Федеральное государственное бюджетное 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 выпускной квалификационной работе

Улучшенная оптимизация совместного сопоставления с образцом в компиляторе Рефала-5-лямбда

Научный руководитель: _____ (А. В. Коновалов)
(подпись, дата)

Студент группы ИУ9-82: _____ (П. А. Савельев)
(подпись, дата)

Москва, 2018

АННОТАЦИЯ

В данной работе рассматриваются различные подходы и стратегии по оптимизации образцовых выражений в функциональном языке программирования Рефал-5λ. Основная задача – увеличение производительности программ порождаемых компилятором. В работе описывается текущий подход, рассматриваются его недостатки и возможные способы упрощения и модификации. Приводится сравнение производительности итоговой реализации с оригинальным методом оптимизации.

Содержание

ВВЕДЕНИЕ **3**

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ **5**

1.1 Общие понятия 7

1.2 Стратегии обобщения 11

1.2.1 Построение дерева 11

1.2.2 Анализ внешнего вида образцов 15

1.2.3 Построение ГСО и группировка 21

1.3 Генерация команд сопоставления с образцом 21

2 РАЗРАБОТКА **23**

2.1 Требования к оптимизации 23

2.2 Обобщение классов $c(m, n)$ и $c(k)$ 24

2.3 Улучшенный алгоритм вычисления ГСО 25

2.3.1 Построение жестких выражений 25

2.3.2 Построение изображений 27

2.3.3 Многорезультатное обобщение 28

2.3.4 Коллапс 29

2.3.5 Функция обобщенного сопоставления с образцом . . . 32

2.4 Совместное сопоставление с образцом 35

2.4.1 Интеграция улучшенного алгоритма вычисления ГСО 35

2.4.2 Формирование групп предложений 36

2.4.3 Рекурсивное разбиение предложений на группы . . . 39

2.5 Оптимизация программы 41

3 ТЕСТИРОВАНИЕ **43**

3.1 Оценка оптимизации 46

ЗАКЛЮЧЕНИЕ **48**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49

ПРИЛОЖЕНИЕ 51

ВВЕДЕНИЕ

Представленная работа ставит своей целью исследование и оптимизацию процесса кодогенерации команд совместного сопоставления с образцом в языке Рефал-5λ. В работе происходит рассмотрение и анализ существующих подходов. Формулируются требования к алгоритму, разрешающему недостатки рассмотренных стратегий построения команд совместного сопоставления с образцом. Описывается процесс и результат разработки алгоритма. Оптимизируется код функций, выполняющих разбиение предложений на группы и обобщение подстановок. Производится тестирование полученной оптимизации и сравнение с предыдущей оптимизацией, а так же на примере демонстрируется корректность работы программы.

В разделе 1 выполняется обзор предметной области и формулируется задача дипломной работы. Происходит анализ имеющихся алгоритмов оптимизации. Рассматриваются существующие стратегии обобщения команд сопоставления с образцом, описываются алгоритмы стратегий, а также формулируются их недостатки. Предлагается способ развития реализованного на данный момент подхода с целью упрощения и увеличения его производительности.

В разделе 2 обозначаются требования к новому алгоритму совместного сопоставления с образцом. Описывается процесс разработки и конечная реализация алгоритма построения жесткого выражения, алгоритма обобщенного сопоставления и алгоритма построения ГСО (глобального сложнейшего обобщения). Описываются временные характеристики улучшенного алгоритма ГСО, а так же алгоритм группировки предложений.

Так же в этом разделе описывается проведенная оптимизация, которая повысила производительность итоговой программы и упростила интерфейс реализованных функций.

В разделе 3 оценивается результат оптимизации. Происходит сравнение результата оптимизации с предыдущей реализацией оптимизации совмест-

ного сопоставления с образцом на примере различных машин и операционных систем и различных этапов работы компилятора. Также рассматривается кодогенерация с оптимизацией на примере функции из исходного текста компилятора.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

РЕФАЛ — РЕкурсивных Функций АЛгоритмический язык программирования, ориентированный на обработку символьных данных. Он соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ.[2] Основной структурой данных языка является «объектное выражение», представляющее собой двунаправленный список. [1]

Язык является точным надмножество Рефал-5. Любая программа, написанная на Рефал-5, должна выполняться точно так же и на Рефал-5λ. Основной особенностью этого диалекта является поддержка функций высшего порядка.

Функция на Рефал-5λ состоит из набора предложений. Каждое предложение разделяется на образцовую и результатную части. При вызове функции ее аргумент поочередно сопоставляется с образцом в левой части предложения. Управление передается на правую результатную часть первого предложения, для которого сопоставление было успешным. Если таких предложений не нашлось, то программа завершится с ошибкой.

При такой реализации каждое предложение компилируется в независимый фрагмент кода. Анализ каждого образца также происходит независимо. В случае, если предложения имеют схожую структуру, операции анализа левой части выполняются по несколько раз, что приводит к серьезной потере в производительности.

Есть ряд причин, почему многие функции имеют похожую структуру предложений. Функции в языке могут принимать лишь одно объектное выражение в качестве аргумента. Если требуется передать несколько значений в качестве аргумента, формируется единый объект. Получается, что каждый образец будет состоять из частей, каждая из которых уточняет общий формат аргумента.

К примеру, функция совместного сопоставления с образцом имеет сле-

дующего вида:

Листинг 1: Пример формирования единого объекта в качестве аргумента

```
HighLevelRASL-Function-Conjoint {
  s.FnGenSubst s.FnGenResult
  s.ScopeClass (e.Name) e.Sentences
    = ...;
}
```

Здесь для успешного сопоставления с образцом функции требуется передать в качестве аргумента объект, содержащий названия функций, которые будут использоваться для генерации команд, а так же сами предложения. В данном случае можно заметить, что часть значений, которые передаются в аргумент функции, были взяты в скобки. Это сделано намеренно, чтобы иметь возможность «расклеить» переданные значения в образце.

Другой причиной является текущая реализация замыканий. Каждая вложенная функция преобразуется в некоторую глобальную функцию. Для того, чтобы передать контекст функции, в начало каждой из образцовых частей добавляются переменные, которые соответствуют переменным контекста. Например, замыкание вида:

$$\begin{aligned} &\{ \\ &\ 1 \quad = \quad e.X; \\ &\ 2 \quad = \quad e.Y; \\ &\} \end{aligned}$$

содержит переменные e.X и e.Y в правой части предложения. Их значения определяются из контекста и глобальная функция будет иметь следующий

вид:

$$
\begin{array}{l}
\{ \\
\quad (e.X)(e.Y) \quad 1 \quad = \quad e.X; \\
\quad (e.X)(e.Y) \quad 2 \quad = \quad e.Y; \\
\}
\end{array}
$$

На данный момент уже была произведена определенная работа в направлении поиска общей последовательности команд сопоставления с образцом. В результате, сначала анализируется общая часть всех предложений, а уже затем особенности каждого предложения. При таком подходе, если одно из предложений неудачно сопоставится с аргументом функции, повторный анализ общей части производиться не будет.

Задачей данной работы является изменение процесса кодогенерации таким образом, чтобы выделять группы образцов, обладающих более сложной общей частью, по сравнению с общей частью всех предложений.

Введем понятия, необходимые для дальнейшего описания алгоритмов.

1.1 Общие понятия

Переменные или свободные переменные в языке Рефал-5λ делятся на 3 типа в зависимости от их возможных значений. Каждая переменная имеет следующий формат: 'Тип.Индекс', где Индекс означает имя, а Тип — это один из символов s, t или e. К s-переменным относятся литеры, числа, имена функций, идентификаторы, которые принято называть словом «символ». Значением t-переменных (термов) может быть s-переменная или выражение в круглых скобках. e-переменная может быть отождествлена с любым выражением, в том числе с пустой последовательностью. Следующее выражение представляет собой пример различных переменных:

'w' 1 Ident s.Symb (Value s.Value) t.Term e.variable.

Переменные, которые встречаются в одном выражении несколько раз будем называть повторными. Объектным выражением является последовательность символов и абстрактных или структурных скобок.

Функции — это последовательность предложений, которые имеют вид «образец = результат;». Образец — объектное выражение, которое может содержать свободные переменные, а результатное выражение дополнительно может содержать скобки вызова функций. Анализ данных в языке происходит за счет поочередного сравнения образцов и аргумента функции. При этом проверяется возможно ли отождествить переменные образца с объектным выражением. Если такое отождествление возможно, то выполняется результатная часть предложения. Такой процесс называется сопоставлением с образцом.

Жесткий образец — это образец, который отождествляется однозначно. Такой образец может иметь повторные переменные, но не имеет открытых. Жесткое выражение — выражение, которое на каждом уровне вложенности скобок содержит не более одной открытой e-переменной и не содержит повторных переменных. [3]

В дальнейшем оптимизации рассматриваются лишь для образцов, являющихся жесткими выражениями.

Подстановка — набор замен переменных в образце на некие образцы,

$$\sigma = v_i \to P_i, \quad i = 1...N.$$

При этом переменные должны соответствовать типу образца. Запись $\sigma(P_1) = P_2$ означает, что образец P_2 получен из образца P_1 подстановкой σ.

Уточнение или сужение образца — это операция замены переменных в образце. Образец P_2 уточняет образец P_1 , если существует подстановка σ, которая переводит P_1 в P_2: $P_1 \Rightarrow^+ P_2$. Такое уточнение называется «стро-

гим». Уточнение $P_1 \Rightarrow^* P_2$, когда образец может уточняться в себя, будем называть «нестрогим».

Обобщение образца — операция, обратная к сужению. Аналогично, обобщение может быть строгим или нестрогим.

Если при уточнении $P_1 \Rightarrow^+ P_3$ оказалось, что не существует такого образца P_2, что $P_1 \Rightarrow^+ P_2 \Rightarrow^+ P_3$, то такое уточнение назовем минимальным $P_1 \underset{min}{\Longrightarrow} P_3$. Аналогично определяется минимальное обобщение.

Теорема 1. *Минимальное уточнение возможно лишь следующим набором подстановок:*

$$e \to \epsilon; \tag{1}$$

$$e \to te; \tag{2}$$

$$e \to et; \tag{3}$$

$$t \to (e); \tag{4}$$

$$t \to s; \tag{5}$$

$$s \to X, \tag{6}$$

где X — «символ», а ϵ — пустое выражение.

Доказательство. Подстановки (1), (4), (5), (6) являются минимальными, так как представляют собой непосредственно определение e-, t- и s- переменных. e-переменная по определению может представлять собой последовательность термов, а t-переменная - терм. Тогда присоединение t-переменной к e-переменной приведет к e-переменной. То есть такая подстановка минимальна.

Докажем, что других подстановок нет. Для s-переменной других подстановок нет по определению. Для t переменной существует подстановка $t \to X$, но она не будет минимальной, так является сочетанием подстановок (5) и (6). Аналогично для e-переменной подстановки, переводящие в s-

переменную или «символ» не будут минимальными. Подстановка нескольких термов вместо e-переменной не будет минимальной подстановкой, так как такая операция будет композицией подстановок (2) и (3). □

Сложность жесткого выражения — числовая характеристика, определяемая по формуле

$$C(P) = n_t + 2 \times n_s + 3 \times n_X + 3 \times n_() - n_e + 1,$$

где n_t, n_s, n_e – количество, соответственно, t-, s-, e-переменных, $n_()$ – количество пар скобок, n_X — количество «символов». При этом, можно заметить, что каждая минимальная подстановка увеличивает сложность выражения на 1. Таким образом, сложность жесткого выражения — это количество минимальных уточнений, переводящих выражение «e» в некоторое конкретное выражение.

Теорема 2. *Если жесткое выражение P уточняется в выражение Q и существует 2 цепочки минимальных уточнений длиной M и N, таких что:*

$$P \underset{min}{\Longrightarrow} R_1^1 \underset{min}{\Longrightarrow} ... \underset{min}{\Longrightarrow} R_N^1 \underset{min}{\Longrightarrow} Q,$$
$$P \underset{min}{\Longrightarrow} R_1^2 \underset{min}{\Longrightarrow} ... \underset{min}{\Longrightarrow} R_M^2 \underset{min}{\Longrightarrow} Q,$$

то $M = N$.

Доказательство. Пусть сложность выражений P и Q — C(P) и C(Q) соответственно. Каждая минимальная подстановка увеличивает сложность выражения на 1. Тогда верны следующие равенства:

$$C(Q) = C(P) + N,$$
$$C(Q) = C(P) + M.$$

Значит N = M. □

Следует отметить, что из равенства длин цепочек не следует их эквивалентность. Например, цепочки минимальных уточнений:

$$e \to te \to tte \to tse \to ts,$$
$$e \to et \to es \to ets \to ts.$$

имеют одинаковую длину, но различный набор подстановок.

1.2 Стратегии обобщения

Рассмотрим существующие способы обобщения образцов. Опишем их плюсы и минусы.

1.2.1 Построение дерева

Вариант данной оптимизации был рассмотрен в диссертации С. А. Романенко «Машинно-независимый компилятор с языка рекурсивных функций». [4] Для функции, состоящей из N предложений, каждое предложение независимо переводится на язык сборки, схожий с языком RASL, который используется в компиляторе Рефал-5λ. Через a_{ij} обозначается j-тый оператор языка сборки в переводе i-го предложения. Затем строится упорядоченное дерево, каждая дуга которого помечена a_{ij} (Рис. 1).

Такое дерево иллюстрирует то, каким образом происходит передача управления при работе операторов отождествления. Если сопоставление произошло успешно, то выполняется переход по дуге до ближайшего узла, после чего рассматривается переход по самой левой дуге, выходящей из этого узла. В случае неудачи осуществляется переход против направления дуги до тех пор, пока не будет найдена вершина, у которой можно рассмотреть следующую дугу за той, по которой произошел спуск.

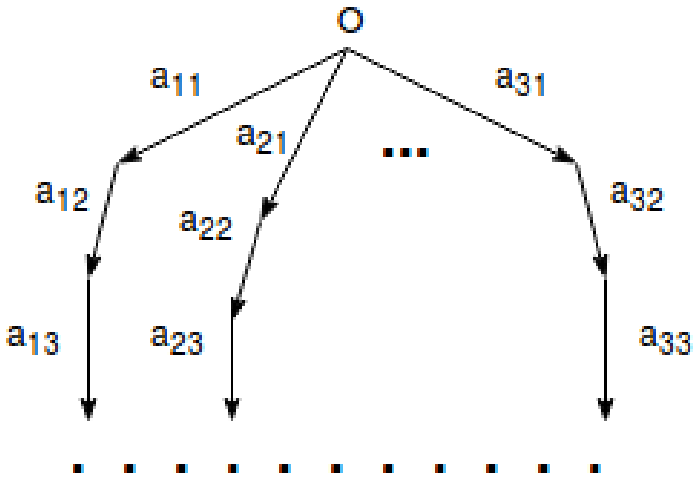

Рис. 1: Представление функции на языке сборки в виде дерева

Процесс оптимизации рассматривается на примере двух дуг, выходящих из одной вершины и помеченных одним и тем же оператором. Тогда 2 узла, в которые направлены эти дуги, сливаются в один. На Рис. 2 «X_1» и «X_2» — это поддеревья, подвешенные к вершинам B_1 и B_2 соответственно.

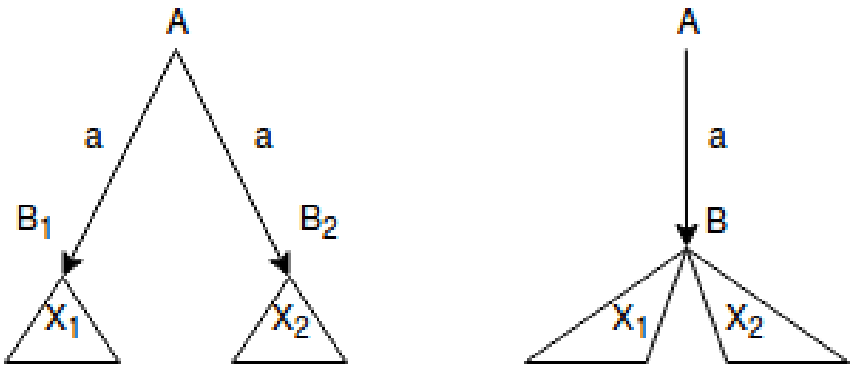

Рис. 2: Объединение совпадающих операторов языка сборки

Рассмотрим на примере, каким образом будет строиться дерево команд сопоставления в языке Рефал-5λ. Пусть имеются 2 образца:

$$(e.X \quad s.1 \quad e.Y) \quad s.1 \quad e.Z \quad = \quad < F \quad (e.X \quad s.1 \quad e.Y) \quad e.Z >; \quad (7)$$

$$(e.X \quad) \quad s.1 \quad e.Z \quad = \quad < F \quad (e.X \quad s.1) \quad e.Z >; \quad (8)$$

$$(e.X \quad) \quad = \quad e.X; \quad (9)$$

Тогда для образца предложения (7) будет построен следующий набор команд сопоставления:

$$\begin{array}{rl} & e.0 \rightarrow (e.1) \quad e.0 \\ & e.0 \rightarrow s.2 \quad e.0 \\ Cycle & e.1 \rightarrow e.1 \quad s.2 \quad e.1 \end{array}$$

Для образца предложения (8) набор команд будет похожим:

$$\begin{array}{l} e.0 \rightarrow (e.1) \quad e.0 \\ e.0 \rightarrow s.2 \quad e.0 \end{array}$$

И для образца предложения (9) набор команд будет совпадать лишь по одной подстановке:

$$\begin{array}{l} e.0 \rightarrow (e.1) \quad e.0 \\ e.0 \rightarrow \epsilon \end{array}$$

При построении дерева команд будет выделена общая часть для всех образцов в виде подстановки $e.0 \rightarrow (e.1) \quad e.0$. Затем для первых двух образцов дополнительно будет выделена общая часть $e.0 \rightarrow s.2 \quad e.0$. Итоговый вид дерева команд сопоставления представлен на Рис. 3.

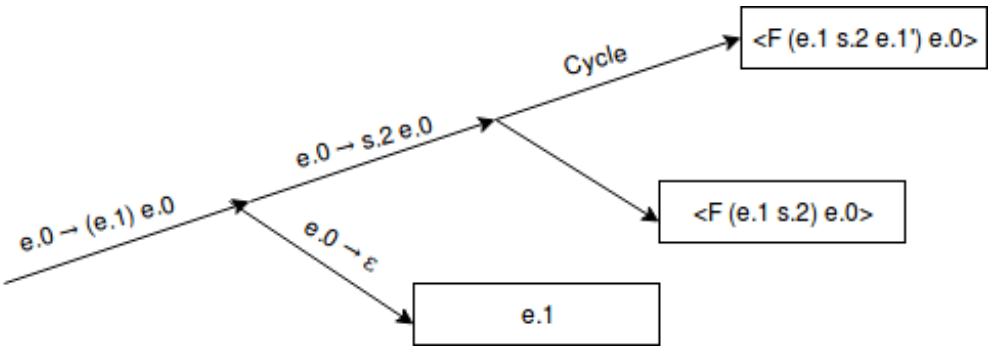

Рис. 3: Общий вид дерева после оптимизации

Однако, из-за двунаправленности образцов и из-за того, что изначально каждое предложение компилируется независимо, а уже затем строится дерево команд, данный подход не дает оптимальное построение для некоторых случаев. Рассмотрим недостаток на примере. Пусть имеется 2 образца:

$$\begin{array}{cccc} & X & Y & Z = ... \\ e.A & X & Y & Z = ... \end{array}$$

Тогда для первого предложения будет построен примерно следующий алгоритм в терминах элементарных сопоставлений:

$$\begin{array}{ll} e.0 \to X & e.0 \\ e.0 \to Y & e.0 \\ e.0 \to Z & e.0 \\ e.0 \to \epsilon & \end{array}$$

Для второго предложения набор команд сопоставления будет следующим:

$$e.0 \to e.0 \quad Z$$
$$e.0 \to e.0 \quad Y$$
$$e.0 \to e.0 \quad X$$
$$e.0 \to e.A$$

Общих команд здесь не будет, хотя сходство образцов очевидно. К примеру, если бы переменная $e.A$ находилась не слева, а справа, то обобщение было бы выполнено идеально. К плюсам данного метода все же можно отнести, то, что он позволяет «ветвить» последовательности команд сопоставления для многих схожих образцов.

1.2.2 Анализ внешнего вида образцов

Определяется образец общего вида, анализируя внешний вид отдельных образцов. К этому способу можно отнести стратегию, примененную в суперкомпиляторе SCP4 и стратегию построения ГСО, реализованную ранее в Рефале-5λ.

«Стратегия, примененная в суперкомпиляторе SCP4».

1. на каждом скобочном уровне выполняется обобщение терм за термом;
2. выбираются самые левые и самые правые термы;
3. обобщаются те из них, обобщение которых приведет к наименьшей потере информации.

Данный подход описан в книге А. П. Немытых «Суперкомпилятор SCP4: Общая структура». [8] Рассмотрим данную стратегию на следующем примере:

$$A \quad B \quad A = ...$$
$$A \quad B \quad X \quad Y \quad A \quad B \quad A = ...$$

Согласно алгоритму, на каждой итерации будем рассматривать пары левых и правых термов. На первой итерации это будут совпадающие пары термов «A» и «A» слева и справа. Так как обобщение в обоих случаях не приведет к потере информации, будет выбран вариант обобщения слева, как вариант по умолчанию. Следующая итерация — сравниваем «B» и «B» слева и «A» и «A» справа. Ситуация аналогична предыдущей, выбираем пару левых термов. На следующем шаге сравниваем обобщение термов «A» и «X» до s-переменной слева и «A» и «A» до «A» справа. Очевидно, что обобщение слева приведет к большей потере информации, поэтому обобщаем термы справа.

Итоговым обобщением оказывается выражение «$AB \quad e.1 \quad A$» и подстановки $\{e.1 \to \epsilon,\ e.1 \to XYAB\}$.

Из-за того, что в процессе работы алгоритма происходит построение ЛСО (см. далее), у данного подхода также существует недостаток, который может привести к неоптимальному обобщению. Рассмотрим следующий пример:

$$\begin{array}{llll} Abc & ((t.1) \quad e.2 \quad s.3) & (t.4 \quad e.5) & = ... \\ (e.X) & ((t.1) \quad e.2 \quad s.3) & & = ... \end{array}$$

Оба приведенных образца обладают одинаковой и достаточно сложной общей частью $((t.1) \quad e.2 \quad s.3)$. Однако из-за способа, которым выбираются термы, сопоставление осуществится следующим образом:

$$
\begin{array}{lllllll}
Abc & ((t.1) & e.2 & s.3 &) & (t.4 \quad e.5) & = ... \\
 & (\quad e.X \quad) & & & & ((t.1) \quad e.2 \quad s.3) & = ...
\end{array}
$$

В данном случае будет получено обобщение вида

$$e \quad (e) \quad (t \quad e).$$

В случае, если бы обобщение происходило согласно «первоначальному выравниванию», было бы получено обобщение

$$t \quad ((t) \quad e \quad s) \quad e.$$

Сложность первого выражения равна 5, а второго — 9. Получается, что сложность второго выражения выше. Таким образом, применение данного алгоритма для оптимизации будет приводить к тому, что в общую часть будет выноситься меньшее количество элементарных команд, а значит будут выполняться лишние сопоставления.

Очевидно, что обобщение, построенное таким образом не будет оптимальным.

«Построение ГСО». Суть стратегии заключается в вычислении обобщения для всех образцов. Так как данный подход к обобщению образцов реализован на данный момент в рамках ВКР бакалавра= И. Скрыпникова, опишем его подробно. [5] Для начала введем понятия локального и глобального сложнейшего обобщения.

Локальным сложнейшим обобщением (далее ЛСО) образцов $P_1...P_N$ является образец P', который обобщает каждый образец из набора $P_1...P_N$, но при этом не существует образца Q, который обобщает образцы $P_1...P_N$ и уточняет образец P'.

Глобальным сложнейшим обобщением (далее ЛСО) образцов $P_1...P_N$ является образец P', который обобщает каждый образец из набора $P_1...P_N$,

но при этом не существует образца Q, который обобщает образцы $P_1 ... P_N$ и сложность Q больше, чем сложность P': $C(Q) > C(P')$.

Из определений следует, что ГСО является подмножеством ЛСО.

Определим представление образцов при помощи синтаксических классов $c(m, n)$ и $c(k)$.

Будем говорить, что образец относится к синтаксическому классу $c(k)$, если он имеет вид $P = T_1 \dots T_n$, где T_i — терм. Образец относится к синтаксическому классу $c(m, n)$, если он имеет вид $P = L_1 \dots L_n e R_m \dots R_1$, где L_i и R_i — термы, а e — е-переменная. Так как е-переменная может быть пустой последовательностью, то возможно представление синтаксического класса $c(k)$ через $c(m, n)$. В таком случае $c(k)$ сужается до следующих классов:

$$c(0, k), c(1, k-1), \dots, c(k-1, 1), c(k, 0).$$

Здесь и далее для краткости слово «синтаксические» при упоминании классов будем опускать.

Быстрое обобщение (далее БО) двух образцов P_1 и P_2 — это образец, построенный по следующим правилам:

- если образцы являются термами, то обобщение строится согласно Таблице 1;
- если образцы принадлежат классу $c(k)$, то БО образцов также будет принадлежать классу $c(k)$, где внутренние термы обобщаются попарно через БО;
- если образцы принадлежат классу $c(m, n)$, то БО образцов также будет принадлежать классу $c(m, n)$; для левых и правых частей попарно вычисляется БО, при этом длина полученных выражений — минимум среди длин соответствующих выражений.

Таблица 1: Построение быстрого обобщения для двух термов

P_1/P_2	X	$Y \neq X$	s	t	P_1^*
X	X	s	s	t	t
s	s	s	s	t	t
t	t	t	t	t	t
P_2^*	t	t	t	t	(БО(P_1, P_2))

- иначе обобщение равно ϵ.

Для набора образцов $P_1...P_N$ выполняется рекурсия:

$$\text{БО}(P_1...P_N) = \text{БО}(\text{БО}(P_1...P_{N-1})P_N).$$

Основой алгоритма вычисления ГСО является следующая теорема:

Теорема 3. *Для набора образцов $P_1...P_N$ верно:*

$$БО(P_1...P_N) \Rightarrow^* ГСО(P_1...P_N).$$

Доказательство. Данное утверждение приводится без доказательства. □

Следствием Теоремы 3 является тот факт, что быстрое обобщение может быть использовано для упрощения построения глобального сложнейшего обобщения

Приведем непосредственно сам алгоритм глобального сложнейшего обобщения. На первом этапе для набора образцов вычисляются жесткие выражения. Затем для полученного набора жестких выражений строится быстрое обобщение и вычисляются подстановки, переводящие обобщение в исходные образцы. Если в полученном выражении нет е-переменных с несколькими альтернативами, то обобщение является ГСО. Иначе вычисляются классы дальнейшего поиска ГСО. Для каждого из допустимых классов ГСО: каждый образец накладывается на текущий класс ($c(m, n)$

или $c(k)$), затем для полученных образцов рекурсивно вычисляется потермовое ГСО — частичное ГСО. Это становится возможным благодаря тому, то все наложенные образцы принадлежат одному классу.

Среди всех найденных частичных ГСО находим сложнейшее по введенной ранее метрике сложности. После выполнения этих действий для всех подстановок e-переменных получим ГСО для начального набора образцов.

Рассмотрим работу алгоритма на следующем примере:

$$\begin{array}{llll} X & ((e.1) \quad A \quad e.2 \quad (B) &) \quad Y & = ... \\ X & (A \quad (C) &) \quad Z & = ... \end{array}$$

Быстрым обобщением этих образцов будет выражение «$X \quad (e.3) \quad s.1$», а подстановками $\sigma_1 = \{e.3 \to (e.1) \quad A \quad e.2 \quad (B)\}$ и $\sigma_2 = \{e.3 \to A \quad (C)\}$. Вычислим допустимые классы частичного ГСО для переменной $e.3$ с несколькими альтернативами. Для первой и второй подстановки получим классы $c(2,1)$ и $c(2)$ соответственно. Тогда допустимыми классами становятся классы $c(1,1)$ и $c(2,0)$. После наложения каждого из классов на образцы, вычислим частичное ГСО: для класса $c(1,1)$ — это выражение «$t.1 \quad e.6 \quad (B)$» со сложностью 7 и для класса $c(2,0)$ — это выражение «$t.1 \quad t.2 \quad e.6$» со сложностью равной 2.

Получается, что ГСО для данных образцов будет выражение

$$X(t.1 \quad e.6 \quad (B)) \quad s.1.$$

К недостаткам данной стратегии можно отнести сложность реализации алгоритма, а также то, что обобщение вычисляется для образцов всех предложений. Если же среди образцов найдется один, с отличающейся от других образцов структурой, то полученное обобщение не будет оптимальным.

Приведу пример. Следующие предложения содержат образец, тело которого представляет собой пустую строку. Подобные образцы достаточно

часто встречаются в программах на Рефал-5λ.

$$
\begin{aligned}
&'A' \quad e.X =' B' \quad &< Fab \quad e.X >;\\
&s.Z \quad e.X = s.Z \quad &< Fab \quad e.X >;\\
&/^*empty_*/ = /^*empty_*/;
\end{aligned}
$$

Тогда ГСО для данного набора образцов будет выражение «$e.0$». Очевидно, если бы при вычислении обобщения можно было разделить предложения на 2 группы (1 и 2 — в первой группе, а 3 — во второй), то полученные обобщения были гораздо точнее.

1.2.3 Построение ГСО и группировка

Как было сказано ранее, стратегию вычисления ГСО можно улучшить, если вычислять обобщение образцов не для всех предложений, а для некоторых наборов предложений. Затем, можно произвести повторную группировку для предложений в выделенных группах. Таким образом можно будет избавиться от недостатка предыдущего подхода.

Последняя стратегия является прямой оптимизацией реализованной на текущий момент стратегии «Построение ГСО». Определим ожидаемый результат оптимизации по сравнению со стратегией «Построение ГСО».

1.3 Генерация команд сопоставления с образцом

Пусть задана функция Fab следующим образом:

$$
\begin{aligned}
&'A' \quad e.X =' B' \quad &< Fab \quad e.X >;\\
&s.Z \quad e.X = s.Z \quad &< Fab \quad e.X >;\\
&/^*empty_*/ = /^*empty_*/;
\end{aligned}
$$

Она выполняет поиск и замену символов 'A' на 'B'. Генерация команд для образцов данной функции с оптимизацией без группировки предложений будет выглядеть так, как представлено на Листинге 2.

Листинг 2: Генерация команд для функции Fab без группировки

```
//  < Fab  e.idx  > - общая часть для всех предложений
...
(CmdSentence
  //  <Fab  'A' e.X > - первое предложение
  ...
)
(CmdSentence
  //  <Fab s.Z  e.X  > - второе предложение
  ...
)
//  <Fab  > - третье предложение
...
```

Как можно увидеть, общая часть для всех образцов - e-переменная. Каждое следующее предложение производит подстановку всего образца вместо этой переменной. Последнее предложение подставляет пустую строку. Рассмотрим, как могла бы выглядеть кодогенерация с дополнительной группировкой предложений.

Листинг 3: Генерация команд для функции Fab с группировкой

```
//  < Fab  e.new00  > - общая часть для всех предложений
...
(CmdSentence
  //  <Fab  s.new10  e.new11 > - общая часть первой группы
  ...
```

```
  (CmdSentence
    //  <Fab  'A' e.X > - первое предложение
    ...
  )
  (CmdSentence
    //  <Fab  s.Z e.X > - второе предложение
    ...
  )
)
//  <Fab  > - третье предложение
...
```

На Листинге 3 представлено выделение группы из первых двух предложений. Для более сложных примеров таких групп может быть больше и они могут быть вложены друг в друга.

2 РАЗРАБОТКА

2.1 Требования к оптимизации

В описанном ранее алгоритме вычисления ГСО без группировки одновременно с вычислением ГСО отслеживались и подстановки, переводящие ГСО в каждое из предложений, что существенно усложняло логику алгоритма.

Улучшенный алгоритм, как и реализованный на данный момент, должен решать задачу построения ГСО. Определим ряд требований, направленных на исправление существующих недостатков реализованного алгоритма построения ГСО:

- подстановки не должны храниться внутри переменных;
- вычисление ГСО должно происходить без этапа вычисления быстрого

обобщения;

- вычисление ГСО должно выполняться последовательно: сначала для пары первых образцов, затем для результата и третьего образца и так далее.

2.2 Обобщение классов $c(m,n)$ и $c(k)$

Введем обозначения: $c(k)$ будем записывать как $(\mathbf{K} : L, \quad T_1 \dots T_L)$, а $c(m,n)$ как $(\mathbf{MN} : M+N, \quad L_1 \dots L_M, \quad R_N \dots R_1)$, где M+N и L — это длины соответствующих классов в термах. В зависимости от длины различных классов будет изменяться способ их обобщения.

Если несколько образцов относятся к классам $c(m_i, n_i)$, то их ГСО также будет входить в класс $c(m^*, n^*)$, очевидно, что $m^* = min(m_i)$, $n^* = min(n_i)$. При этом левая и правая половины обобщения будут попарными обобщениями термов слева и справа соответственно.

Если несколько образцов относятся к различным классам $c(k_i)$, то ГСО будет находиться в одном из классов $c(j, k^* - j)$, где $k^* = min(k_i)$. Левая и правая половины — попарные обобщения исходного образца слева и справа соответственно. Другими словами, при рассмотрении образцов классов $c(k_i)$ различной длины мы их обобщаем точно также, как и образцы классов $c(k_i, k_i)$ с ограничением на то, что окончательный результат должен иметь класс $c(m,n)$, где $m+n = k^*$.

ГСО пары образцов классов $c(m,n)$ и $c(k)$ находится в классе $c(m^*, n^*)$, причём при $m+n > k$ величины m^* и n^* будут лежать в пределах $max(n-k, 0) \leq m^* \leq min(m,k)$ и $max(m-k, 0) \leq n^* \leq min(n,k)$. Неравенства вытекают из соображения, что $m^*+n^* = k$. Если же $m+n \leq k$, то $m^* = m$ и $n^* = n$.

То есть при обобщении пары образцов классов $c(k)$ и $c(m,n)$ мы вправе обобщать их как и $c(k,k)$ с $c(m,n)$, налагая на результат ограничение, что $m^* + n^* \leq k$.

Следовательно, если мы обобщаем набор образцов одного и того же класса $c(k)$, то результат неизбежно будет находиться в классе $c(k)$. Иначе (k_i разной длины, смешанные типы образцов) все образцы класса $c(k_i)$ рассматриваем как образцы класса $c(k_i, k_i)$. При этом на класс окончательного результата $c(m^*, n^*)$ накладываем ограничение $m^* + n^* \leq k^*$, где $k^* = min(k_i)$.

2.3 Улучшенный алгоритм вычисления ГСО

Улучшенный алгоритм построения глобального сложнейшего обобщения можно разделить на три этапа: построение изображений, многорезультатное обобщение и коллапс.

2.3.1 Построение жестких выражений

Два основных принципа алгоритма построения жесткого выражения:

1. у всех переменных удаляются индексы;
2. участки выражения вида «e ... e» заменяются на «e».

Данный алгоритм удалось практически дословно перенести на Рефал-5λ. Реализация алгоритма в рамках функции HardSentence представлена на Листинге 4.

Листинг 4: Реализация функции HardSentence

```
HardSentence {
  (TkVariable 'e' e.LIndex) e.Rest (TkVariable 'e' e.RIndex)
    = (TkVariable 'e');

  (TkVariable 'e' e.LIndex) e.Rest t.Term
```

```
    = <HardSentence (TkVariable 'e' e.LIndex) e.Rest>
      <HardTerm t.Term>;

  t.Term e.Rest = <HardTerm t.Term> <HardSentence e.Rest>;

  /* пусто */ = /* пусто */;
}

HardTerm {
  t.Symbol, <IsSymbol t.Symbol> : True = t.Symbol;

  (TkVariable 's' e.Index) = (TkVariable 's');
  (TkVariable 't' e.Index) = (TkVariable 't');
  (TkVariable 'e' e.Index) = (TkVariable 'e');

  (ADT-Brackets t.Name e.Rest)
    = (ADT-Brackets t.Name <HardSentence e.Rest>);

  (Brackets e.Rest) = (Brackets <HardSentence e.Rest>);
  (Closure e.Rest) = (Closure <HardSentence e.Rest>);
  (CallBrackets e.Rest) = (CallBrackets <HardSentence e.Rest>);
}
```

К примеру, для выражения вида

$$s.Symb \quad (Value \quad s.Value) \quad e.Var \quad t.Term \quad e.variable \quad s.Num$$

будет построен следующее жесткое выражение

$$s \quad (Value \quad s) \quad e \quad s$$

Введем многорезультатное представление образцов.

2.3.2 Построение изображений

Скобочный терм $(t_1 \dots t_K)$ класса K изображается как $(\mathbf{K} : |t'_1 \dots t'_K|, t'_1 \dots t'_K)$, где t'_i — изображение соответствующего терма. Скобочный терм $(t_{L1} \dots t_{LM} \quad e.X \quad t_{RN} \dots t_{R1})$ изображается как $(\mathbf{MN} : \quad (M{+}N), \quad t'_{L1} \dots t'_{LM}, t'_{RN} \dots t'_{R1})$. Символ отображается самим собой. S- и t-переменные также отображаются сами в себя.

Скобочный терм $(\mathbf{K} : \quad \dots)$ описывает скобочный терм с образцом класса $c(k)$. Скобочный терм $(\mathbf{MN} : \quad K, \quad left, \quad right)$ описывает образец класса $c(m, n)$, на который наложено ограничение, что $m + n \leq k$. При переходе от скобочного терма «$left \quad e.X \quad right$» к его изображению $(\mathbf{MN} : \quad K, \quad left, \quad right)$, k будем считать равным |left| + |right|.

Получается, что время стадии построения изображений линейно от длины записи образца.

Описание типов функции BuildImage в рамках которой был реализован описанный алгоритм представлено на Листинге 5.

Листинг 5: Описание типов функции BuildImage

```
<BuildImage t.Term*> == t.Image*

  t.Term ::=
      t.Symbol
    | (TkVariable 't')
    | (TkVariable 's')
    | (ADT-Brackets t.Name t.Term*)
    | (Brackets t.Term*)
    | (Closure t.Term*)
    | (CallBrackets t.Term*)
    | (t.Term* (TkVariable 'e') t.Term*)
    | (t.Term*)
```

```
t.Symbol ::=
    (TkIdentifier e.Name)
  | (TkName t.SrcPos e.Name)
  | (TkNumber s.Number)
  | (TkChar s.HChar)

t.Image ::=
    (K ':' s.Len ',' (t.Image*))
  | (MN ':' s.Len ',' (t.Image*) ',' (t.Image*))
  | t.Symbol
  | (TkVariable 't')
  | (TkVariable 's')
  | (ADT-Brackets t.Name t.Image*)
  | (Brackets t.Image*)
  | (Closure t.Image*)
  | (CallBrackets t.Image*)
```

Далее рассмотрим обобщение полученных изображений.

2.3.3 Многорезультатное обобщение

При анализе пар термов возможны различные ситуации. Опишем результаты для каждой из них:

- одинаковые символы обобщаются в себя;
- разные символы обобщаются в s-переменную;
- символ/s-переменная и скобочный терм обобщаются в t-переменную;
- s-переменная поглощает символы и s-переменные;
- t-переменная поглощает символы, скобки, s- и t-переменные;

- термы ($\mathbf{K}: \quad N, \quad terms_1$) и ($\mathbf{K}: \quad N, \quad terms_2$) равной длины N обобщаются в ($\mathbf{K}: \quad N, \quad terms^*$), а термы внутри обобщаются попарно;

- термы ($\mathbf{MN}: \quad K_1, \quad left_1, \quad right_1$) и ($\mathbf{MN}: \quad K_2, \quad left_2, \quad right_2$) обобщаются в ($\mathbf{MN}: \quad K^*, \quad left^*, \quad right^*$), где $K^* = min(K_1, K_2)$, а цепочки термов обобщаются попарно (соответственно, слева направо и справа налево);

- термы ($\mathbf{MN}: \quad K_1, \quad left_1, \quad right_1$) и ($\mathbf{K}: \quad N, \quad terms_2$) обобщаются также, как термы ($\mathbf{MN}: \quad K_1, \quad left_1, \quad right_1$) и ($\mathbf{MN}: \quad N, \quad terms_2, \quad terms_2$);

- термы ($\mathbf{K}: \quad K_1, \quad terms_1$) и ($\mathbf{K}: \quad K_2, \quad terms_2$) обобщаются также, как термы ($\mathbf{MN}: \quad K_1, \quad terms_1, \quad terms_1$) и ($\mathbf{MN}: \quad K_2, \quad terms_2, \quad terms_2$).

Физический смысл многорезультатного обобщения следующий. Выражение вида ($\mathbf{MN}: \quad K, \quad terms_1, \quad terms_2$) означает то, что оно является образом целого семейства выражений вида ($terms'_1 \quad e \quad terms'_2$), при этом длина выражения $terms'_1$ равна m, длина выражения $terms'_2$ равна n, а $m+n=k$. $terms'_1$ является префиксом $terms_1$, а $terms'_2$ является суффиксом $terms_2$.

2.3.4 Коллапс

Возврат от ассоциативного представления назовем коллапсом. Коллапс описывается следующими правилами:

- изображения символов и s-, t-переменных коллапсируют в себя;

- скобочные термы вида ($\mathbf{K}: \quad N, \quad t'_1 \ldots t'_K$) коллапсируют в ($t_1 \ldots t_K$), где внутренние термы коллапсируют рекурсивно;

- скобочные термы ($\mathbf{MN}: \quad K, \quad t'_{L1} \dots t'_{LM}, \quad t'_{RN} \dots t'_{R1}$) соответствуют набору ЛСО. Сначала рекурсивно коллапсируют вложенные термы, затем строятся образцы вида $t_{L1} \dots t^{*}_{LM} \quad e.X \quad t^{*}_{RN} \dots t_{R1}$ классов (M^*, N^*), где $M^* + N^* \leq K$, $M^* \leq M$, $N^* \leq N$. Среди этих образцов выбирается образец $t_{L1} \dots t^{**}_{LM} \quad e.X \quad t^{**}_{RN} \dots t_{R1}$, обладающий наибольшей сложностью. Таким образом, семейство ЛСО коллапсирует в единственный скобочный терм ($t_{L1} \dots t^{**}_{LM} \quad e.X \quad t^{**}_{RN} \dots t_{R1}$), являющийся ГСО. Если несколько вариантов имеют одинаковую наибольшую сложность, то выбирается произвольный.

С получившегося в результате скобочного терма снимаются скобки, результат становится ГСО для данного набора жестких образцов. Время стадии коллапса линейно от длины записи изображения,так как по построению нам достаточно получить лишь одно ГСО, а не все их семейство.[9]

Рассмотрим на примере, каким образом происходит выбор ГСО среди ЛСО. Пусть в результате обобщения был получен образ

$$(\mathbf{MN}: \quad 4, \quad s \quad (e) \quad t \quad t, \quad t \quad t \quad t \quad s).$$

Тогда этот образ соответствует 5 ЛСО: 0 + 4, 1 + 3, 2 + 2, 3 + 1, 4 + 0. Первая цифра — длина первой половины, вторая — правой.

$(\mathbf{MN}: \quad 4,$	$s \quad (e) \quad t \quad t,$		$t \quad t \quad t \quad s)$	
$0+4:$		e	$ttts,$	сложность 5,
$1+3:$	s	e	$tts,$	сложность 6,
$2+2:$	$s(e)$	e	$ts,$	**сложность 7**,
$3+1:$	$s(e)t$	e	$s,$	**сложность 7**,
$4+0:$	$s(e)tt$	$e,$		сложность 6.

Следовательно ГСО:

$$\{s \quad (e) \quad e \quad t \quad s, \quad s \quad (e) \quad t \quad e \quad s\}$$

.

Функция «LocalGen», осуществляющая построения ЛСО приводится на Листинге 6.

Листинг 6: Реализация функции LocalGen

```
LocalGen {
  s.Num (e.TermsL) (e.TermsR)
    = <DoLocalGen 0 s.Num (e.TermsL) (e.TermsR)>;
}

DoLocalGen {
  s.NumL s.NumR (e.TermsL) (e.TermsR)
    = <First s.NumL e.TermsL> : (e.FirstTermsL) e.RestL
    = <Last s.NumR e.TermsR> : (e.RestR) e.LastTermsR
    = e.FirstTermsL (TkVariable 'e') e.LastTermsR : e.LocalGen
    = (<CalcComplexity e.LocalGen> e.LocalGen) : t.Result
    = s.NumR
    : {
        0 = t.Result;
        s.NumR^ = t.Result
                  <DoLocalGen
                    <Inc s.NumL>
                    <Dec s.NumR>
                    (e.TermsL) (e.TermsR)
                  >;
      };
}
```

Для упрощения алгоритма построения глобального сложнейшего обобщения потребуется функция, которая будет строить подстановки, переводящие ГСО в конкретный образец.

2.3.5 Функция обобщенного сопоставления с образцом

В качестве аргумента функция GenericMatch будет принимать два образцовых выражения: образец и его глобальное сложнейшее обобщение. После выполнения функция должна вернуть один из следующих результатов:

- признак неуспешного сопоставления;
- признак безусловного успешного сопоставления и набор подстановок.

В качестве признака неуспешного сопоставления был выбран идентификатор «Failed». Если же сопоставление оказалось успешным, то результатом будет являться пара: идентификатор «Clear» и набор объектных выражений, уточняющих ГСО до образца. Описание типов функции GenericMatch представлено на Листинге 7.

Листинг 7: Описание типов функции GenericMatch

```
<GenericMatch (e.Pattern) (e.LPattern)>
    == Clear (e.Val ':' t.Var)*
    == Failure

  e.Pattern, e.LPattern, e.Val ::= t.PatternTerm*
  t.PatternTerm ::=
        (#TkChar s.Char)
```

```
    | (#TkNumber s.Number)
    | (#TkName e.Name)
    | (#TkIdentifier e.Name)
    | (#Brackets e.Expression)
    | (#ADT-Brackets (e.ADTName) e.Expression)
    | (#CallBrackets e.Expression)
    | (#TkVariable s.Mode e.Index)
    | (#Closure e.Body)

t.Var ::= (s.VarType e.Index)
s.VarType ::= 's' | 't' | 'e'
```

Алгоритм обобщенного сопоставления с образцом можно описать следующим образом. При сопоставлении выражения общего вида E и жесткого выражения He (обозначается как $E \quad He$) жесткое выражение имеет вид $Ht'_1 \dots Ht'_N \quad e.X \quad Ht'_M \dots Ht'_1$ или $Ht_1 \dots Ht_N$, где Ht — жесткий терм, а $e.X$ — e-переменная.

Нужно рассмотреть следующие случаи:

- жёсткое выражение начинается на жёсткий терм
- жёсткое выражение заканчивается на жёсткий терм,
- жёсткое выражение состоит из одной e-переменной,
- жёсткое выражение пустое.

«Жёсткое выражение имеет вид $Ht \quad He'$, где Ht — жёсткий терм». Если сопоставляемое выражение имеет вид $T \quad E'$, где T — некий терм (символ, выражение в скобках, переменные s или t, замыкание), то выполняем сопоставления соответственно $T : Ht$ и $E' : He'$.

Иначе, если выражение E начинается на e-переменную или на вызов функции, имеем неудачу сопоставления и сразу возвращаем Failure.

«Жёсткое выражение имеет вид $He' \quad Ht$, где Ht — жёсткий терм».

Случай аналогичен предыдущему, сопоставляемое выражение тоже должно заканчиваться на терм.

«Жёсткое выражение состоит из e-переменной $e.X$».

Присваиваем этой переменной сопоставляемое выражение $E \leftarrow e.X$. Всегда успешно.

«Жёсткое выражение пустое».

Если сопоставляемое выражение пустое, то сопоставление успешно. Иначе неудача.

Далее нужно рассмотреть алгоритм попарного сопоставления термов. Пусть имеем сопоставление $T : Ht$:

- Если Ht является t-переменной $t.X$, то строим присваивание $T \leftarrow t.X$.
- Если Ht является s-переменной, то T может быть только символом, s-переменной или замыканием ($ClosureBrackets$). В случае, если T — скобочный терм ($Brackets$ или $ADT - Brackets$), имеем неудачу сопоставления.
- Если Ht является символом (идентификатором, именем функции, числом или литерой), то T может быть только той же литерой. Иначе неудача.
- Если Ht является выражением в круглых скобках (He'), то T может быть только выражением в круглых скобках (E'). Соответственно, сопоставление выполняется рекурсивно $E' : He'$.
- Если Ht является выражением в абстрактных скобках $[XHe']$, то T может быть только выражением в абстрактных скобках с той же меткой $[XE']$. Далее аналогично рекурсивно для оставшихся типов.

2.4 Совместное сопоставление с образцом

2.4.1 Интеграция улучшенного алгоритма вычисления ГСО

Для полученных предложений находим образец. Затем преобразуем их в жесткие образцы. Строим ГСО для набора жестких образцов и добавляем уникальные индексы для полученного выражения. Как и раньше найдем общие команды для образцов всех предложений. Для каждой найденной в последствие подгруппы будем производить подстановки новых уточненных переменных в набор общих команд сопоставления. Таким образом для каждого уровня вложенности групп будет свой набор общих команд. Результат интеграции представлен на Листинге 8.

Листинг 8: Функция совместного сопоставления с образцом

```
HighLevelRASL-Function-ConjointExt {
  s.FnGenSubst s.FnGenResult s.ScopeClass (e.Name) e.Sentences

    /* Получаем из предложений образцы и преобразуем в жесткие
   образцы */
    = <Map
        {
          ((e.Pattern) e.Conditions (e.Result))
            = (<HardSentence e.Pattern>);
        }
        e.Sentences
      >
    : e.HardPatterns

    /* Для жестких образцов строим ГСО */
    = <GlobalGen e.HardPatterns> : e.GlobalGen
```

```
    /* Добавляем к ГСО уникальные идентификаторы */
    = <EnumerateVars e.GlobalGen> : e.GlobalGen^

    /* Сопоставляем образец каждого предложения с ГСО */
    = <Map
        {
          ((e.Pattern) e.Conditions (e.Result))

            /* Строим подстановки, переводящие ГСО в образец */
            = <GenericMatch (e.Pattern)(e.GlobalGen)>
              : Clear e.Substs

            /* Заменяем образцы на подстановки */
            = ((e.Substs)() e.Conditions (e.Result));
        }
        e.Sentences
      >
    : e.SentencesWithSubst

    ...;
}
```

2.4.2 Формирование групп предложений

Будем рассматривать способ группировки, который используется в компиляторах и суперкомпиляторах Рефала. Сначала будем строить последовательность команд сопоставления для каждого предложения, а затем эти цепочки будем превращать в префиксное дерево. Основной проблемой является то, что для одного и того же образца можно построить различные последовательности команд. Например, для образца

$$A \quad B \quad e.X \quad C \quad D$$

возможны следующие цепочки команд:

$$(e.0 \to A \quad e.0)(e.0 \to B \quad e.0)(e.0 \to e.0 \quad D)(e.0 \to e.0 \quad C)(e.0 \to e.X)$$
$$(e.0 \to e.0 \quad D)(e.0 \to A \quad e.0)(e.0 \to e.0 \quad C)(e.0 \to e.0 \quad B)(e.0 \to e.X)$$
$$...$$

Соответственно для одних вариантов ветви могут быть длиннее, а для других короче.

Пусть на нулевой итерации имеется набор предложений. Вычисляем для них ГСО и подстановки. Тогда часть предложений будет иметь дублирующиеся операции сопоставления. Требуется выделить группу из нескольких первых предложений, у которых будет достаточно много общих операций сопоставления. Для группы можно найти ГСО для каждой из подстановок, чтобы вычислить его отдельно. Такое разбиение на группы рекурсивно выполняем для оставшихся предложений, а также для предложений первой группы.

Для разбиения на группы будем отщеплять по одному предложению с конца и для образцов оставшихся предложений будем сравнивать ГСО значений подстановок и их обобщение. Если обобщение стало точнее (или сложнее с точки зрения метрики введенной ранее для образцов) то считаем, что разбиение на группы найдено. Иначе, пока количество предложений в группе не стало равным 1 продолжаем процесс отщепления и оценки. Листинг 9 показывает устройство функции сравнения ГСО подстановок и их начального обобщения.

Листинг 9: Реализация функции CompareSubsts

```
CompareSubsts {
```

```
(/* подстановки закончились*/) e.OtherSentences = Continue;

e.Sentences
  /* Отделяем по одной подстановке слева
     и берем только её значение */
  = <MapReduce
      {
        ((e.PrevVal) ':' e.Var)
        (
          (e.Val ':' (s.VarType e.Index)) e.OtherSubsts
        )
          = ((e.PrevVal (e.Val)) ':' (TkVariable s.VarType))
            (e.OtherSubsts);
      }
      ((/* Значения подстановок */) ':' /* обобщение */)
      e.Sentences
    >
  : ((e.SubstsVals) ':' t.Var) e.Sentences^

  /* Получаем из значений подстановок жесткие подстановки */
  = <Map
      { (e.SubstsVal) = (<HardSentence e.SubstsVal>); }
      e.SubstsVals
    >
  : e.HardSubsts

  /* Строим ГСО для подстановок */
  = <GlobalGen e.HardSubsts> : e.GlobalGenOfSubsts

  /* Сравним ГСО для подстановок группы
```

```
        и для всех предложений */
    = e.GlobalGenOfSubsts
    : {
        /* Если ГСО точнее не стало,
           рассмотрим остальные подстановки */
        t.Var = <CompareSubsts e.Sentences>;

        /* Иначе, разбиение на группы найдено */
        e.Else = Stop;
      };
}
```

Идентификатор Stop в качестве возвращаемого значения используется, если разбиение найдено. Иначе передается идентификатор Continue.

Для групп, которые разбить больше нельзя, выполним итоговую генерацию команд сопоставления с образцом.

2.4.3 Рекурсивное разбиение предложений на группы

После того, как группы предложений были определены, нужно проанализировать их длину. В случае, если длина первой группы оказалась равной нулю, то это означает, что все предложения имеют достаточно схожую структуру и их разбиение не требуется. Достаточно выполнить генерацию команд сопоставления с образцом для предложений первой и второй групп. Аналогично, выполняем генерацию команд сопоставления с образцом предложений группы в случае, если в ней осталось всего одно предложение после разбиения.

Листинг 10: Генерация команд для первой группы

```
/* Генерация команд для первой группы */
= s.Num1
```

```
: {
  1 = <ReturnCommandsOfLastSentences
        s.FnGenSubst s.FnGenResult
        (e.MarkedPattern) (e.SentencesWithSubst1) s.ContextSize
      >
    : s.FirstSentenceContext e.FirstSentenceCommands
    = s.FirstSentenceContext (CmdSentence e.
   FirstSentenceCommands);

  s.Num1^
    = <GenerateCommands
        s.FnGenSubst s.FnGenResult
        (e.MarkedPattern) (e.SentencesWithSubst1) (e.Patterns1)
        (e.GlobalGen)
        s.ContextSize <Inc s.BaseNum>
      >
    : s.FirstGroupContext e.FirstGroupCommands
    = s.FirstGroupContext (CmdSentence e.FirstGroupCommands);
  }
: s.FirstPartContext e.FirstPartCommands
```

В отличие от обработки первой группы (Листинг 10), предложения второй группы не оборачиваются в защищенный блок (Листинг 11).

Листинг 11: Генерация команд для второй группы

```
/* Генерация команд для второй группы */
= s.Num2
: {
  1 = <ReturnCommandsOfLastSentences
        s.FnGenSubst s.FnGenResult
```

```
        (e.MarkedPattern) (e.SentencesWithSubst2) s.ContextSize
      >
    : s.SecondSentenceContext e.SecondSentenceCommands
    = s.SecondSentenceContext e.SecondSentenceCommands;

  s.Num2^
    = <Last s.Num2 e.SubstsForSentences>
    : (e.SubstsForSentences1) e.SubstsForSentences2

    = <GenerateCommandsOfGroups
        (e.GlobalGen) (e.SubstsForSentences2)
        s.FnGenSubst s.FnGenResult
        (e.MarkedPattern) (e.SentencesWithSubst2) (e.Patterns2)
        s.ContextSize s.BaseNum
      >
    : s.SecondGroupContext e.SecondGroupCommands
    = s.SecondGroupContext e.SecondGroupCommands;
  }
: s.SecondPartContext e.SecondPartCommands
```

2.5 Оптимизация программы

В процессе разработки и тестирования функции, выполняющей сравнение глобального сложнейшего обобщения для нескольких первых предложений и для всех предложений, чтобы найти нужное разбиение на группы, было обнаружено, что сравнение выполняется достаточно сложным и ненадежным способом. Опишу его подробно.

Для того, чтобы понять, является ли обобщение группы предложений более сложным по сравнению с обобщением всех предложений, сначала производится обобщение подстановок, а затем выполняются подстановки в

обобщение для всех предложений. Затем полученный результат сравнивается с исходным обобщением.

Данный подход является сложным из-за того, что при построении подстановок, теряется общая структура обобщенного предложения. Так происходит в силу алгоритма построения подстановок. Поэтому для каждой подстановки приходится выполнять рекурсивный поиск по термам и затем рекурсивно заменять найденные вхождения на значение подстановки.

В качестве оптимизации было решено производить сравнение еще на этапе обобщения подстановок. Таким образом, обобщение набора подстановок сравнивается с изначальным обобщением подстановки.

Такая оптимизация значительно упростила интерфейсы функций и всю логику сравнения обобщений. Более того, в процессе введения оптимизации, были исправлены баги, связанные с множественной кодогенерацией команд, необходимых для сохранения контекста.

Эти баги не влияли на работоспособность программы, так как приводили лишь к повторной записи одних и тех же значений в ячейки памяти. Однако, исключение повторных операций позитивно сказалось на итоговой производительности программы и внешнем виде генерируемого кода.

Листинг 12: Сравнение интерфейса функции до и после оптимизации

```
/* Интерфейс до оптимизации */
CompareSubsts {
  e.Sentences (/* GlobalGen */) = Continue;
  (/* подстановки закончились*/) e.OtherSentences (e.GlobalGen)
    = Continue;

  e.Sentences (t.GlobalGenFirst e.GlobalGen)
    = ...;
```

```
/* Интерфейс после оптимизации */
CompareSubsts {
  (/* подстановки закончились*/) e.OtherSentences
    = Continue;

  e.Sentences
    = ...;
```

3 ТЕСТИРОВАНИЕ

Для тестирования итоговой оптимизации требуется выполнить замеры на различных системах и компиляторах. Проверка производительности будет выполняться на следующих операционных системах:

1. Машина 1. Компьютер Intel® Core™ i5-5200U CPU @ 2.20GHz, Кэши L1/L2/L3: 128 Кбайт/512 Кбайт/3,0 Мбайт, ОЗУ 8 Гбайт. Операционная система Windows 10.

2. Машина 2. Компьютер Intel® Core™ i7-5600U CPU @ 2.60GHz × 4, 4 Мбайта кэш памяти, ОЗУ 16 Гбайт. Ubuntu 16.04 LTS.

Компилятор на первой машине — «Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland».

Компилятор на второй машине — «GCC 5.4.0».

Запуск тестов производительности выполняется следующим скриптом:

Листинг 13: Скрипт запуска тестов производительности

```
set SRMAKE_FLAGS=
call benchmark.bat 13 no-opt
set SRMAKE_FLAGS=-X-OP
call benchmark.bat 13 OP
set SRMAKE_FLAGS=-X-OQ
```

```
call benchmark.bat 13 OQ
```

Выполняется сборка компилятора без флага оптимизации и затем замер производительности различных операций. После этого происходит сборка компилятора с флагом оптимизации «OP», который соответствует предыдущей реализации оптимизации совместного сопоставления с образцом. Так же собирается информация о времени выполнения операций. И наконец последняя сборка происходит для оптимизации, реализованной в рамках данной дипломной работы. После чего также сохраняются данные теста.

В результате получатся три файла с расширением *.time, где находятся отсортированные времена работы. Среднее по порядку значение (7-е для 13) — медиана, середины половин (4-е и 10-е) — доверительный интервал.

Для первой машины были получены следующие данные. Чистое время Рефала (т.е. без библиотечных функций):

- без оптимизаций — 11,158 секунд, доверительный интервал 11,013. . . 12,297
- OP — 10,059 с, доверительный интервал 9,878. . . 10,223
- OQ — 9,595 с, доверительный интервал 9,352. . . 9,788

Замеры можно считать достоверными, так как доверительные интервалы не пересекаются. Оптимизация «ничего»→«OP» даёт прирост 9,8%, оптимизация «OP»→«OQ» — 4,6%. Оптимизация «ничего»→«OQ» — 14,0 %.

Линейное время сопоставления с образцом:

- без оптимизаций — 5,103, доверительный интервал 4,698. . . 5,290
- OP — 3,924, интервал 3,751. . . 4,031
- OQ — 3,501, интервал 3,443. . . 3,671

Опять статистически достоверные результаты. Прирост производительности:

- «ничего»→«OP»: 5,103−3,924 = 1,179; 1,179 / 5,103 = 23,1 %
- «OP»→«OQ»: 3,924−3,501 = 0,423; 0,423 / 3,924 = 10,8 %
- «ничего»→«OQ»: 5,103−3,501 = 1,602; 1,602 / 5,103 = 31,4 %

Теперь оценим данные для второй машины. Чистое время Рефала (т.е. без библиотечных функций):

- без оптимизаций — 32,35 секунд, доверительный интервал 32.291. . . 32.410
- OP — 30,754 с, доверительный интервал 30.732. . . 30.776
- OQ — 27,486 с, доверительный интервал 27.387. . . 27.584

Доверительные интервалы не пересекаются, а значит замеры также достоверные. Оптимизация «ничего»→«OP» даёт прирост 4,9%, оптимизация «OP»→«OQ» — 10,6%. Оптимизация «ничего»→«OQ» — 15,0 %.

Линейное время сопоставления с образцом:

- без оптимизаций — 20,6035, доверительный интервал 20.556. . . 20.651
- OP — 19,0645, интервал 19.034. . . 19.095
- OQ — 15,7505, интервал 15.713. . . 15.788

Опять статистически достоверные результаты. Прирост производительности:

- «ничего»→«OP»: 20,6035−19,0645 = 1,539; 1,539 / 20,6035 = 7,4 %
- «OP»→«OQ»: 19,0645−15,7505 = 3,314; 3,314 / 19,0645 = 17,4 %
- «ничего»→«OQ»: 20,6035−15,7505 = 4,853; 4,853 / 20,6035 = 23,6 %

Общий прирост производительности присутствует на замерах для двух машин. Таким образом наглядно показана эффективность реализованного алгоритма с точки зрения производительности. Рассмотрим как изменилась генерация команд сопоставления с образцом.

3.1 Оценка оптимизации

Для примера, который был описан в главе 1.3, полученный в итоге набор команд сопоставления соответствует ожидаемому (Листинг 3). Рассмотрим интересный пример того, как новый алгоритм обобщил набор образцов:

$$
\begin{aligned}
'ENUM' &= (TkDirectiveEnum);\\
'EENUM' &= (TkDirectiveEEnum);\\
'ENTRY' &= (TkDirectiveEntry);\\
'EXTERN' &= (TkDirectiveExtern);\\
'FORWARD' &= (TkDirectiveForward);\\
'SWAP' &= (TkDirectiveSwap);\\
'ESWAP' &= (TkDirectiveESwap);\\
'LABEL' &= (TkDirectiveIdent);\\
'INCLUDE' &= (TkDirectiveInclude);\\
'SCOPEID' &= (TkNumberCookie1)(TkNumberCookie2);
\end{aligned}
$$

Для такого набора образцов компилятор смог выделить подгруппу:

$$
\begin{aligned}
'ENTRY' &= (TkDirectiveEntry);\\
'EXTERN' &= (TkDirectiveExtern);\\
'FORWARD' &= (TkDirectiveForward);
\end{aligned}
$$

И обобщить ее как

$$e.1 \quad s.2 \quad s.3 \quad 'R' \quad s.4.$$

То есть компилятор заметил во всех трех предложениях, что предпоследний символ образцов — 'R' и составил соответствующее обобщение.

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены существующие стратегии обобщения образцов и способы их группировки. В результате был разработан и реализован улучшенный алгоритм совместного сопоставления с образцом в языке Рефал-5λ. Требования, предъявленные к разрабатываемому алгоритму, были выполнены. В перспективе планируется реализация более сложного, но и более эффективного способа группировки образцов.

Список литературы

1 А. В. Коновалов. Пользовательская документация для языка Простой Рефал [Электронный ресурс] — Режим доступа: https://github.com/Mazdaywik/simple-refal/blob/master/doc/manul.pdf — Дата обращения: 05.03.2018

2 История РЕФАЛ-компилятора [Электронный ресурс]— Режим доступа: http://refal.net — Дата обращения: 22.04.2018

3 Р. Гурин, С. Романенко. Язык программирования Рефал Плюс. Курс лекций. Учебное пособие для студентов университета города Переславля. — Переславль-Залесский: «Университет города Переславля» им.А.К.Айламазяна: 2006. — 222 с.

4 С.А. Романенко. Машинно-независимый компилятор с языка рекурсивных функций — Москва: 1978. — 98 с.

5 И. Скрыпников. ВКР по теме Оптимизация совместного сопоставления с образцом [Электронный ресурс] — Режим доступа: «https://github.com/Mazdaywik/simple-refal/blob/master/doc/OptPattern/Скрыпников Записка 2016.pdf» — Дата обращения: 12.02.2018

6 В. Ф. Турчин. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ. В сб.: Труды симпозиума “Теория языков и методы построения систем программирования” — Киев-Алушта: 1972. — 31 с.

7 В. Ф. Турчин. Эквивалентные преобразования программ на РЕФАЛе. В сб.: Труды ЦНИПИАСС “Автоматизированная система управления строительством”, выпуск 6 — М: 1974. — 36 с.

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

9 Т. Кормен, Ч. Лейзерсон, Р. Ривест «Алгоритмы: построение и анализ» — М.: МЦНМО, 2001. — 960 с

```
Generalization {
  t.Image1 t.Image2 e.Rest
    = <Generalization <TermGeneralization t.Image1 t.Image2> e.
   Rest>;
  e.Image = e.Image;
}

TermGeneralization {
  t.Symbol t.Symbol, <IsSymbol t.Symbol> : True = t.Symbol;

  t.Term1 t.Term2
    , <IsSVarSubset t.Term1> : True
    , <IsSVarSubset t.Term2> : True
    = (TkVariable 's');

  (Brackets e.BodyL) (Brackets e.BodyR)
    = (Brackets <Generalization e.BodyL e.BodyR>);

  (ADT-Brackets t.Name e.BodyL) (ADT-Brackets t.Name e.BodyR)
    = (ADT-Brackets t.Name <Generalization e.BodyL e.BodyR>);

  t.Term1 t.Term2
    , <IsTVarSubsetOrTVar t.Term1> : True
    , <IsTVarSubsetOrTVar t.Term2> : True
    = (TkVariable 't');

  (K ':' s.Num ',' t.Left)
  (K ':' s.Num ',' t.Right)
    = (K ':' s.Num ','
```

```
      (<GeneralizationInPairs t.Left t.Right>)
    );

(K ':' s.NumL ',' t.Left)
(K ':' s.NumR ',' t.Right)
  = <Min s.NumL s.NumR> : s.Num
  = t.Left t.Right : e.Pairs
  = (
      MN ':' s.Num ','
      (<GeneralizationInPairs Left e.Pairs>) ','
      (<GeneralizationInPairs Right e.Pairs>)
    );

(MN ':' s.NumL ',' (e.TermsL1) ',' (e.TermsL2))
(MN ':' s.NumR ',' (e.TermsR1) ',' (e.TermsR2))
  = <Min s.NumL s.NumR> : s.Num
  = (e.TermsL1) (e.TermsR1) : e.LPairs
  = (e.TermsL2) (e.TermsR2) : e.RPairs
  = (
      MN ':' s.Num ','
      (<GeneralizationInPairs Left e.LPairs>) ','
      (<GeneralizationInPairs Right e.RPairs>)
    );

(MN ':' s.NumL ',' (e.TermsL1) ',' (e.TermsL2))
(K ':' s.NumR ',' t.TermsR)
  = <Min s.NumL s.NumR> : s.Num
  = (e.TermsL1) t.TermsR : e.LPairs
  = (e.TermsL2) t.TermsR : e.RPairs
  = (
```

```
        MN ':' s.Num ','
        (<GeneralizationInPairs Left e.LPairs>) ','
        (<GeneralizationInPairs Right e.RPairs>)
      );

  (K ':' s.NumL ',' t.TermsL)
  (MN ':' s.NumR ',' (e.TermsR1) ',' (e.TermsR2))
    = <Min s.NumL s.NumR> : s.Num
    = t.TermsL (e.TermsR1) : e.LPairs
    = t.TermsL (e.TermsR2) : e.RPairs
    = (
        MN ':' s.Num ','
        (<GeneralizationInPairs Left e.LPairs>) ','
        (<GeneralizationInPairs Right e.RPairs>)
      );
}
```

```
Collapse {
  e.Generalization
    = <Map
        {
          t.Symbol, <IsSymbol t.Symbol> : True = t.Symbol;
          (TkVariable 't') = (TkVariable 't');
          (TkVariable 's') = (TkVariable 's');

          (Brackets e.Terms) = (Brackets <Collapse e.Terms>);

          (ADT-Brackets t.Name e.Terms)
            = (ADT-Brackets t.Name <Collapse e.Terms>);

          (K ':' s.Num ',' (e.Terms)) = <Collapse e.Terms>;

          (MN ':' s.Num ',' (e.Terms1) ',' (e.Terms2))
            = <Collapse e.Terms1> : e.TermsC1
            = <Collapse e.Terms2> : e.TermsC2
            = <GlobalGenFromLocalGen
                <LocalGen s.Num (e.TermsC1) (e.TermsC2)>
              >;
        }
        e.Generalization
      >;
}
```

```
GenerateCommandsOfGroups {
  (e.GlobalGen) (e.SubstsForSentences)
  s.FnGenSubst s.FnGenResult
  (e.MarkedPattern) (e.SentencesWithSubst) (e.Patterns) s.
   ContextSize s.BaseNum

    /* Ищем разбиение предложений на 2 группы */
    = <FindDivision (e.SubstsForSentences)> : e.FirstGroup

    = e.SubstsForSentences : e.FirstGroup e.SecondGroup

    = <Len e.FirstGroup><Len e.SecondGroup>
    : {
        /* Дальнейшее разбиение невозможно.
           Формируем команды для предложений */
        0 s.Num2 = <ReturnCommandsOfLastSentences
              s.FnGenSubst s.FnGenResult
              (e.MarkedPattern) (e.SentencesWithSubst)
              s.ContextSize
            >;


        s.Num1 s.Num2
          = <First s.Num1 e.SentencesWithSubst>
          : (e.SentencesWithSubst1) e.SentencesWithSubst2

          = <First s.Num1 e.Patterns>
          : (e.Patterns1) e.Patterns2
```

```
/* Генерация команд для первой группы */
= s.Num1
: {
    1 = <ReturnCommandsOfLastSentences
          s.FnGenSubst s.FnGenResult (e.MarkedPattern)
          (e.SentencesWithSubst1) s.ContextSize
        >
      : s.FirstSentenceContext e.FirstSentenceCommands
      = s.FirstSentenceContext
        (CmdSentence e.FirstSentenceCommands);


    s.Num1^
      = <GenerateCommands
          s.FnGenSubst s.FnGenResult
          (e.MarkedPattern) (e.SentencesWithSubst1)
          (e.Patterns1) (e.GlobalGen)
          s.ContextSize <Inc s.BaseNum>
        >
      : s.FirstGroupContext e.FirstGroupCommands
      = s.FirstGroupContext
        (CmdSentence e.FirstGroupCommands);
  }
: s.FirstPartContext e.FirstPartCommands


/* Генерация команд для второй группы */
= s.Num2
: {
```

```
      1 = <ReturnCommandsOfLastSentences
            s.FnGenSubst s.FnGenResult
            (e.MarkedPattern) (e.SentencesWithSubst2)
            s.ContextSize
          >
        : s.SecondSentenceContext
          e.SecondSentenceCommands
        = s.SecondSentenceContext
          e.SecondSentenceCommands;


      s.Num2^
        = <Last s.Num2 e.SubstsForSentences>
        : (e.SubstsForSentences1) e.SubstsForSentences2

        = <GenerateCommandsOfGroups
            (e.GlobalGen) (e.SubstsForSentences2)
            s.FnGenSubst s.FnGenResult (e.MarkedPattern)
            (e.SentencesWithSubst2) (e.Patterns2)
            s.ContextSize s.BaseNum
          >
        : s.SecondGroupContext e.SecondGroupCommands
        = s.SecondGroupContext e.SecondGroupCommands;
    }
  : s.SecondPartContext e.SecondPartCommands


= <Max s.FirstPartContext s.SecondPartContext>
: s.ContextSize^
```

```
            = s.ContextSize e.FirstPartCommands
              e.SecondPartCommands;
        }
    : s.MaxMemory e.ResultCommands

    = s.MaxMemory e.ResultCommands;
}
```