENTRY Go {
  /* empty */ = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
```

*Листинг 11 Исходное абстрактное синтаксическое дерево*

После первых двух проходов специализатора получаем следующее синтаксическое дерево (см. Листинг 12). В нём были охлаждены рекурсивные вызовы внутри определения функции `Replace` без порождения специализированных версий, поскольку их сигнатура тривиальна. Так же был создан первый вариант специализированной функции `Replace@1` для вызова внутри функции `F`, этот вызов был заменён на сформированный вызов функции `Replace@1`. В самой же функции остался вызов исходной версии `Replace`.

```

Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1
    = e.t#1 <* &Replace (e.f#1) (e.t#1) e.Tail#1*>;
  (e.f#1) (e.t#1) t.X#1 e.Tail#1
    = t.X#1 <* &Replace (e.f#1) (e.t#1) e.Tail#1*>;
  (e.f#1) (e.t#1) = /* empty */;
}

F {
  s.1#1 s.2#1 e.X#1 = <* &Replace@1 s.1#1 s.2#1 (e.X#1)*>;
}

$ENTRY Go {
  /* empty */ = <* &F 1 5 1 1 1 2 2 2 3 3 3*>;
}

Replace@1 {
  s.1#1 s.2#1 (s.1#1 s.2#1 e.Tail#1)
    = s.2#1 s.1#1 <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.Tail#1>;

  s.1#1 s.2#1 (t.X#1 e.Tail#1)
    = t.X#1 <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.Tail#1>;

  s.1#1 s.2#1 () = /* empty */;
}

```

*Листинг 12 Промежуточное синтаксическое дерево*

Для завершения специализации потребовался всего ещё одна итерация специализатора. В результирующем синтаксическом дереве после прохождения специализатора (см. Листинг 13) все вызовы исходной версии Replace заменены на рекурсивные вызовы Replace@1.

```

Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1 = e.t#1 <Replace (e.f#1) (e.t#1)
e.Tail#1>;

  (e.f#1) (e.t#1) t.X#1 e.Tail#1 = t.X#1 <Replace (e.f#1) (e.t#1)
e.Tail#1>;

  (e.f#1) (e.t#1) = /* empty */;
}

F {
  s.1#1 s.2#1 e.X#1 = <Replace@1 s.1#1 s.2#1 (e.X#1)>;
}

$ENTRY Go {
  /* empty */ = <F 1 5 1 1 1 2 2 2 3 3 3>;
}

Replace@1 {
  s.1#1 s.2#1 (s.1#1 s.2#1 e.Tail#1)
  = s.2#1 s.1#1 <Replace@1 s.1#1 s.2#1 (e.Tail#1)>;

  s.1#1 s.2#1 (t.X#1 e.Tail#1) = t.X#1 <Replace@1 s.1#1 s.2#1
(e.Tail#1)>;

  s.1#1 s.2#1 () = /* empty */;
}

```

*Листинг 13 Результирующее абстрактное синтаксическое дерево*

Отметим, что повторные вхождения переменных в фактические значения статических аргументов успешно убраны из образцов и фактических аргументов

новых вызовов. Для однозначного распознавания в сформированном вызове и образцовых выражениях порождённой функции, выражения соответствующие динамическим параметрам заворачиваются в круглые скобки.

## 5.2 Проведение эксперимента на самоприменимом компиляторе

Эксперимент проводился на примере раскрутки компилятора для следующих случаев:

- раскрутка неспециализированным компилятором без специализаций; этот пример компиляции и выполнения программы без каких-либо оптимизаций;
- раскрутка неспециализированным компилятором со специализацией; этот пример покажет, как может измениться время компиляции при проведении оптимизации специализации библиотечных функций Map, MapAccum, Reduce;
- раскрутка специализированным компилятором без специализаций; этот пример покажет, как может измениться время выполнения программы со специализированными библиотечными функциями Map, MapAccum, Reduce;
- раскрутка специализированным компилятором со специализацией; данный пример даёт возможность оценить пользу применения специализации без использования дополнительных оптимизаций, например оптимизации прогонки.

Для каждого случая делалось 13 прогонов, выбирались среднее значение и границы доверительного интервала. Результаты представлены в Таблица 1.

Таблица 1 Результаты тестирования алгоритма специализации на примере раскрутки компилятора

Параметры замера	Среднее значение	Доверительный интервал	
		Левое значение	Правое значение
неспециализированный компилятор, без специализаций	14.60297	13.45032	16.53564
неспециализированный компилятор, со специализацией	20.45314	19.60385	20.77855
специализированный компилятор, без специализаций	14.24620	13.09759	15.45941
Специализированный компилятор, со специализацией	19.10623	18.59413	19.78853

Во-первых, доверительные интервалы для измерений не пересекаются, следовательно результат измерений статистически достоверен. Во-вторых, как ожидалось, проведение самой специализации замедлило компиляцию, в нашем случае на 40%. Однако, неожиданным оказалось то, что специализированный компилятор отработал практически за то же время, что и несспециализированный. Это связано с тем, что мы заворачиваем динамические параметры в скобки, если параметр задан как e-переменная. В списковой реализации эти скобки потребуют больше времени, чем экономия от исключения s-переменной.

## 6 Руководство пользователя

### 6.1 Установка и раскрутка самоприменимого компилятор Рефала-5λ

Для описанных ниже действий необходимо скачать и установить `git`, любой компилятор C++, любой редактор. Рекомендуется использовать в качестве компилятора C++ – компилятором `g++`, а в качестве редактора – IntelliJ IDEA. Если выбор `g++` обоснован только привычкой и тем, что для UNIX-подобных систем он идёт из коробки<sup>1</sup>, то IntelliJ IDEA имеет бесспорное преимущество в виде плагина для подсветки кода, автоподстановки при вводе и валидации кода до компиляции.<sup>2</sup>

Самоприменимый компилятор Рефала-5λ собирается из исходников, которые можно выгрузить из репозитория на `github`<sup>3</sup> командой:

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

Для раскрутки терминала установки компилятора нужно выполнить в терминале в корневой директории компилятора:

Windows	UNIX-like
<code>bootstrap.bat -no-tests</code>	<code>./bootstrap.sh --no-tests</code>

После этого нужно проверить настройки компилятора в скрипте `c-plus-plus.conf.bat (sh)`. Если нужны правки, то после них выполнить пересборку. Последнее, что нужно, – добавить подкаталог `bin` с исполняемыми файлами компилятора к списку каталогов переменной среды `PATH`.

Вот и всё, компилятор готов.

---

<sup>1</sup> Да и тем, что по умолчанию компилятор настроен именно на `g++`

<sup>2</sup> <https://github.com/bmstu-iu9/RefalFiveLambdaPlugin.git>

<sup>3</sup> <https://github.com/bmstu-iu9/refal-5-lambda.git>

## 6.2 Объявление специализации функции

Чтобы компилятор мог специализировать некоторую функцию, в файле с определением функции должна располагаться спецификация её формата в виде:

```
$SPEC ИмяФункции ЖёсткоеВыражение;
```

Здесь *ЖёсткоеВыражение* — это образец, не содержащий двух *e*-переменных на одном скобочном уровне. Также в жёстком выражении запрещено наличие повторных переменных. Имена переменных в этом выражении значения не имеют, за исключением первого символа их индекса. Если первый символ индекса начинается с большой латинской буквы, то параметр, описываемый этой переменной, является специализируемым. Если начинается с малой латинской буквы, дефиса, прочерка или цифры, то параметр считается обычным.

На предложения специализируемой функции накладывается следующее ограничение. Все образцовые выражения должны быть уточнениями указанного формата, при этом в подстановках, переводящих формат функции в каждую из левых частей, специализируемые параметры должны заменяться на жёсткие выражения соответствующего типа. Обычные параметры в подстановках могут заменяться на всё, что угодно (в соответствии с их типом, по определению уточнения).

Специализируемые функции должны быть либо непосредственно определены в текущей единице трансляции, либо включены через `$INCLUDE`.

### 6.3 Несоответствие формата объявления специализации

При несоответствии формата объявления специализации выдаются синтаксические и семантические ошибки, компиляция останавливается по окончании всех проверок. Типовые ошибки можно увидеть в Таблица 2.

Таблица 2 Ошибки при несоответствии формата объявления специализации функции

Причина	Текст ошибки
Переопределение специализации функции $F_n$	Specialization of function $F_n$ is redefined
Специализируемая функция $F_n$ не определена	Orphan function specialization for $F_n$
В шаблоне специализации указаны повторные переменные (повторение переменной $Mode.Index$ )	Repeated variable $Mode.Index$ in specialization pattern
В качестве шаблона специализации функции $F_n$ указано нежёсткое выражение	Pattern of function specialization for $F_n$ must be hard
Не выполнены ограничения на соответствие образцового выражения и шаблона специализации функции $F_n$	Bad function specialization for $F_n$
Не соответствие типов статического параметра $Mode.Index$ шаблона специализации и переменных из образцового выражения функции $F_n$	Type mismatching of static parameter $Mode.Index$ in specialization of function $F_n$
В статическом параметре $Mode.Index$ обнаружено второе или более вхождение переменной из образца предложения функции $F_n$	Repeating variables match static parameter $Mode.Index$ in specialization of function $F_n$

Если обобщённое сопоставление фактического аргумента вызова с шаблоном специализации не выполнилось с результатом безусловно успешно, то для данного вызова функция не специализируется, сообщение об ошибке не выдаётся.

#### 6.4 Компиляция программы с оптимизацией специализации и логирование компиляции

Для компиляции программы с оптимизацией специализации необходимо указать ключ компиляции `-OS`. Ключ можно указать через опции командной строки или через переменные окружения `SRMAKE_FLAGS`, `SREFC_FLAGS`. Для передачи ключей компиляции через переменную окружения `SRMAKE_FLAGS` необходимо использовать префикс `-X`, то есть `-X-OS`.

Для отладки программ удобно использовать логирование, которое вызывается после основных стадий компиляций. Для того, чтобы получить лог, нужно установить значение соответствующего ключа компиляции – имя файла, куда сохраниться лог. Например, через ту же переменную окружения `SRMAKE_FLAGS`:

```
SRMAKE_FLAGS=-X-log=log.txt
```

Подробнее про ключи компиляции и логирование можно прочитать в документации компилятора Рефала-5λ (ссылка на документацию).

## ЗАКЛЮЧЕНИЕ

В результате работы были сформированы требования к формату объявления специализации, к соответствию шаблона специализации и определению специализируемой функции, был реализован алгоритм специализации функций по заданному шаблону в самоприменимом компиляторе языка Рефал-5λ.

Алгоритм специализации был протестирован на раскрутку компилятора. Компиляция со специализацией стала дольше на 40%, при этом из-за заворачивания в круглые скобки динамических параметров, описанных как переменные, выгода от специализации в процессе выполнения программы практически полностью уходит за счёт появления новых узлов при сопоставлении.

Для получения выигрыша по скорости работы программы при оптимизации только специализаций необходима доработка алгоритма, чтобы можно было убрать по крайней мере одни дополнительные скобки. Но возможно получение выигрыша по скорости работы программы при комплексной оптимизации методами специализации, встраивания и прогонки.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Климов Андрей Викторович Введение в метавычисления и суперкомпиляцию [Раздел книги] // Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям. - Москва : КомКнига, 2008. - ISBN 978-5-484-01028-8.
2. Климов Андрей Викторович и Романенко Сергей Анатольевич Метавычислитель для языка Рефал. Основные понятия и примеры [Книга]. - Москва : ИПМ им.М.В.Келдыша АН СССР, 1987. - препринт N 71 : стр. 32.
3. Турчин Валентин Фёдорович Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ [Конференция] // Труды симпозиума "Теория языков и методы построения систем программирования". - Киев-Алушта : [б.н.], 1972. - стр. 31-42.
4. Ершов Андрей Петрович О сущности трансляции [Статья] // Программирование, №5. - 1977 г.. - стр. 21-39.
5. Futamura Yoshihiko Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler [Статья] // Systems, Computers, Controls 2, №5. - 1971 г.. - стр. 45-50.
6. Jones Neil D., Gomard Carsten K. и Sestoft Peter Partial Evaluation and Automatic Program Generation [Книга]. - [б.м.] : Prentice-Hall, 1993.
7. Jones Neil D., Sestoft Peter и Sondergaard Harald An Experiment In Partial Evaluation: The Generation Of A Compiler Generator [Статья] // Sigplan Notices, Vol.20, №8. - 1985 г.. - стр. 82-87.
8. Sestoft Peter The Structure Of A Self-Applicable Partial Evaluator [Конференция] // Programs As Data Objects, Copenhagen, Lecture Notes In Computer Science 217 / ред. Ganzinger H. и Jones Neil D.. - [б.м.] : Springer-Verlag, 1986. - стр. 236-256.

9. Turchin Valentin Fedorovich The algorithm of generalization in the supercompiler [Статья] // Partial Evaluation and Mixed Computation / ред. Bjørner Dines, Ershov Andrei Petrovich и Jones Neil D.. - North-Holland : Elsevier Science Publishers B.V., 1988 г.. - стр. 531-549.
10. Turchin Valentin Fedorovich The language Refal, the theory of compilation and metasystem analysis. Technical Report 20 [Книга]. - New York University : Courant Institute of Mathematical Sciences, 1980.
11. Немытых Андрей Петрович О суперкомпиляции (к 80-летию со дня рождения В. Ф. Турчина) [Конференция] // Международная конференция "Современные проблемы математики, информатики и биоинформатики", посвященная 100-летию со дня рождения А. А. Ляпунова. - Академгородок, Новосибирск : [б.н.], 11 - 14 октября 2011. - ISBN 978-5-905569-03-6.
12. Романенко Сергей Анатольевич Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру [Отчет] / ИПМ им.М.В.Келдыша АН СССР. - Москва : [б.н.], 1987.

## ПРИЛОЖЕНИЕ А

Листинг 1 Определение шаблона на языке C++ .....	13
Листинг 2 Определение функции Add1 сложения чисел в унарной системе счисления.....	18
Листинг 3 Определение библиотечных функций Map и Apply .....	22
Листинг 4 Объявление модельного языка для объяснения алгоритма специализации.....	28
Листинг 5 Исходная функция на модельном языке .....	29
Листинг 6 Получение сигнатуры вызова .....	30
Листинг 7 Формирование вызова специализированной функции.....	30
Листинг 8 Формирование определения специализированной функции, промежуточный результат .....	31
Листинг 9 Формирование определения специализированной функции, конечный вид .....	32
Листинг 10 Исходный код функции replace.....	34
Листинг 11 Исходное абстрактное синтаксическое дерево .....	35
Листинг 12 Промежуточное синтаксическое дерево .....	36
Листинг 13 Результирующее абстрактное синтаксическое дерево .....	37
Таблица 1 Результаты тестирования алгоритма специализации на примере раскрутки компилятора.....	39
Таблица 2 Ошибки при несоответствии формата объявления специализации функции.....	42