



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПISКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:

***«Улучшение результатов суперкомпиляции
посредством препроцессирования исходных
программ»***

Студент

(Подпись, дата)

Рудикова Т.В.
(И.О.Фамилия)

Руководитель ВКР

(Подпись, дата)

Коновалов А.В.
(И.О.Фамилия)

Консультант

(Подпись, дата)

(И.О.Фамилия)

Консультант

(Подпись, дата)

(И.О.Фамилия)

Нормоконтролер

(Подпись, дата)

(И.О.Фамилия)

2019 г.

Аннотация

Тема: «Улучшение результата суперкомпиляции посредством препроцессирования исходных программ».

Цель данной работы – разработать препроцессор, позволяющий сохранять больше информации в конфигурациях, получающихся в процессе суперкомпиляции, и затем исследовать влияние препроцессирования на результаты суперкомпиляции.

Для достижения поставленной цели были изучены основные понятия суперкомпиляции, разработаны и реализованы два препроцессора и рассмотрено влияние препроцессирования на процесс суперкомпиляции и остаточные программы, получающиеся в результате.

Объём расчётно-пояснительной записки к выпускной квалификационной работы составляет 71 страницу, на которых размещены 29 рисунков и 3 таблицы. Список использованных источников состоит из 12 названий.

Содержание

Аннотация	2
Введение.....	5
1. Постановка задачи	7
2. Суперкомпиляция	8
2.1. Прогонка.....	8
2.2. Свёртка.....	10
2.2.1. Вложение	10
2.2.2. Обобщение	12
3. Разработка.....	14
3.1. Первый препроцессор	14
3.2. Второй препроцессор	15
4. Реализация	18
4.1. Лексический анализатор	18
4.2. Первый препроцессор	21
4.3. Второй препроцессор	25
5. Примеры.....	30
5.1. Программа замены литеры 'А' на литеру 'В'	30
5.1.1. Первый препроцессор	33
5.1.2. Второй препроцессор	34
5.1.3. Постановка задачи как задачи верификации	39
5.2. Программа вычисления целочисленного логарифма	41
5.2.1. Первый препроцессор	44
5.2.2. Второй препроцессор	46
5.2.3. Исследование примера на суперкомпиляторе MSCP-A.....	47

5.3. Криптографический протокол.....	49
5.3.1. Основная теоретическая информация.....	49
5.3.2. Упрощенный криптографический протокол	51
5.3.3. Первый препроцессор	54
5.3.4. Второй препроцессор	56
5.3.5. Результаты суперкомпиляции уточненным суперкомпилятором.....	58
5.4. Результаты.....	59
6. Руководство пользователя	61
6.1. Компиляция и использование	61
6.2. Псевдокомментарий первого препроцессора	61
6.3. Псевдокомментарий второго препроцессора	64
Заключение	67
Список использованных источников	68
Приложение А	70

Введение

Суперкомпиляция – это способ преобразования и анализа программ, который был предложен В.Ф. Турчиным в 1970-х годах [1]. В основе суперкомпиляции лежит выполнение следующих действий: прогонка, зацикливание и обобщение.

Прогонкой называется попытка выполнения программы для неизвестных входных данных. Для этого строится так называемое «дерево конфигураций» [2], узлы которого содержат «конфигурации» – описания множеств мгновенных состояний вычислительного процесса, а рёбрам соответствуют действия и проверки, происходящие в процессе выполнения программы.

Зацикливание и обобщение – действия, применяемые, чтобы избежать построения бесконечного дерева, которое получается в том случае, если исходная программа содержит циклы или рекурсию. В результате этих действий потенциально бесконечное дерево преобразуется в конечный граф конфигураций.

Полученный в результате конечный граф конфигураций преобразуется в остаточную программу, которая и будет являться результатом работы суперкомпилятора.

На действия, выполняемые суперкомпилятором, влияет информация, хранящаяся в конфигурациях. Это значит, что чем больше информации предоставляет конфигурация, тем более глубокий анализ может провести суперкомпилятор. Дополнительная информация в конфигурации может быть добавлена разными способами, в том числе и посредством самого суперкомпилятора, однако, более интересны способы, которые не будут требовать преобразований суперкомпилятора.

В данной работе рассматриваются способы препроцессирования исходного текста программы, которая будет анализироваться суперкомпилятором: программа преобразуется в эквивалентную, возможно, с точностью до кодирования данных. При этом в поле зрения рефал-машины будет находиться некоторая дополнительная информация, не влияющая на ход вычислений. Но эта

же информация будет храниться и в конфигурациях, и она будет влиять на решения суперкомпилятора.

В работе рассматривается суперкомпиляция языка программирования Рефал [3] и в частности суперкомпиляторы SCP4[4] и MSCP-A[5].

1. Постановка задачи

В рамках данной работы были поставлены следующие цели:

- изучить принципы работы суперкомпилятора;
- написать препроцессор программ, написанных на языке Рефал-5, позволяющий обогатить конфигурации, без изменения кода самого суперкомпилятора;
- исследовать насколько и как улучшается результат суперкомпиляции после применения препроцессора;

Для достижения этих целей должны быть решены следующие задачи:

- определить факторы, влияние на которые может позволить улучшить результаты суперкомпиляции;
- разработать и реализовать препроцессор, позволяющий влиять на выделенные факторы;
- сравнить результаты суперкомпиляции исходных программ и соответствующих препроцессированных программ.

2. Суперкомпиляция

Суперкомпиляция выполняет преобразование программ путём моделирования процесса выполнения программы для некоторого множества неизвестных данных. После выполнения суперкомпиляции получается остаточная программа, которая может обладать какими-нибудь интересными свойствами.

Одним из применений суперкомпиляции является специализация программ [6]. Под специализацией подразумевается преобразование исходной программы, для которой часть входных данных известна, таким образом, чтобы получившаяся программа зависела бы только от неизвестных входных данных исходной программы, а процесс её выполнения описывался бы с некоторых известных свойств о неизвестной части входных данных.

Ещё одним применением суперкомпиляции можно назвать верификацию программ [7]. Пусть дана некоторая функция F , результат этой функции может иметь некоторое свойство, которое нужно проверить. Для того, чтобы это сделать пишется функция $Pred$, которая проверяет это свойство и возвращает $True$, если свойство выполняется или $False$, если не выполняется. Затем производится суперкомпиляция композиции $\langle Pred \langle F \ e.0 \rangle \rangle$, где $e.0$ – какие-то входные данные для функции F . Если остаточная программа не содержит символа $False$, то свойство считается доказанным.

Рассмотрим общие принципы действий, выполняемых при суперкомпиляции.

2.1. Прогонка

Прогонка параметрической конфигурации является одним из основных действий, выполняемых суперкомпилятором [8]. Под параметрической конфигурацией понимается параметризованное (частично неизвестное) состояние стека вызовов функций суперкомпилируемой программы.

Под прогонкой понимается выполнение программы не для конкретных данных, а для некоторых параметризованных выражений, то есть можно сказать, что программа выполняется в общем виде. Прогонка продельывает разбор всех

возможных в данной конфигурации вычислений, и затем на основе этого строит конечное дерево, называемое гроздью прогонки. Ребра этого дерева упорядочены слева направо и означают последовательность выбора вариантов вычисления, вершинами являются результирующие конфигурации.

Рассмотрим первый этап прогонки для программы, представленной в листинге 1. Получившаяся гроздь прогонки представлена на рисунке 1.

Листинг 1 – Программа замены литеры 'a' на 'b'

```
Fab {
  (e.Res) 'a' e.X = <Fab ('b') e.X> ;
  (e.Res) s.1 e.X = <Fab (s.1) e.X>;
  (e.Res) = e.Res ;
}
```

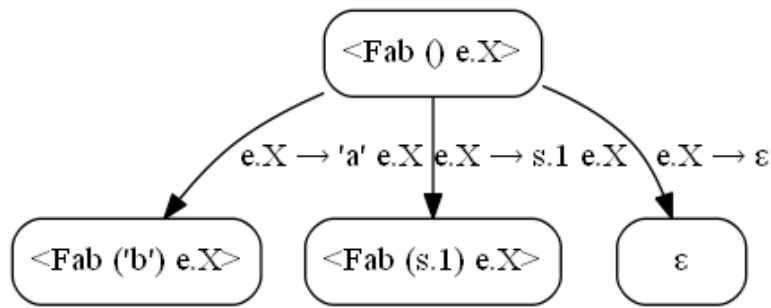


Рисунок 1 – Пример прогонки

Если гроздь содержит вершину, из которой выходит только одно ребро, помеченное некоторым условием перехода, то эта вершина называется полутранзитной. Если это единственное ребро, выходящее из вершины, не помечено никаким условием, тогда эта вершина называется транзитной. Суперкомпилятор может выполнять оптимизацию, посредством которой все полутранзитные вершины удаляются, а входящее в вершину и исходящее из неё ребро соединяются, то есть два ребра объединяются в одно (рисунки 2, 3).

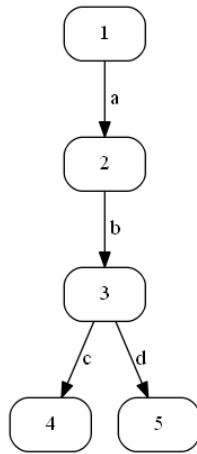


Рисунок 2

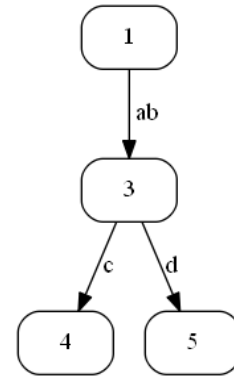


Рисунок 3

Так как в процессе прогонки порождаются новые параметрические конфигурации, то, не смотря на конечность каждой грозди, процесс прогонки может продолжаться бесконечно.

2.2. Свёртка

Процесс суперкомпиляции должен быть конечным и давать в итоге некоторую остаточную программу, однако в общем случае дерево, которое строится посредством просмотра всех возможных вычислений программы, может получиться бесконечным. Это означает, что необходимы способы приведения потенциально бесконечного дерева конфигураций к конечному графу.

Под свёрткой будем понимать процесс сведения бесконечного дерева к конечному графу. Основными инструментами свёртки являются вложение и обобщение [9].

2.2.1. Вложение

В процессе построения дерева могут возникать конфигурации, которые можно свести друг к другу с помощью подстановки параметров.

Пусть C_1 и C_2 – параметризованные конфигурации, и C_2 можно свести к C_1 подстановкой параметров, тогда конфигурация C_2 вкладывается в конфигурацию C_1 .

Стоит отметить, что конфигурация может представлять собой не только единственный вызов функции, но и стек функций. Пусть стек функций разделён

на две части: начальный отрезок ненулевой длины и оставшуюся часть. Для вложения ищется предыдущая конфигурация, для которой существует подстановка параметров, сводящая описание предыдущего стека к описанию начального отрезка рассматриваемого стека. Таким образом начальный отрезок вкладывается в предыдущую конфигурацию, а оставшаяся часть определяет отдельную задачу на суперкомпиляцию.

Вложение означает, что при помощи подстановки параметров все возможные вычисления, определяемые текущей конфигурацией, можно свести ко всем возможным вычислениям, которые были определены ранее.

Рассмотрим пример вложения на основе программы, описанной в листинге 2.

Листинг 2 – Функция-пример

```
Func {
  (e.1 'I') 'X' = (e.1) 'X';
  () 'X' = 'X';
}
```

Если строить дерево конфигураций для данной программы без использования вложений, то будет получен результат, представленный на рисунке 4.

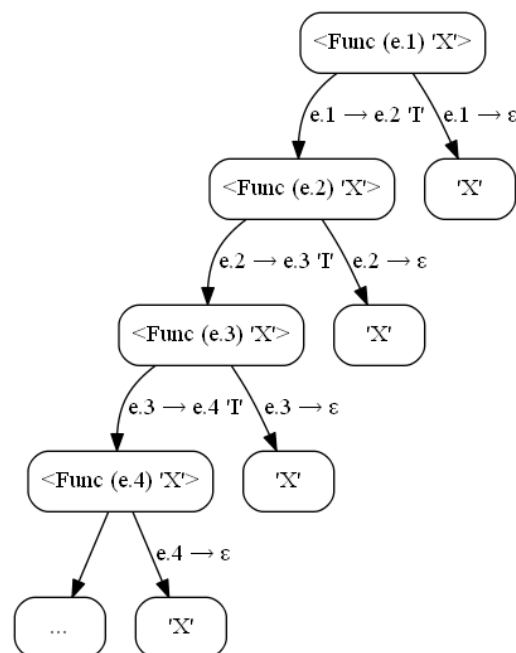


Рисунок 4 – Развитие дерева конфигураций для функции Func

Можно заметить, что это дерево будет расти бесконечно.

Рассмотрим теперь вариант с вложением. Конфигурацию $\langle \text{Func } (e.2) \text{ 'X'} \rangle$ можно получить из конфигурации $\langle \text{Func } (e.1) \text{ 'X'} \rangle$ с помощью подстановки $e.2 \leftarrow e.1$. После этого будет получен конечный граф, представленный на рисунке 5.

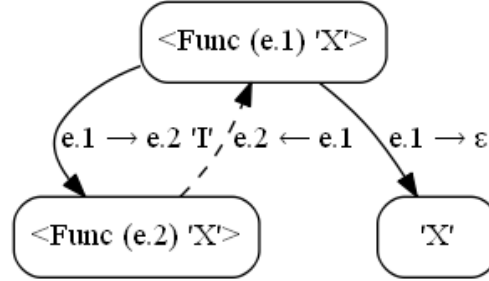


Рисунок 5 – Конечный граф конфигураций для функции Func

Можно увидеть, что при вложении в графе появляются обратные стрелки, то есть образуются циклы. Поэтому свёртка графа с помощью нахождения вкладывающихся конфигураций называется зацикливанием.

2.2.2. Обобщение

В процессе суперкомпиляции могут появляться конфигурации, которые нельзя получить друг из друга прямой подстановкой, но которые являются в некотором смысле «похожими».

Рассмотрим отношение похожести, называемое отношение Турчина. Данное отношение ищет похожие по своей функциональной структуре стеки.

Пусть каждому вызову функции F приписано время его создания t в процессе развития дерева конфигураций, обозначим это как F_t . Пусть даны два стека предыдущий Prev и текущий Curr , тогда эти стеки похожи по отношению Турчина, если их можно представить в виде

$\text{Prev} = \text{PrevTop}; \text{Context};$

$\text{Curr} = \text{CurrTop}; \text{Middle}; \text{Context};$

Дно стеков Context является некоторой общей частью максимальной длины, а вершины стеков совпадают с точностью до времён создания вызовов. В этом случае данные стеки будут похожими по отношению Турчина.

Вершину PrevTop можно назвать точкой входа в цикл, Middle содержит путь от предыдущего состояния стека к текущему, Context – вычисления, которые будут произведены после завершения цикла. Будем называть PrevTop и CurrTop префиксами.

Отношение Турчина предполагает, что, начиная с текущей конфигурации, процесс возможных вычислений будет повторяться, а значит текущий префикс необходимо свести к предыдущему с помощью подстановки параметров. Однако, такой подстановки может не существовать, поэтому необходимо отношение похожести, которое позволяло бы сравнивать параметрическое описание аргументов разных вызовов одной функции. Роль такого отношения играет отношение Хигмана-Крускала.

Пусть родительская конфигурация описывается выражением E_1 , а дочерняя выражением E_2 , тогда эти конфигурации похожи по Хигману-Крускалу, если в E_2 можно стереть некоторые элементы (символы, пары скобок, переменные и т.д.) таким образом, что получится E_1 .

Таким образом, если для префиксов соответствующие элементы похожи по отношению Хигмана-Крускала, то производится их «обобщение», то есть строится конфигурация, в которую вкладываются оба префикса.

На рисунке 6 представлен пример отношения Хигмана-Крускала.

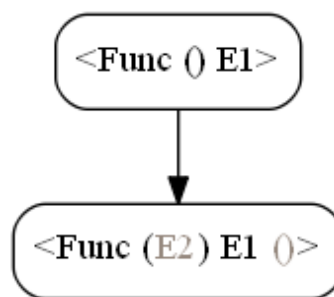


Рисунок 6 – Отношение Хигмана-Крускала

Если в конфигурации $\langle Func (E2) E1 () \rangle$ стереть $E2$ и вторые скобки. То получим конфигурацию $\langle Func () E1 \rangle$.

Таким образом с помощью обобщения можно свернуть дерево, в котором изначально нет вложений.

3. Разработка

3.1. Первый препроцессор

Изначальной задачей было написание препроцессора, который позволял бы улучшать результаты суперкомпиляции при помощи учёта конструкторов данных в отношении Турчина. Данную гипотезу не удалось подтвердить, так как не удалось найти примеры, которые бы показывали влияние препроцессора именно на отношение Турчина. Однако, именно эта задача является основой для выбора преобразований программы, которые осуществляет первый препроцессор.

Изначальными преобразованиями, которые должен был делать первый препроцессор были: замена константных символов и круглых скобок на вызовы функций в результатных выражениях.

Эти преобразования позволяют вынести аргументы функции в стек имён функций, что теоретически должно повлиять на отношение Турчина, так как изначально аргументы никак не участвуют в этом отношении. На самом деле оказалось, что эти функций практически не задерживаются в стеке из-за простоты их структуры. То есть суперкомпилятору нет необходимости держать эти функции в стеке, так как при необходимости он может сразу вычислить результат их работы. Из-за этого подбор примера стал достаточно нетривиальной задачей, которая в рамках этой работы не была решена.

Несмотря на неудачу с отношением Турчина первый препроцессор всё же показал своё влияние на результаты суперкомпиляции. Ответ лежал в отношении Хигмана-Крускала. Замены, осуществляемые препроцессором, позволили сохранять информацию о происхождении конструкторов, что позволяет делать более точные обобщения.

Для большего влияния на процесс суперкомпиляции было решено добавить возможность управлять совершаемыми преобразованиями, а также добавить преобразование, заключающееся в добавлении вызовов функций, возвращающих пустое выражение (т.е. «строку длины 0»).

3.2. Второй препроцессор

Так как первый препроцессор изначально создавался для улучшения результатов для отношения Турчина, а его влияние на отношение Хигмана-Крускала было найдено в процессе исследования, значит можно разработать препроцессор, задачей которого будет улучшение результатов именно для отношения Хигмана-Крускала.

Проблема первого препроцессора, заключающаяся в том, что конструкторы функций легко стираются и «забываются», оказывает влияние и на отношение Хигмана-Крускала. Рассмотрим пример, представленный в листинге 3, показывающий это.

Листинг 3 – Функция-пример

```
F {  
    (e.X) e.Y = RightPart;  
    e.Z = <G e.Z>;  
}  
c3 { = True }
```

Рассмотрим конфигурацию $\langle F \langle c3 \rangle e.1 s.2 \rangle$. Для того чтобы продолжить работу супекомпилятор должен будет сопоставить аргумент с первым предложением – $\langle c3 \rangle e.1 s.2 : (e.X) e.Y$. Так как аргумент начинается с вызова функции, то суперкомпилятор попытается вычислить значение этой функции и в результате получим конфигурацию $\langle F True e.1 s.2 \rangle$. Первое предложение неприменимо, поэтому выполнится второе предложение. После этого получается конфигурация $\langle G True e.1 s.2 \rangle$. Для наглядности распишем всю цепочку: $\langle F \langle c3 \rangle e.1 s.2 \rangle \rightarrow \langle F True e.1 s.2 \rangle \rightarrow \langle G True e.1 s.2 \rangle \rightarrow \dots$ Информация о происхождении символа True была утеряна.

Для того чтобы сохранить информацию каждый символ можно заменить на пару символ-координата, а каждый скобочный терм – на пару терм-координата.

В программе это будет представляться следующим образом: каждый символ может заменяться на пару «символ-координата», которая представляется в виде скобочного терма (*значение координата*), где координата – это идентификатор вида K-N, где N = 1, 2, 3 ..., а скобочные термы представляются в виде (*значение*) \rightarrow ((*значение*) координата).

Для подробного представления работы препроцессора опишем две функции TP (transform pattern) – для преобразования образцов и TR (transform result) – для преобразования результатов. Также добавим функцию NEXT(), которая вычисляет новый символ для координаты (K-1, K-2, K-3 ...). Запись s.NEXT() означает генерацию переменной с новым уникальным индексом, например, s. K-1.

Рассмотрим для начала функцию преобразования образцов:

- $TP[\varepsilon] = \varepsilon$ – пустое выражение отображается в пустое выражение;
- $TP[sym] = (sym \ s.NEXT())$ – для символов в образце генерируется переменная с новым индексом;
- $TP[(expr)] = ((TP[expr]) \ s.NEXT())$ – для скобочных выражений в образце генерируем переменную с новым индексом и рекурсивно обрабатываем выражение внутри скобок;
- $TP[\ e.X \] = e.vX$, $TP[\ t.X \] = t.vX$ – у t- и e-переменных к индексу приписывается буква v;
- $TP[s.X] = (s.vX \ s.NEXT())$ – для s-переменных генерируется новая переменная, которая будет сопоставляться с координатой;

Теперь рассмотрим функцию преобразования результата:

- $TR[\varepsilon] = \varepsilon$ – пустота отображается в пустоту;
- $TR[sym] = (sym \ NEXT())$ – для символов вычисляется новая координата;
- $TR[(expr)] = ((TR[expr]) \ NEXT())$ – для скобочных выражений, вычисляем новую координату и рекурсивно обрабатываем выражение внутри скобок;
- $TR[\ e.X \] = e.vX$, $TR[\ t.X \] = t.vX$ – у t- и e-переменных к индексу приписывается буква v;
- $TR[s.X] = (s.vX \ LOOKUP[s.X])$ – для s-переменных в результатных выражениях запись LOOKUP[s.X] обозначает поиск в текущей области видимости имени переменной-координаты, соответствующей заданной s-переменной;

Такой препроцессор позволяет лучше сохранять информацию в конфигурации, однако усложняет остаточные программы из-за появления новых переменных. Но это усложнение не всегда влияет на сложность анализа остаточной программы.

Этот препроцессор также предоставляет возможность управлять совершаемыми преобразованиями, а именно можно добавлять уникальные координаты только символам или только скобочным термам.

4. Реализация

4.1. Лексический анализатор

Лексический анализ реализован в функции `Lexex`, которая преобразует текст исходной программы в последовательность лексем, которые представляются скобочными термами вида `(s.Type t.Pos e.Rep)`, где `s.Type` — тип токена, `t.Pos` — позиция вида `(s.Row s.Col)`, где `s.Row` — номер строки, `s.Col` — номер колонки, `e.Rep` — текстовое представление токена.

В Рефале-5 можно выделить следующие лексические домены:

- разделяющие символы – пробел, табуляция, перевод строк;
- однострочные комментарии – строки, у которых в первой позиции находится символ `*`;
- многострочные комментарии – набор строк, заключенный в `/* ... */`;
- идентификаторы – начинаются с латинской буквы, далее могут состоять из латинских букв, цифр, а также знаков `«_»` и `«-»`;
- целые числа – последовательности десятичных цифр;
- знаки пунктуации: `«{»`, `«}»`, `«(»`, `«)»`, `«>»`, `«.»`, `«;»`, `«:»`, `«=»`;
- левые скобки вызова `<Func`, где `Func` – идентификатор, обозначающий имя вызываемой функции или один из знаков арифметических операций;
- ключевые слова: `$ENTRY`, `$EXTERN`, `$EXTERNAL`, `$EXTRN`;
- переменные – состоят из типа переменной, точки и индекса, где тип переменной обозначается одной из букв `«s»`, `«t»` или `«e»`, а индексом является непустая последовательность из букв, цифр, прочерков и дефисов;
- литеры – символы, заключенные в одинарные кавычки. Последовательность подряд идущих литер может находиться внутри одной пары кавычек, внутри пары кавычек допустимы следующие escape-последовательности `\r`, `\n`, `\t`, `\\`, `\'`, `\"`, `\<`, `\>`, `\(`, `\)`, `\xHH`;

- составные символы – заключены в двойные кавычки и считаются одним монолитным символом, такие символы могут содержать те же escape-последовательности, что и литеры;

Опишем работу функции $\langle \text{Lexer } t.\text{Pos } (e.\text{Line}) e.\text{Lines} \rangle == e.\text{Tokens}$, где $t.\text{Pos}$ обозначает позицию текущего рассматриваемого символа, $(e.\text{Line})$ содержит текущую строку, а $e.\text{Lines}$ – набор еще не рассмотренных строк исходной программы.

Сначала задаются начальные значения: для $t.\text{Pos}$ – $(1\ 1)$, а $(e.\text{Line}) e.\text{Lines}$ – содержат весь текст исходной программы, разбитый построчно на последовательность скобочных термов. Таким образом $(e.\text{Line})$ будет содержать первую строку исходной программы, а $e.\text{Lines}$ все последующие.

Затем идёт посимвольный анализ текущей рассматриваемой строки, для лексических доменов, представленных одиночными символами, как например знаки пунктуации, токен задаётся сразу, а для более сложных вызываются вспомогательные функции, позволяющие определять принадлежность символа или последовательности символов к лексическому домену.

Для каждого токена $t.\text{Pos}$ присваивается местоположение первого символа, а $e.\text{Rep}$ полностью повторяет представление токена в исходном тексте программы, а для ошибок содержит их описание.

Рассмотрим вспомогательные функции:

- $\langle \text{LexKeyword } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа ключевых слов, переход в эту функцию осуществляется при встрече в тексте исходной программы символа «\$», затем эта функция проверяет является ли последовательность далее идущих символов ключевым словом. Если является, то $s.\text{Type}$ присваивается одно из следующих значений в зависимости от ключевого слова: Entry, Extern, External, Extn, иначе $s.\text{Type}$ присваивается значение Error;
- $\langle \text{LexVar } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа переменных, переход в эту функцию осуществляется при встрече

в тексте исходной программы одного из символов «s», «t», «e» и следующим за ним символа «.». Данная функция проверяет правильность индекса переменной. Если индекс является правильным, то s.Type присваивается значение Variable, иначе s.Type присваивается значение Error;

- $\langle \text{LexIdent } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа идентификаторов, переход в эту функцию осуществляется при встрече в тексте исходной программы буквы латинского алфавита и проверяет далее идущие символы на возможность принадлежности идентификатору. Как только встречается недопустимый символ, s.Type присваивается значение Ident и функция прекращает работу;
- $\langle \text{LexLeftCall } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа левой скобки вызова, переход в эту функцию осуществляется при встрече в тексте исходной программы символа «<», данная функция проверяет можно ли из идущих далее символов составить идентификатор или является ли следующий символ арифметической операцией. Если имя функции является пустым, то s.Type присваивается значение Error, иначе s.Type присваивается значение LeftCall;
- $\langle \text{LexBlockComment } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (\text{BlockComment } t.\text{Pos } e.\text{Rep})$
 $== (\text{Error } t.\text{Pos } e.\text{Rep}) (\text{EOF } t.\text{Pos EOF})$ – функция анализа многострочных комментариев, переход в эту функцию осуществляется при встрече в тексте исходной программы символов «/*», в функции просматриваются символы до тех пор, пока не будет встречена последовательность «*/», обозначающая конец комментария или пока не будет встречен конец файла. В первом случае, будет возвращен первый формат результата, а во втором второй. Также при втором варианте будет завершена работа лексического анализатора;

- $\langle \text{LexNumber } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа целых чисел, переход в эту функцию осуществляется при встрече в тексте исходной программы десятичной цифры. Функция просматривает далее идущие символы пока они являются десятичными числами, и как только встречается какой-либо другой $s.\text{Type}$ присваивается значение Number и функция прекращает работу;
- $\langle \text{LexChars } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})^+$ – функция анализа литер, переход в эту функцию осуществляется при встрече в тексте исходной программы символа «'». Функция просматривает символы, пока не встретит «'». Для каждого символа создаётся отдельный скобочный терм, в котором $s.\text{Type}$ присваивается значение Char . Также в функции проверяется допустимость escape-последовательностей и отсутствие перевода строки внутри последовательности. Если была встречена недопустимая escape-последовательность или перевод строки, то создаётся скобочный терм, в котором $s.\text{Type}$ присваивается значение Error , однако в случае escape-последовательности процесс обработки последовательности литер продолжается, а при встрече перевода строки он прекращается;
- $\langle \text{LexCompound } t.\text{Pos } (e.\text{Scanned}) (e.\text{Line}) e.\text{Lines} \rangle == (s.\text{Type } t.\text{Pos } e.\text{Rep})$ – функция анализа составных символов, переход в эту функцию осуществляется при встрече в тексте исходной программы символа «"». Работа функции похожа на работу функции LexChars . С тем исключением, что скобочный терм создается для всей последовательности символов, а $s.\text{Type}$ присваивается значение Compound .

4.2. Первый препроцессор

Первый препроцессор преобразует программы, написанные на языке Рефал-5 следующим образом: в зависимости от выбора пользователя в результатных выражениях круглые скобки и константные символы, могут заменяться на вызовы функций, а также может добавляться вызов функций, возвращающих пустое выражение, после каждого токена результатного выражения.

Первым пунктом работы препроцессора является проверка наличия специального псевдокомментария `*$PREPROCESS1` и анализ его аргументов, на основе этих аргументов выбирается способ преобразования программы. Синтаксис псевдокомментария подробнее описан в разделе 6.2. Если комментарий не был найден, или он содержал недопустимые аргументы, то препроцессор выдаёт ошибку, возвращает исходный текст программы без изменений и прекращает свою работу.

Если комментарий препроцессирования был успешно найден и проанализирован, то далее начинает работу функция `<Preprocess (e.Type) e.Tokens> == e.TokensResult`, где `e.Type` содержит идентификаторы `e`, `s`, `b` в любых сочетаниях. Данные идентификаторы указывают на то, какого рода преобразования нужно применить к исходной программе: `e` – добавление вызовов пустых функций, `s` – замена константных символов на вызов функции, `b` – замена круглых скобок на вызовы функций. `e.Tokens` содержит последовательность токенов, которая была получена в результате лексического анализа, `e.TokensResult` содержит преобразованную последовательность токенов, на основе которой можно восстановить препроцессированную программу.

Функция `Preprocess` при необходимости разделяет препроцессирование на два прохода, в первом выполняется преобразование программы, связанное с добавлением вызовов пустых функций, а во втором преобразование, связанное с заменой константных символов и скобок.

У каждого прохода есть два режима анализа: первый для пропуска токенов, не относящихся к результатным выражениям и второй для анализа результатного выражения и проведения необходимых изменений. Переход из одного режима в другой происходит при встрече определённого токена.

Для прохода, осуществляющего преобразования, связанные с добавлением вызовов пустых функций, режимы описываются функциями `<PreprocessEmpty s.NextEmpty t.CurToken e.Tokens> == e.TokensResult` для первого режима и `<ResultPrepEmpty s,NextEmpty t.CurToken e.Tokens> == e.ChangedTokens` для

второго, где `s.NextEmpty` является целым числом, представляющим собой номер следующей добавленной функции, `t.CurToken` – текущий рассматриваемый токен, `e.Tokens` – список еще не обработанных токенов, `e.TokensResult` – токены, на основе которых можно восстановить препроцессорованную программу, `e.ChangedTokens` – набор токенов, описывающих изменённое результатное выражение.

Функция `PreprocessEmpty` пропускает все токены, пока `t.CurToken` не окажется токеном, соответствующим символам «=» или «,», после этого за текущим токеном добавляется токен, описывающий пустую функцию и вызывается функция `ResultPrepEmpty` обработки результатного выражения, для которой значение `s.NextEmpty` будет увеличено на единицу. Так как символ «,» может использоваться для разделения аргументов после ключевого слова `$EXTERN`, то во избежание путаницы все токены, содержащиеся между токеном, соответствующим ключевому слову, и токеном, соответствующим символу «;», пропускаются одновременно.

Функция `ResultPrepEmpty` после каждого токена, не описывающего комментарий, пробельный символ или символы «;», «:», «}», добавляет токен пустой функции и увеличивает значение переменной `s.NextEmpty` на единицу. Токены, описывающие комментарии и пробельные символы, пропускаются, а токены, описывающие символы «;», «:», «}», сигнализируют о переходе в первый режим.

После того как будут просмотрены все токены вызывается функция `<MakeWrappersEmpty s.Num> == e.Tokens`, которая составляет токены, описывающие строки программы, в которых реализованы пустые функции. `s.Num` – число еще не реализованных функций, `e.Tokens` – список составленных функцией токенов.

Для прохода, осуществляющего преобразования, связанные с заменой константных символов и скобок на вызовы функций, режимы описываются функциями `<PreprocessSymBr (e.Type) s.NextBrac s.NextSym (e.Table) t.CurToken e.Tokens> == e.TokensResult` для первого режима и `<ResultPrepSymBr`

(e.Type) s.NextBrac s.NextSym (e.Table) t.CurToken e.Tokens> == e.ChangedTokens для второго, где e.Type описывает какие замены должны быть осуществлены, s.NextBrac и s.NextSym – целые числа, представляющие собой номер следующей функции, заменяющей скобку или символ соответственно, e.Table представляет собой набор скобочных термов вида (s.Num s.Type), s.Num – номер функции, s.Type – тип замены, t.CurToken – текущий рассматриваемый токен, e.Tokens – список еще не обработанных токенов, e.TokensResult – токены на основе которых можно восстановить препроцессированную программу, e.ChangedTokens – набор токенов, описывающих изменённое результатное выражение.

Работа функции PreprocessSymBr практически полностью повторяет работу функции PreprocessEmpty, с тем исключением, что не производится добавления токенов пустых функций.

Работа функции ResultPrepSymBr зависит от того какие идентификаторы содержатся в переменной e.Type. Если e.Type содержит идентификатор b, то в процессе работы ищутся токены, соответствующие символам «(» и «)». При обнаружении токена соответствующего символу «(» он заменяется на токен, соответствующий левому вызову функции, значение переменной s.NextBrac увеличивается на единицу, а в (e.Table) добавляется скобочный терм вида ('b' s.NextBrac). Для токена соответствующего символу «)» производится его замена на токен, соответствующий символу «>». Если e.Type содержит идентификатор a, то ищутся токены, описывающие целые числа, литеры, составные символы и идентификаторы, каждый такой токен заменяется на пару токенов, соответствующих левому вызову функции и символу «>», значение переменной s.NextSym увеличивается на единицу, а в (e.Table) добавляется скобочный терм вида ('c' s.NextSym e.Rep), где e.Rep – текстовое представление заменяемого токена.

После того, как будут просмотрены все токены вызывается функция <MakeWrappers t.Cur e.Table> == e.Tokens, которая составляет токены, описывающие строки программы, в которых реализованы функции, на которые

были заменены круглые скобки и константные символы. $t.Cur$ – текущий элемент $e.Table$, описывающий функцию, которую необходимо добавить, $e.Table$ – последовательность еще не добавленных функций, $e.Tokens$ – список составленных функцией токенов.

4.3. Второй препроцессор

Второй препроцессор преобразует программы, написанные на языке Рефал-5 следующим образом: каждый символ может заменяться на пару «символ-координата», которая представляется в виде скобочного термина (*значение координата*), где координата – это идентификатор вида $K-N$, где $N = 1, 2, 3 \dots$, а скобочные термины представляются в виде (*значение*) \rightarrow ((*значение*) *координата*).

Препроцессор начинает свою работу с проверки наличия специального псевдокомментария $*\$PREPROCESS2$ и затем, на основе аргументов данного псевдокомментария, определяется, какие преобразования необходимо применить к программе. Синтаксис псевдокомментария подробнее описан в разделе 6.2. Если псевдокомментарий отсутствует или содержал ошибки, то препроцессор выдаёт ошибку, возвращает исходный текст программы без изменений и прекращает свою работу.

После успешного обнаружения и анализа псевдокомментария $*\$PREPROCESS$, начинает работу функция $\langle Preprocess(e.Type) s.GlobalCount(e.Stack) e.Tokens \rangle == e.TokensResult$, где $e.Type$ – описывает тип изменений, которые нужно применить к исходной программе, $s.GlobalCount$ – целое число, использующееся в качестве нумерации идентификаторов-координат и переменных-координат, $e.Stack$ – содержит скобочные термины, состоящие из пар $(s.X s.K-NUM)$, $s.X$ – значение переменной, $s.K-NUM$ – переменная-координата для переменной $s.X$, $e.Tokens$ – список еще не обработанных токенов, $e.TokensResult$ – список токенов, по которым можно восстановить препроцессированную программу.

У препроцессора есть три режима анализа токенов: первый – пропуск токенов, находящихся вне определения функции, второй – обработка токенов,

находящихся в образце и третий – обработка токенов, находящихся в результирующем выражении.

Первый режим реализован в самой функции `Preprocess`, в которой пропускаются все токены, пока не будет встречен токен, соответствующий символу «{», так как этот токен означает, что дальше последуют токены, относящиеся к определению функции. После того как данный токен будет встречен происходит переход во второй режим – режим обработки образцов.

Режим обработки образцов реализован в функции `<Pattern (e.Type) s.GlobalCount (e.Stack) e.Tokens> == e.TokensResult`, где значение всех переменных, являющихся аргументами функции соответствует их значению в функции `Preprocess`, а `e.TokensResult` – это список токенов, описывающих изменения в образце функции.

Функция `Pattern` работает следующим образом:

- при встрече токена, соответствующего символу «(», этот токен заменяется на пару токенов соответствующих «(»;
- при встрече токена, соответствующего символу «)», этот токен заменяется на последовательность из трёх токенов, где первый и последний являются токенами, соответствующими символу «)», а второй токен описывает s-переменную, являющуюся координатой. Значение `s.GlobalCount` увеличивается на единицу. Таким образом добавляются переменные-координаты к скобочным выражениям;
- при встрече токена, соответствующего целому числу, литере, составному символу или идентификатору, он заменяется на последовательность из четырех токенов, где первый и последний соответствуют символам «(» и «)», на место второго токена копируется заменяемый, а третий описывает s-переменную, являющуюся координатой. Затем значение `s.GlobalCount` увеличивается на единицу;
- при встрече токена, соответствующего s-переменной, производится его замена на последовательность из четырех токенов, где первый и последний

соответствуют символам «(» и «)», второй токен почти повторяет заменяемый, с тем исключением, что ему в e.Rep после точки приписывается символ «v», третий описывает s-переменную, являющуюся координатой. После этого значение s.GlobalCount увеличивается на единицу, в вершину стека помещается пара, содержащая e.Rep текущего токена и определённую для него переменную-координату;

- при встрече токена, соответствующего t- или e-переменной, этому токenu в e.Rep перед индексом приписывается символ «v»;
- при встрече токена, соответствующего символу «{», на вершину стека помещается пустой элемент;
- при встрече токена, соответствующего символу «=» или «,», препроцессор переходит в режим обработки результатных выражений;
- при встрече токена, соответствующего символу «}», со стека снимается верхний элемент, если после этого стек стал пустым, то осуществляется переход в режим обработки токенов вне функции, иначе осуществляется переход в режим обработки результатных выражений;
- токены, соответствующие разделяющим символам и комментариям, не изменяются;
- для остальных токенов выдаётся ошибка;

Режим обработки результатных выражений реализован в функции <Result (e.Type) s.GlobalCount (e.Stack) e.Tokens> == e.TokensResult, где значение всех переменных, являющихся аргументами функции соответствует их значению в функции Preprocess, а e.TokensResult – это список токенов, описывающих изменения в результатном выражении.

Функция Result работает следующим образом:

- при встрече токена, соответствующего символу «(», этот токена заменяется на пару токенов соответствующих «(»;
- при встрече токена, соответствующего символу «)», этот токен заменяется на последовательность из трёх токенов, где первый и последний являются

токенами, соответствующими символу «)», а второй токен описывает идентификатор, являющийся координатой. Если переменная `e.Type` содержит идентификатор `b`, то каждому скобочному выражению будет соответствовать уникальный идентификатор-координата, иначе идентификатор-координата будет одинаковым для всех скобочных выражений. Значение `s.GlobalCount` увеличивается на единицу, в случае уникальных идентификаторов-координат, иначе не изменяется;

- при встрече токена, соответствующего целому числу, литере, составному символу или идентификатору, он заменяется на последовательность из четырех токенов, где первый и последний соответствуют символам «(» и «)», на место второго токена копируется заменяемый, а третий описывает `s`-переменную, являющуюся координатой. Если переменная `e.Type` содержит идентификатор `s`, то идентификатор-координата будет уникальным, иначе идентификатор-координата будет одинаковым для всех токенов такого вида. Значение `s.GlobalCount` увеличивается на единицу, в случае уникальных идентификаторов-координат, иначе не изменяется;
- при встрече токена, соответствующего `s`-переменной, производится его замена на последовательность из четырех токенов, где первый и последний соответствуют символам «(» и «)», второй токен почти повторяет заменяемый, с тем исключением, что ему в `e.Rep` после точки приписывается символ «v», третий описывает `s`-переменную, являющуюся координатой. Если переменная `e.Type` содержит идентификатор `s`, то идентификатор-координата будет искаться в стеке по значению `e.Rep`. При наличии в стеке нескольких таких элементов, будет выбран первый из них. Иначе идентификатор-координата будет одинаковым для всех `s`-переменных и совпадать с идентификатором-координатой, определённым для набора токенов из предыдущего пункта.
- при встрече токена, соответствующего `t`- или `e`-переменной, этому токenu в `e.Rep` перед индексом приписывается символ «v»;

- при встрече токена, соответствующего символу «;» вершина стека удаляется, а затем на вершину помещается пустой элемент, и препроцессор переходит в режим анализа образца;
- при встрече токена, соответствующего символу «:» препроцессор переходит в режим анализа образца;
- при встрече токена, соответствующего символу «}», со стека снимается верхний элемент, если после этого стек стал пустым, то осуществляется переход в режим обработки токенов вне функции, иначе осуществляется переход в режим обработки результатных выражений;
- токены, соответствующие разделяющим символам, комментарию, левому вызову функции и символу «>» не изменяются;
- для остальных токенов выдаётся ошибка;

5. Примеры

В данном разделе будут рассмотрены примеры суперкомпиляции исходных и соответствующих препроцессированных программ. Для обоих препроцессоров примеры рассматриваются в режиме изменений, влияющих на символы и скобки.

Для суперкомпиляции будут использоваться суперкомпиляторы SCP4 и MSCP-A. Так как примеры подробно рассматривались для суперкомпилятора SCP4, то будет подразумеваться, что процесс суперкомпиляции рассматривается именно для этого суперкомпилятора. Если суперкомпиляция рассматривается для MSCP-A, то это будет сказано отдельно.

5.1. Программа замены литеры 'A' на литеру 'B'

Рассмотрим программу, представленную в листинге 4.

Листинг 4 – Программа замены литеры 'A' на литеру 'B'
\$ENTRY Fab { e.X = <DoFab e.X ('B')> }

```
DoFab {  
    e.X 'A' (e.Res) = <DoFab e.X ('B' e.Res)>;  
    e.X s.1 (e.Res) = <DoFab e.X (s.1 e.Res)>;  
    /* empty */ (e.Res) = e.Res;  
}
```

Данная программа производит замену литеры 'A' на литеру 'B', причем изначально в накопитель *e.Res* кладётся литера 'B'. Таким образом, в результате последним элементом всегда будет литера 'B'.

Рассмотрим процесс суперкомпиляции данной программы (рисунок 7).

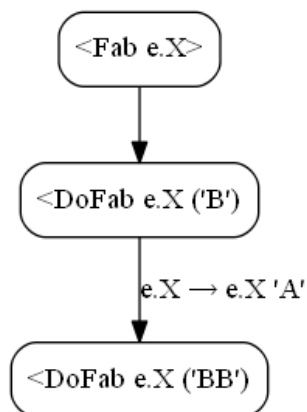


Рисунок 7 – Граф конфигураций исходной программы

Конфигурации $\langle DoFab\ e.X\ ('B') \rangle$ и $\langle DoFab\ e.X\ ('BB') \rangle$ являются похожими по отношению Хигмана-Крускала, следовательно, они должны быть обобщены. В данном случае возможно два обобщения: $\langle DoFab\ e.0\ ('B'\ e.1) \rangle$ и $\langle DoFab\ e.0\ (e.1\ 'B') \rangle$. У суперкомпилятора нет возможности понять, какой из этих вариантов лучше, поэтому предположим, что будет выбран первый вариант.

В результате получаем граф суперкомпиляции, представленный на рисунке 8.

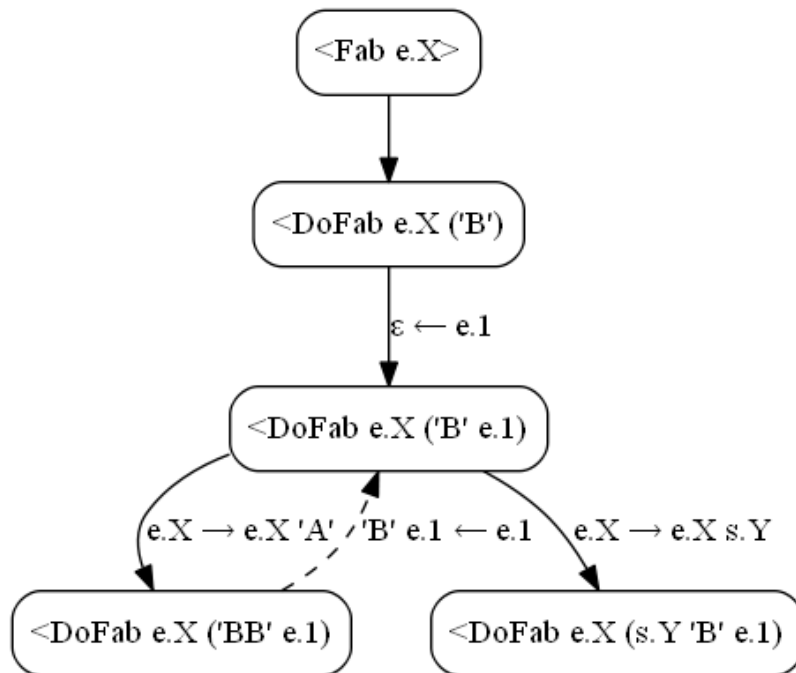


Рисунок 8 – Граф конфигураций исходной программы

Конфигурация $\langle DoFab\ e.X\ ('BB'\ e.1) \rangle$ получается подстановкой из конфигурации $\langle DoFab\ e.X\ ('B'\ e.1) \rangle$, поэтому происходит заикливание данной ветки.

Далее рассмотрим вторую ветку. Можно заметить, что конфигурации $\langle DoFab\ e.X\ ('B'\ e.1) \rangle$ и $\langle DoFab\ e.X\ (s.Y\ 'B'\ e.1) \rangle$ похожи по Хигману-Крускалу, следовательно, нужно их обобщить. Обобщим эти конфигурации следующим образом: $\langle DoFab\ e.X\ (s.3\ e.1) \rangle$. Граф, получающий после применения обобщения, представлен на рисунке 9.

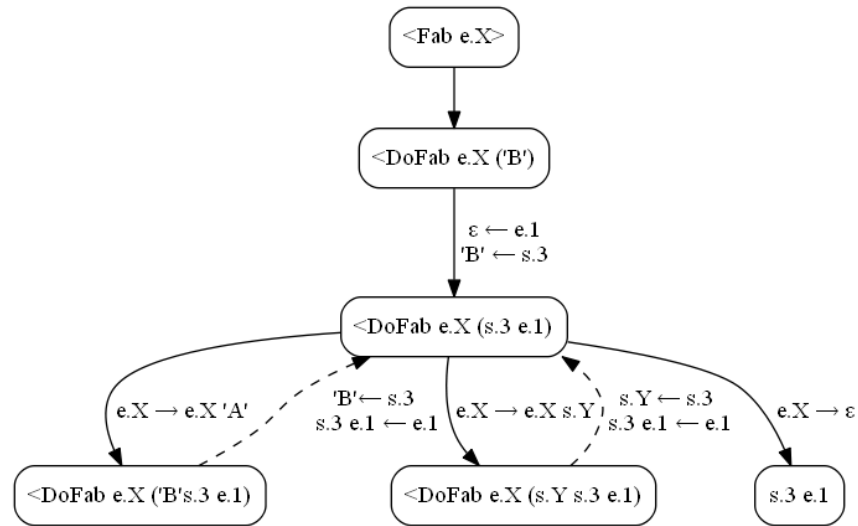


Рисунок 9 – Итоговый граф конфигураций исходной программы

Остаточная программа, полученная с помощью SCP4, очевидно будет отличаться от остаточной программы, полученной ручной суперкомпиляцией. Однако данный граф позволяет понять каким образом получается программа, представленная в листинге 5 (программа была отформатирована вручную), являющаяся результатом работы SCP4.

Листинг 5 – Остаточная программа

```

$ENTRY Fab {
  e.1 'A', <F50 (e.1) 'BB'> : s.2 s.3 (e.4) = s.2 s.3 e.4;
  e.1 s.5, <F50 (e.1) s.5 'B'> : s.6 s.7 (e.8) = s.6 s.7 e.8;
  = 'B';
}
F50 {
  (e.7 'A') s.8 s.9 e.10,
  <F50 (e.7) 'B' s.8 s.9 e.10> : s.11 s.11 (e.12) = s.11 s.11 (e.12);
  (e.7 s.10) s.8 s.9 e.10,
  <F50 (e.7) s.10 s.8 s.9 e.10> : s.13 s.14 (e.15) = s.13 s.14 (e.15);
  () s.8 s.9 e.10 = s.8 s.9 (e.10);
}

```

Можно увидеть, что в данной программе теряется информация о том, что последним элементом результата обязательно будет литера 'B'. Это происходит из-за того, что обобщаются разные по происхождению литеры 'B'. В конфигурации <DoFab e.X ('B')> литера 'B' была сформирована в функции *Fab*, а в конфигурации <DoFab e.X ('BB')> первая 'B' была сформирована в функции *DoFab*, а вторая в функции *Fab*. Однако при обобщении <DoFab e.X ('B' e.1)> две разные по происхождению литеры 'B' обобщаются, что ведёт к потере информации.

Препроцессирование добавляет дополнительную информацию в конфигурации, таким образом влияя на совершаемые суперкомпилятором действия. В данном случае препроцессирование позволит различать разные по происхождению литеры 'B'.

5.1.1. Первый препроцессор

Рассмотрим программу, представленную в листинге 6, после её обработки первым препроцессором в режиме замены скобок и символов.

Листинг 6 – Программа, препроцессированная первым препроцессором

```
$ENTRY Fab { e.X = <DoFab e.X <b1 <c1>>> }
DoFab {
  e.X 'A' (e.Res) = <DoFab e.X <b2 <c2> e.Res>>;
  e.X s.1 (e.Res) = <DoFab e.X <b3 s.1 e.Res>>;
  /* empty */ (e.Res) = e.Res;
}

b1 { e.X = (e.X);}
c1 { = 'B';}
b2 { e.X = (e.X);}
c2 { = 'B';}
b3 { e.X = (e.X);}
```

Процесс суперкомпиляции для данной программы будет следующим (рисунок 10):

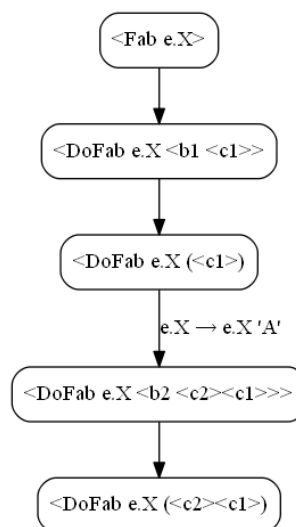


Рисунок 10 – Граф конфигураций для первого препроцессора.

Конфигурации $\langle DoFab\ e.X\ (<c1>) \rangle$ и $\langle DoFab\ e.X\ (<c2><c1>) \rangle$ похожи по Хигману-Крускалу, поэтому нужно провести их обобщение, однако в данном

случае существует только один вариант обобщения, а именно $\langle DoFab\ e.X\ (e.1\ \langle c1 \rangle) \rangle$. Граф конфигураций представлен на рисунке 11.

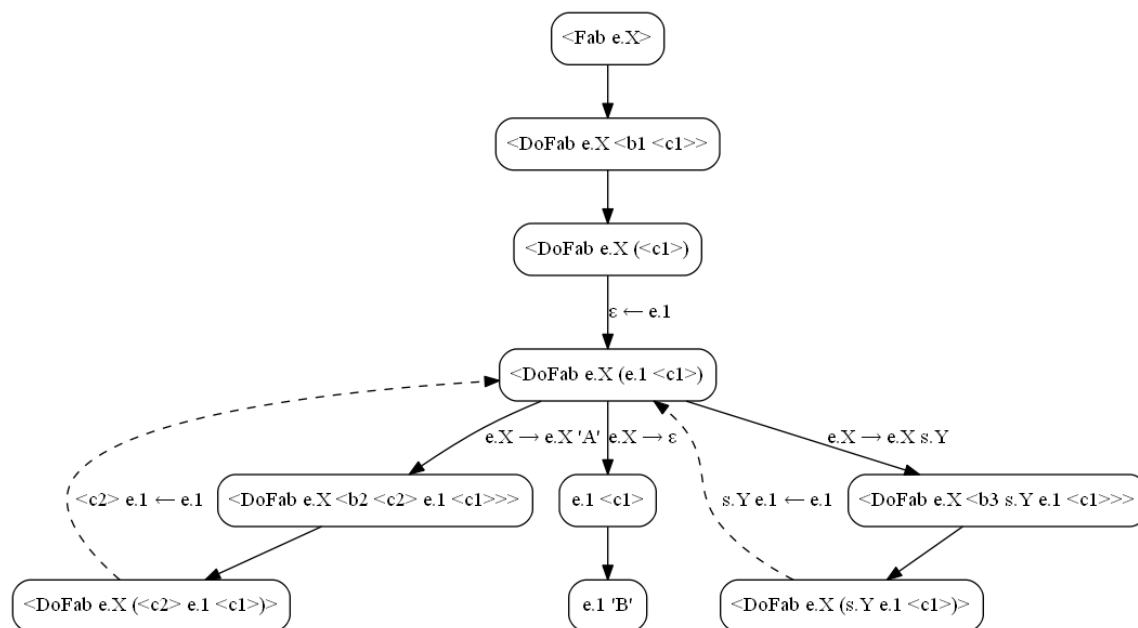


Рисунок 11 – Итоговый граф конфигураций для первого препроцессора

В результате получаем конфигурацию $e.1\ 'B'$, показывающую, что результат будет содержать последним элементом литеру $'B'$. Это отражается и на остаточной программе, которая представлена в листинге 7.

Листинг 7 – Остаточная программа

```
$ENTRY Fab {
    e.41 'A' = <F84 (e.41) 'B'> 'B';
    e.41 s.101 = <F84 (e.41) s.101> 'B';
    = 'B';
}

F84 {
    (e.133 'A') e.134 = <F84 (e.133) 'B' e.134>;
    (e.133 s.135) e.134 = <F84 (e.133) s.135 e.134>;
    () e.134 = e.134;
}
```

Можно заметить, что литера $'B'$ была выведена за пределы рекурсии. Таким образом, было показано, что препроцессор улучшил результат суперкомпиляции.

5.1.2. Второй препроцессор

В листинге 8 представлена программа, обработанная вторым препроцессором.

Листинг 8 – Программа, обработанная вторым препроцессором
 $\$ENTRY$ Fab { $e.vX = \langle DoFab\ e.vX\ (((B'\ K-1))\ K-2) \rangle$ }

```
DoFab {
  e.vX ('A' s.K-3) ((e.vRes) s.K-4)
    =  $\langle DoFab\ e.vX\ (((B'\ K-5)\ e.vRes)\ K-6) \rangle$ ;
  e.vX (s.v1 s.K-7) ((e.vRes) s.K-8)
    =  $\langle DoFab\ e.vX\ (((s.v1\ s.K-7)\ e.vRes)\ K-9) \rangle$ ;
  /* empty */ ((e.vRes) s.K-10) = e.vRes;
}
```

Для сокращения записи введём следующие обозначения: $(Sym\ K-1)$ будем записывать как Sym_1 , $((e.Expr)\ K-2)$ как $({}_2e.Expr)_2$ и $(s.Var\ s.K-3)$ как $s.Var_{s.K-3}$.
 Граф конфигураций представлен на рисунке 12.

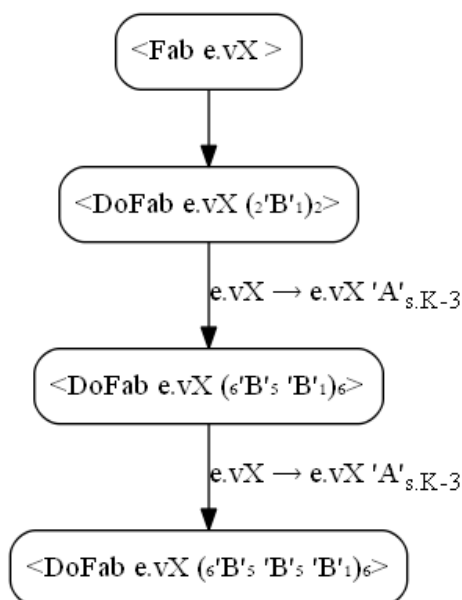


Рисунок 12 – Граф конфигураций для второго препроцессора

Конфигурации $\langle DoFab\ e.vX\ ({}_2'B'_1)_2 \rangle$ и $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_1)_6 \rangle$ не похожи по Хигману-Крускалу, так как скобки имеют разные координаты. А вот конфигурации $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_1)_6 \rangle$ и $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_5'B'_1)_6 \rangle$ являются похожими, поэтому производится их обобщение. Обобщим эти конфигурации следующим образом: $\langle DoFab\ e.vX\ ({}_6'B'_5\ e.1\ 'B'_1)_6 \rangle$.
 Подученный граф представлен на рисунке 13.

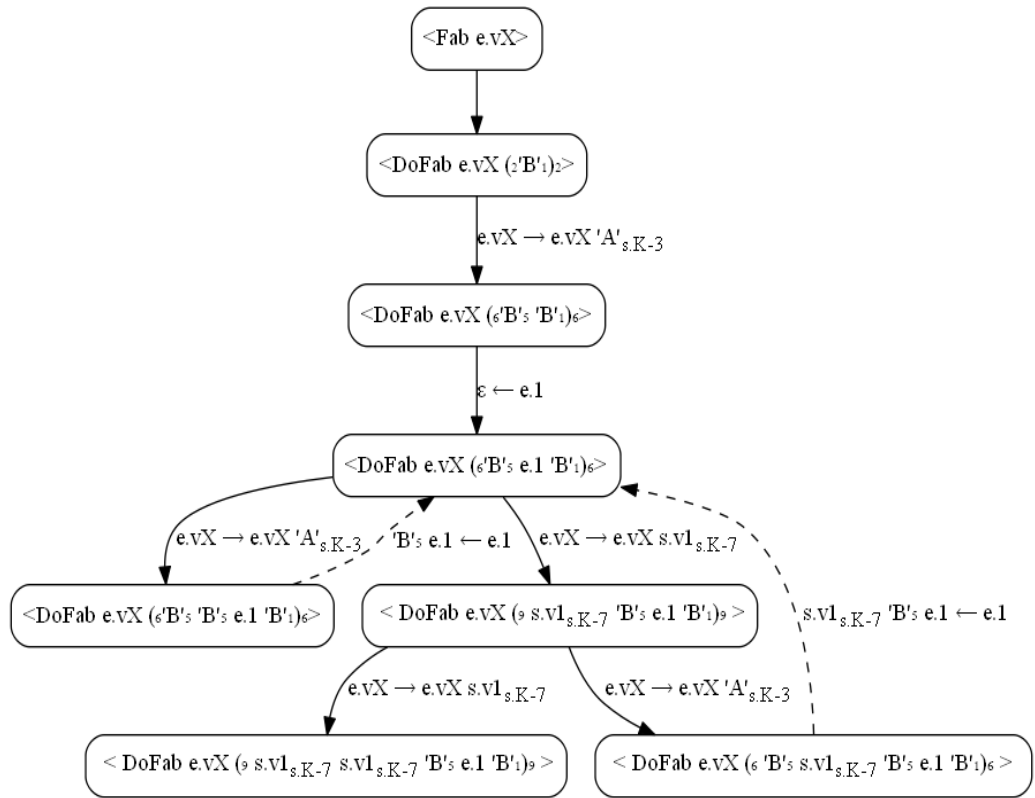


Рисунок 13 – Граф конфигураций для второго препроцессора

Так как конфигурация $\langle DoFab e.vX ({}_6'B'_5 'B'_5 e.1 'B'_1)_6 \rangle$ получается подстановкой из конфигурации $\langle DoFab e.X ({}_6'B'_5 e.1 'B'_1)_6 \rangle$, происходит заикливание данной ветки. Конфигурация $\langle DoFab e.vX ({}_6'B'_5 s.v1_{s.K-7} e.1 'B'_1)_6 \rangle$ тоже получается подстановкой из $\langle DoFab e.X ({}_6'B'_5 e.1 'B'_1)_6 \rangle$, поэтому данная ветка заикливается.

Конфигурации $\langle DoFab e.vX ({}_9 s.v1_{s.K-7} 'B'_5 e.1 'B'_1)_9 \rangle$ и $\langle DoFab e.vX ({}_9 s.v1_{s.K-7} s.v1_{s.K-7} 'B'_5 e.1 'B'_1)_9 \rangle$ похожи по Хигману-Крускалу, следовательно для них строится обобщение $\langle DoFab e.vX ({}_9 s.v1_{s.K-7} s.1_{s.K-X} e.1 'B'_1)_9 \rangle$.

Граф, получающийся в результате суперкомпиляции представлен на рисунке 14. Процесс суперкомпиляции для второй ветки почти полностью совпадает с процессом суперкомпиляции для первой, поэтому не будем расписывать процесс для данной ветки.


```

    () s.145 s.146 s.147 s.148 s.149 s.150 e.151
      = (s.145 s.146) (s.148 s.147) (s.150 s.149) e.151;
  }
F44 {
  (e.125 ('A' s.131)) s.126 s.127 e.128
    = <F44 (e.125) K-5 'B' (s.127 s.126) e.128>;
  (e.125 (s.130 s.131)) s.126 s.127 e.128
    = <F70 (e.125) s.130 s.131 K-5 'B' s.126 s.127 e.128>;
  () s.126 s.127 e.128 = ('B' K-5) (s.127 s.126) e.128;
}

```

Для того, чтобы понять причины несовпадения ожидаемого и реального результатов, был просмотрен файл трассировки работы суперкомпилятора, и было обнаружено, что суперкомпилятор делает неточное обобщение. Конфигурации $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_1)_6 \rangle$ и $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_5'B'_1)_6 \rangle$ обобщались следующим образом: $\langle DoFab\ e.vX\ ({}_6'B'_5\ 'B'_{s.1}\ e.2\)_6 \rangle$. При таком обобщении полностью стёрлась информация о происхождении $'B'_1$, поэтому препроцессор не смог как-либо улучшить результаты суперкомпиляции.

О проблеме с обобщениями было написано создателю суперкомпилятора, после чего он прислал версию суперкомпилятора, в которой обобщение было уточнено. Далее будем называть версию суперкомпилятора, на которой изначально проводились эксперименты «исходной», а версию с уточнённым обобщением «уточнённой». Если обе версии суперкомпилятора дают одинаковый результат, то остаточные программы будут приводиться без указания версии, с помощью которой они были получены.

Уточнённая версия делала правильное обобщение для конфигураций $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_1)_6 \rangle$ и $\langle DoFab\ e.vX\ ({}_6'B'_5'B'_5'B'_1)_6 \rangle$, и в итоге в остаточной программе был получен ожидаемый результат. Остаточная программа представлена листинге 10.

Листинг 10 – Остаточная программа

```

$ENTRY Fab {
  e.41 ('A' s.103) = <F23 (e.41)>;
  e.41 ('A' s.147) (s.102 s.103) = <F23 (e.41) (s.102 s.103)>;
  e.41 (s.146 s.147) (s.102 s.103)
    = <F49 (e.41) s.146 s.147 s.103 s.102>;
  (s.102 s.103) = (s.102 s.103) ('B' K-1);
  = ('B' K-1);
}

```

```

F49 {
  (e.128 ('A' s.138)) s.131 s.132 s.133 s.134 e.135
    = <F23 (e.128) (s.131 s.132) (s.134 s.133) e.135>;
  (e.128 (s.137 s.138)) s.131 s.132 s.133 s.134 e.135
    = <F49 (e.128) s.137 s.138 s.132 s.131 (s.134 s.133) e.135>;
  () s.131 s.132 s.133 s.134 e.135
    = (s.131 s.132) (s.134 s.133) e.135 ('B' K-1);
}
F23 {
  (e.110 ('A' s.115)) e.112 = <F23 (e.110) ('B' K-5) e.112>;
  (e.110 (s.114 s.115)) e.112
    = <F49 (e.110) s.114 s.115 K-5 'B' e.112>;
  () e.112 = ('B' K-5) e.112 ('B' K-1);
}

```

Можно заметить, что данная программа в точках выхода из рекурсии добавляет в качестве последнего элемента элемент ('B' K-1). Таким образом, второй препроцессор также позволяет улучшить результат суперкомпиляции.

5.1.3. Постановка задачи как задачи верификации

Можно заметить, что из остаточной программы может быть довольно сложно понять удовлетворяет ли её результат ожидаемым свойствам. Для более простой проверки можно сформулировать задачу как задачу верификации.

В нашем случае проверяется свойство о том, что результат программы всегда будет содержать литеру 'B' в качестве последнего символа. В листинге 11 приложена часть программы, описывающая предикат, проверяющий данное свойство.

```

Листинг 11 - Предикат
$ENTRY Test {
  e.X = <Check <Fab e.X>>
}
...
Check {
  e.X 'B' = True;
  e.X = False;
}

```

Рассмотрим часть остаточной программы, которая получается после суперкомпиляции исходной (листинг 12).

```

Листинг 12 - Остаточная программа
$ENTRY Test {
  e.41 'A' = <F51 (e.41) 'BB'>;
  e.41 s.101 = <F51 (e.41) s.101 'B'>;
}

```

```

    = True;
}
F51 {
...
() s.119 s.120 e.121 = False;
}

```

В остаточной программе сохранился символ False, а это значит, что свойство программы не удалось доказать.

Рассмотрим теперь остаточную программу, которая получается после обработки исходной первым препроцессором (листинг 13).

Листинг 13 – Остаточная программа

```

$ENTRY Test {
    e.41 'A' = True;
    e.41 s.101 = True;
    = True;
}

```

Получилась очень простая программа, которая доказывает проверяемое свойство.

Рассмотрим теперь результаты для второго препроцессора. В листинге 24 представлена часть остаточной программы, получающаяся после суперкомпиляции исходной версией суперкомпилятора.

Листинг 14 – Остаточная программа

```

$ENTRY Test {
    e.41 ('A' s.103) = <F45 (e.41) K-1 'B'>;
    ...
}
...
F45 {
    ...
    () s.126 s.127 e.128 = (False K-13);
}

```

Как и можно было ожидать, остаточная программа содержит символ False, так как неточное обобщение стирает важную для проверки свойства информацию.

Покажем теперь результат суперкомпиляции для уточнённой версии (листинг 15).

Листинг 15 – Остаточная программа

```

$ENTRY Test {

```



```

e.41 ('A' s.103) = <F24 e.41>;
e.41 ('A' s.147) (s.102 s.103) = <F24 e.41>;
e.41 (s.146 s.147) (s.102 s.103) = <F50 e.41>;
(s.102 s.103) = (True K-12);
= (True K-12);
}
F50 {
  e.128 ('A' s.138) = <F24 e.128>;
  e.128 (s.137 s.138) = <F50 e.128>;
  = (True K-12);
}
F24 {
  e.110 ('A' s.115) = <F24 e.110>;
  e.110 (s.114 s.115) = <F50 e.110>;
  = (True K-12);
}

```

В получившейся программе не осталось ни одного символа False, значит проверяемое свойство было доказано.

Было показано, что препроцессирование позволило суперкомпилятору решить поставленную задачу верификации. Переформулировка исходной задачи в задачу верификации помогла сделать пример более наглядным и простым для понимания, так как пропала необходимость разбираться в запутанных остаточных программах.

5.2. Программа вычисления целочисленного логарифма

Рассмотрим программу, представленную в листинге 16.

Листинг 16 – Вычисление целочисленного логарифма

```

$ENTRY Go {
  = <FlatLog () () 'III'>
}

FlatLog {
  (e.Acc) () /* empty */ = e.Acc;
  (e.Acc) () 'I' e.X
    = <FlatLog (e.Acc 'I') (Half) 'I' e.X ()>;
  (e.Acc) (Half) e.X 'II' (e.Res)
    = <FlatLog (e.Acc) (Half) e.X (e.Res 'I')>;
  (e.Acc) (Half) e.X (e.Res) = <FlatLog (e.Acc) () e.Res>;
}

```

Данная программа осуществляет вычисление целочисленного логарифма по базе два. На вход подаётся число три, поэтому суперкомпилятор должен отработать в качестве интерпретатора и в остаточной программе оставить только

результат работы данной программы. В данном случае ответом является результат вида 'II'.

Рассмотрим процесс суперкомпиляции данной программы (рисунки 15, 16). В процессе суперкомпиляции будут появляться транзитные вершины, поэтому для суперкомпиляции была отключена оптимизация, позволяющая удалять такие вершины.

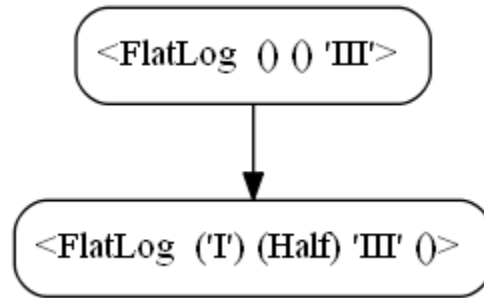


Рисунок 15 – Граф конфигураций для исходной программы.

Конфигурации $\langle FlatLog () () 'III' \rangle$ и $\langle FlatLog ('I') (Half) 'III' () \rangle$ похожи по Хигману-Крускалу, поэтому они обобщаются. Их обобщение будет выглядеть следующим образом: $\langle FlatLog (e.0) (e.1) 'III' e.2 \rangle$.

Однако при таком обобщении теряется информация о том, что на вход подаётся определённое конечное число и в дальнейшем суперкомпиляция проходит так, будто на вход подаются неизвестные данные.

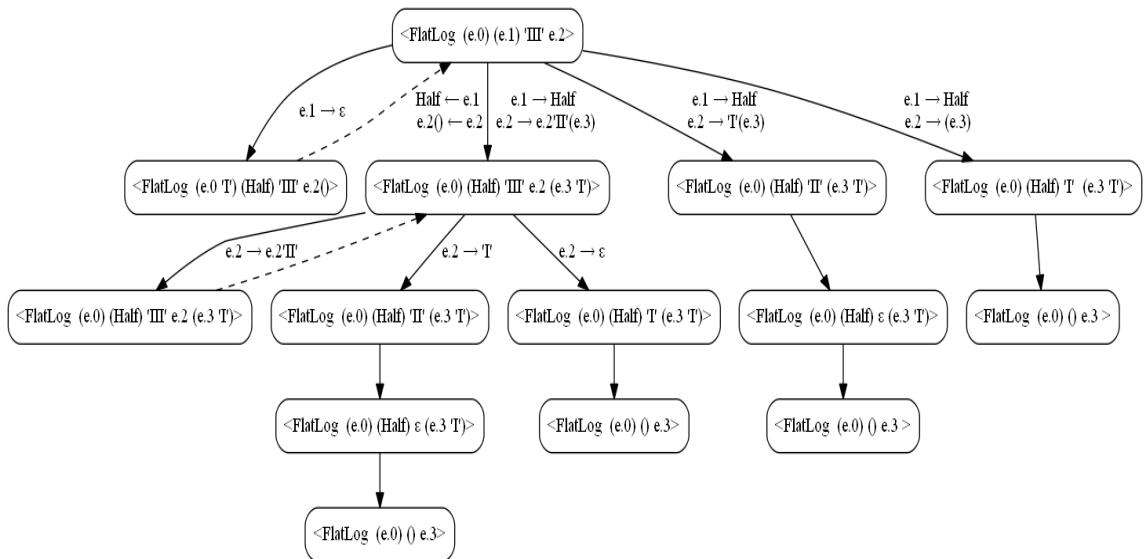


Рисунок 16 – Граф конфигураций для исходной программы

Конфигурацию $\langle FlatLog (e.0 \ T') (Half) \ III' \ e.2() \rangle$ можно получить из конфигурации $\langle FlatLog (e.0) (e.1) \ III' \ e.2 \rangle$ с помощью подстановки, поэтому данная ветвь закидывается. Ещё одно закидывание происходит с конфигурациями $\langle FlatLog (e.0) (Half) \ III' \ e.2(e.3 \ T') \rangle$ и $\langle FlatLog (e.0) (Half) \ III' \ e.2(e.3 \ T') \rangle$, так как эти конфигурации полностью совпадают.

Так как листовые вершины получились одинаковыми, рассмотрим процесс суперкомпиляции для них отдельно (рисунок 17).

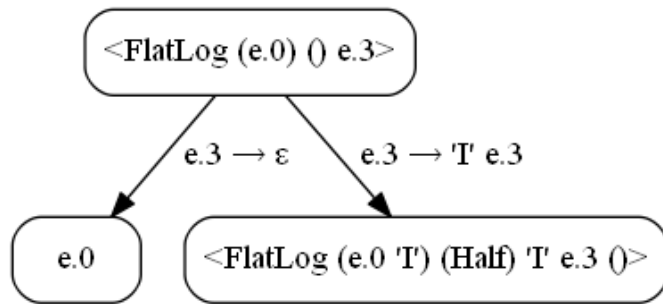


Рисунок 17 – Граф конфигураций для исходной программы

$\langle FlatLog (e.0) () e.3 \rangle$ и $\langle FlatLog (e.0 \ T') (Half) \ T' \ e.3 () \rangle$ – похожие по Хигману-Крускалу конфигурации, поэтому строим для них обобщение. Обобщением этих двух конфигураций будет конфигурация $\langle FlatLog (e.0) (e.4) e.5 \rangle$. На рисунке 18 представлен получившийся граф конфигураций.

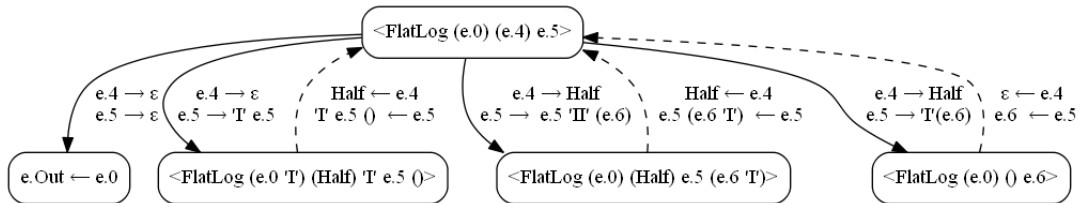


Рисунок 18 – Граф конфигураций для исходной программы

Все конфигурации $\langle FlatLog (e.0 \ T') (Half) \ T' \ e.5 () \rangle$, $\langle FlatLog (e.0) (Half) \ T' \ e.5 (e.6 \ T') \rangle$, $\langle FlatLog (e.0) () e.6 \rangle$ можно получить из конфигурации $\langle FlatLog (e.0) (e.4) e.5 \rangle$ с помощью подстановки, поэтому все эти ветви закидываются.

В результате, из-за потери информации о конечности входных данных после суперкомпиляции, получаем программу, которая повторяет процесс вычисления логарифма (листинг 7).

Листинг 17 – Остаточная программа

```
$ENTRY Go {  
    = <F122 () () 'III'>;  
}  
F122 {  
    (e.113) () = e.113;  
    (e.113) () 'I' e.115 = <F122 (e.113 'I') (Half) 'I' e.115 ()>;  
    (e.113) (Half) e.115 'II' (e.118)  
        = <F122 (e.113) (Half) e.115 (e.118 'I')>;  
    (e.113) (Half) e.115 (e.118) = <F122 (e.113) () e.118>;  
}
```

Можно заметить, что получившаяся после суперкомпиляции программа в точности повторяет исходную. Это означает, что в данном случае суперкомпиляция не принесла никакого результата.

5.2.1. Первый препроцессор

В листинге 18 представлена исходная программа, обработанная первым препроцессором.

Листинг 18 – Программа, обработанная первым препроцессором

```
$ENTRY Go {  
    = <FlatLog <b1 > <b2 > <c1><c2><c3>>  
}  
FlatLog {  
    (e.Acc) () /* empty */ = e.Acc;  
    (e.Acc) () 'I' e.X  
        = <FlatLog <b3 e.Acc <c4>> <b4 <c5>> <c6> e.X <b5 >>;  
    (e.Acc) (Half) e.X 'I' 'I' (e.Res)  
        = <FlatLog <b6 e.Acc> <b7 <c7>> e.X <b8 e.Res <c8>>>;  
    (e.Acc) (Half) e.X (e.Res) = <FlatLog <b9 e.Acc><b10 > e.Res>;  
}  
b1 { e.X = (e.X);}  
b2 { e.X = (e.X);}  
c1 { = 'I';}  
c2 { = 'I';}  
c3 { = 'I';}  
b3 { e.X = (e.X);}  
c4 { = 'I';}  
b4 { e.X = (e.X);}  
c5 { = Half;}  
c6 { = 'I';}  
b5 { e.X = (e.X);}  
b6 { e.X = (e.X);}  
b7 { e.X = (e.X);}  
c7 { = Half;}  
b8 { e.X = (e.X);}  
c8 { = 'I';}  
b9 { e.X = (e.X);}  
b10 { e.X = (e.X);}
```

Рассмотрим процесс суперкомпиляции данной программы (рисунки 19, 20).

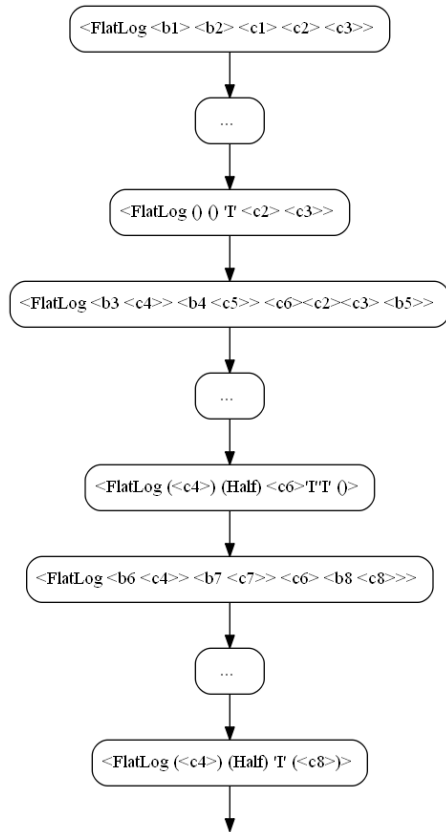


Рисунок 19 – Начало графа конфигураций

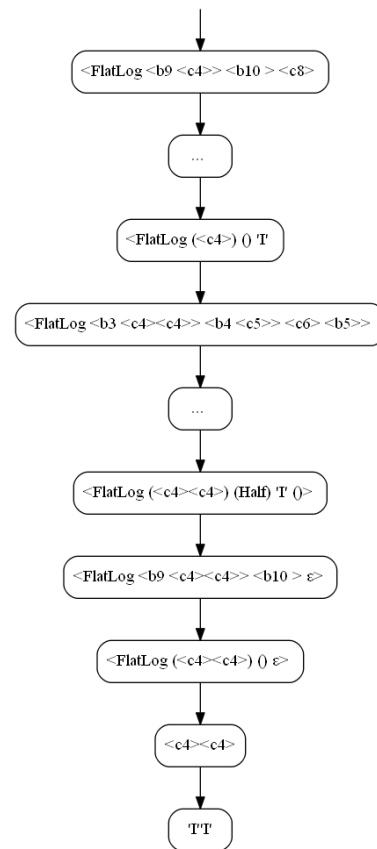


Рисунок 20 – Окончание графа конфигураций

Можно заметить, что в процессе суперкомпиляции не появляются похожие конфигурации, поэтому не строятся обобщения, а так как входные данные являются константными, то прогонка будет транзитной, так как у узла будет только один потомок, а ребро не будет помечено никаким сужением. Также в процессе не возникают конфигурации, которые можно получить подстановкой из предыдущих, поэтому в графе не появляются заикливания.

В итоге после суперкомпиляции получается результат работы исходной программы, а именно 'II'. Остаточная программа представлена в листинге 19.

```
Листинг 19 – Остаточная программа
$ENTRY Go {
    = 'II';
}
```

5.2.2. Второй препроцессор

Рассмотрим теперь работу второго препроцессора. В листинге 20 представлена программа, получающаяся после препроцессирования исходной.

Листинг 20 – Программа, обработанная вторым препроцессором

```
$ENTRY Go {
= <FlatLog (( ) K-1) (( ) K-2) ('I' K-3)('I' K-4)('I' K-5)>
}
FlatLog {
  ((e.vAcc) s.K-6) (( ) s.K-7) /* empty */ = e.vAcc;
  ((e.vAcc) s.K-8) (( ) s.K-9) ('I' s.K-10) e.vX
    = <FlatLog ((e.vAcc('I' K-11)) K-12) (((Half K-13)) K-14)
      ('I' K-15) e.vX (( ) K-16)>;
  ((e.vAcc) s.K-17) (((Half s.K-18)) s.K-19)
e.vX ('I' s.K-20)('I' s.K-21) ((e.vRes) s.K-22)
    = <FlatLog ((e.vAcc) K-23) (((Half K-24)) K-25)
      e.vX ((e.vRes ('I' K-26)) K-27)>;
  ((e.vAcc) s.K-28) (((Half s.K-29)) s.K-30)
e.vX ((e.vRes) s.K-31)
    = <FlatLog ((e.vAcc) K-32) (( ) K-33) e.vRes>;
}
```

Для описания процесса суперкомпиляции воспользуемся теми же сокращениями, которые были введены в пункте 5.1.2. Граф конфигураций приведён на рисунке 21.

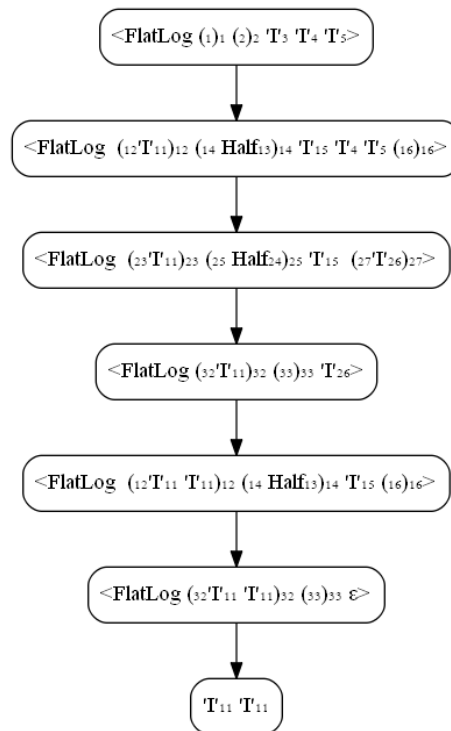


Рисунок 21 – Граф конфигураций для второго препроцессора

В данном случае также не возникает каких-либо похожих конфигураций, которые нужно обобщать или конфигураций, получающихся подстановкой из предыдущих, которые нужно заикливать. В итоге опять был получен конечный результат, с точностью до кодирования данных повторяющий результат работы исходной программы.

Таким образом остаточная программа, представленная в листинге 21, содержит только результат, вычисленный с точностью до кодирования данных.

Листинг 21 – Остаточная программа

```
$ENTRY Go {
    = ('I' K-11) ('I' K-11);
}
```

5.2.3. Исследование примера на суперкомпиляторе MSCP-A

Суперкомпилятор MSCP-A не поддерживает возможность отключения удаления транзитных шагов. Чтобы это обойти программа вычисления целочисленного логарифма была переписана таким образом, чтобы преобразовать транзитные шаги в нетранзитные. Для наглядности задача была переформулирована в задачу верификации, которая проверяет правильность ответа, возвращаемого программой. Программа представлена в листинге 22.

Листинг 22 – Программа верификации

```
$ENTRY Go {
    e.Steps = <BoolOnly <Check <FlatLog (e.Steps) () () 'III'>>>
}
FlatLog {
    (S e.Steps) e.Arg = <FlatLog-Aux (e.Steps) e.Arg>;
    (STOP) e.X = STOPPED;
}
FlatLog-Aux {
    (e.Steps) (e.Res) () /* empty */ = e.Res;
    (e.Steps) (e.Res) () 'I' e.X
        = <FlatLog (e.Steps) (e.Res 'I') (Half) 'I' e.X>;
    (e.Steps) (e.Res) (Half e.Half) 'II' e.X
        = <FlatLog (e.Steps) (e.Res) (Half e.Half 'I') e.X>;
    (e.Steps) (e.Res) (Half e.Half) e.X
        = <FlatLog (e.Steps) (e.Res) () e.Half>;
}
Check {
    'II' = True;
    STOPPED = STOPPED;
    e.X = False;
```

```

}
BoolOnly {
    True = True;
    False = False;
}

```

Программа представляет собой эмуляцию пошагового выполнения вычисления логарифма. S символизирует переход к следующему шагу, а STOP указывает на прерывание вычислений. Таким образом, каждая конфигурация имеет одну ветку, обозначающую продолжение вычислений, и одну указывающую на прерывание. Это значит, что конфигурации перестают быть транзитными.

Рассмотрим остаточную программу, которая получается после суперкомпиляции исходной (листинг 23).

Листинг 23 – Остаточная программа

```

$ENTRY Go {
    e.x1 = <Let_1 (e.x1) () ()>;
}
Let_1 {
    (S S S e.x1) (e.x2) () = <Let_2 (e.x1) (e.x2) ()>;
    (S S S S S e.x1) ('I') (Half) = True;
    (S S S S S e.x1) (e.x2) (Half) = False;
}
Let_2 {
    (S S S e.x1) () () = True;
    (S S S e.x1) (e.x2) () = False;
    (S S e.x1) ('I') (Half) = True;
    (S S e.x1) (e.x2) (Half) = False;
}

```

Данная программа показывает, что суперкомпилятор не выполняет вычисление результата для конечных входных данных, а рассматривает все возможные варианты вычислений.

По техническим причинам эксперименты для первого препроцессора с данным суперкомпилятором провести не удалось, поэтому рассмотрим результат для второго препроцессора (листинг 24).

Листинг 24 – Остаточная программа

```

$ENTRY Go {
    e.x1 = <InputFormat_0 e.x1>;
}
InputFormat_0 {
    (S s.z1) (S s.z2) (S s.z3) (S s.z4) (S s.z5) (S s.z6) e.x1

```


= (True K-51);
 }

По получившейся остаточной программе видно, что после препроцессирования суперкомпилятор смог вычислить результат работы программы для подаваемых входных данных.

5.3. Криптографический протокол

5.3.1. Основная теоретическая информация

Криптографический протокол – это множество правил, которые задают поведение участников процесса обмена сообщениями в сети таким образом, что злоумышленнику было как можно труднее выдавать себя за правомерного пользователя [10].

В примере будет использоваться пинг-понг протокол [11]. Данные протоколы требуют, чтобы секретное сообщение, передаваемое участниками, представлялось единственной строкой, изменяемой конечным множеством операций. Каждый участник протокола применяет к секретному сообщению операции из этого множества и пересылает результат другому участнику. Множество операций и количество пересылок определяются используемым протоколом.

Рассмотрим процесс обмена данными по сети между участниками протокола. Данные представляют собой слова в некотором конечном алфавите, обозначим множество таких слов как \mathbb{S} . Множество всех пользователей сети обозначим U_{sr} . Множество операторов $f: \mathbb{S} \rightarrow \mathbb{S}$, доступных пользователю $A \in U_{sr}$, назовём словарём A и обозначим Σ_A . Пустое действие обозначим λ .

Пусть дана функция $f: U_{sr} \times \mathbb{S} \rightarrow \mathbb{S}$, ограничение этой функции на множество $\{A\} \times \mathbb{S}$, где $A \in U_{sr}$, $f_A: \mathbb{S} \rightarrow \mathbb{S}$, назовём параметрическим оператором.

Дадим определение p -стороннего пинг-понг протокола $P[x_1, \dots, x_p]$. $P[x_1, \dots, x_p]$ – это последовательность пар $((y_1, \alpha_1[x_1, \dots, x_p]), \dots, (y_l, \alpha_l[x_1, \dots, x_p]))$, где p – число участников, y_i – переменная из U_{sr} и $\alpha_i[x_1, \dots, x_p]$ – последовательность параметризованных операторов из словаря пользователя y_i .

В рассматриваемом пинг-понг протоколе будут использоваться следующие операторы:

- Оператор шифрования открытым ключом E_x . $E_x(Y)$ зашифровывает Y ключом пользователя x ;
- Оператор расшифровки D_x . $D_x(Y)$ расшифровывает Y ключом пользователя x ;
- Оператор приписывания имени a_x . $a_x(Y)$ приписывает имя пользователя x к Y ;
- Оператор удаления имени d_x . $d_x(Y)$ стирает префикс Y , соответствующий имени пользователя x .

D_x является обратным для E_x , то есть выполняются следующие равенства: $D_x E_x = \lambda$ и $E_x D_x = \lambda$. d_x является левым обратным для a_x , то есть выполняется $d_x a_x = \lambda$. E_x, a_x, d_x – открытые операторы, доступные каждому пользователю сети, D_x – секретный и доступен только пользователю x .

Под атакой на протокол будем понимать попытку злоумышленника провести анализ сообщений протокола с целью получения секретного сообщения, пересылаемого участниками.

В примере используется протокол $P[x_1, x_2] = ((x_1, E_{x_2} a_{x_1} E_{x_2}), (x_2, E_{x_1} D_{x_2} d_{x_1} D_{x_2}))$, описанный в [11]. Суть протокола заключается в следующем: пользователь x_1 шифрует сообщение открытым ключом пользователя x_2 , затем приписывает своё имя и ещё раз шифрует и затем отправляет его пользователю x_2 . Пользователь x_2 расшифровывает сообщение своим ключом, удаляет имя пользователя x_1 , ещё раз расшифровывает сообщение, а затем шифрует его открытым ключом пользователя x_1 и отправляет сообщение этому пользователю.

Рассмотрим атаку на этот протокол. Пусть в процессе обмена сообщениями участвуют правомерные пользователи A и B и злоумышленник I . На рисунке 22 графически представлена атака на протокол.

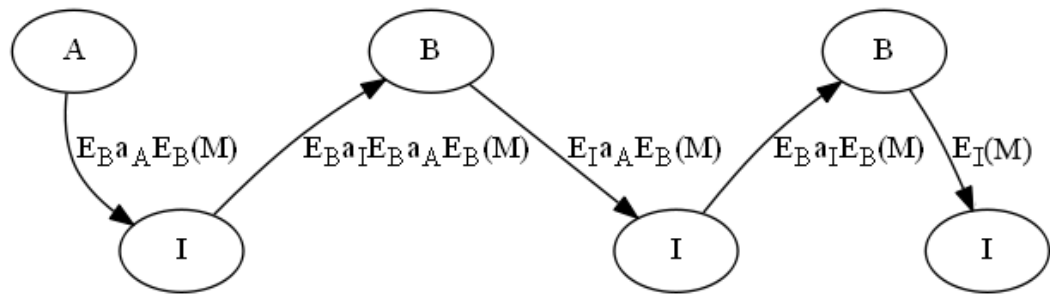


Рисунок 22 – Пример атаки на протокол

В данном случае пользователь A отправляет сообщение пользователю B , однако злоумышленник перехватывает сообщение $E_B a_A E_B(M)$, и после этого приписывает к полученному сообщению своё имя и повторно шифрует ключом E_B . B будет считать присланное ему сообщение продолжением правомерного взаимодействия с I , поэтому, поступив в соответствии с протоколом отошлёт I сообщение $E_I a_A E_B(M)$. I расшифровывает сообщение ключом D_I и стирает a_A , а затем вновь повторяет свои предыдущие действия. После ответа B злоумышленник I получает сообщение $E_I(M)$, которое может расшифровать и прочесть секретное сообщение M .

5.3.2. Упрощенный криптографический протокол

Задачей при исследовании этого примера будет являться нахождение атаки на протокол при помощи суперкомпиляции. Этот пример будет более подробно рассмотрен для исходной версии суперкомпилятора, так как изначально он исследовался именно на ней. Результаты для уточнённой версии будут рассмотрены в отдельном подразделе. В листинге 25 представлен упрощённый криптографический протокол.

Листинг 25 – Упрощённый криптографический протокол

```

$ENTRY Go {e.R = <Exec (e.R)(BE)(aP)(BE)>; }

Exec {
  (R1 e.R)(BE)(iP)(BE)e.Other = <Exec (e.R)(IE)e.Other>;
  (e.R)(IE)e.Other = <Exec (e.R) e.Other>;
  (e.R)(aP) e.Other = <Exec (e.R) e.Other>;
  (e.R)/ * EMPTY */ = FALSE;
  (R5 e.R) e.Other = <Exec (e.R)(BE)(iP)e.Other>;
  (R0 e.R) e.Other = <Exec (e.R)(Nonsense)e.Other>;
  e.Other = TRUE;
}

```

R1, R5, R0 – это номера правил переписывания, применяемых к протоколу.

Здесь R0 – это произвольное бессмысленное действие, не имеющее обратного.

R1 – это использование злоумышленником правила протокола: Пользователь В, получив сообщение от Х, расшифровывает это сообщение, читает идентификатор пославшего и еще раз расшифровывает, а потом зашифровывает ключом пославшего.

R5 – это приписывание злоумышленником к сообщению собственного имени, а затем зашифровывание сообщения.

Есть правила, которые злоумышленник применяет по умолчанию, то есть безальтернативно. Злоумышленник сразу расшифровывает сообщение, зашифрованное его ключом, а также сразу стирает имя пользователя А из сообщения.

ХЕ – это зашифровка открытым ключом участника Х. Т.е. если $X=A$, то пользователя А, если $X=B$, пользователя В, если $X=I$ – ключом злоумышленника.

хР – приписывание идентификатора участника Х.

Атака на этот протокол описывается последовательностью действий R5 R1 R5 R1, если злоумышленник применит эти действия к исходному сообщению, он сможет получить секретный текст.

Рассмотрим процесс суперкомпиляции для протокола (рисунок 23).

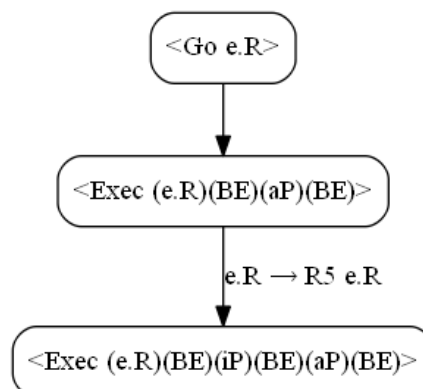


Рисунок 23 – Граф конфигураций исходной программы

Конфигурации $\langle Exec (e.R)(BE)(aP)(BE) \rangle$ и $\langle Exec (e.R)(BE)(iP)(BE)(aP)(BE) \rangle$ похожи по Хигману-Крускалу, поэтому они обобщаются.

Суперкомпилятор строит для них следующее обобщение: $\langle Exec(e.R)(BE)(s.1)(BE)e.2 \rangle$. Получившийся граф представлен на рисунке 24.

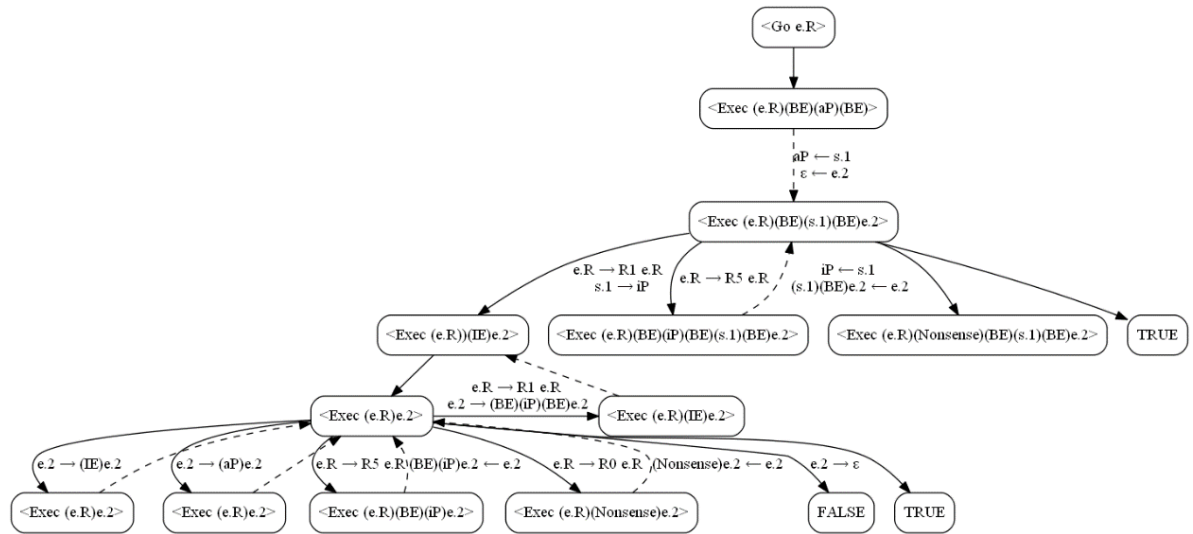


Рисунок 24 – Граф конфигураций исходной программы

Из-за того, что в процессе обобщения aP заменяется на $s.1$, теряется информация о начальном действии участника протокола В. Эту информация теперь не может быть получена обратно, поэтому дальнейший поиск атаки на протокол может быть значительно затруднён.

Для данного примера программы подвергаются нескольким последовательным суперкомпиляциям. Причины этого заключаются в результатах, полученных препроцессированием.

Для исходной программы остаточная, представленная в листинге 26, была получена с помощью трёх последовательных суперкомпиляций.

Листинг 26 – Остаточная программа

```
$ENTRY Go {
  R5 R1 R5 e.41 = <F30 () e.41>;
  R5 R1 R0 e.41 = <F39 ((Nonsense) (BE)) e.41>;
  R5 R1 e.41 = TRUE;
  R5 R5 e.41 = <F39 ((BE) (iP) (BE) (iP) (BE) (aP) (BE)) e.41>;
  R5 R0 e.41 = <F39 ((Nonsense) (BE) (iP) (BE) (aP) (BE)) e.41>;
  R5 e.41 = TRUE;
  R0 e.41 = <F39 ((Nonsense) (BE) (aP) (BE)) e.41>;
  e.41 = TRUE;
}
F39 {
  ((BE) (iP) (BE) e.118) R1 e.119 = <F39 (e.118) e.119>;
  ((IE) e.118) e.119 = <F39 (e.118) e.119>;
  ((aP) e.118) e.119 = <F39 (e.118) e.119>;
}
```

```

() e.119 = FALSE;
((BE) e.118) R5 R1 e.119 = <F39 (e.118) e.119>;
(e.118) R5 R5 e.119 = <F30 ((iP) e.118) e.119>;
(e.118) R5 R0 e.119 = <F39 ((Nonsense) (BE) (iP) e.118) e.119>;
(e.118) R5 e.119 = TRUE;
(e.118) R0 e.119 = <F39 ((Nonsense) e.118) e.119>;
(e.118) e.119 = TRUE;
}
F30 {
(e.118) R1 e.119 = <F39 (e.118) e.119>;
(e.118) R5 e.119 = <F30 ((iP) (BE) e.118) e.119>;
(e.118) R0 e.119=<F39 ((Nonsense) (BE) (iP) (BE) e.118) e.119>;
(e.118) e.119 = TRUE;
}

```

Можно будет сказать, что атака найдена, если предложение, равное символу False, будет находиться в функции Go. Тогда в этом предложении будет описана последовательность действий, которая приведёт злоумышленника к получению секретного сообщения.

5.3.3. Первый препроцессор

В листинге 27, представлен протокол, обработанный первым препроцессором.

Листинг 27 – Протокол, обработанный первым препроцессором

```

$ENTRY Go {e.R = <Exec <b1 e.R><b2 <c1>><b3 <c2>><b4 <c3>>>; }
Exec {
(R1 e.R)(BE)(iP)(BE)e.Other = <Exec <b5 e.R><b6 <c4>>>e.Other>;
(e.R)(IE)e.Other = <Exec <b7 e.R> e.Other>;
(e.R)(aP) e.Other = <Exec <b8 e.R> e.Other>;
(e.R)/ * EMPTY */ = <c5>;
(R5 e.R) e.Other = <Exec <b9 e.R><b10 <c6>><b11 <c7>>>e.Other>;
(R0 e.R) e.Other = <Exec <b12 e.R><b13 <c8>>>e.Other>;
e.Other = <c9>;
}
b1 { e.X = (e.X);}
...
b13 { e.X = (e.X);}
c1 { = BE;}
c2 { = aP;}
c3 { = BE;}
c4 { = IE;}
c5 { = FALSE;}
c6 { = BE;}
c7 { = iP;}
c8 { = Nonsense;}
c9 { = TRUE;}

```

Для упрощения графов суперкомпиляции будем рассматривать только ту ветку, по которой может быть найдена атака. На рисунке 25 представлен граф конфигураций рассматриваемой программы.

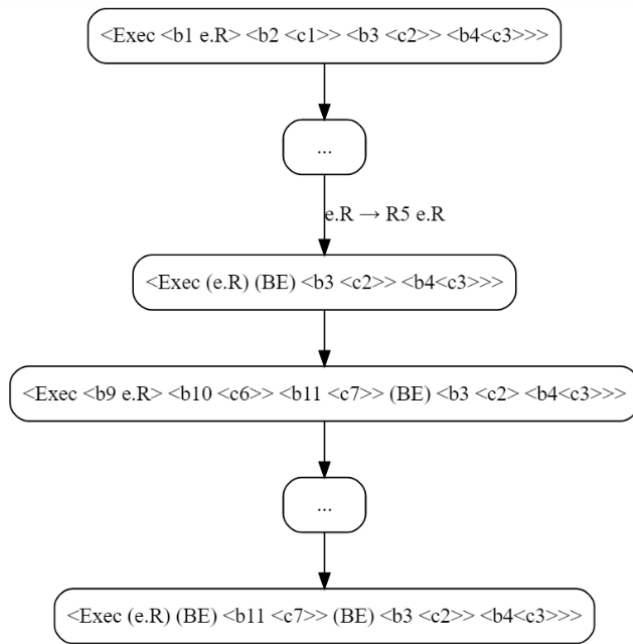


Рисунок 25 – Граф конфигураций для первого препроцессора

Конфигурации $\langle \text{Exec } (e.R) (BE) \langle b3 \langle c2 \rangle \rangle \langle b4 \langle c3 \rangle \rangle \rangle$ и $\langle \text{Exec } (e.R) (BE) \langle b11 \langle c7 \rangle \rangle (BE) \langle b3 \langle c2 \rangle \rangle \langle b4 \langle c3 \rangle \rangle \rangle$ похожи по Хигману-Крускалу, поэтому строится их обобщение $\langle \text{Exec } (e.R) (BE) e.1 \langle b3 \langle c2 \rangle \rangle \langle b4 \langle c3 \rangle \rangle \rangle$.

Получим граф конфигураций, представленный на рисунке 26.

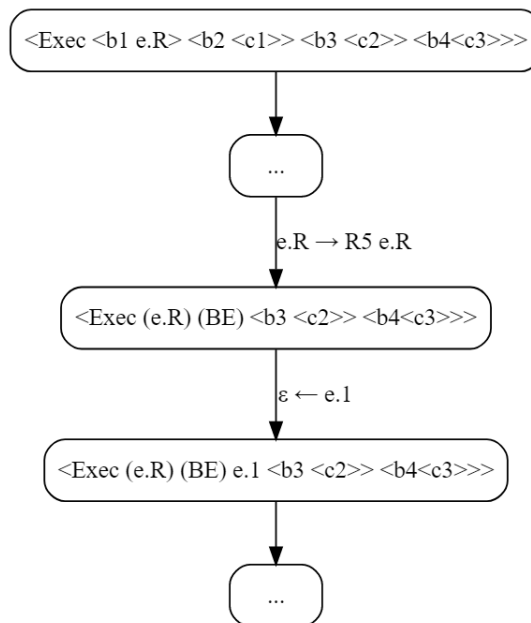


Рисунок 26 – Граф конфигураций для первого препроцессора

Из-за получившегося обобщения была потеряна некоторая часть информации, однако большая часть исходных данных была сохранена. Получившаяся после трёх последовательных суперкомпиляций остаточная программа показывает, что оставшейся информации достаточно, чтобы в итоге получить атаку на протокол. Часть остаточной программы, представлена в листинге 28.

Листинг 28 – Остаточная программа

```
$ENTRY Go {
    R5 R1 R5 R1 e.41 = FALSE;
    ...
}
```

В итоге при помощи препроцессирования была найдена атака на протокол. Таким образом препроцессирование улучшило результат суперкомпиляции.

5.3.4. Второй препроцессор

В листинге 29 представлен протокол после обработки вторым препроцессором.

Листинг 29 – Протокол, обработанный вторым препроцессором

```
$ENTRY Go {e.vR
=<Exec ((e.vR)K-1)(((BE K-2))K-3)(((aP K-4))K-5)(((BE K-6)) K-7)>;
}
Exec {
    (((R1 s.K-8) e.vR) s.K-9)(((BE s.K-10)) s.K-11)(((iP s.K-12)) s.K-
13)(((BE s.K-14)) s.K-15)e.vOther
    = <Exec ((e.vR) K-16)(((IE K-17)) K-18)e.vOther>;
    ((e.vR) s.K-19)(((IE s.K-20)) s.K-21)e.vOther
    = <Exec ((e.vR) K-22) e.vOther>;
    ((e.vR) s.K-23)(((aP s.K-24)) s.K-25) e.vOther
    = <Exec ((e.vR) K-26) e.vOther>;
    ((e.vR) s.K-27)/* EMPTY */ = (FALSE K-28);
    (((R5 s.K-29) e.vR) s.K-30) e.vOther
    =<Exec((e.vR) K-31) (((BE K-32))K-33) (((iP K-34)) K-35)
e.vOther>;
    (((R0 s.K-36) e.vR) s.K-37) e.vOther
    = <Exec ((e.vR) K-38)(((Nonsense K-39)) K-40)e.vOther>;
    e.vOther = (TRUE K-41); /*<Rec_Imp>;*/
}
```

Рассмотрим процесс суперкомпиляции для этой программы. Граф суперкомпиляции представлен на рисунке 27.

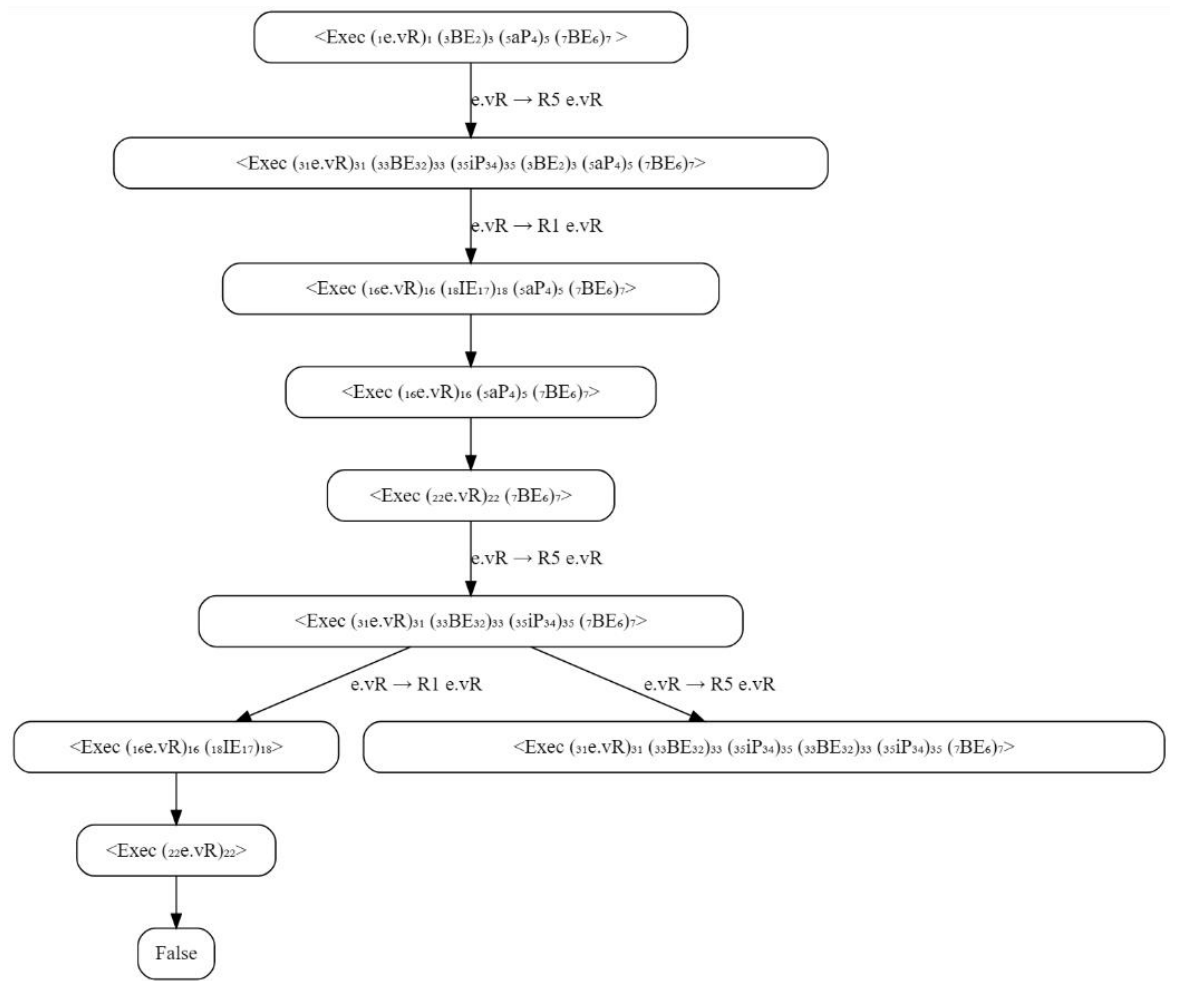


Рисунок 27 – Граф конфигураций для первого препроцессора

Можно заметить, что препроцессор позволил сразу найти атаку, однако суперкомпилятор ничего не знает о необходимом нам результате, поэтому он сравнивает конфигурации $\langle \text{Exec } (31e.vR)_{31} (33BE_{32})_{33} (35iP_{34})_{35} (7BE_6)_7 \rangle$ и $\langle \text{Exec } (31e.vR)_{31} (33BE_{32})_{33} (35iP_{34})_{35} (33BE_{32})_{33} (35iP_{34})_{35} (7BE_6)_7 \rangle$, видит, что они похожи по Хигману-Крускалу и обобщает их до конфигурации $\langle \text{Exec } (31e.vR)_{31} (33BE_{32})_{33} (35iP_{34})_{35} (5.2BE_{5.1})_{5.2} e.3 \rangle$. На рисунках 28 и 29 представлена часть графа конфигураций, получающаяся после этого обобщение.

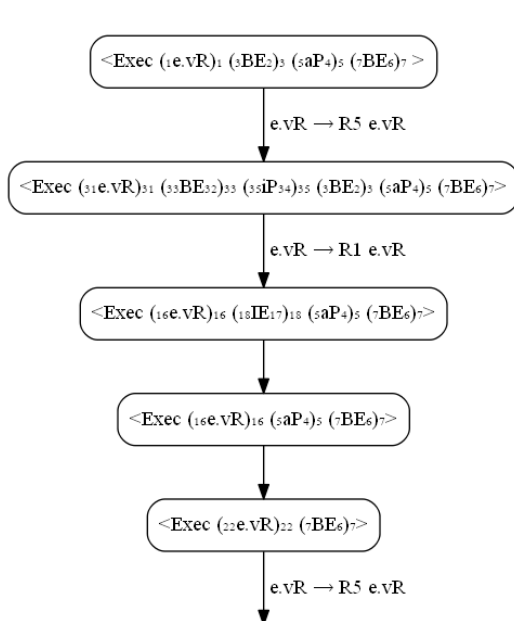


Рисунок 28 – Начало графа конфигураций

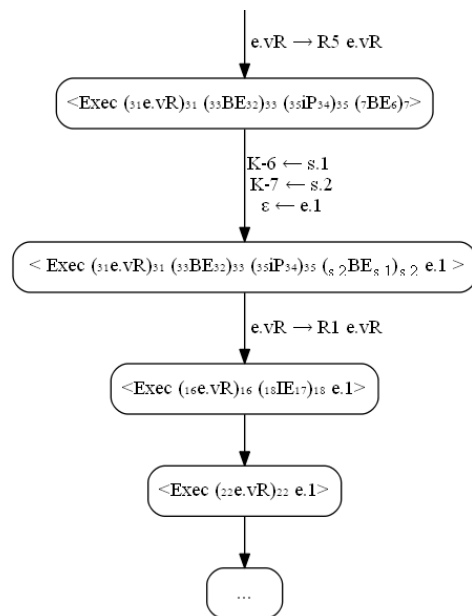


Рисунок 29 – Окончание графа конфигураций

С одной стороны, информация об атаке была потеряна из-за обобщения, однако никакой исходной информации потеряно не было, и атака на самом деле может быть найдена довольно быстро. Поэтому второй процессор даёт лучший результат, чем первый, и находит атаку уже после второй суперкомпиляции. Часть остаточной программы после суперкомпиляции представлена на листинге 30.

Листинг 30 – Остаточная программа

```

$ENTRY Go {
(R5 s.103) (R1 s.106) (R5 s.109) (R1 s.145) e.41
= (FALSE K-28);
...
}

```

5.3.5. Результаты суперкомпиляции уточненным суперкомпилятором

Изначально опыт над протоколом проводился с использованием исходной версии. Для этой версии, как уже было показано, суперкомпилятор не смог найти атаку для исходной программы.

После того, как была получена версия суперкомпилятора, с уточнённым обобщением, оказалось, что эта версия может найти атаку даже для исходной программы.

Ещё более интересным оказалось то, что для первого препроцессора атаку наоборот найти не удалось. Для второго препроцессора режимы, которые

добавляют уникальные координаты только к символам и только к скобкам, позволяют найти атаку, причем режим, добавляющий уникальные координаты только к скобкам даёт очень маленькую и простую программу, представленную в листинге 31.

Листинг 31 – Остаточная программа

```
$ENTRY Go {  
    (R5 s.103) (R1 s.106) (R5 s.129) (R1 s.131) e.41 = (FALSE K);  
    (R5 s.103) (R1 s.106) e.41 = (TRUE K);  
    (R5 s.110) (R5 s.113) e.41 = (TRUE K);  
    (R5 s.117) (R0 s.120) e.41 = (TRUE K);  
    (R5 s.124) e.41 = (TRUE K);  
    (R0 s.127) e.41 = (TRUE K);  
    e.41 = (TRUE K);  
}
```

Для режима добавления уникальных координат и к символам, и к скобкам суперкомпилятор, к сожалению, не может выдать конечный результат.

Результат, получившийся для уточнённой версии суперкомпилятора, хоть и является отрицательным в рамках рассматриваемой задачи, является довольно интересным для дальнейшего изучения процесса суперкомпиляции.

5.4. Результаты

Подведём итоги рассмотренных примеров.

Первый пример с программой замены литеры 'А' на литеру 'В' показывает, что суперкомпилятор в некоторых случаях не может построить наиболее оптимальное обобщение из-за недостатка информации. Было показано, что препроцессор помогает суперкомпилятору выбрать более правильное обобщение.

Второй пример с программой вычисления целочисленного логарифма демонстрирует случай, когда суперкомпилятор обнаруживает ложное заикливание на цепочке транзитных шагов. В этом случае препроцессирование тоже показало хороший результат, не давая суперкомпилятору делать ложные заикливания.

Пример с протоколом показывает возможности улучшения суперкомпиляции с помощью препроцессирования, для задач связанных с исследованием криптографических протоколов.

Как уже говорилось у препроцессоров есть способы настройки изменений, применяемых к препроцессируемым программам, но подробный разбор влияния каждого варианта изменений является слишком трудоёмкой задачей. Таблица 3 представляет собой сводную таблицу влияния различных изменений на результат препроцессора. Первый и второй примеры представлены в виде задачи верификации, для них знак плюс в таблице означает, что препроцессор помог доказать исследуемое свойство, знак минус – что не помог. Для примера с протоколом знак плюс означает, что препроцессор помог найти атаку на протокол. В – препроцессор применяет изменения, связанные с преобразованием скобок, S – изменения, связанные с преобразованиями символов, E – для первого препроцессора означает добавление вызовов функций, возвращающих пустое выражение, в программу.

Таблица 1 – Результаты препроцессирования для разных режимов

Пример	Исходная программа	Первый препроцессор							Второй препроцессор		
		S	B	E	SB	SE	BE	SBE	S	B	SB
Замена 'A' на 'B'	—	+	—	—	+	—	—	—	+	—	+
Целочисленный логарифм	—	+	+	+	+	+	+	+	+	+	+
Криптографический протокол	—	+	+	—	+	—	—	—	—	+	+

Данная таблица показывает, что результаты препроцессирования зависят от того, в какие изменения исходной программы были выбраны. Это значит, что даже если для какого-то режима препроцессор не показал результата, то можно изменить режим и, возможно, это поможет улучшить результат суперкомпиляции.

6. Руководство пользователя

6.1. Компиляция и использование

Для получения исполняемого файла препроцессора из исходного кода на компьютере пользователя должен быть установлен компилятор языка Рефал-5λ [12].

Исходный код первого препроцессора содержится в файле *ref_preprocess1.ref*, второго в файле *ref_preprocess2.ref*, также имеется файл *lexer.ref*, содержащий исходный код лексического анализатора.

Для компиляции выбранного препроцессора выполняется команда

```
srefc ref_preprocessX lexer
```

где X соответствует номеру выбранного препроцессора. После компиляции появится исполняемый файл.

После этого препроцессор используется следующим образом:

- для Windows *ref_preprocessX path_to_file/filename.ref*
- для Linux *./ref_preprocessX path_to_file/filename.ref*

Файл *filename.ref* должен содержать соответствующий псевдокомментарий препроцессирования. После применения препроцессора в папку с файлом *filename.ref* будет помещён файл *filename_updatedX.ref*. Этот файл будет содержать препроцессированную программу, если псевдокомментарий был написан верно, или исходную программу, если он был написан неверно или отсутствовал.

6.2. Псевдокомментарий первого препроцессора

Первый препроцессор предоставляет три способа преобразования исходных программ, написанных на языке Рефал-5: первый – добавление вызова функций, возвращающих пустоту, в результатные выражения, второй – замена константных символов на вызовы функций и третий – замена круглых скобок на вызовы функций.

Выбор вариантов преобразования программы производится с помощью специального псевдокомментария `*$PREPROCESS1` с соответствующими аргументами.

Таблица 2 – Аргументы псевдокомментария `*$PREPROCESS1`

Аргумент	Преобразование
Empty	Добавление вызовов функций, возвращающих пустоту.
Symbols	Замена константных символов на вызовы функций.
Brackets	Замена круглых скобок на вызовы функций.

Существует возможность записи псевдокомментария двумя способами:

- `*$PREPROCESS1 Symbols, Brackets;` - аргументы записываются в одну строку через запятую;
- `*$PREPROCESS1 Symbols;`
`*$PREPROCESS1 Brackets;` - пишется несколько псевдокомментариев, каждый из которых содержит аргументы;

Способы записи можно комбинировать, то есть можно написать

`*$PREPROCESS1 Empty, Symbols;`

`*$PREPROCESS1 Brackets;`

и в таком случае все три преобразования будут применены.

Аргументы могут повторяться, в таком случае повторные аргументы игнорируются.

Рассмотрим пример использования препроцессора. В листинге 32 представлен исходный текст программы.

Листинг 32 – Исходная программа

```
1. **$PREPROCESS1 Brackets;  
2. **$PREPROCESS1 Symbols;  
3. **$PREPROCESS1 Brackets, Symbols;  
4. **$PREPROCESS1 Empty;  
5. $ENTRY Go {
```

```

6.      /*empty*/ = <Prout <Func Symb>>;
7.  }
8.
9.  Func {
10.     Symb = ';' -> ')';
11.     Comp = ":-)";
12.     Ident = NewIdent;
13.     Numb  = 42;
14.     Bracket = ("\x22");
15.     s.X = s.X;
16.     e.X = (e.X)
17. }

```

Если раскомментировать только строки 1 и 2 программы, то в результате препроцессирования будет получена программа, представленная в листинге 33.

Листинг 33 – Программа, препроцессированная первым препроцессором

```

1.  *$PREPROCESS1 Brackets;
2.  *$PREPROCESS1 Symbols;
3.  **$PREPROCESS1 Brackets, Symbols;
4.  **$PREPROCESS1 Empty;
5.  $ENTRY Go {
6.      /*empty*/ = <Prout <Func <c1>>>;
7.  }
8.
9.  Func {
10.     Symb = <c2><c3><c4>;
11.     Comp = <c5>;
12.     Ident = <c6>;
13.     Numb  = <c7>;
14.     Bracket = <b1 <c8>>;
15.     s.X = s.X;
16.     e.X = <b2 e.X>
17. }
18. c1 { = Symb;}
19. c2 { = ';' ;}
20. c3 { = '-' ;}
21. c4 { = ')' ;}
22. c5 { = ":-)";}
23. c6 { = NewIdent;}
24. c7 { = 42;}
25. b1 { e.X = (e.X);}
26. c8 { = "\x22";}
27. b2 { e.X = (e.X);}

```

Если раскомментировать только строки 3 и 4 исходной программы, то в результате препроцессирования будет получена программа, представленная в листинге 34.

Листинг 34 – Программа, препроцессированная первым препроцессором

```
1. **$PREPROCESS1 Brackets;
2. **$PREPROCESS1 Symbols;
3. *$PREPROCESS1 Brackets, Symbols;
4. *$PREPROCESS1 Empty;
5. $ENTRY Go {
6.     /*empty*/ =<e1> <Prout<e2> <Func<e3> <c1><e4>><e5>><e6>;
7. }
8.
9. Func {
10.     Symb =<e7> <c2><e8><c3><e9><c4><e10>;
11.     Comp =<e11> <c5><e12>;
12.     Ident =<e13> <c6><e14>;
13.     Numb =<e15> <c7><e16>;
14.     Bracket =<e17> <b1 <e18><c8><e19>><e20>;
15.     s.X =<e21> s.X<e22>;
16.     e.X =<e23> <b2 <e24>e.X<e25>><e26>
17. }
18. e1 { = ;}
19. ...
20. e26 { = ;}
21. c1 { = Symb;}
22. c2 { = ';' ;}
23. c3 { = '-' ;}
24. c4 { = ')' ;}
25. c5 { = ":-)";}
26. c6 { = NewIdent;}
27. c7 { = 42;}
28. b1 { e.X = (e.X);}
29. c8 { = "\x22";}
30. b2 { e.X = (e.X);}
```

Остальные варианты преобразования программы представлены в Приложении А.

6.3. Псевдокомментарий второго препроцессора

Второй препроцессор предоставляет два способа преобразования исходных программ, написанных на языке Рефал-5: первый – преобразование символов в пары «символ-координата», в которых координата является уникальной для каждого символа, второй – преобразование скобочных термов следующим образом: *(значение)* → *((значение) координата)*, где координата является уникальной для каждого скобочного терма.

Выбор вариантов преобразования программы производится с помощью специального псевдокомментария `*$PREPROCESS2` с соответствующими аргументами.

Таблица 3 – Аргументы псевдокомментария `*$PREPROCESS2`

Аргумент	Преобразование
Symbols	Замена символов на пару вида (значение координата), где координата является уникальной для каждого символа.
Brackets	Замена скобочного термина вида (значение) → ((значение) координата), где координата является уникальной для каждого скобочного термина.

Правила записи псевдокомментария совпадают с правилами записи псевдокомментария для первого препроцессора.

Рассмотрим пример работы препроцессора. Листинг 35 содержит текст исходной программы, к которой будет применяться препроцессор.

Листинг 35 – Исходная программа

```

1. **$PREPROCESS2 Brackets;
2. **$PREPROCESS2 Brackets, Symbols;
3. $ENTRY Go { = <Prout <Fab 'AA'>>;}
4. Fab { e.X = <DoFab e.X ('B')> }
5. DoFab {
6.     e.X 'A' (e.Res) = <DoFab e.X ('B' e.Res)>;
7.     e.X s.Y (e.Res) = <DoFab e.X (s.Y e.Res)>;
8.     /*empty*/(e.Res) = ((e.Res));
9. }
```

Если раскомментировать первую строку программы, то будет получена программа, в которой символы заменятся на пары с одинаковыми координатами, а скобочные термины заменятся на пары с уникальными координатами. Препроцессированная программа представлена в листинге 36.

Листинг 36 – Программа, препроцессированная вторым препроцессором

```

1. **$PREPROCESS2 Brackets;
2. **$PREPROCESS2 Brackets, Symbols;
```

```

3. $ENTRY Go { = <Prout <Fab ('A' K)('A' K)>>;}
4. Fab { e.vX = <DoFab e.vX (((('B' K)) K-1)> }
5. DoFab {
6. e.vX ('A' s.K-2) ((e.vRes) s.K-3) = <DoFab e.vX (((('B' K) e.vRes) K-
  4)>;
7. e.vX (s.vY s.K-5) ((e.vRes) s.K-6) = <DoFab e.vX (((s.vY K) e.vRes)
  K-7)>;
8. /*empty*/((e.vRes) s.K-8) = (((((e.vRes) K-9)) K-10);
9. }

```

Если дополнительно раскомментировать вторую строку исходной программы, то будет получена программа, в которой и символы, и скобочные термы заменяются на пары с уникальными координатами. Препроцессированная программа представлена в листинге 37.

Листинг 37 – Программа, препроцессированная вторым препроцессором

```

1. *$PREPROCESS2 Brackets;
2. *$PREPROCESS2 Brackets, Symbols;
3. $ENTRY Go { = <Prout <Fab ('A' K-1)('A' K-2)>>;}
4. Fab { e.vX = <DoFab e.vX (((('B' K-3)) K-4)> }
5. DoFab {
6. e.vX ('A' s.K-5) ((e.vRes) s.K-6) = <DoFab e.vX (((('B' K-7) e.vRes)
  K-8)>;
7. e.vX (s.vY s.K-9) ((e.vRes) s.K-10) = <DoFab e.vX (((s.vY s.K-9)
  e.vRes) K-11)>;
8. /*empty*/((e.vRes) s.K-12) = (((((e.vRes) K-13)) K-14);
9. }

```

Заключение

В ходе выпускной квалификационной работы бакалавра были изучены основные механизмы работы суперкомпилятора Рефала. Были выявлены неточности работы этих инструментов и предложены способы их уточнения.

Были разработаны и реализованы препроцессоры, позволяющие обогащать конфигурации путём преобразования исходных программ. Препроцессированные программы содержат избыточную информацию, которая не влияет на результат вычислений, но учитывается суперкомпилятором.

Было исследовано и проанализировано влияние разработанных препроцессоров на результаты суперкомпиляции. Анализ показал, что препроцессирование действительно позволяет улучшить результаты, позволяя суперкомпилятору более точно различать конфигурации и строить обобщения.

Список использованных источников

1. Turchin V. F. The Concept of a Supercompiler // ACM Transactions on Programming Languages and Systems. — 1986. — Vol. 8, no. 3. — P. 292–325.
2. Климов А. В., Романенко С. А. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М. В. Келдыша. — 2018. — № 111. — 36 с.
3. Turchin V. F. Refal-5: programming guide and reference manual. 1999, (Revised and extended edition of the issue published by New England Publishig Co., Holyoke 1989)
4. Суперкомпилятор SCP4 [Электронный ресурс] – Режим доступа: <http://www.botik.ru/pub/local/scp/refal5/refal5.html>
5. Суперкомпилятор MSCP-A [Электронный ресурс] – Режим доступа: <http://refal.botik.ru/mscp/>
6. Немытых А. П. Специализация функциональных программ методами суперкомпиляции: диссертация ... кандидата физико-математических наук: 05.13.17 / Немытых Андрей Петрович; [Место защиты: Ин-т програм. систем РАН]. — Переславль-Залесский, 2007. — 170 с.
7. Ключников И. Г. Выявление и доказательство свойств функциональных программ методами суперкомпиляции: диссертация ... кандидата физико-математических наук: 05.13.11 / Ключников Илья Григорьевич; [Место защиты: Ин-т прикладной математики им. М.В. Келдыша РАН]. — Москва, 2010. — 189 с.
8. Немытых А. П. О некоторых понятиях суперкомпиляции – методе специализации программ // Сборник трудов по функциональному языку программирования Рефал, том II // Под редакцией А. П. Немытых. — Переславль-Залесский: Издательство «СБОРНИК», 2015. — 156 с.
9. Немытых А. П. Суперкомпилятор SCP4: Общая структура — М.: Издательство ЛКИ, 2015. — 152 с.

10. Непейвода А. Н. Верификация пинг-понг протоколов в модели префиксных грамматик с помощью суперкомпиляции. // Сборник трудов по функциональному языку программирования Рефал, том II // Под редакцией А. П. Немытых. — Переславль-Залесский: Издательство «СБОРНИК», 2015. — 156 с.
11. Dolev D., S. Even S., Karp R.M. On the security of ping-pong protocols // Information and Control Volume 55, Issues 1–3 October–December 1982. doi: 10.1016/S0019-9958(82)90401-6
12. Компилятор Рефал-5λ[Электронный ресурс] – Режим доступа: <https://github.com/bmstu-iu9/refal-5-lambda>

Приложение А

Листинг 38 – Режим преобразования символов

```
*$PREPROCESS1 Symbols;
$ENTRY Go { empty*/ = <Prout <Func <c1>>>; }
Func {
    Symb = <c2><c3><c4>;
    Comp = <c5>;
    Ident = <c6>;
    Numb = <c7>;
    Bracket = (<c8>);
    s.X = s.X;
    e.X = (e.X)
}
c1 { = Symb;}
c2 { = ';' ;}
c3 { = '-' ;}
c4 { = ')' ;}
c5 { = ":-)";}
c6 { = NewIdent;}
c7 { = 42;}
c8 { = "\x22";}
```

Листинг 39 - Режим преобразования скобок

```
*$PREPROCESS1 Brackets;
$ENTRY Go { /*empty*/ = <Prout <Func Symb>>; }
Func {
    Symb = ';' '-' ')';
    Comp = ":-)";
    Ident = NewIdent;
    Numb = 42;
    Bracket = <b1 "\x22">;
    s.X = s.X;
    e.X = <b2 e.X> }
b1 { e.X = (e.X);}
b2 { e.X = (e.X);}

```

Листинг 40 – Режим добавления вызовов функций, возвращающих пустоту

```
*$PREPROCESS1 Empty;
$ENTRY Go { /*empty*/ =<e1> <Prout<e2> <Func<e3> Symb<e4>><e5>><e6>; }
Func {
    Symb =<e7> ';' <e8>'-'<e9>')'<e10>;
    Comp =<e11> ":-)"<e12>;
    Ident =<e13> NewIdent<e14>;
    Numb =<e15> 42<e16>;
    Bracket =<e17> (<e18>"\x22"<e19>)<e20>;
    s.X =<e21> s.X<e22>;
    e.X =<e23> (<e24>e.X<e25>)<e26> }
e1 { = ;}
...
e26 { = ;}
```

Листинг 41 – Режим преобразования символов и добавления вызовов функций, возвращающих пустоту

```

*$PREPROCESS1 Symbols, Empty;
$ENTRY Go { /*empty*/ =<e1> <Prout<e2> <Func<e3> <c1><e4>><e5>><e6>; }
Func {
Symb =<e7> <c2><e8><c3><e9><c4><e10>;
Comp =<e11> <c5><e12>;
Ident =<e13> <c6><e14>;
Numb  =<e15> <c7><e16>;
Bracket =<e17> (<e18><c8><e19>)<e20>;
s.X =<e21> s.X<e22>;
e.X =<e23> (<e24>e.X<e25>)<e26>
}
e1 { = ;}
...
e26 { = ;}
c1 { = Symb;}
c2 { = ';' ;}
c3 { = '-' ;}
c4 { = ')' ;}
c5 { = ":-)";}
c6 { = NewIdent;}
c7 { = 42;}
c8 { = "\x22";}

```

Листинг 42 – Режим преобразования скобок и добавления вызовов функций, возвращающих пустоту

```

*$PREPROCESS1 Brackets, Empty;
$ENTRY Go { /*empty*/ =<e1> <Prout<e2> <Func<e3> Symb<e4>><e5>><e6>; }
Func {
Symb =<e7> ';'<e8>'-'<e9>')'<e10>;
Comp =<e11> ":-)"<e12>;
Ident =<e13> NewIdent<e14>;
Numb  =<e15> 42<e16>;
Bracket =<e17> <b1 <e18>"\x22"<e19>>><e20>;
s.X =<e21> s.X<e22>;
e.X =<e23> <b2 <e24>e.X<e25>>><e26>
}
e1 { = ;}
...
e26 { = ;}
b1 { e.X = (e.X);}
b2 { e.X = (e.X);}

```