



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:

“Встраивание функций в компиляторе РЕФАЛ5-λ”

Студент _____ ИУ9-82
(Группа)

(Подпись, дата) _____ К.А.Ситников
(И.О.Фамилия)

Руководитель ВКР

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

Консультант

(Подпись, дата) _____ (И.О.Фамилия)

Консультант

(Подпись, дата) _____ (И.О.Фамилия)

Нормоконтролер

(Подпись, дата) _____ (И.О.Фамилия)

2019 г.

АННОТАЦИЯ

Темой данной работы является “Встраивание функций в компиляторе РЕФАЛ-5λ”. Объем работы составляет 55 страниц.

Основной объект исследования – эквивалентные преобразования функций в языке РЕФАЛ, а точнее – преобразования прогонки и встраивания. С помощью этих преобразований исходный код программы на языке преобразуется с целью ускорить выполнение скомпилированной программы.

Дипломная работа состоит из трех глав. Первая глава посвящена теоретическим сведениям, необходимым для реализации прогонки, результатам исследования эквивалентных преобразований функций в других диалектах языка РЕФАЛ и их расширение на язык РЕФАЛ-5λ. В главе “Разработка” описывается модификация компилятора языка, необходимые для осуществления преобразования, а также тонкости реализации алгоритма оптимизации. В главе “Тестирование” рассматриваются результаты применения оптимизации встраивания и прогонки к самому компилятору и оценивается прирост производительности при выполнении программы.

Содержание

ВВЕДЕНИЕ.....	4
1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ.....	6
1.1 Обзор языка РЕФАЛ5-λ.....	6
1.1.1 Введение.....	6
1.1.2 Символы и выражения.....	6
1.1.3 Сопоставление с образцом.....	8
1.1.4 Функции в языке РЕФАЛ-5λ.....	9
1.2 Абстрактная рефал-машина.....	12
1.2.1 Обзор работы.....	12
1.2.2 Фаза анализа.....	12
1.2.2 Фаза синтеза.....	14
1.2.4 Сущность прогонки.....	15
1.3 Алгоритм обобщенного сопоставления.....	17
1.3.1 L-выражения.....	17
1.3.2 Сужения и присваивания.....	19
1.3.3 Алгоритм обобщенного сопоставления.....	20
1.3.4 Эквивалентные преобразования функций.....	27
1.4 Архитектура компилятора РЕФАЛ5-λ.....	34
2 РАЗРАБОТКА.....	36
2.1 Модификации языка, необходимые для оптимизации.....	36
2.1.1 Реализация поддержки ключевых слов \$DRIVE,\$INLINE.....	36
2.1.2 Реализация поддержки синтаксиса списка \$ENTRY.....	39
2.2 Разработка оптимизационного алгоритма.....	41
2.2.1 Реализация алгоритма решения уравнений.....	41
2.2.2 Реализация встраивания и прогонки функций.....	42
2.2.3 Руководство пользователя.....	48
3 ТЕСТИРОВАНИЕ.....	52
ЗАКЛЮЧЕНИЕ.....	54
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	55

ВВЕДЕНИЕ

Разработка программного обеспечения – сложная задача. Есть несколько критериев качества программного обеспечения. Выделим из них:

- Производительность;
- Сопровождаемость.

Сопровождаемость ПО включает в себя качество исходного кода программного продукта, возможность поддерживающих этот продукт программистов легко разбираться в коде и вносить в него изменения. К сожалению, качественно реализованный исходный код программного продукта – не всегда является оптимальным решением. А оптимальные решение очень часто являются малоподдерживаемыми. В этом случае можно организовать процессы оптимизации программ непосредственно в компиляторе языка программирования. Именно поэтому разработка алгоритмов оптимизации для компиляторов является очень важной и актуальной задачей.

В языке C++ есть ключевое слово `inline`, которое, будучи записано перед именем функции, делает её встраиваемой. Вызовы встраиваемых функций компилятор заменяет на непосредственно тела этих функций. В данной работе рассматриваются способы реализации похожей функциональности для языка РЕФАЛ-5λ. Цель работы – организация оптимизации встраивания и прогонки функций в компиляторе этого языка.

В рамках работы требуется решить следующие задачи:

- Изменение грамматики языка РЕФАЛ-5λ и организация поддержки новых синтаксических конструкций, необходимых для реализации оптимизации в компиляторе.

- Разработка алгоритма оптимизации встраивания и прогонки функций в компиляторе.
- Оптимизация самоприменимого компилятора языка РЕФАЛ-5λ с помощью разработанных решений.
- Оценка результатов оптимизации прогонки и встраивания функций.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор языка РЕФАЛ5-λ

1.1.1 Введение

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

Язык РЕФАЛ-5λ [1] является диалектом языка РЕФАЛ-5 и одновременно его точным надмножеством. Это значит, что компилятор языка может компилировать программы, написанные на языке РЕФАЛ-5, без изменений семантических свойств. Основным отличием диалекта РЕФАЛ-5λ является поддержка анонимных функций и функций высшего порядка. Это значит что функции в этом случае являются данными и могут быть использованы в качестве аргументов вызовов других функций.

1.1.2 Символы и выражения

Символ – базовый и минимальный синтаксический элемент для описания данных в языке РЕФАЛ. Символы имеют несколько видов:

- идентификатор – последовательность знаков, которая может включать в себя буквенные символы, цифры, дефисы и нижние подчеркивания.

Примером корректных идентификаторов являются: `True`, `False`, `Post-Process`;

- печатные знаки – все печатные знаки, используются в качестве символов. Для отличия от идентификаторов заключаются в одинарные кавычки. При этом несколько печатных знаков, заключенных в кавычки являются не отдельным символом, а набором символов. Пример: `'ABC'` – набор символов `'A'`, `'B'`, `'C'`;
- неотрицательные целые числа (макроцифры);
- действительные числа;
- указатели на функции – представляют собой знак `'&'` и следующий за ним идентификатор, который в свою очередь является именем некоторой определенной в области видимости функции. Пример: `&Palindrome`.

Основным синтаксическим элементом для организации структур данных в языке РЕФАЛ-5λ являются структурные скобки. Структурные скобки могут быть вложенными. Пример выражения со структурными скобками: `(A (C)) E F`. Также существуют и именованные структурные скобки, использующиеся для построения абстрактных типов данных. Они представляют из себя выражение в квадратных скобках, первым символом которых является идентификатор, именуемый абстрактный тип данных, представляющий из себя имя функции в текущей области видимости.

Пример выражения с именованными скобками: `[Person 'Name' 'Age']`.

Объектным выражением – называется некоторая последовательность *термов*, где каждый терм представляет из себя либо произвольный символ, либо выражение в структурных или именованных скобках. Пустое выражение, не содержащее ни одного символа, также является объектным выражением.

1.1.3 Сопоставление с образцом

Другим типом выражений в языке являются выражения образцы. Образцы отличаются от объектных выражений тем что могут содержать переменные. Переменная имеет тип и идентификатор. Всего в языке РЕФАЛ-5λ есть три типа переменных:

- *s-переменные* – значением переменной этого типа может быть только один символ;
- *t-переменные* – значением переменной этого типа может быть произвольный терм;
- *e-переменные* – значением переменной этого типа может быть произвольное объектное выражение;

Переменные имеют следующий формат: TYPE ' . ' INDEX, где TYPE – тип переменной, а INDEX – идентификатор или целое положительное число.

Выражение-образец также определяет множество объектных выражений языка, которые могут быть подставлены в данный образец при условии, что все значения переменных будут допустимыми. Процесс декомпозиции объектного выражения и присваивания значений переменным из выражения-образца называется *сопоставлением с образцом*. Приведем пример этого процесса. Пусть имеется выражение образец:

$$s.1 \ e.2 \ s.1 \quad (1)$$

В этом случае множество объектных выражений, которые могут быть сопоставлены с данным образцом состоит из выражений, у которых первый и

последний символ выражения одинаковы. Пусть имеется следующее объектное выражение:

$$A \ (A \ B \ C \ () \ (E) \ A) \quad (2)$$

При сопоставлении выражения 2 с образцом 1 переменная $s.1$ примет значение A , а переменная $e.2 - (A \ B \ C \ () \ (E) \ A$. Если не существует набора значений, которые с учетом ограничений могут принять переменные из выражения-образца, то сопоставление считается неудачным. Пример: пусть имеется объектное выражение $A \ B \ C$ и выражение-образец $s.1 \ s.2$. Сопоставление с образцом завершится неудачей, так как множество объектных выражений, которые могут сопоставиться с этим выражением представляют из себя выражения, состоящие только из двух символов.

1.1.4 Функции в языке РЕФАЛ-5λ

Основным элементом программ, написанных на языке РЕФАЛ-5λ являются функции [2]. Определение регулярной функции имеет следующий формат:

Function := ` \$ENTRY` ? Name { Block }

Модификатор $\$ENTRY$ используется для указания области видимости функции. При его наличии функция является глобальной, а при отсутствии – локальной, то есть может использоваться только в рамках одной единицы трансляции. У каждой функции есть уникальное имя. Блок представляет из себя упорядоченный набор предложений. Предложение имеет следующую структуру:

Sentence := Left = Right

Левая часть предложения – это выражение-образец, с последующими условиями, правая часть предложения является *результатным выражением*. Результатное выражение отличается от выражения-образца тем, что может включать в себя еще и вызовы функций:

$$\text{FunctionCalls} := \langle \text{FunctionName Arg}^* \rangle$$

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

Пусть имеется функция F , которая определяется следующим набором предложений:

$$L_1 = R_1$$

$$L_2 = R_2$$

$$\dots$$

$$L_n = R_n$$

Пусть в некоторой части программы есть вызов функции с аргументом A . Тогда процесс вычисления функции упрощенно можно описать следующим образом:

1. Положим $i=1$.
2. Выполняется попытка сопоставления выражения A с образцом L_i . Если сопоставление успешно, то результатом функции будет результат вычисления выражения R_i .
3. Если сопоставление завершилось неудачей, то переходим на шаг 1, увеличивая счетчик: $i:=i+1$.

4. Если $i > n$, то программа завершается с ошибкой, так как не нашлось необходимого образца для сопоставления.

Пример функции, написанной на языке РЕФАЛ-5λ описан в листинге 1.1.

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

Листинг 1.1 Пример функции в языке РЕФАЛ-5λ

В примере представлена функция, определяющая, является ли последовательность символов палиндромом. Определения функций в этом языке очень близки к математическим описаниям алгоритмов. В этом случае утверждается, что пустые последовательности и последовательности, состоящие из одного символа являются палиндромами. Если последовательность начинается и заканчивается на один и тот же символ, необходимо проверить, является ли палиндромом подпоследовательность без первого и последнего символа. В противном случае последовательность символов палиндромом не является.

1.2 Абстрактная рефал-машина

1.2.1 Обзор работы

Процесс выполнения программы описанной на языке РЕФАЛ-5λ можно описать с помощью абстрактной рефал-машины [3].

Рабочая память абстрактной рефал машины называется *полем зрения*. В поле зрения могут находиться только *определенные* выражения. *Определенные* выражения отличаются от объектных выражений тем, что могут содержать в себе вызовы функций с аргументом, который представляет из себя объектное выражение.

Процесс исполнения программы на языке РЕФАЛ-5λ является конечной последовательностью шагов этой машины. Последовательность шагов представляет собой последовательность чередующихся между собой фаз *анализа* и *синтеза*.

1.2.2 Фаза анализа

Пусть на этом этапе в поле зрения находится определенное выражение вида $E = s_1..s_k < F a_0 > s_{k+1}...s_n$, а функция F определяется набором предложений вида:

$$L_1 = R_1$$

$$L_2 = R_2$$

...

$$L_m = R_m$$

Фаза анализа разбивается на более мелкие шаги:

1. Рефал машина выбирает в поле зрения *активный* вызов функции. Активным в некотором объектном выражении называют вызов функции закрывающая скобка которого находится ближе всего к началу определенного выражения. Пусть для определенного выражения E активным вызовом является $\langle F a_0 \rangle$.

2. Теперь для каждого из предложений функции последовательно осуществляется сопоставление с образцом. Стоит отметить, что в некоторых случаях решений по сопоставлению с образцом оказывается несколько.

е-переменные, для которых существует несколько решений при сопоставлении с образцом, называются *открытыми* переменными. Рефал-машина однозначно решает, какие значения должны быть присвоены открытым переменным. Пусть имеется левая часть предложения L_i функции F , при этом $L_i = l_1 e_1 l_2 e_2 \dots l_n e_n l_{n+1}$, где подвыражения $l_1 \dots l_{n+1}$ являются объектными выражениями, не содержащими открытых переменных, а $e_1 \dots e_n$ – открытые переменные. В этом случае производится сопоставление с образцом и выбирается решение, в котором переменная e_1 заменяется на последовательность с наименьшей длиной. Пусть этим выражением является $s_1 \dots s_l$. Далее производится подстановка этого выражения в $L_i = l_1 s_1 \dots s_l l_2 e_2 \dots l_n e_n l_{n+1}$, и процесс продолжается до тех пор, пока всем свободным переменным выражения не будут присвоены значения. В случае неуспешного сопоставления с образцом осуществляется переход к сопоставлению выражения a_0 с выражением образцом L_{i+1} .

Если $i > n$, то рефал-машина аварийно завершается с ошибкой “Отождествление невозможно”.

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

1.2.2 Фаза синтеза

Пусть имеется предложение вида $L_j = R_j$, набор подстановок для всех переменных, содержащихся в L_j , а в поле зрения рефал-машины находится выражение $E = s_1 \dots s_k < F a_0 > s_{k+1} \dots s_n$. На этом этапе происходит применение подстановок к правой части предложения R_j , и затем преобразованное выражение R_c вставляется в поле зрения рефал-машины вместо вызова функции F . Поле зрения приобретает вид $E' = s_1 \dots s_k R_c s_{k+1} \dots s_n$.

В начале работы абстрактной рефал машины в ее поле зрения находится функция G_0 , так как она является входной точкой в программу, реализованную на языке РЕФАЛ-5λ.

Рефал машина работает до тех пор, пока в поле зрения не окажется вызовов функций. В этом случае определенное выражение, которое находится в поле зрения рефал-машины является результатом ее работы.

Ниже в листинге 1.2 представлен пример программы, а таблице 1.1 представлено ее последовательное выполнение на рефал-машине.

```

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

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

```

Листинг 1.2 Пример программы

Таблица 1.1 Последовательное выполнение программы рефал-машиной

№ шага	Присваивания	Поле зрения
1		<Go >
2	'A' ← s.1 'BACB' ← e.m	<IsPalindrome 'ABACBA'>
3	'B' ← s.1 'AC' ← e.m	<IsPalindrome 'BACB'>
4	False ← e.Other	<IsPalindrome 'AC'>
5		False

1.2.4 Сущность прогонки

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

$$A_1 S_1 A_2 S_2 \dots A_n S_n$$

Как известно из предыдущих рассуждений, на шаге анализа аргумент вызова функции разделяется на составные части, затем происходит синтез – соединение этих составных частей. На следующем шаге анализа аргумент вызова

опять разделяется на составные части, при этом часто на те же самые, что и в предыдущем шаге. Перестройка выражений может занимать значительное время. Также после выполнения каждого шага рефал-машины вся информация о разборе выражения на составные части теряется. Это приводит к ухудшению производительности программы.

Основной задачей прогонки является композиция нескольких аналогичных шагов рефал-машины в один [4]. В этом случае процесс работы рефал машины можно представить цепочкой $A'S'...A_nS_n$, в котором шаг анализа A' является композицией шагов A_1 и A_2 , а шаг синтеза S' является композицией шагов S_1 и S_2 . Таким образом, оптимизация прогонки позволяет программе выполняться быстрее, путем уменьшения количества промежуточных вызовов функций.

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

1.3 Алгоритм обобщенного сопоставления

1.3.1 L-выражения

В статье В.Ф.Турчина [5] рассматривались система преобразований функций в некотором подмножестве языка РЕФАЛ. На это подмножество накладывались некоторые ограничения:

- исключено использование t-переменных;
- исключено использование открытых и повторных e-переменных выражения.

В частности давалось следующее определение *L-выражения*. Объектное выражение называется L-выражением, если:

- любое подвыражение этого выражения не должно содержать более одной e-переменной, не входящей в скобки;
- любая e-переменная выражения не должна входить в него более одного раза.

В рамках этой работы предлагается расширить подмножество языка РЕФАЛ для проведения прогонки и встраивания функции, а именно снять ограничение на использование t-переменных. Далее L-выражением будем называть объектное выражение следующего вида:

- любое подвыражение этого выражения не должно содержать более одной e-переменной, не входящей в скобки;
- e-переменные и t-переменные не должны входить в выражение более одного раза.

Примерами L-выражений являются:

$$\begin{array}{c} A \ B \ s.1 \ B \ C \\ s.Y \ (e.1 \ t.2) \ e.3 \ s.X \end{array}$$

Примеры выражений-образцов, которые не являются L-выражениями:

$$\begin{array}{c} e.1 \ s.m \ e.2 \\ t.1 \ (s.X \ t.1) \end{array}$$

В ограниченном рефале левые части предложений могут быть только L-выражениями, и эквивалентные преобразования функций рассматриваются только над этим подмножеством языка РЕФАЛ. В рамках этой работы также будут рассмотрено эквивалентное преобразование функций с помощью прогонки без для более широкого подмножества языка РЕФАЛ.

Теорема. Каждое подвыражение L-выражения также является L-выражением.

Теорема. Пусть E_i L-выражение, которое переходит в объектное выражение E_i при замене переменных на некоторые значения. Тогда этот набор замен является единственным.

Каждое выражение-образец E_i определяет множество объектных выражений, которые могут быть с ним сопоставлены. Это множество далее будем называть *классом* E_i .

Подстановка – замена всех переменных, входящих в E_i на некоторые объектные выражения, которые называются их *значениями*, при условии, что значения всех вхождений одной переменной должны совпадать. Применение множества подстановок A к выражению E_i будем обозначать как $A // E_i$.

Теорема. Если E_t – выражение-образец, а A – множество подстановок, то $A // E_t \subseteq E_t$.

Пример. Пусть есть выражение следующего вида:

e.X e.Y t.1 s.3 e.X

и подстановки $[e.X \rightarrow t.2]$, $[t.1 \rightarrow s.2]$. После применения подстановок к выражению оно примет вид:

t.2 e.Y s.2 s.3 t.2

1.3.2 Сужения и присваивания

Вспомогательной задачей для осуществления оптимизации прогонки является поиск пересечения некоторых классов выражений. Пусть имеется выражение-образец P и произвольное выражение E . Необходимо найти $E \cap P$. Пересечение можно представить в виде:

$$E \cap P = C_1 // E \cup \dots \cup C_r // E$$

где $C_1..C_r$ множества подстановок, и эти множества исчерпывают пересечение классов $E \cap P$. Множества $C_i // E$ также являются попарно непересекающимися, то есть $\forall i, j: C_i \cap C_j = \emptyset$.

Элементы множеств C_i будем называть *сужениями* и обозначать их следующим образом:

$$[v \rightarrow E_i]$$

где v – переменная, которая входит в выражение E , а E_i – объектное выражение, соответствующее типу переменной.

Каждому множеству сужений C_i соответствует множество A_i некоторых подстановок, таких что:

$$A_i // P = C_i // E$$

Элементы множеств этих подстановок будем называть *присваиваниями* и записывать следующим образом:

$$(E_i \leftarrow v)$$

1.3.3 Алгоритм обобщенного сопоставления

Пусть необходимо уравнение вида $E:P$. Эта задача сводится к поиску пересечений классов $E \cap P$.

Будем называть пару $\langle C, A \rangle$, где C – список сужений следующего вида:

$$\{[v_1 \rightarrow L_1] .. [v_k \rightarrow L_k]\}$$

где v_i – переменные, из E , а A – список присваиваний следующего вида:

$$\{(L_{k+1} \leftarrow v_{k+1}) .. (L_m \leftarrow v_m)\}$$

где, v_i – переменные из P , *решением сопоставления*.

Тройку $\langle C, A, Q \rangle$, где Q – набор уравнений вида $\{E_1:P_1..E_n:P_n\}$ будем называть *частично разрешенным сопоставлением*. Алгоритм обобщенного сопоставления принимает на вход уравнение, а результатом его работы является набор решений сопоставления [6].

В процессе работы алгоритм оперирует с системой независимых между собой частично разрешенных сопоставлений, выглядящей следующим образом:

$$S = \begin{bmatrix} \langle C_1, A_1, Q_1 \rangle \\ \langle C_2, A_2, Q_2 \rangle \\ \dots \\ \langle C_n, A_n, Q_n \rangle \end{bmatrix}$$

Каждое частично разрешенное сопоставление обрабатывается алгоритмом независимо. В начале работы алгоритма его конфигурация выглядит следующим образом:

$$S_0 = \langle \emptyset, \emptyset, \{E_i:E_i\} \rangle$$

Ниже приведены шаги алгоритма обобщенного сопоставления:

1. Выбираем из S первый кортеж $\langle C_i, A_i, Q_i \rangle$, для которого Q_i содержит хотя бы одно уравнение, и далее выберем из этого множества первое уравнение q_j .
2. Далее решаем уравнение q_j . В зависимости от решения уравнения, конфигурация S может измениться:
 - из S может быть удален кортеж;
 - в S может быть добавлено несколько новых кортежей;
 - в некоторый кортеж может быть добавлено несколько уравнений;

- из некоторого кортежа может быть удалено некоторое уравнение;
 - в кортежи могут быть добавлены новые сужения и присваивания;
3. Выполняем шаги 1 и 2 до тех пор пока для каждого кортежа множество уравнений не станет пустым.
 4. Для каждого кортежа в отдельности выполняем постобработку решения.
 5. Формируем общее решение уравнения на основе результатов постобработки.

Стоит отметить, что добавление нового сужения $[v \rightarrow e]$ в кортеж $\langle C, A, Q \rangle$ должно сопровождаться его модификацией. Для всех уравнений применяется подстановка. Все вхождения переменной v в левой и правой частях уравнений заменяются на e :

$$Q := \{ E_i : P_i \in Q : [v \rightarrow e] // E_i : [v \rightarrow e] // P_i \}$$

Также модификации подвергаются и присваивания. Ко всем левым частям присваиваний применяется подстановка:

$$A := \{ (e_i \leftarrow v_i) \in A : ([v \rightarrow e] // e_i \leftarrow v_i) \}$$

Пример. Пусть имеется кортеж следующего вида:

$$\begin{aligned} C &= \{ \}, \\ A &= \{ s.x \leftarrow s.y, s.x \leftarrow s.z \}, \\ Q &= \{ s.x \ s.x : s.1 \ s.2 \} \end{aligned}$$

Необходимо добавить сужение $[s.X \rightarrow A]$, в этом случае кортеж после преобразования примет следующий вид:

$$\begin{aligned}
C &= \{ s.X \rightarrow A \} \\
A &= \{ A \leftarrow s.y, A \leftarrow s.z \} \\
Q &= \{ A \ A : s.1 \ s.2 \}
\end{aligned}$$

Рассмотрим непосредственно решения уравнений. Для начала рассмотрим уравнения вида $T_e:T_p$, где T_e и T_p – являются термами. Под термами подразумеваются:

- t- и s-переменные;
- выражения в структурных скобках и именованных скобках;
- символы (включая указатели на функции).

Для всех вариантов решения уравнения термов исходное уравнение удаляется из кортежа. Все варианты решения уравнения указаны в приоритетном порядке.

1. Если уравнение имеет вид $S:S$, где S – символ, то в этом случае конфигурация не модифицируется.
2. Если уравнение имеет вид $t.X:S$ или $s.X:S$, где S – символ, то в этом случае в кортеж добавляется новое сужение $[t.X \rightarrow S]$ или $[s.X \rightarrow S]$, соответственно.
3. Если уравнение имеет вид $s.X:s.Y$ или $s.X:t.Y$, то в этом случае в кортеж добавляется новое присваивание $(s.X \leftarrow s.Y)$ или $(s.X \leftarrow t.Y)$, соответственно.
4. Если уравнения имеет вид $S:s.X$, где S – символ, то в этом случае в кортеж добавляется новое присваивание $(S \leftarrow s.X)$.
5. Если уравнение имеет вид $s.X:T$, где T – скобочный терм, то в этом случае уравнение решений не имеет и из конфигурации необходимо удалить кортеж, из которого было взято исходное уравнение.

6. Если уравнение имеет вид $(E_r):(P_r)$, то в этом случае необходимо в исходный кортеж добавить новое уравнение $E_r:P_r$. Аналогично обрабатываем ситуацию с именованными скобками, при условии совпадения имен этих скобок.
7. Если уравнение имеет вид $t.X:s.Y$, то в этом случае необходимо в исходный кортеж добавить новое сужение $[t.X \rightarrow s.Y]$.
8. Если уравнение имеет вид $T:t.X$, где T – терм, то в этом случае в исходный кортеж добавляется новое присваивание $(T \leftarrow t.X)$.
9. Если уравнение имеет вид $t.X:(P_r)$, то в этом случае необходимо добавить новое сужение $[t.X \rightarrow (e.N)]$ и новое уравнение $e.N:P_r$. В этом случае переменная $e.N$ – новая сгенерированная переменная, которой нет в выражениях E, P .
10. В остальных случаях решений уравнения нет , необходимо удалить исходный кортеж из конфигурации.

Далее рассмотрим общие варианты решения уравнений для произвольных выражений. В этих вариантах исходное уравнение также всегда удаляется из конфигурации.

Если уравнение имеет вид $T_e E_e : T_p E_p$, где T_e, T_p – термы, то в этом случае в исходный кортеж необходимо добавить новые уравнения:

$$\begin{array}{l} T_e : T_p \\ E_e : E_p \end{array}$$

Аналогично обрабатывается ситуация, когда термы стоят в правых частях выражений: $E_e T_e : E_p T_p$.

Если уравнение имеет вид $E_e:e.X$, то в этом случае необходимо в кортеж добавить новое присваивание $(E_e \leftarrow e.x)$.

Если уравнение имеет вид $e.1..e.n:\epsilon$, то в этом случае на каждую e -переменную необходимо добавить новое сужение $[e.i \rightarrow \epsilon]$.

Если уравнение имеет вид $\epsilon:P_l T P_r$, где T - терм, то в этом случае решений нет, и исходный кортеж необходимо удалить из конфигурации.

Если уравнение имеет вид $e.x E_e:T_p E_p$, то в этом случае происходит *ветвление*. В конфигурацию добавляются новые кортежи, которые являются модифицированными исходными кортежами, а сам исходный кортеж удаляется:

$$\left[\begin{array}{l} [e.x \rightarrow \epsilon] \\ E_e:T_p E_p \\ [e.x \rightarrow t.New1 e.New2] \\ t.New1:T_p \\ e.New2 E_e:E_p \end{array} \right]$$

Аналогично обрабатываем ситуацию для уравнений типа $E_t e.x:E_l T_l$.

Далее для каждого найденного кортежа вида $\langle C_i, A_i \rangle$ необходимо выполнить постобработку решений. Постобработка решений является альтернативой логике использования “чужих” переменных.

Выполняем постобработку решений кортежа $\langle C_i, A_i, Q_i \rangle$ до тех пор, пока ни одно из следующих условий не будет выполнено:

1. Если среди присваиваний есть $(s.X \leftarrow v)$ и $(X \leftarrow v)$, то в этом случае необходимо добавить новое сужение $[s.X \rightarrow v]$ и удалить первое присваивание. Стоит отметить, что добавление нового сужения в кортеж осуществляется аналогично добавлению сужения в частично решенное сопоставление.

2. Если среди присваиваний есть $(s.A \leftarrow v)$ и $(s.B \leftarrow v)$, то в этом случае необходимо добавить новое сужение $[s.B \rightarrow s.A]$, а второе присваивание удалить.
3. Если среди присваиваний есть два идентичных присваивания $(e \leftarrow v)$, то необходимо одно из них удалить.
4. Если среди присваиваний есть противоречивые – $(X \leftarrow v)$ и $(Y \leftarrow v)$, где $X \neq Y$, то в этом случае сопоставление считается неудачным и необходимо удалить этот кортеж из конфигурации.
5. Если среди присваиваний есть присваивания с повторными t- или e-переменными, причем присваивания в разные выражения, то в этом случае, мы не можем точно сказать, есть ли решение у системы, поэтому требуются дополнительные исследования для частных случаев. Результат постобработки кортежа – “Неизвестное решение”.

После постобработки каждого отдельного кортежа производится их общий анализ:

1. Если множество решений пусто, то алгоритм возвращает результат “Ошибочное сопоставление”.
2. Если результатом постобработки хотя бы одного кортежа является “Неизвестное решение”, то результатом работы всего алгоритма сопоставления является “Неизвестное решение”.
3. В остальных случаях алгоритм возвращает множество кортежей вида $\langle C_i, A_i \rangle$, где C_i – сужения, A_i – присваивания.

Стоит также отметить что на части уравнения, которые решает алгоритм не накладываются ограничения, в частности – правая часть уравнения может не быть L-выражением. Это ограничение снято потому, что в процессе работы алгоритма может выясниться, что система несовместна, и в дальнейшем этот вызов будет оптимизирован.

Пример. Рассмотрим следующую систему уравнений:

$$A \text{ e.1} : B \text{ e.2} \text{ e.3} \text{ e.2}$$

Уже на первом шаге алгоритма, выяснится, что решение несовместно, так как левая и правая части уравнений начинаются с разных символов.

Также повторные в результате работы алгоритма могут получиться присваивания повторным t- и e-переменным одних и тех же выражений.

Пример. Рассмотрим следующую систему уравнений:

$$s.1 (s.X) s.X : A (t.Y) t.Y$$

В этом случае решение уравнения будет следующий набор сужений и присваиваний:

$$\begin{aligned} C &= \{ [s.1 \rightarrow A] \} \\ A &= \{ (s.X \leftarrow t.Y) \} \end{aligned}$$

1.3.4 Эквивалентные преобразования функций

Алгоритм обобщенного сопоставления используется для эквивалентных преобразований функций. Функции F, F' называются эквивалентными если для любых входных данных результат их вычисления одинаков. Одним из таких эквивалентных преобразований является преобразование *прогонки* [7].

Пусть имеется функция F , определение которой выглядит следующим образом:

$$F = \{$$

$$L_{F1} = R_L < G a_0 > R_R$$

$$L_{F2} = R_{F2}$$

$$\dots$$

$$L_{Fn} = R_{Fn}$$

$$\}$$

Функция G имеет определение:

$$G = \{$$

$$L_{G1} = R_{G1}$$

$$L_{G2} = R_{G2}$$

$$\dots$$

$$L_{Gm} = R_{Gm}$$

$$\}$$

Рассмотрим случай, когда решение уравнения $a_0 : L_{G1}$ непусто, и сопоставление успешно. Тогда решение уравнения имеет следующий вид:

$$a_0 : L_{G1} = \left[\begin{array}{l} < C_1, A_1 > \\ < C_2, A_2 > \\ \dots \\ < C_k, A_k > \end{array} \right.$$

В этом случае можно преобразовать исходную функцию F в функцию F' , так, что функции F, F' будут эквивалентными:

$$\begin{aligned}
F' = \{ & \\
& C_1 // L_{F1} = C_1 // R_l A_1 // R_{G1} C_1 // R_R \\
& \dots \\
& C_k // L_{F1} = C_k // R_l A_k // R_{G1} C_k // R_R \\
& L_{F1} = R_l < G^1 a_0 > R_R \\
& \dots \\
& L_{Fn} = R_{Fn} \\
& \}
\end{aligned}$$

Функцией G^i будем обозначать функцию, построенную из G путем исключения из нее первых i предложений. То есть:

$$\begin{aligned}
G^i = \{ & \\
& L_{Gi+1} = R_{Gi+1} \\
& L_{Gi+2} = R_{Gi+2} \\
& \dots \\
& L_{Gm} = R_{Gm} \\
& \}
\end{aligned}$$

Если же множество решений уравнения пусто, то это означает, что не существует элемента класса множества a_0 , который может успешно сопоставиться с выражением-образцом L_{F1} , результат вычисления функции G в этом контексте не может быть R_{G1} , и поэтому можно просто заменить вызов функции G на вызов G^1 . В этом случае функция F' будет выглядеть следующим образом:

$$\begin{aligned}
F' = \{ & \\
& L_{F1} = R_l < G^1 a_0 > R_R \\
& \dots \\
& L_{Fn} = R_{Fn} \\
& \}
\end{aligned}$$

Этот процесс преобразования функций $F \rightarrow F' \rightarrow F'' \rightarrow \dots \rightarrow F^*$ продолжается до тех пор, пока не будут рассмотрены варианты сопоставления со всеми левыми частями функции G , или не возникнет условие останова – левая часть предложения функции F не является L-выражением, или результат сопоставления не определен. После прогонки итоговая функция F^* будет иметь следующий вид:

$$F^* = \{ \begin{aligned} &C_{11} // L_{F1} = C_{11} // R_l A_{11} // R_{G1} C_{11} // R_R \\ &\dots \\ &C_{1k_1} // L_{F1} = C_{k_1} // R_l A_{1k_1} // R_{G1} C_{k_1} // R_R \\ &\dots \\ &C_{m1} // L_{F1} = C_{m1} // R_l A_{m1} // R_{G1} C_{m1} // R_R \\ &\dots \\ &C_{mk_m} // L_{F1} = C_{mk_m} // R_l A_{mk_m} // R_{G1} C_{mk_m} // R_R \\ &\dots \\ &L_{Fn} = R_{Fn} \end{aligned} \}$$

Стоит отметить, что в таких случаях на левые части предложений функции F накладываются ограничения – они должны быть L-выражениями. В противном случае при преобразовании у функции может измениться семантика.

Пример. В листинге 1.3 приведен пример функции для которой меняется семантика после преобразования прогонки. До преобразования функция F при передаче в нее аргумента `C A` приводит к ошибке, так как переменная $s.X$ после сопоставления принимает значение C , а это значение находится вне области определения функции G . После преобразования при передаче функции F этого же аргумента, он успешно подходит под первое предложения и результат вычисления функции F будет 1.

```

F {
    e.A s.X e.B = <G s.X>;
    e.A = Z;
}

G { A = 1; B = 2; }

/* После преобразования */
G*2 {
/* пустая функция, вызов приводит к ошибке */
}

F {
    e.A A e.B = 1;
    e.A B e.B = 2;
    e.A s.X e.B = <G*2 s.X>
    e.A = Z;
}

```

Листинг 1.3 Преобразование функции, меняющее семантику

Прогонка функций для не L-выражений также может привести к ухудшению производительности, если в левой части предложения есть повторная е-переменная, так как прогонка может привести к увеличению количества предложений. Пример представлен в листинге 1.4. В этом случае семантика функции не меняется, но переменную *e.X* приходится сравнивать дважды.

```

F {
    (e.X) e.X s.X e.Q = <G s.X>;
    e.A = Z;
}

$DRIVE G;
G {
    A = 1;
    B = 2;
}

```

```

/* После прогонки */
F {
    (e.X) e.X A e.Q = 1;
    (e.X) e.X B e.Q = 2;
    (e.X) e.X s.N e.Q = <G*2 s.N>;
    e.A = Z;
}
G*2 {}

```

Листинг 1.4 Пример прогонки с повторными переменными

Однако даже для выражений, не являющихся L-выражениями можно произвести эквивалентные преобразования. Если результатом решения уравнения $a_0:L_{G1}$ является единственный кортеж $\langle \emptyset, A_1 \rangle$, то это означает, что аргумент a_0 однозначно сопоставится с выражением-образцом L_{G1} , пустое множество сужений означает, что левая часть предложения L_{F1} не будет изменена, что также приводит к невозможности изменения семантических свойств. В этом случае вызов функции G можно заменить на ее результат вычисления с применением сужений и исходная функция F примет после преобразования следующий вид:

$$\begin{aligned}
 F = \{ \\
 & L_{F1} = R_l A_1 // R_{G1} R_R \\
 & L_{F2} = R_{F2} \\
 & \dots \\
 & L_{Fn} = R_{Fn} \\
 & \}
 \end{aligned}$$

Пример. В листинге 1.4 приведен пример эквивалентного преобразования функции F у которой левая часть одного из предложений не является L-выражением.


```

F {
    e.A s.X e.B = <Eq s.X s.X>;
}

Eq {
    s.Y s.Y = True s.Y;
    e.Other = False;
}

/* после преобразования */

F {
    e.A s.X e.B = True s.X;
}

```

Листинг 1.4 Эквивалентное преобразование без сужений

В это случае необходимо вызов функции `Eq` был заменен на результат вычисления этой функции и изменения семантических свойств функции нет.

1.4 Архитектура компилятора РЕФАЛ5-λ

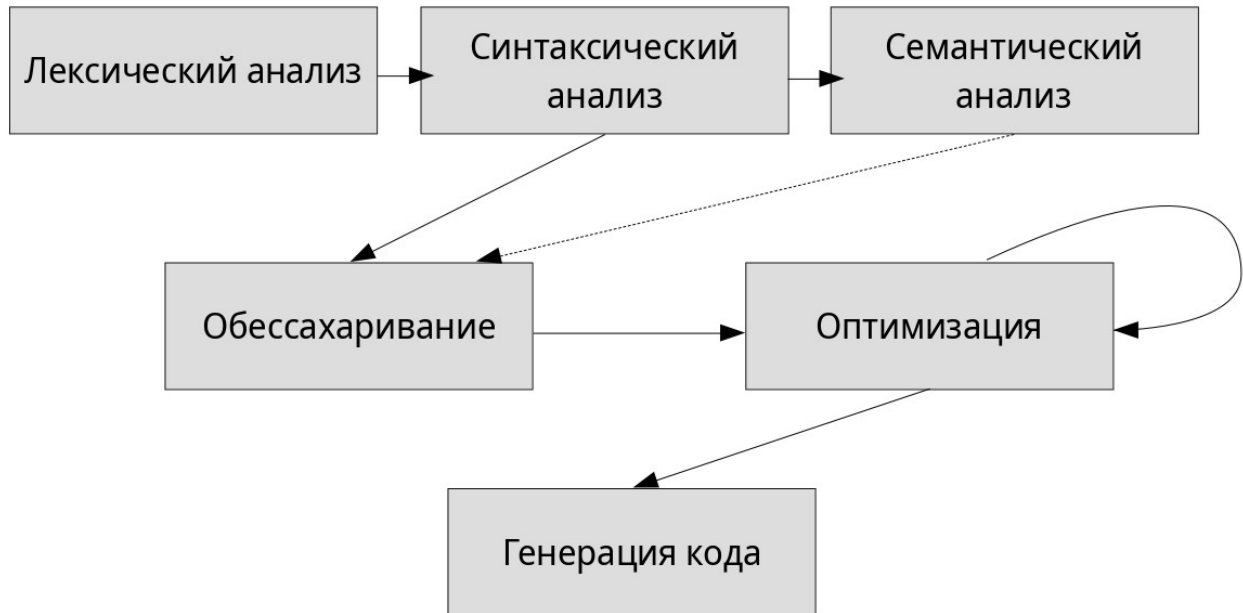


Рисунок 1.1 Архитектура компилятора

Компиляция программы на языке состоит из нескольких этапов.

1. *Лексический анализ*. На этом этапе из входного потока символов формируется выходной поток токенов, корректных для языка.
2. *Синтаксический анализ*. На этом этапе на основе грамматики языка из потока токенов строится абстрактное синтаксическое дерево.
3. *Семантический анализ*. На этом этапе проходит проверка семантики построенного синтаксического дерева. Пример: поиск объявлений функций без их определения и.т.д.
4. *Обессахаривание* абстрактного синтаксического дерева. Принимает дерево, порожденное на этапе синтаксического анализа, при условии, что

семантический анализ успешно завершился. На этом этапе некоторые синтаксические конструкции являющиеся «синтаксическим сахаром», преобразуются, и результатом этого этапа является упрощенное синтаксическое дерево. Пример: все анонимные функции именуются, их тела выносятся из тел других функций на верхний уровень и заменяются их вызовами.

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

2 РАЗРАБОТКА

2.1 Модификации языка, необходимые для оптимизации

2.1.1 Реализация поддержки ключевых слов \$DRIVE,\$INLINE

Для проведения оптимизации прогонки и встраивания необходимо средство в языке, с помощью которого можно передавать информацию компилятору, о том, что вызовы той или иной функции должны быть подвергнуты оптимизации прогонки или встраивания. В связи с этим было принято решение добавить в язык два новых ключевых слова – \$DRIVE и \$INLINE.

Эти ключевые слова должны сопровождаться списком функций, вызовы которых должны быть оптимизированы, при этом все функции, указанные в этом списке, должны быть определены в рамках этой единицы трансляции, в противном случае компилятор, выдаст сообщение об ошибке. Ниже в листингах 2.1 и 2.2 приведены пример корректного и некорректного использования ключевых слов.

```
$INLINE A, B;  
  
A { ... }  
  
B { ... }  
  
C { ... }  
  
$DRIVE C;
```

Листинг 2.1 Корректное использование ключевых слов

```
$INLINE A;  
  
C { ... }  
  
$DRIVE B;
```

Листинг 2.2 Некорректное использование ключевых слов

Для поддержки ключевых слов, были сделаны изменения практически во всех стадиях компилятора.

Лексический анализ. На этой стадии при анализе входного потока символов для ключевых слов `$INLINE` и `$DRIVE` генерируются токены `TkInline` и `TkDrive`.

Синтаксический анализ. На этой стадии, если во входном потоке присутствует один из вышеупомянутых токенов, и за ним следует список имен для каждого имени, на верхнем уровне синтаксического дерева генерируется узел одного из следующих форматов:

```
(Drive t.Pos t.ScopeClass e.Name)  
(Inline t.Pos t.ScopeClass e.Name)
```

Семантический анализ. На этой стадии в последовательные проходы по синтаксическому дереву программы был добавлен новый – проверка всех меток оптимизации прогонки и встраивания. Если в некоторой единице трансляции присутствуют метки для оптимизации, а сама функция не определена, то компилятор выдает сообщение об ошибке.

Обессахариватель. Для стадии были реализованы функции для модификации синтаксического дерева посредством удаления избыточных меток оптимизации. Если для одной функции присутствуют несколько меток, то

избыточные метки удаляются. Если для одной функции присутствуют метки и `Inline` и `Drive`, то приоритет имеет метка `Drive`. Она остается в дереве, остальные метки удаляются. Также на этой стадии выполняется вынесение анонимных функций в отдельные функции. Так как все анонимные функции по умолчанию являются прогоняемыми, то для всех этих функций на этом этапе генерируются метки `Drive`.

Оптимизация. На стадии оптимизации модифицируется синтаксическое дерево программы с целью построить более эффективную его версию для стадии генерации кода, без изменения семантических свойств программы. Количество итераций оптимизации программы является параметром запуска компилятора и по умолчанию оно равно 100.

В компилятор была добавлена новая стадия оптимизации для прогонки и встраивания, состоящая из нескольких процедур:

- подготовительная процедура;
- выполнение оптимизации.

В рамках подготовительной процедуры компилятор осуществляет проход по синтаксическому дереву и собирает информацию, необходимую для последующих оптимизаций. Ниже описан формат данных, используемый для оптимизации:

```
t.FunctionName* (t.SpecificInfo*)
t.SpecificInfo := (s.Label s.ScopeClass (e.Name) e.Body)
```

Информация для оптимизации состоит из имен прогоняемых функций и специфической информации. Она состоит из следующих частей:

- `s.Label` – метка оптимизации `Drive` или `Inline`;
- `s.ScopeClass` – метка видимости функции (глобальная или локальная);

- `e.Name` – имя функции;
- `e.Body` – тело функции, состоящее из набора предложений.

Стоит отметить, что список имен оптимизируемых функций состоит не только из прогоняемых или встраиваемых. Пусть в прогоняемой функции F есть N предложений. В этом случае в список имен оптимизируемых функций добавляются также специфические имена вида $F * i$, где $i = 1 \dots N$. Такая запись обозначает функцию F без первых i предложений. Эти особые функции используются впоследствии в алгоритме оптимизации, но как отдельные единицы синтаксического дерева они не существуют. Во время оптимизации алгоритм может породить такие функции, на основе тел функций, находящихся в специфической информации для оптимизации.

Каждая итерация оптимизации возвращает модифицированное синтаксическое дерево. Оптимизация продолжается до тех пор пока синтаксическое дерево не перестает изменяться на каждой итерации, или до тех пор, пока не будет превышено максимальное количество итераций.

2.1.2 Реализация поддержки синтаксиса списка `$ENTRY`

Также для удобства проведения оптимизации была разработана поддержка синтаксиса меток `$ENTRY` в отдельности от объявления функции. По умолчанию в компиляторе все функции являются локальными и могут быть использованы только в рамках отдельной единицы трансляции. Вышеупомянутое ключевое слово используется для пометки функции как *глобальной*. Если функция является глобальной, то ее область видимости можно расширить для других единиц трансляции с помощью ключевого слова `$EXTERN`. При использовании предыдущей версии компилятора метку глобальной функции можно было

использовать только непосредственно перед именем функции. В новой версии компилятора была добавлена поддержка синтаксиса списка глобальных функций:

```
$ENTRY A, B, C;
```

В этом случае каждая функция из списка помечается как глобальная.

При разработке поддержки этого синтаксиса в компиляторе была изменена лишь стадия синтаксического анализа. На этой стадии при обработке такого списка для каждой функции из него на верхний уровень синтаксического дерева добавляются метки вида:

```
(Entry t.Pos GN-Entry e.Name)
```

Стоит отметить, что функции, помеченные как глобальные, должны быть объявлены в той же единице трансляции, иначе компилятор выдаст сообщение об ошибке.

Пример использования этого синтаксиса вместе с оптимизацией прогонки или встраивания представлен в листинге 2.4. В этом случае при включении файла `LibraryEx.srefi` в код, определенные в нем функции будут иметь локальную область видимости, а при отдельной трансляции библиотеки они будут глобальными.

```
//file LibraryEx.refi
Apply { ... }
$INLINE Apply;

//file LibraryEx.sref
$INCLUDE "LibraryEx"
$ENTRY Apply;
```

Листинг 2.4. Пример использования списка `$ENTRY`

2.2 Разработка оптимизационного алгоритма

2.2.1 Реализация алгоритма решения уравнений

Основной частью оптимизации прогонки и встраивания является алгоритм решения уравнений. В процессе разработки этот алгоритм был реализован в соответствии с его описанием в пункте 1.3.3. Ниже представлена сигнатура функции `Solve`, осуществляющей решение одного уравнения:

```
<Solve (e.UsedVars) t.Equation>  
  == Success t.Result* | Undefined | Failure  
t.Contr ::= (t.Var ':' e.Val)  
t.Equation ::= ((e.Expr) ':' (e.LExpr))  
t.Assign ::= (e.Val ':' t.Var)  
t.Result ::= ((t.Contr*) (t.Assign*))
```

Функция принимает список переменных, используемых в уравнении и непосредственно уравнение, которое необходимо решить. Список используемых переменных нужен для того, чтобы избежать коллизий в индексах переменных при формировании новых переменных при осуществлении обобщенного сопоставления.

Результат работы функции может иметь один из трех следующих видов:

- `Success t.Result*` – сопоставление успешно, `t.Result` представляет собой набор сужений и присваиваний;
- `Failure` – сопоставление невозможно;
- `Undefined` – решение неизвестно, оптимизация вызова невозможна.

Исходная функция решает только одного уравнения. Для обработки отдельных множества кортежей вида $\langle C, A, Q \rangle$ используется вспомогательная рекурсивная функция `Solve-Aux`, которая вызывает сама себя и несколько раз и соединяет результаты своих вычислений. Пример обработки ситуации с ветвлением в функции `Solve-Aux` представлен в листинге 2.5.

```
(e.UsedVars) (e.Contrs) ((e.Pe t.Pt) ':' (e.He t.Ht))
e.Equations (e.Assigns)
...
    = <Solve-Aux
      (e.UsedVars)
      e.Branch1
    >
    <Solve-Aux
      (e.UsedVars)
      e.Branch2
    >;
```

Листинг 2.5 Пример ветвления в функции `Solve-Aux`

2.2.2 Реализация встраивания и прогонки функций

Оптимизация прогонки и встраивания базируется на эквивалентном преобразовании прогонки из пункта 1.3.3.

Далее будут рассмотрены действия, выполняемые алгоритмом оптимизации в рамках одного оптимизационного прохода по синтаксическому дереву.

Для начала рассмотрим оптимизацию прогонки. Алгоритм проходит по всему синтаксическому дереву и пытается оптимизировать предложения всех функций у которых в правой части есть *пассивный* вызов прогоняемой функции, а соответствующая левая часть представляет собой простое выражение-образец без условий. Пассивным называется вызов, аргументы которого на момент вычисления известны, то есть среди аргументов нет вызовов других функций.

Сначала алгоритм извлекает аргументы $e.Args$ из оптимизируемого вызова и преобразует правое предложение путем замены вызова на специфический узел вида:

$$(TkVariable \ 'e' \ DRIVEN \ 0)$$

который впоследствии будет заменен на некоторое выражение, в зависимости от результатов оптимизации.

Затем извлекается тело прогоняемой функции из вспомогательной информации для оптимизации. Пусть тело выглядит следующим образом:

$$\begin{aligned} e.Left1 &= e.Right1; \\ &\dots \\ e.LeftN &= e.RightN; \end{aligned}$$

Далее, осуществляется попытка решения уравнения $e.Args : e.Left1$ и в зависимости от решения уравнения исходная функция модифицируется.

Если результат решения уравнения однозначен и представляет из себя одно множество сужений и присваиваний $(e.Contrs)(e.Assigns)$, причем множество сужений пусто, то в этом случае в множество сужений добавляется новое сужение вида:

$$(TkVariable \ 'e' \ DRIVEN \ 0) \rightarrow e.Assigns \ // \ e.Right1 \ (2.1)$$

Далее сужения применяются к левой и правой частям выражения, таким образом обеспечивается применение сужений к сегментам правой части предложения, расположенным слева и справа от вызова, соответственно. Набор предложений

функции не модифицируется. Пример одного прохода прогонки для этого случай представлен в листинге 2.6.

```
$DRIVE H;

H {
    s.1 A = True;
    s.1 s.1 = False;
}

G {
    s.1 e.1 = <H s.1 A>
}

/* Преобразованный код функции G */

G {
    s.1 e.1 = True;
}
```

Листинг 2.6 Пример прогонки без сужений

Если результат решения уравнений представляет несколько множеств сужений и присваиваний, то это означает, что сопоставление неоднозначно и заменить вызов функции на результат ее вычисления можно лишь с некоторыми ограничениями. Также в этом случае существуют ограничения на левую часть оптимизируемого предложения – она должно быть L-выражением, иначе в процессе оптимизации может поменяться семантика функции (пункт 1.3). Если же левая часть не удовлетворяет этому условию, то обработка решения уравнения происходит аналогично обработке неизвестного результата.

Для каждого множества вида (e.Contrs) (e.Assigns) конструируется новое предложение путем добавления нового сужения вида 2.1 и применения этих сужений к левой и правой части предложения. В полученное множество предложений также добавляется новое предложение, в котором вызов

оптимизируемой функции G заменяется на вызов новой функции G^{*1} , которая представляет из себя функцию G , за исключением первого предложения. Если же функция, вызов которой мы оптимизируем уже имеет вид G^{*N} , то ее вызов заменяется на вызов функции G^{*N+1} . С помощью этих преобразований обеспечивается последовательная прогонка всех предложений функции G . Также в этом случае эти новые функции добавляются в синтаксическое дерево. Пример такого преобразования представлен в листинге 2.7.

```
$DRIVE EQ;

EQ {
    s.X s.X = True s.X;
    s.X s.Y = False s.X s.Y;
}

F {
    s.A e.B s.C = <EQ s.A s.C>;
}

/* После одного прохода прогонки */

EQ*1 {
    s.X s.Y = False s.X s.Y;
}

F {
    s.A e.B s.A = True s.A;
    s.A e.B s.C = <EQ*1 s.A s.C>;
}
```

Листинг 2.7 Пример прогонки с сужениями

Отсутствие решения у уравнения (результат – Failure) означает, что ни при каких значениях переменных результатом вычисления функции G выражение $e.Right1$ быть не может. В этом случае в оптимизируемом предложении вызов

функции G , заменяется на вызов функции G^*1 . Пример оптимизации при отсутствии решений уравнений представлен в листинге 2.8.

```
$DRIVE H;

H {
    s.1 A = True;
    s.1 s.1 = False;
}

G {
    s.1 e.1 = <H s.1 B>
}

/* После прогонки */

G {
    s.1 e.1 = <H*1 s.1 B>
}

$DRIVE H*1;

H*1 {
    s.1 s.1 = False;
}
```

Листинг 2.8 Пример прогонки при отсутствии решений

Если же результат решения уравнений не известен, то в этом случае оптимизация функции невозможно, поэтому исходное предложение заменяется на новое, в которой вызов функции заменяется на *холодный* вызов. Холодным называется вызов, который впоследствии при следующих итерациях прогонки, оптимизирован не будет.

Стоит отметить, что после оптимизации функции все холодные вызовы функций в правой части оптимизируемого предложения заменяются на обычные, так как на последующих итерациях эти функции могут быть оптимизированы.

Далее в листинге 2.9 представлен пример полной прогонки функции за несколько итераций.

```
$DRIVE TEST;
TEST {
    (e.A) F e.B (e.C F) = e.A e.B e.C;
    (e.A) A e.B (A e.C) = e.A e.B e.C;
    (A e.A) B e.B (C e.C) = e.A e.B e.C;
}

TARGET {    e.X = <TEST (e.X) e.X (e.X)>}

/* После прогонки */

TARGET {
    F = F;

    F e.0#0 F = F e.0#0 F e.0#0 F F e.0#0;

    A e.#0 = A e.#0 e.#0 e.#0;

    e.X#1 = <TEST*3 (e.X#1) e.X#1 (e.X#1)>;
}

TEST*3 {}
```

Листинг 2.9 Пример полной прогонки функции

Оптимизация встраивания отличается от оптимизации прогонки только тем, что в ней не рассматриваются неоднозначные решения уравнений, то есть решения, в которых есть сужения. Пример успешного встраивания представлен в листинге 2.10.

```
$INLINE H;

H {
    s.X s.X = True;
    e.Other = False;
}
```

```
G {
    s.1 s.1 = <H s.1 s.1>
}

/* После встраивания */

G {
    s.1 s.1 = True;
}
```

Листинг 2.10 Оптимизация встраивания

2.2.3 Руководство пользователя

Не для всех функций оптимизация прогонки или встраивания может привести к уменьшению времени их выполнения, поэтому ниже изложены советы по использованию разработанной оптимизации.

В частности, стоит аккуратно использовать эти оптимизации для рекурсивных функций, так как для них прогонка без ограничений на количество итераций могут выполняться бесконечно, при условии что в рекурсивных вызовах происходит последовательное сужение некоторой переменной. При наличии ограничений будет разрастаться правая часть прогоняемого предложения, что может привести к ухудшению производительности.

Рассмотрим функцию `Apply`. Ее исходное определение и результат встраивания представлен в листинге 2.11.

```
$INLINE Apply;

Apply {
    s.Fn e.Argument = <s.Fn e.Argument>;

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



```
F { e.Args = <Apply Foo e.Args>}  
  
/* после встраивания */  
  
F { e.Args = <Foo e.Args>}
```

Листинг 2.11 Встраивание рекурсивной функции

Функция `F` успешно оптимизирована, и дополнительный вызов функции в правой части предложения был устранен. Если исходная функция `Apply` помечена как прогоняемая, то второе предложение функции на каждой итерации оптимизируется, и переменная `t.Closure` будет на каждой итерации сужаться до `(t.Closure ...)`, что приводит к разрастанию правой части предложения и ухудшению производительности. Стоит отметить, что если последовательного сужения переменных нет, то можно подвергнуть оптимизации прогонки и рекурсивную функцию.

Рассмотрим пример функции, для которой оптимизация встраивания также может привести к ухудшению производительности:

```
$INLINE Loop;  
  
Loop {  
    e.X = <Loop e.X A>;  
}  
  
$ENTRY Go {  
    = <Exit 0> <Loop>;  
}
```

Листинг 2.12 Ухудшение производительности для встраивания

В этом случае функция `Loop` никогда не будет вызвана, но прежде чем программа завершится, в поле зрения будет помещено выражение `<Loop A A ... A>`, в котором находится 100 символов `A`.

Встраивание функций целесообразно использовать для функций, которые не являются рекурсивными, так как в этом случае преобразование не может привести к ухудшению производительности, а также для рекурсивных функций, у которых на каждой итерации аргумент функции уменьшается.

Прогонка для нерекурсивных функций также может привести к ухудшению производительности, так как прогонка может увеличить количество предложений в функции. Рассмотрим пример прогонки функции F , представленный в листинге 2.13.

```
G {
  (e.A) A e.B (e.C A) = A;
  (e.A) B e.B (e.C B) = B;
  (e.A) C e.B (C e.C) = C;
}

F {
  e.X (e.B) (e.C) = <G (e.X) e.X (e.X)>
}

/* после прогонки */

TARGET {
  A (e.B#1) (e.C#1) = A;
  A e.0#0 A (e.B#1) (e.C#1) = A;
  B (e.B#1) (e.C#1) = B;
  B e.0#0 B (e.B#1) (e.C#1) = B;
  C e.#0 (e.B#1) (e.C#1) = C;
  e.X#1 (e.B#1) (e.C#1) =
    <TEST*3 (e.X#1) e.X#1 (e.X#1)>;
}
```

Листинг 2.13 Ухудшение производительности после прогонки

Если в этом примере в функции G чаще всего вызывается аргументы, подходящие под третье предложение, то после оптимизации прогонки количество

сопоставлений с образцом, которые нужно сделать, чтобы достичь этого предложения, увеличивается, что и приводит к ухудшению производительности.

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

`-O`flags``

где ``flags`` - набор флагов оптимизации.

Для организации оптимизации прогонки и встраивания было введено два новых флага – `I` и `D`.

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

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

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

3 ТЕСТИРОВАНИЕ

Компилятор языка РЕФАЛ является самоприменимым поэтому для тестирования оптимизаций проводилось на сборке компилятора оптимизированной программой.

Для тестирования оптимизаций для некоторых функций из стандартной библиотеки были расставлены метки `$INLINE`. Метки `$DRIVE` были расставлены вручную для некоторых функций модуля `Generator-RASL.ref`. В остальных модулях неявно прогонялись вложенные функции.

Тестирование проводилось на компьютере со следующими характеристиками:

- частота процессора – 2.9ГГц;
- объем оперативной памяти – 8ГБ;
- ОЗУ – Linux Ubuntu 18.04;
- компилятор языка C++ - g++.

Тестирование оптимизаций производилось с помощью специальной утилиты для тестирования производительности, которая осуществляет несколько раз осуществляет сборку компилятора и дает информацию о процессе сборки, в частности:

- количество шагов рефал-машины, выполненных при сборке;
- время затраченное на компиляцию.

Сначала тестирование было проведено для компилятора с отключенными ключами оптимизаций, затем с ключом оптимизации встраивания, и далее – с ключом оптимизации пргонки. Сборка компилятора в каждом случае производилась 17 раз. Для времени компиляции были измерены медиана и

значения на границах первого и четвертого квартилей. Также в каждом из случаев было измерено количество шагов рефал-машины.

Таблица 3.1 Результаты тестирования

Сборка	Кол-во шагов	Время компиляции, медиана, с	Время компиляции, I квартиль, с	Время компиляции IV квартиль, с
Без оптимизаций	27607449	29.45	29.04	29.48
Оптимизация встраивания	25545258	26.23	26.22	26.25
Оптимизация прогонки	24683318	25.45	25.43	25.49

Исходя из результатов тестирования:

- С включенной оптимизацией встраивания количество шагов сократилось на 2062191, медианное время компиляции уменьшилось на 3.22 секунды или на 10.9%.
- С включенной оптимизацией прогонки количество шагов сократилось на 2924131, медианное время компиляции уменьшилось на 4 секунды или на 13.5%.

ЗАКЛЮЧЕНИЕ

В процессе выполнения этой работы были выполнены все установленные задачи. В частности, были изучен диалект РЕФАЛ-5λ, способы эквивалентного преобразования функций в других диалектах языка РЕФАЛ. Компилятор этого языка был модифицирован: реализована поддержка синтаксиса для прогоняемых и встраиваемых функций, и разработаны соответствующие им алгоритмы оптимизации функций, и произведено их тестирование.

Алгоритмы оптимизации были успешно применены к некоторым функциям из исходного компилятора, что обеспечило прирост производительности программы. Другие функции из компилятора РЕФАЛ-5λ также могут быть встроены или прогнаны, что теоретически может дать еще больший прирост его производительности. Также разработанные алгоритмы могут быть использованы разработчиками для оптимизации других программ.

В дальнейшем работа предполагает расширение подмножества языка РЕФАЛ-5λ, которое может быть подвергнуто оптимизации встраивания и прогонки, а также обработку частных случаев решения уравнений сопоставления, что может дать еще больший рост производительности программам на языке РЕФАЛ-5λ. Также теоретическую часть работы можно использовать для реализации оптимизаций в языках, основанных на сопоставлении с образцом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Компилятор Рефал-5 λ [Электронный ресурс] – Режим доступа: <https://github.com/bmstu-iu9/r-5-lambda>. Дата обращения: 19.03.2019.
2. V.F.Turchin. REFAL-5 programming guide & reference manual [Электронный ресурс] – Режим доступа: <http://refal.botik.ru/book/html>. Дата обращения: 23.04.2019.
3. А.П.Немытых. Лекции по языку программирования Рефал. Сборник трудов по функциональному языку программирования Рефал. Том №1 — Переславль-Залесский: Сборник, 2014.
4. С.А.Романенко. Прогонка для программ на РЕФАЛе-4. Препринт №211 — Институт Прикладной Математики АН СССР, 1987.
5. В.Ф.Турчин. Эквивалентные преобразования программ на РЕФАЛе — ЦНИПИАС, 1974.
6. V.F.Turchin. The Basics of Metacomputation — Obninsk, 1990.
7. В.Ф.Турчин. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ — Киев-Алушта, 1972.