Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ*

КАФЕДРА *ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему:

Оптимизация образцовых выражений в языке программирования РЕФАЛ

Студент *Батусов П. В.*

Руководитель курсового проекта *Коновалов А. В.*

Москва 2014

Аннотация

В данной работе рассматриваются возможные методы и подходы для оптимизации образцовых выражений в языке программирования РЕФАЛ. Цель поставленной задачи — доработка компилятора, порождающего более быструю исполняемую программу. В работе описывается и обосновывается необходимость данной оптимизации, а также производится оценка производительности оптимизированных программ.

Содержание

ВВЕДЕНИЕ

РЕФАЛ — РЕкурсивных Функций АЛгоритмический язык программирования, является одним из старейших функциональных языков, ориентированный на символьные вычисления: обработку символьных строк (например, алгебраические выкладки); перевод с одного языка (искусственного или естественного) на другой; решение проблем, связанных с искусственным интеллектом. Соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ [1].

Первая версия РЕФАЛа была создана в 1966 году Валентином Турчиным в качестве метаязыка для описания семантики других языков. Впоследствии, в результате появления достаточно эффективных реализаций на ЭВМ, он стал находить практическое использование в качестве языка программирования [2].

РЕФАЛ программа может состоять из одного или нескольких модулей (файлов), каждый из которых, в свою очередь, состоит из функций. РЕФАЛ-функция представляет собой упорядоченный набор предложений, состоящих из образца и шаблона. На вход функции подается некоторое выражение; вычисление функции состоит в сопоставлении выражения поочередно образцам, взятым из первого, второго и т. д. предложений. Если очередное сопоставление проходит успешно, то на основании шаблона, взятого из того же предложения, формируется новое РЕФАЛ-выражение, которое и будет результатом функции. В случае, если ни с одним из имеющихся образцов аргумент функции сопоставить не удалось, фиксируется ошибка (аварийно завершается вся программа). Во избежание этого часто в конце функции помещают предложение, с образцом которого можно сопоставить вообще произвольное выражение. В некоторых современных реализациях РЕФАЛа (например, РЕФАЛ+) неуспех любого выражения функции вместо ошибки порождает неуспех самой функции.

В РЕФАЛе, как и во многих других функциональных языках программирования, в отличие от императивных языков, отсутствуют циклы. Для описания циклических процессов используется рекурсия. Зачастую, при таком подходе, тело функции, реализующей цикл, содержит большое число предложений, с очень похожими образцами. Если оптимизировать их вычисление, например создать дерево из левых частей предложений (в пределах одной функции), чтобы операторы, общие для нескольких предложений, выполнялись только один раз, это может дать большой прирост скорости выполнения программы.

1 Базисный РЕФАЛ

РЕФАЛ является одним из старейших функциональных языков, отличительная черта которого — использование сопоставления с образцом и переписывания термов как основного способа определения функций.

1.1 Предложения и функции

Предложением называется конструкция вида: «образцовое выражение = результатное выражение;». Образец часто называют левой частью, результат — правой частью, в которой могут использоваться только те переменные, которые определены в левой части.

РЕФАЛ-функция представляет собой упорядоченный набор таких предложений. Ее выполнение заключается в сопоставлении переданного аргумента с левой частью первого предложения. Если сопоставление оказывается успешным, то значения переменных, определенных в образце, подставляются в правую часть и вызов функции заменяется на получившийся результат. При неудачном сопоставлении с образцом, точно таким же образом выполняется второе предложение и т. д. Если сопоставление не удалось в левой части последнего предложения, то программа прекращает работу с ошибкой невозможности сопоставления (recognition impossible).

Обычно в левых частях выражений рассматриваются различные варианты допустимых аргументов, поэтому ошибка невозможности сопоставления часто говорит о том, что или на вход функции был передан некорректный аргумент, или сама функция написана с ошибкой. Таким образом, в РЕФАЛ четко разделены операции анализа аргумента, выполняемые при помощи сопоставления с образцом, и операции синтеза, осуществляемые при помощи подстановки значений переменных в результатное выражение.

В качестве аргументов функции передается последовательность (возможно пустая) термов, которыми могут быть:

— обыкновенные символы — буквы, цифры и т. д.;

— символы-метки (идентификаторы);

— цифры — цифровая запись неотрицательных целых чисел, не превышающих предельное значение;

— выражение в структурных скобках;

— активное выражение (означающее вызов функции);

1.2 Сопоставление с образцом

Образцы РЕФАЛа могут содержать свободные переменные, которые состоят из указателя типа, точки и индекса. Индексом переменной является последовательность букв и цифр. Есть три указателя типа. Они обозначаются малыми буквами s, t и e, а соответствующие им переменные называются s-, t- и e-переменными. Различие между ними состоит в множествах их допустимых значений. Значением s-переменной может служит только один символ, а t-переменной — любой терм. Значением e-переменной может быть любое выражение [3].

Если в образце присутствует несколько вхождений переменных с одинаковым типом и одинаковым именем, то такие переменные называются повторными. Все вхождения повторной переменной должны иметь одинаковое значение. В образце не может быть переменных, имеющих одинаковое имя, но при этом разные типы.

Сопоставлением объектного выражения E с образцом P называется поиск значений переменных, входящих в образец P, подстановка которых в P дает выражение E. Если такие значения найти невозможно, то сопоставление считается неуспешным.

Если существует несколько вариантов сопоставления, то выбирается тот, в котором первая переменная (при чтении слева-направо) принимает кратчайшее значение (имеется ввиду длина в термах). Если это правило не устраняет неоднозначности, то анализируется следующая переменная и т. д. При таком соглашении сопоставление в РЕФАЛе становится однозначной операцией.

Для разных образцовых выражений операция сопоставления будет выполняться за разное время. Так, например простые операции: сопоставление с атомом, s-переменной, с парой скобок выполняются за постоянное время, а сопоставления каждой открытой e-переменной линейно зависит от длины «сканируемого» объектного выражения, поэтому одна e-переменная добавляет линейную сложность, две — квадратичную, три — кубическую. Повторная t- или e-переменная требует рекурсивного сравнения объектных термов или объектных выражений, соответственно, поэтому сложность сопоставления линейно зависит от числа атомов и скобок, входящих в сравниваемые выражения. Пара повторных переменных требует одного сравнения, 3 переменных — двух сравнений и т. д.

1.3 Необходимость оптимизации

Синтаксис РЕФАЛа не позволяет функциям принимать более одного аргумента — объектного выражения. При передачи нескольких параметров, они «склеиваются» в одно объектное выражение, которое потом в теле функции приходится разбирать на части соответствующие входным аргументам. Таким образом зачастую можно встретить ситуацию, когда несколько образцовых частей предложения одной функции имеют похожую структуру, в результате чего одни и те же операции анализа аргумента функции выполняются несколько раз. Это крайне неэффективно, особенно если такая функция описывает некоторый циклический процесс. Кроме этого, оптимизация необходима в случае если компилятор РЕФАЛа порождает код для каждого предложения функции отдельно (пример — рис. 1).

Код на Рефале

```
// Объявления библиотечных функций
$EXTERN WriteLine, Dec, Mul;
// Объявление локальной функции
$FORWARD Fact;
// Точка входа в программу
$ENTRY Go {
  = <WriteLine '6! = ' <Fact 6>>;
}

Fact {
  0 = 1;
  s.Number =
    <Mul
      s.Number
      <Fact <Dec s.Number>>
    >;
}
```

Код на C++

```
// Automatically generated file. Don't edit!
#include "refalrts.h"

extern refalrts::FnResult WriteLine(refalrts::Iter
    arg_begin, refalrts::Iter arg_end);

extern refalrts::FnResult Dec(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

extern refalrts::FnResult Mul(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

static refalrts::FnResult Fact(refalrts::Iter arg_begin,
    refalrts::Iter arg_end);

refalrts::FnResult Go(refalrts::Iter arg_begin,
    refalrts::Iter arg_end) {
  Код предложения
  return refalrts::cRecognitionImpossible;
}

static refalrts::FnResult Fact(refalrts::Iter arg_begin,
    refalrts::Iter arg_end) {
  Код первого предложения
  Код второго предложения
  return refalrts::cRecognitionImpossible;
}
```

Рис. 1 – Генерация кода в простом РЕФАЛе

1.4 Постановка задачи

Задачей данного курсового проекта является разработка алгоритма оптимизации, устраняющего избыточность операции анализа аргумента функции для предложений, имеющих похожую структуру образцовых частей.

2 Алгоритм оптимизации

Одним из возможных методов удаления избыточных сопоставлений с образцом является обобщение левых частей подряд идущих предложений.

2.1 Жесткий образец

Введем некоторые определения. *Объектное выражение* (или, для краткости, просто «выражение») — это последовательность из термов, каждый из которых может быть либо атомарным, либо скобочным термом, т.е. выражением, заключенным в круглые скобки. *Атомарные термы* (атомы) — это термы, которые невозможно разбить на составные части путем сопоставления с образцом, обычно к ним относятся атомы-числа, атомы-символы, атомы-имена и т.д. Будем обозначать атомы как $X_1, X_2, \ldots, X_n$. Сами объектные выражения будем обозначать латинской буквой E, иногда с индексом.

Образцовое выражение (образец) — это синтаксическая конструкция, позволяющая описать некоторое множество объектных выражений. Оно записывается как объектное выражение, но некоторые части заменяются особыми подстановочными знаками — переменными. Переменная может заменять либо любой атом (так называемая s-переменная), либо любой терм (t-переменная), либо некоторый правильный (с правильной расстановкой скобок) фрагмент выражения (e-переменная). Переменные могут иметь имена для ссылок на них из результатных выражений, либо для указания того факта, что две одноименные переменные (они должны быть одного типа) заменяют одну и ту же часть объектного выражения.

Жесткий образец — выражение без открытых и повторных переменных. Таким образом жесткое выражение на каждом уровне скобок содержит не более чем одну e-переменную, а индексы всех переменных, входящих в одно и то же жесткое выражение, должны быть попарно различны. Таким образом, жесткие образцы имеют следующий вид:

Образец := Терм* [e-переменная Терм*].

Терм := Атом | t-переменная | s-переменная | (Образец).

Атом := X | s-переменная.

2.2 Уточнение образцов

Подстановка — замкнутая на множестве жестких образцов замена переменных в образце соответствующими образцами ($S = \{e_1 \to p_1, t_2 \to p_2, \ldots\}$ $P \xrightarrow{S} P'$). Один из возможных примеров подстановки (1).

$$\begin{aligned} P &= t_1 e_2 t_3 \\ S &= \{t_1 \to (e_4), e_2 \to s_5 e_6\} \\ P' &= (e_4) s_5 e_6 t_3 \end{aligned} \tag{1}$$

Если один образец P_1 можно получить из другого образца P_2 подстановкой, то будем говорить, что P_1 *уточняет* P_2, а P_2, соответственно, *обобщает* P_1. Введем следующие обозначения для операции уточнения образцов:

$P_1 \Rightarrow^+ P_2$ — P_1 уточняет P_2

$P_1 \Rightarrow^* P_2$ — нестрогое уточнение, $P_1 \Rightarrow^+ P_2$ или $P_1 = P_2$

Не трудно догадаться, что e обобщает любой образец: $\forall P e \Rightarrow^* P$.

Пусть $P_1 \Rightarrow^+ P_2$, и при этом не существует такого образца P_3, что $P_1 \Rightarrow^+ P_3 \Rightarrow^+ P_2$. Тогда P_1 будем называть *минимальным уточнением* P_2 (и, соответственно, P_2 — *минимальным обобщением* P_1) и обозначать как $P_1 \underset{\min}{\Rightarrow} P_2$.

Заметим, что $P_1 \underset{\min}{\Rightarrow} P_2$ тогда и только тогда, когда P_1 можно получить из P_2 только одной из следующих замен:

$$\begin{aligned} e &\to te \\ e &\to et \\ t &\to (e) \\ t &\to s \\ s &\to X \end{aligned} \tag{2}$$

Если $P \Rightarrow^+ Q$ и можно построить цепочку минимальных уточнений от Q до P двумя способами, т. е.

$$P \underset{\min}{\Rightarrow} R_1 \underset{\min}{\Rightarrow} R_2 \underset{\min}{\Rightarrow} \ldots \underset{\min}{\Rightarrow} R_n \underset{\min}{\Rightarrow} Q$$
$$P \underset{\min}{\Rightarrow} R'_1 \underset{\min}{\Rightarrow} R'_2 \underset{\min}{\Rightarrow} \ldots \underset{\min}{\Rightarrow} R'_m \underset{\min}{\Rightarrow} Q$$

то $n = m$.

Чтобы это показать, ведём числовую характеристику образца $C(P)$, которую определим как

$$C(P) = n_t + 2n_s + 3n_X + 3n_{()} - n_e + 1 \tag{3}$$

Здесь n_t, n_s, n_e — число, соответственно t-переменных, s-переменных, e-переменных, $n_{()}$ — число пар круглых скобок, n_X — число литералов атомов. Нетрудно убедиться, что для каждого преобразования (2) величина $C(P)$ возрастает на 1, следовательно, минимальное уточнение увеличивает данную величину на 1. Поскольку $C(P)$ и $C(Q)$ зависят только от внешнего вида образца, число элементарных уточнений между P и Q не будет зависеть от «траектории» перехода. Определим *сложность сопоставления* для образца P как число минимальных уточнений от e до P — $C(P)$. Заметим, что $C(e) = 0$.

2.3 Обобщение образцов

Рассмотрим множество образцов $P_1 \dots P_N$. Будем называть *глобальным сложнейшим обобщением (ГСО)* такое $P^* \Rightarrow^* P_i, i = 1 \dots N$, что $\nexists Q \Rightarrow^* P_i$, $C(Q) > C(P^*)$. Аналогичным образом определим *локально сложнейшее обобщение (ЛСО)* $P^* \Rightarrow^+ Q$. Заметим, что $\text{ГСО}(P_1 \dots P_N) \subseteq \text{ЛСО}(P_1 \dots P_N)$. Ниже (4), представлен пример ЛСО и ГСО дл двух образцов $P_1 = stt$ и $P_2 = st$.

$$\begin{aligned} &\text{ЛСО}(stt, st) = \{ste, set, ett\} \\ &\text{ГСО}(stt, st) = \{ste, set\} \\ &e_1 t_2 t_3 \xrightarrow{e_1 \to s} st_2 t_3 \\ &e_1 t_2 t_3 \xrightarrow{e_1 \to \varepsilon, t_2 \to s} st_3 \end{aligned} \tag{4}$$

Пусть $S_1 \dots S_N$ — подстановки переменных $v_1 \dots v_k$, $S_i = \{v_j \to P_{ij}\}$, где P — это некоторый образец с этими переменными. Обозначим за P_i подстановку S_i в образец P. Если $P \xrightarrow{S^*} P^*$, где $S^* = \{v_j \to P_j^*\}$, то $P^* \in \text{ЛСО}(P_1 \dots P_N)$ тогда и только тогда, когда $P_j^* \in \text{ЛСО}(P_{1j} \dots P_{Nj})$. Аналогично верно и для ГСО.

Определим *быстрое обобщение (БО)* для двух образцов P_1 и P_2 следующим образом:

1) если P_1 и P_2 являются термами, то БО определется согласно таблице 1.

2) Если P_1 и P_2 имеют следующий вид:

$$P_1 = L_1^1 \dots L_{M_1}^1 e R_{N_1}^1 \dots R_1^1$$

$$P_2 = L_1^2 \dots L_{M_2}^2 e R_{N_2}^2 \dots R_1^2$$

то $\text{БО}(P_1, P_2) = L_1^* \dots L_{min(M_1,M_2)}^* e R_{min(N_1,N_2)}^* \dots R_1^*$, где $L_i^* = \text{БО}(L_i^1, L_i^2)$, $R_i^* = \text{БО}(R_i^1, R_i^2)$.

Таблица 1: Правила быстрого обобщения для двух термов

P_1	P_2			
	x	s	(P_2')	t
x	x	s	t	t
y $\neq$ x	s	s	t	t
s	s	s	t	t
(P_1')	t	t	(БО(P_1', P_2'))	t
t	t	t	t	t

3) Если P_1 и P_2 имеют вид:

$$P_1 = T_1^1 \dots T_k^1$$

$$P_2 = T_1^2 \dots T_k^2$$

то БО$(P_1, P_2) = T_1^* \dots T_k^*$, где $T_i^* =$ БО(T_i^1, T_i^2).

4) Во всех остальных случаях БО$= e$.

Заметим следующие свойства быстрого обобщения образцов:

— сложность алгоритма $O(len(P_1) + len(P_2))$, где $len(P)$ — длина образца образца P

— БО(P_1, БО(P_2, P_3)) = БО(БО(P_1, P_2), P_3)

Определим БО$(P_1 \dots P_N)$ как БО(БО$(P_1 \dots P_{N-1})$, P_N).

БО$(P_1 \dots P_N) \Rightarrow^*$ ЛСО$(P_1 \dots P_N)$

2.4 Сложнейший жесткий образец

Обозначим $\mathbb{P}$ — множество всех образцов, $\mathbb{H}$ — множество жестких образцов. *Сложнейший жесткий образец (СЖО)* $P_H \in \mathbb{H}$ для образца $P \in \mathbb{P}$ — это такой образец, что $P \Rightarrow^* P_H$ и $\nexists P_H' \in \mathbb{H}$ $P \Rightarrow^* P_H'$, $C(P_H') > C(P_H)$. На листинге 1 представлен псевдокод алгоритма получения сложнейшего жесткого образца.

Алгоритм 1 Алгоритм получения СЖО

```
1:      ▷ S — множество подстановок
2:      ▷ next — максимальный индекс переменной в P + 1
3:      ▷ P_H — сложнейший жесткий образец
4: S ← ∅
5: next ← 1
6: P_H ← CHS(P)
7: function CHS(P)
8:     if P имеет вид (P') then
9:         return (CHS(P'))
10:    if P имеет вид v_i, где v — s или t then
11:        S ← {v_next → v_i} ∪ S
12:        next++
13:        return v_next
14:    if P имеет вид TP', где T — терм then
15:        return CHS(T) + CHS(P')
16:    if P имеет вид P'T, где T — терм then
17:        return CHS(P') + CHS(T)
18:    if P = ϵ then
19:        return ϵ
20:    S = S ∪ {e_next → P}
21:    next++
22:    return e_next
23: end function
```

2.5 Классы образцов

Образец вида $L_1 \dots L_N e R_M \dots R_1$, где L_i, R_j — термы, будем называть *образцом класса* (M, N).

Образец вида $T_1 \dots T_K$ будем называть *образцом класса* (K).

Пусть $P_1 \dots P_K$ — образцы класса (N_i, M_i), тогда ЛСО относится к классу $(min(N_i)$,

$min(M_i))$, $i = 1 \dots k$. Если же все образцы относятся к классу (K), то аналогично предыдущему случаю их ЛСО относится к классу $(j, min(K_i) - j), i = 1 \dots k$, $j = 1 \dots min(K_i)$.

Рассмотрим случаи, когда в исходном множестве образцов $P_1 \dots P_k$ встречаются образцы обоих классов. Пусть $K_{min} = min(K_i)$, где индекс i пробегает по всем образцам класса (K), а M_{min} и N_{min} — соответственно минимум среди всех M и N образцов класса (M, N). В таком случае возможны следующие варианты класса полученного ЛСО:

— $K_{min} > M_{min} + N_{min}$, ЛСО относится к классу (M_{min}, N_{min})

— $K_{min} < M_{min} + N_{min}$, и $K_{min} > M_{min}$, $K_{min} > N_{min}$, ЛСО относится к классам $(i, K_{min} - i)$, $i = K_{min} - N_{min} \dots M_{min}$

— $K_{min} < M_{min} + N_{min}$, и $K_{min} < M_{min}$, $K_{min} > N_{min}$, ЛСО относится к классам $(i, K_{min} - i)$, $i = K_{min} - N_{min} \dots K_{min}$

— $K_{min} < M_{min} + N_{min}$, и $K_{min} > M_{min}$, $K_{min} < N_{min}$, ЛСО относится к классам $(K_{min} - i, i)$, $i = K_{min} - M_{min} \dots K_{min}$

— $K_{min} < M_{min} + N_{min}$, и $K_{min} < M_{min}$, $K_{min} < N_{min}$, ЛСО относится к классам $(i, K_{min} - i)$, $i = 0 \dots K_{min}$

2.6 Алгоритм вычисления ГСО

Первый этап алгоритма вычисления глобального сложнейшего обобщения заключается в получении быстрого обобщения $P^* = \text{БО}(P_{H_1} \dots P_{H_n})$, $P^* \xrightarrow{S_i} P_{H_i}$, $S_i = \{v_j \to P_{ij} \mid j = 1 \dots k\}$, где $P_{H_1} \dots P_{H_n}$ — жесткие образцы.

Второй этап — разбор полученного образца P^*. Если в нем нет переменных с несколькими альтернативами, то P^* и есть ГСО $P_{H_1} \dots P_{H_n}$.

В противном случае переходим к рассмотрению всех переменных v_i образца P^* и их подстановок в $P_{H_1} \dots P_{H_n}$, $v_i \to P_{H_{i_1}} \dots P_{H_{i_n}}$. Для каждой подстановки производится вычисление классов, в которых происходит дальнейший поиск ГСО (пункт 2.5).

Следующий шаг — работа с вычисленными классами:

1. Наложение каждого образца на вычисленные классы.

   Наложение на класс (m, n) — это выбор первых m и последних n термов из исходного образца, а вся оставшаяся часть T заменяется подстановкой вида: $e \to T$.

   Наложение на класс (k) возможно только для образца класса (k), соответственно данное наложение возможно только если все образцы имеют класс (k).

Заметим, что алгоритм выбора класса (2.5) гарантирует, возможность наложения.

2. Для каждого терма вычислить ГСО рекурсивно.
3. Вычислить сложность полученного результата («частичного ГСО»).

   Сложность образца вычисляется по формуле 3.

Третий этап — выбор среди всех вычисленных «частичных ГСО» образца с максимальной сложностью. Он и будет являться результатом глобального сложнейшего обобщения. Для пояснения алгоритма, рассмотрим пример (5):

$$\begin{gathered}\text{Первый образец: (A e.1) (B e.2)}\\ \text{Второй образец: (B e.3)}\end{gathered} \tag{5}$$

Вычисляем быстрое обобщение.

$$\text{Быстрое обобщение: e.X} \rightarrow \text{[(A e.1) (B e.2) | B (e.3)]}$$

Поскольку в быстром обобщении получилась e переменная с альтернативой, то переходим к вычислению классов, которым будет принадлежать ЛСО.

Для образцов класса (1, 1) и (1) ЛСО будут принадлежать либо классу (1, 0), либо (0, 1)

Производим наложение на каждый из этих двух классов, после чего повторно вычисляем потермово ГСО и считаем сложность полученных образцов.

Класс (1, 0)

Наложения образцов будут иметь вид (A e.1) e.4 и (B e.3) e.5

Результат потермово вычисленного ГСО — (s.6 e.7) e.8

Сложность результата C((s.6 e.7) e.8) = 1 + 0 + 2 + 0 + 3 - 2 = 4

Класс (0, 1)

Наложения образцов будут иметь вид e.9 (B e.2) и e.10 (B e.3)

Результат потермово вычисленного ГСО — e.11 (B e.12)

Сложность результата C(e.11 (B e.12)) = 1 + 0 + 0 + 3 + 3 − 2 = 5

В итоге, из (s.6 e.7) e.8 и e.11 (B e.12) выбираем второй. ГСО((A e.1) (B e.2), (B e.3)) = e.11 (B e.12).

3 Реализация алгоритма

3.1 Внутренне представление компилируемой программы

Изначально алгоритм был протестирован на модели с указанным ниже представлением данных, а позже перенесен на модель данных компилятора. Различия между моделями не большие и не оказывают серьезного влияния на логику работы алгоритма.

3.1.1 Лексический анализ

Перед оптимизацией очередной функции, происходит построение лексической свертки образцовых частей предложений. Каждая лексема является скобочным термом, снабженным тегом и атрибутом (в зависимости от типа лексемы (6)).

```
e.Tokens ::= t.Token*
t.Token ::= (s.Tag e.Attr)                                    (6)
s.Tag ::= '(' | ')' | #S | #T | #E | #Atom
```

3.1.2 Синтаксический анализ

Синтаксический анализ строит синтаксическое дерево (7). Образец есть последовательность термов, терм — атом, переменная или скобки. Скобки — скобочный терм, внутри которого находится символ Brackets, и «тело» скобок, имеющего ту же структуру, что и для образца.

```
e.Pattern ::= t.Term*
t.Term ::= t.Atom | t.Variable | (#Brackets e.Pattern)
t.Atom ::= (#Atom s.Attr)                                     (7)
t.Variable ::= (s.VariableType e.Index)
s.VariableType ::= #S | #E | #T
```

Для того, чтобы хранить подстановки, переводящие жёсткие образцы в реальные, формат переменной был изменен следующим образом:

```
t.Variable ::= (s.VariableType (e.Index) (e.Pattern))
```

3.1.3 Пример синтаксического дерева

На рисунке 2 показано синтаксическое дерево, построенное для образца:

((a)) e.name1 ((a) t.name2)

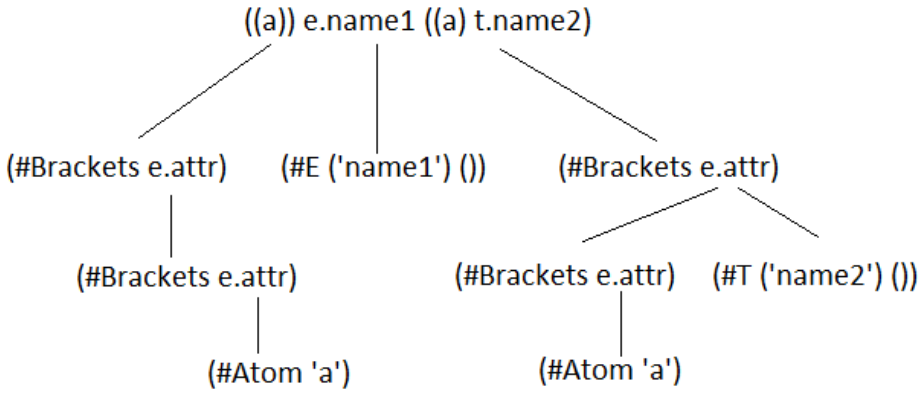

Рис. 2 – Пример синтаксического дерева

3.2 Обобщение образцов

3.2.1 Генерация жестких образцов

Алгоритм получения СЖО описан в пункте 2.4. Для того что бы не «протаскивать» между вызовами счётчик, инкрементируемый для каждой переменной, ее новое имя представляется уникальной последовательностью букв 'l', 'c', и 'r'. Функция генерации индексов принимает на вход образец, разбитый на лексемы и последним аргументом в структурных скобках — текущий индекс. Если образец можно разбить таким образом:

e.smth1 (e.smth2) e.smth3

то для каждой из частей происходит рекурсивный вызов функции генерации индексов со следующими добавлениями к текущему индексу

e.smth1 - текущий индекс + 'l' - спуск в левую часть

e.smth2 - текущий индекс + 'c' - спуск «по центру»

e.smth3 - текущий индекс + 'r' - спуск в правую часть

Если скобок нет, то отщипывается терм слева, а для оставшейся правой части опять происходит рекурсивный вызов, с добавлением буквы 'r' к текущему индексу. Если «оторванный» терм - переменная, то ее имя меняется на текущий индекс + 'l'.

В примере 8 показано, как будет выглядеть СЖО для образца с открытыми и повторными переменными.

$$\begin{array}{l} \text{Образец: s.1 X (t.1) e.1 () e.2 () e.1} \\ \text{Жесткий образец: s.Index\_l X (t.Index\_c) e.Index\_r} \\ \text{Подстановка для s.Index\_l} \rightarrow \text{[s.1]} \\ \text{Подстановка для t.Index\_c} \rightarrow \text{[t.1]} \\ \text{Подстановка для e.Index\_r} \rightarrow \text{[e.1 () e.2 () e.1]} \end{array} \tag{8}$$

3.2.2 Быстрое обобщение

Реализация быстрого обобщения, практически дословно повторяет описанный в пункте 2.3 алгоритм. На листинге 1 представлена основная часть алгоритма на РЕФАЛе.

На вход функция получает образцовые части предложений, которые «обернуты» в структурные скобки:

$$(e.Pattern1)(e.Pattern2)\ldots(e.PatternN)$$

Отдельно рассматриваются случаи, когда тело функции представленно лишь одним предложением, или состоит из пустых образцов.

Листинг 1: Функция получения быстрого обобщения

```
FastGen {
  /* 0. Тело функции состоит из одного предложения или из пустых образцов */
  (e.1) = (e.1);
  () () = ();

  /* 1. Предложения являются термами; троим с БО согласно таблице */
  (t.1) (t.2) e.Tail = <FastGen (<FastGen-Terms (t.1) (t.2)>) e.Tail>;

  /* 2. Предложения класса (M,N) */
  (t.left1 e.1 (#E (e.name1) (e.pattern1)) e.2 t.right1)
```

```
  (t.left2 e.3 (#E (e.name2) (e.pattern2)) e.4 t.right2)
  e.Tail =
    <FastGen
      (<FastGen-MeN
        (t.left1 e.1 (#E (e.name1) (e.pattern1)) e.2 t.right1)
        (t.left2 e.3 (#E (e.name2) (e.pattern2)) e.4 t.right2)
      >)
      e.Tail
    >;

  /* 3. Предложения класса (M) */
  (t.term1 e.1) (t.term2 e.2) e.Tail =
    <FastGen
      (<FastGen-M
        <Check (t.term1 e.1) (t.term2 e.2)>
        (t.term1 e.1) (t.term2 e.2)
      >)
      e.Tail
    >;

  /* 4. Все остальное */
  (e.1) (e.2) = ((#E ('$') (#Alt (e.1) (e.2))));
}
```

3.2.3 Глобальное сложнейшее обобщение

Синтаксис «Простого РЕФАЛа» позволяет использовать лямбда-функции, что значительно упрощает разработку кода. На листинге 2 представлен код функции генерации ГСО. На вход она принимает результат быстрого обобщения. Вычисленные классы ЛСО для каждой переменной по очереди передаются в лямбда-функцию, где происходит подсчет сложности каждого «частичного ГСО» (листинг 3) и выбор максимума.

Листинг 2: Функция получения глобального сложнейшего обобщения

```
GlobalGen {
  (e.FastGen) =
    <Map
      { (e.body) = <GetMax <CalcComplexity <GSOagain e.body>>>; }
      <CreateSuperposition
        (<GetAllVars e.FastGen>)
        (<CalcClasses <GetAllVars e.FastGen>>)
      >
    >;
}
```

Листинг 3: Функция подсчета сложности образцов

```
CalcComplexity-aux {
  s.comp (#S e.body) e.tail =
    <CalcComplexity-aux <Add s.comp 2> e.tail>;
  s.comp (#E e.body) e.tail =
    <CalcComplexity-aux <Sub s.comp 1> e.tail>;
  s.comp (#T e.body) e.tail =
    <CalcComplexity-aux <Add s.comp 1> e.tail>;
  s.comp (#Atom e.body) e.tail =
    <CalcComplexity-aux <Add s.comp 3> e.tail>;
  s.comp (#Brackets e.body) e.tail =
    <CalcComplexity-aux <Add s.comp 3> e.body e.tail>;
  s.comp = s.comp;
}
```

3.3 Генерация кода

Рассмотрим программу на листинге 4. Это вариант «калькулятора» для единичной системы счисления.

Листинг 4: Тестовая программа

```
$EXTERN WriteLine;

MySum {
  ( e.X ) ( '0' ) = e.X;
  ( '0' ) ( e.Y ) = e.Y;
  ( e.X ) ( e.Y ) = e.X e.Y;
}

$LABEL ERROR;

MySub {
  ( '1' e.X ) ( '1' e.Y ) = <MySub (e.X) (e.Y)>;
  ( '1' e.X ) ( ) = '1' e.X;
  ( '1' e.X ) ( '0' ) = '1' e.X;
  ( '0' ) ( '0' ) = '0';
  ( '0' ) ( e.X ) = #ERROR;
  ( ) ( e.X ) = #ERROR;
}

MyMul {
  ( '0' ) ( e.X ) = '0';
  ( e.X ) ( '0' ) = '0';
  ( e.X ) ( '1' e.Y ) = e.X <MyMul (e.X) ( e.Y )>;
  ( e.X ) ( ) = ;
}

Parse {
  e.a '+' e.b = <MySum (<Parse e.a>) (<Parse e.b>)>;
  e.a '-' e.b '-' e.c = <Parse <Parse e.a '-' e.b> '-' e.c>;
  e.a '-' e.b = <MySub (<Parse e.a>) (<Parse e.b>)>;
```

```
  e.a '*' e.b = <MyMul (<Parse e.a>) (<Parse e.b>)>;
  e.a = e.a;
}


$ENTRY Go {
    = <WriteLine <Parse <Spar '1+1*111-11'>>>;
}
```

В функциях *MySum*, *MySub* и *MyMul* хорошо видно общую структуру всех образцов. При совмещении глобального сложнейшего обобщения и быстрого обобщения, получится следующий образец:

(#Brackets (#E (3) ())) (#Brackets (#E (4) ()))

Для этого образца будет сгенерирован следующий код (листинг 5).

Листинг 5: Код общего образца

```
  //Common Pattern (FastGen & GlobalGen)
    refalrts::Iter bb_0 = arg_begin;
    refalrts::Iter be_0 = arg_end;
    refalrts::move_left( bb_0, be_0 );
    refalrts::move_left( bb_0, be_0 );
    refalrts::move_right( bb_0, be_0 );
    // (~1 e.3 )~1 (~2 e.4 )~2
    refalrts::Iter bb_1 = 0;
    refalrts::Iter be_1 = 0;
    if( ! refalrts::brackets_left( bb_1, be_1, bb_0, be_0 ) )
      return refalrts::cRecognitionImpossible;
    refalrts::Iter bb_2 = 0;
    refalrts::Iter be_2 = 0;
    if( ! refalrts::brackets_left( bb_2, be_2, bb_0, be_0 ) )
      return refalrts::cRecognitionImpossible;
    if( ! empty_seq( bb_0, be_0 ) )
```

```
    return refalrts::cRecognitionImpossible;
  refalrts::Iter copy_bb_3 = bb_1 ;
  refalrts::Iter copy_be_3 = be_1 ;
  refalrts::Iter copy_bb_4 = bb_2 ;
  refalrts::Iter copy_be_4 = be_2 ;
```

Повторный разбор скобок для каждого предложения больше производиться не будет. Результат разбора сохранен в переменные *copy_b(b|e)_3* и *copy_b(b|e)_4*. Какое из предложений будет выполнено — определится в результате сопоставления значений образцов в скобках с сохраненными переменными.

3.4 Тестирование

Ощутимый прирост производительности достигается у программ имеющих «сильно» похожие предложения функций. Чем сложнее полученное обобщение образцовых частей, и чем больше предложений, тем сильнее заметно увеличение производительности программы.

Средний прирост скорости работы программы составляет 5%. Необходимо отметить, что для некоторых программ оптимизация может вообще не увеличить производительность. Однако, если программа описывает циклические процессы, можно получить прирост скорости на 10% и даже больше.

ЗАКЛЮЧЕНИЕ

В данной работе рассматривалась возможность улучшения производительности программ написанных на функциональном языке программирования РЕФАЛ. Оптимизация выполнена для компилятора «Простого РЕФАЛа». Необходимость оптимизации обусловлена тем, что генерация кода предложений производится независимо, что сильно снижает потенциальную производительность для программ, имеющих похожие образцы функций.

Курсовая работа выполнена успешно. Получилось добиться увеличения производительности программ за счет оптимизации операций сопоставления с образцом. Средний прирост скорости работы программы составляет 5%

Данную оптимизацию можно улучшить еще сильнее, за счет добавления ветвлений.

Список литературы

[1] РЕФАЛ [электронная публикация]. — URL: http://refal.net/index.html.

[2] История рефал-компилятора [электронная публикация]. — URL: http://refal.net/romanenko/TheHistoryOfRefalCompiler/index.htm.

[3] Сопоставление с образцом [электронная публикация]. — URL: http://refal.net/rf5_frm.htm.