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

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Автоматическая разметка оптимизируемых функций в компиляторе Рефала-5λ

Студент ИУ9-81 (Группа)	_____ (Подпись, дата)	Е.А. Калинина (И.О.Фамилия)
Руководитель ВКР	_____ (Подпись, дата)	А.В. Коновалов (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Нормоконтролер	_____ (Подпись, дата)	_____ (И.О.Фамилия)

2020 г.

АННОТАЦИЯ

Тема данной работы – «Автоматическая разметка оптимизируемых функций в компиляторе Рефала-5λ». Целью работы является расширение возможностей оптимизаций компилятора Рефала-5λ добавлением автоматической разметки специализируемых и прогоняемых функций.

Объем работы составляет 77 страниц, включая приложения. Для её написания было использовано 13 источников. Работа включает в себя 14 листингов, 3 таблицы и 1 рисунок.

Дипломная работа состоит из четырёх глав. В первой главе содержатся теоретические сведения, необходимые для реализации автоматической разметки, а именно краткий обзор языка Рефал-5λ и его компилятора, а также определение сущности прогонки и встраивания. Во второй главе описываются критерии, которым должна соответствовать итоговая разметка, и приведён алгоритм, реализующий разметку, соответствующую данным критериям. В третьей главе содержится описание разработанной программы и руководство пользователя. В последней главе проведено тестирование работы дополненного компилятора, теперь расставляющего метки специализации и прогонки автоматически, на некоторой пользовательской программе и на самоприменимом компиляторе. Код разработанной программы, которая встраивается в существующий компилятор, приведён в приложениях к работе.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Высокоуровневые языки программирования были созданы для удобства работы программистов. Сложные структуры данных и операции над ними, которые длинны и громоздки в машинном коде или языках низкого уровня, в высокоуровневых языках представлены абстрактными программными конструкциями: функциями, условными операторами и циклами. Такое представление кода делает процесс программирования удобным и понятным для человека.

Однако такое удобство для пользователя существенно снижает скорость работы программы, что является одним из основных недостатков при работе с языками высокого уровня. Поэтому основной задачей компиляторов является не только перевод языка программирования высокого уровня в машинный код, но и его автоматическая оптимизация с целью увеличения скорости работы программы.

Оптимизация – это процесс преобразования фрагмента исходного кода программы в другой фрагмент, который функционально полностью эквивалентен исходному, однако превосходит его с точки зрения одной или нескольких целевых характеристик, оказывающих влияние на скорость и ресурсоемкость программы.

Целями оптимизации программного кода могут являться:

- Сокращение времени выполнения программы;
- Повышение производительности;
- Повышения компактности программного кода;
- Экономия машинной памяти, расходуемой для выполнения программы;
- Оптимизация энергозатрат оборудования;
- Минимизация количества операций ввода-вывода.

Для выполнения поставленных оптимизационных целей компиляторы могут преобразовывать циклы, выражения и рекурсивные функции с целью рационализации алгоритмов, удалять целые блоки излишнего кода, выполнять их слияние или наоборот разделять один блок на несколько. Работа компилятора направлена на оптимизацию кода программы под процессорную архитектуру, чтобы сделать его действительно быстрым и компактным, при этом полностью сохранив функционал.

Процесс автоматической оптимизации кода, написанного на языке программирования высокого уровня, компилятором позволяет программисту, пишущему на данном языке, сфокусироваться на написании читабельного кода, вместо долгого и нудного процесса его ручной оптимизации. По сути программист сможет сосредоточиться на решении конкретных задач, оставив монотонный однотипный труд компьютеру. К тому же, нередки случаи, когда оптимизацию недостаточно провести один раз, может потребоваться повторная оптимизация кода при дальнейшем развитии и поддержке сложных программных продуктов, в данном случае автоматизация оптимизации поможет многократно сократить трудозатраты программистов на выполнение однотипных операций ручной оптимизации.

На данный момент компилятор Рефала-5λ среди прочих поддерживает такие оптимизации, как специализация и прогонка функций. Но при этом пометка функций, которые нужно оптимизировать, не является автоматической, то есть остаётся задачей программиста.

Конечной целью данной работы является расширение возможностей оптимизаций компилятора Рефала-5λ добавлением корректной и безопасной автоматической разметки специализируемых и прогоняемых функций.

Для этого необходимо:

- Выбрать критерии, по которым будет работать разметка и которые обеспечат её корректность и безопасность;
- Реализовать алгоритм, обеспечивающий автоматическую разметку функций, удовлетворяющую выбранным критериям;
- Оценить влияние данной оптимизации в самоприменимом компиляторе и в отдельных программах.

1 Обзор предметной области, алгоритмов и технологий

1.1 Обзор языка

1.1.1 Особенности языка Рефал

Рефал (РЕкурсивных Функций АЛгоритмический язык) – функциональный язык программирования, ориентированный на осуществление символьных вычислений [2] [3]. Разработан Валентином Турчиным в 1966 году и является одним из старейших функциональных языков программирования высокого уровня. Представляет собой достаточно изящное сочетание направленности на практическую реализацию больших и сложных программ с математической простотой языка. Особенно эффективен при выполнении символьных вычислений: обработке символьных строк, переводе с искусственного на естественный язык, решении проблем, связанных с искусственным интеллектом.

Основной отличительной характеристикой языка Рефал от других функциональных языков программирования является использование особой структуры данных – объектного выражения. В то время как подавляющее большинство функциональных языков используют для структурирования данных однонаправленные списки – последовательности элементов, в которой для непосредственной обработки может быть доступен только левый край, Рефал применяет двунаправленные последовательности, в которых не только возможны операции отсечения или приписывания элементов, расположенных на левом и правом крае, но и операции конкатенации и разбиения последовательности в произвольных местах [2].

Второй важной отличительной особенностью языка является сопоставление с образцом (pattern matching) [2] [3] – метод обработки и анализа структурированных данных, основанный на выполнении заданных инструкций в зависимости от совпадения анализируемого значения с одним из нескольких заданных образцов, описывающих аргумент.

Не стоит воспринимать полувековую историю этого языка как сигнал о том, что он безнадёжно устарел. На протяжении всей истории существования языка модернизировался его синтаксис и актуализировался и расширялся набор дополнительных средств языка, что привело к образованию целой группы диалектов, наиболее распространенными из которых являются Рефал-2, Рефал-5 и Рефал+.

Задача данной работы – расширение возможностей компилятора Рефала-5λ, соответственно именно язык Рефал-5λ и будет рассматриваться далее.

Язык Рефал-5λ является диалектом языка Рефал-5 и одновременно его расширением, то есть его точным надмножеством. Это значит, что компилятор языка может компилировать программы, написанные на языке Рефал-5, без изменений семантических свойств.

Основным отличием диалекта Рефал-5λ является поддержка анонимных функций и функций высшего порядка [1]. Благодаря данному изменению функции могут восприниматься программной средой как самостоятельные структуры данных и могут быть использованы в качестве аргументов вызова других функций, что значительно расширяет практическое применение данного языка.

1.1.2 Введение в Рефал-5λ

Каждая программа, написанная на языке Рефал-5λ, представляет собой набор функций [4]. Определение каждой функции состоит из 2 частей: имя функции и тело функции – блок, заключенный в фигурные скобки.

Функция может состоять из одного или нескольких предложений. Каждое предложение в теле функции – отдельное правило, определяющее как построить значение функции на указанном подмножестве аргументов.

Каждое предложение функции в свою очередь состоит из двух частей: левая часть служит образцом, описывающим подмножество значений аргумента функции, к которым применяется данное предложение, а правая часть – результат, который описывает значение функции на заданном подмножестве. В соответствии с синтаксисом языка, левая и правая части предложения разделяются знаком равенства.

Помимо явно заданных символов, констант, язык позволяет использовать в выражениях неизвестные и произвольные фрагменты – переменные. В Рефале есть 3 типа переменных s-, t- и e-переменные, в зависимости от типа множества значений, которые эти переменные могут принимать.

S-переменная – переменная символа, может принимать значение любого одиночного символа. Под символом в семействе языков Рефал принимается неделимый в плане сопоставления с образцом элемент данных. В случае Рефала-5λ символ может являться литерой, словом, числом, замыканием или указателем на функцию.

E-переменная – переменная выражения, может принимать значение любого фрагмента аргумента функции, в том числе пустого. Для работы с выражением как с единым объектом, его заключают в круглые скобки, которые также называются структурными [5]. Объект, заключённый в структурные скобки, называют скобочным термом, он также может являться частью другого выражения, которое в свою очередь тоже может быть другим скобочным термом. Данное правило описывает принцип иерархической вложенности данных в семье языков Рефал.

T-переменная может принимать значение любого одиночного терма, как символа, являющегося термом, так и выражения в структурных скобках.

Запись каждой переменной состоит из 2 частей: признака типа переменной (s, t, или e) и имени переменной, разделённых точкой. Имя

переменной может быть представлено последовательностью букв и цифр и называется индексом переменной.

Процесс выполнения функции в Рефале разделен на 3 этапа:

1. Выбирается предложение, левая часть которого содержит переменные, из которых путем замены на некоторые значения можно получить аргумент функции. Если данному условию удовлетворяют несколько предложений, выбирается то, что записано выше. Если такого предложения не нашлось, программа завершается, выдавая ошибку отождествления.
2. Происходит фиксация значений переменных, при которых левая часть выбранного предложения принимает значение аргумента функции.
3. Переменным правой части выбранного предложения присваиваются их значения, после чего происходит вычисление функций правой части.

1.1.3 Абстрактная Рефал-машина

Процесс выполнения программы, написанной на языке Рефал-5λ может быть описан с помощью абстрактной Рефал-машины [5]. Абстрактная Рефал-машина – это воображаемая вычислительная машина, воспринимающая синтаксис Рефал-программ. В Рефал-машине заданы 2 области памяти: program field – поле программ, где хранятся все определения функций программы, и view field – поле зрения, в котором хранятся состояния вычислений, полученные в конкретный момент выполнения программы. Состояния вычислений в view field описываются в виде активного выражения – выражения языка, содержащего скобки активации (угловые скобки вызова функции), но не содержащего переменных.

Рефал-машина выполняет программу пошагово, где каждый шаг – это выполнение определенной последовательности действий:

1. Поиск первичного активного подвыражения: Рефал-машина находит в поле зрения самую левую пару скобок активации, в которой внутри не заключены другие угловые скобки.
2. Поиск имени функции: Рефал-машина считывает значение справа от левой скобки активации. Если значение не найдено или не удовлетворяет синтаксису языка, машина выдает ошибку «отождествление невозможно» и прекращает работу.
3. Поиск имени функции в поле программ. Рефал-машина воспринимает как функции, написанные на языке Рефал, так и встроенные. Для встроенной функции управление передаётся на процедуру в машинном коде, реализующую логику данной функции. Для функции, написанной на языке Рефал, машина выбирает для исполнения первое предложение функции.
4. Перебор исполняемых предложений. Рефал-машина подбирает такие значения переменных левой части рабочего предложения, чтобы левая часть обратилась в аргумент функции. Если такие значения можно подобрать, машина переходит к следующему этапу (пункт 5). Если такие значения подобрать невозможно, Рефал-машина берет следующее предложение функции и повторяет данную операцию (пункт 4). Если данные условия не выполняются ни на одном из предложений функции, машина останавливает работу с выдачей ошибки «отождествление невозможно».
5. Подстановка значений переменных. Найденные в ходе исполнения пункта 4 значения переменных подставляются в правую часть текущего предложения. Рефал-машина вставляет полученное выражение на место первичного активного подвыражения в поле зрения.
6. Завершение прохода или выход из цикла работы. Если в поле зрения остались необработанные скобки активации, Рефал-машина выполняет следующий шаг работы, возвращаясь к пункту 1. Если скобок

активации в поле зрения не осталось, работа Рефал-машины корректно завершается.

1.2 Самоприменимый компилятор Рефала-5λ

Компилятор Рефала-5λ – оптимизирующий компилятор, поддерживающий возможность как компиляции в промежуточный интерпретируемый код, так и в исходный код на языке C++. [1]

Работа компилятора осуществляется в несколько проходов:

1. Лексический анализ – идентификация последовательностей, называемых токенами. Компилятор анализирует входную последовательность символов, выделяет распознанные группы, лексемы, и формирует из них токены.
2. Синтаксический анализ – обход последовательности токенов с целью сопоставления с формальной грамматикой языка. В результате данного обхода формируется синтаксическое дерево (дерево разбора) [10].
3. Разрешение подключаемых файлов – поиск в исходных текстах директивы $INCLUDE и подгрузка содержимого указанных файлов. Для всех подгружаемых файлов выполняются шаги прохода с 1 по 3й.
4. Проверка контекстных зависимостей – компилятор пополняет список ошибок контекстно-зависимыми ошибками.
5. Редуктор до подмножества – «обессахариватель». Обход элементов кода с целью поиска и устранения «синтаксического сахара». «Синтаксический сахар» – любой имеющийся в языке программирования элемент синтаксиса, который дублирует другой активный элемент языка, но представляет его в более удобном для программиста виде.
6. Высокоуровневая оптимизация – пометка оптимизируемых функций в синтаксическом дереве программы, оптимизации прогонки и

специализации и возвращение оптимизированного синтаксического дерева.

7. Генерация RASL'а высокого уровня. На 7 и 8 проходе компилятора генерируется промежуточное представление – язык сборки RASL. RASL высокого уровня – это императивный язык, выражающий семантику декларативного языка Рефала, отдельные команды теперь выражают абстрактные императивные действия.
8. Генерация RASL'а низкого уровня. RASL низкого уровня представляет собой набор команд, близких к коду на целевом языке. Отдельные команды низкоуровневого RASL'а могут соответствовать строчкам кода в целевом массиве.
9. Генерация целевого кода – преобразование каждой команды промежуточного кода программы в соответствующий фрагмент кода на языке C++. [6]

Для решения поставленной задачи главным образом придётся работать с шестым проходом – оптимизацией дерева.

1.3 Прогонка

Рассмотрим виды оптимизации, для которых нужно добавить автоматическую разметку.

Прогонка – это один из способов оптимизации программы, при котором происходит замена вызова функции в правой части предложения на результат своей работы. При этом возможно расщепление одного предложения на несколько, имеющих более точные образцы. Основной задачей является композиция нескольких аналогичных шагов Рефал-машины в один [7].

В качестве примера возьмем определение библиотечных функций `Inc` (Листинг 1.1) и `Apply` (Листинг 1.2).

```
$ENTRY Inc {
  s.Num
    = <Add s.Num 1>;
}
```

Листинг 1.1 – Библиотечная функция `Inc`

```
$ENTRY Apply {
  s.Fn e.Argument
    = <s.Fn e.Argument>;

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

Листинг 1.2 – Библиотечная функция `Apply`

В целях оптимизации вызовы данных функций при некоторых аргументах можно вычислить ещё на стадии компиляции:

- `<Inc s.Depth>` → `<Add s.Depth 1>`,
- `<Apply s.Func t.Next>` → `<s.Func t.Next>`.

Рассмотрим ещё один пример, чтобы показать расщепление одного предложения на несколько при прогонке.

Возьмём некоторую функцию `DriveMe` (Листинг 1.3), которая выполняет разные действия, в зависимости от того, в скобках находится поступивший на вход аргумент или нет.

```
DriveMe {
  (e.X)
    = <Bracs>;

  t.X
    = <Simp>;
}
```

Листинг 1.3 – Функция `DriveMe`

и другую функцию `Func`, в теле которой есть вызов `DriveMe` (Листинг 1.4).

```
Func {
  t.N
    = t.N;

  t.F t.S t.O
    = t.F <DriveMe t.S> t.O;
}
```

Листинг 1.4 – Функция `Func`, в теле которой есть вызов `DriveMe`

Данную функцию можно прогнать, в результате чего получим следующее:

```
Func {
  t.N
    = t.N;

  t.F (e.S) t.O
    = t.F <Bracs> t.O;

  t.F t.S t.O
    = t.F <Simp> t.O;

  <..>
}
```

Листинг 1.5 – Функция `Func` после прогонки

1.4 Специализация

В широком смысле специализация – это процесс генерации на основании универсальной программы с множеством параметров программы специализированной, в которой значения части параметров известны и фиксированы. При специализации осуществляется процесс обхода программы, в ходе которого происходит подстановка известных значений в код программы с целью оптимизации и повышения эффективности. [8] [9]

Увеличение скорости работы конечной программы происходит за счёт увеличения времени компиляции, так как часть вычислений, которые ранее должны были быть выполнены на этапе исполнения программы, теперь выполняются в ходе её компиляции.

Мы будем рассматривать частный случай: специализацию отдельно взятого определения функции для её вызова в отдельных точках программы.

К примеру, в случае, когда функция f принимает два аргумента x, y и можно предположить, что первый аргумент x принимает значение A, функция может быть специализирована по этому аргументу:

$$f(x, y) \to f_A(y) \to f(A, y),$$

где f – исходная функция, x, y – аргументы исходной функции, A – предполагаемое значение аргумента x функции f, f_A – специализированная функция, результат специализации функции f по первому аргументу x, равному A.

Процесс специализации выполняет метапрограмма SPEC, называемая специализатором, которая принимает на вход исходную функцию f и значение первого аргумента A и в результате процесса специализации порождает соответствующую функцию f_A, также называемую экземпляром:

$$SPEC(f, A) = f_A$$

В данном случае использование специализатора можно задать следующей формулой:

$$f(x, y) = SPEC(f, A)(y)$$

Таким образом, формально, если F – функция, а *F(args)* – вызов функции F, то специализация – такое порождение новой функции F' и замена вызова *F(args)* на вызов *F'(args')*, что в F' учтена часть статически известной информации о её вызове, например, значения некоторых аргументов *args*. Тогда F' – экземпляр функции F, а информация, учтённая о вызове *F(args)* – сигнатура экземпляра F'.

Приведём пример специализации:

Дана исходная функция `Map` (Листинг 1.6):

```
Map {
  s.Fn t.Next e.Items
    = <s.Fn t.Next> <Map s.Fn e.Items>;

  s.Fn /* пусто */
    = /* пусто */;
}
```

Листинг 1.6 – Функция `Map`, которую можно специализировать

И её вызов (Листинг 1.7):

```
<Map &PrintNum e.Numbers>
```

Листинг 1.7 – Пример вызова функции `Map`, которую можно специализировать

где `PrintNum` – другая функция (Листинг 1.8).

```
PrintNum {
  s.Num
    = <StrFromInt s.Num ', ';

  (s.Re s.Im)
    = <StrFromInt s.Re> '+' <StrFromInt s.Im> 'j, ';
}
```

Листинг 1.8 – Вид функции `PrintNum`

Тогда специализированная функция (Листинг 1.9) и её вызов (Листинг 1.10) будут выглядеть следующим образом:

```
Map'{
  t.Next e.Items
    = <PrintNum t.Next> <Map' e.Items>;

  /* пусто */
    = /* пусто */
}
```

Листинг 1.9 – Специализированная функция `Map'`

```
<Map' e.Numbers>
```

Листинг 1.10 – Вызов специализированной функции `Map'`

В разобранном примере функция Map прошла специализацию по аргументу `s.Fn` – функция `PrintNum` из точки вызова перешла в специализированный экземпляр функции.

Объявление специализации состоит из кодового слова `$SPEC`, имени функции и шаблона специализации, описывающего формат аргумента специализируемого вызова. Формат задаётся жёстким выражением, в котором отсутствуют повторные переменные. Первый символ индекса переменных указывает на то, является ли этот параметр статическим (заглавная латинская буква) или динамическим (строчная латинская буква, цифра, дефис или нижнее подчёркивание).

Статические параметры – это такие параметры, которые проецируются в каждом предложении на одну переменную того же типа, динамические – все остальные. Специализация проводится только по статическим параметрам, при этом в теле функции вместо всех вхождений статического параметра подставляется его фактическое значение.

Средства, необходимые для прогонки и специализации функций, уже присутствуют в компиляторе Рефала-5λ, то есть те функции, которым поставлены метки (`DRIVE` или `SPEC`), будут оптимизированы соответствующим образом, если метки проставлены корректно. Автоматическая расстановка меток и является задачей данной работы, о ней и пойдёт речь далее.

2 Конструкторский раздел

2.1 Критерии автоматической разметки

Для начала выделим критерии, которым должна соответствовать итоговая разметка:

- Безопасность – пометка функции не должна вызывать зацикливание оптимизатора. К примеру, если поставить метку `DRIVE` рекурсивной функции, она будет прогоняться сама в себя и оптимизатор уйдёт в бесконечный цикл;
- Корректность – метки должны удовлетворять некоторым семантическим ограничениям;
- Простота и эффективность – алгоритмы не должны быть излишне сложными и запутанными.

Остановимся подробнее на семантических ограничениях меток:

- Шаблон, по которому проводится специализация, должен согласовываться с определением функции. Все образцы предложений должны быть уточнениями. Статические параметры должны отображаться как переменные того же типа;
- Метки специализации не могут быть назначены специализированным функциям, оканчивающимся на `'@' s.N`, в противном случае это приведет к зацикливанию. Также текущая версия специализатора не поддерживает функций с суффиксами, поэтому их специализация также недопустима;
- Все добавленные в синтаксическое дерево метки должны ссылаться на существующие функции;
- Метки не должны конфликтовать с другими метками. Поскольку оптимизаторы `SPEC` и `DRIVE` считают, что для каждой функции в дереве может быть только одна метка оптимизации определённого

типа, нельзя добавлять новую метку оптимизации к функции, которая уже имеет метку оптимизации того же рода (`SPEC` к `SPEC` или `DRIVE` к `DRIVE`). Аналогично нельзя добавлять в дерево метку `DRIVE` для функций, у которых уже присутствуют метки `INTRINSIC` или `INLINE`.

Задача точной разметки алгоритмически неразрешима, поэтому в данном случае имеет место эвристический подход – поиск критериев, которые показывают хорошую практическую применяемость.

2.2 Реализация автоматической разметки прогоняемых функций

Метки `DRIVE` нельзя назначать рекурсивным функциям, как уже говорилось ранее, это приведёт к зацикливанию. Взаимно-рекурсивной функции можно назначить метку `DRIVE` только если она вызывается в программе один и только один раз, тогда прогонка просто сожмёт эту компоненту сильной связности на одно звено.

Решим задачу поиска рекурсивных связей, построив ориентированный граф вызовов функций G, множество вершин которого V, при этом вершины помечены именами функции, и множество рёбер – E (в графе есть ребро от функции F_1 до функции F_2, если в теле функции F_1 есть хотя бы один вызов F_2). Для рекурсивных функций в графе получаются петли.

Пусть n – количество вершин графа, а m – количество его рёбер.

Компонентой сильной связности называется такое максимальное по включению подмножество вершин C, что любые две вершины этого подмножества достижимы друг из друга: $\forall u,v \in C \quad u \mapsto v, v \mapsto u$, где символом $\mapsto$ обозначается достижимость, то есть существование пути из первой вершины во вторую. Рекурсивные и взаимно-рекурсивные функции в графе

будут являться компонентами сильной связности. Таким образом, расстановку меток DRIVE можно свести к анализу свойств этого графа.

Компоненты сильной связности не могут пересекаться, то есть, по сути, являются разбиением всех вершин графа. Благодаря этому можно определить конденсацию G^{SCC} графом, получаемым из исходного графа сжатием каждой компоненты сильной связности в одну вершину.

Для каждой вершины графа конденсации существует G, а ориентированное ребро между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин $u \in C_i, v \in C_j$, между которыми существовало ребро в исходном графе, то есть $(u,v) \in E$.

Функцию можно помечать меткой DRIVE, если она представляет собой вершину графа конденсации или входит в компоненту, но вызывается внутри неё только один раз.

Такая разметка будет безопасной, то есть не приведёт к зацикливанию. Связано это с ацикличностью графа конденсации.

Теорема 1: Граф конденсации является ацикличным.

Доказательство: Предположим, что $C \mapsto C'$. Докажем, что $C' \not\mapsto C$. Из определения конденсации получаем, что найдутся две вершины $u \in C$ и $v \in C'$, что $u \mapsto v$. Доказывать будем от противного, то есть предположим, что $C' \mapsto C$, тогда найдутся две вершины $u' \in C$ и $v' \in C'$, что $v' \mapsto u'$. Но так как u и u' находятся в одной компоненте сильной связности, то между ними есть путь; аналогично для v и v'. В итоге, объединяя пути, получаем, что $v \mapsto u$, и одновременно $u \mapsto v$. Следовательно, u и v должны принадлежать одной компоненте сильной связности, то есть получили противоречие, что и требовалось доказать. ▷

2.2.1 Алгоритм поиска компонент сильной связности

Приведём алгоритм, который выделяет в графе все компоненты сильной связности.

Алгоритм отличается простотой реализации и основан на двух сериях поиска в глубину, поэтому работа осуществляется за время O(n+m).

Во время первого шага алгоритма осуществляется серия обходов в глубину, охватывающая граф целиком. Для этого алгоритм проходит все вершины графа, а из каждой ещё не посещенной вершины выполняется обход в глубину. При обходе для каждой вершины v фиксируется время выхода $t_{out}(v)$. Ключевую роль в алгоритме играет массив времен выхода, данная роль выражена теоремой №2, которая приведена ниже.

Введём условные обозначения: время выхода $t_{out}(C)$ из компоненты C сильной связности определим как максимум из значений $t_{out}(\mathrm{v})$ для всех $v \in C$. Кроме того, в доказательстве теоремы будут упоминаться и времена входа в каждую вершину $t_{in}(v)$, и аналогично определим времена входа $t_{in}(C)$ для каждой компоненты сильной связности как минимум из величин $t_{in}(v)$ для всех $v \in C$.

Теорема 2: Пусть C и C' – две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро (C, C'). Тогда $t_{out}(C) > t_{out}(C')$.

Доказательство: При доказательстве рассмотрим два принципиально различных случая в зависимости от того, в какую из компонент первой зайдёт обход в глубину, то есть в зависимости от соотношения между $t_{in}(C)$ и $t_{in}(C')$:

1. Случай, когда первой достигается компонента C. В данном случае в определенный момент времени обход в глубину достигает некоторую

вершину v компоненты C, при этом остальные вершины компонент C и C' пока остаются непосещёнными. Поскольку по условию в графе существует ребро (C,C'), то из вершины v может быть достигнута не просто вся компонента C, но и вся компонента C'. Следовательно, при запуске из вершины v обход в глубину охватит все вершины компоненты C и компоненты C', отсюда следует, что они будут потомками вершины v в дереве обхода. Таким образом, для любой вершины $u \in C \bigcup C', u \neq v$ выполняется неравенство $t_{out}(v) > t_{out}(u)$, что и требовалось доказать.

2. Во втором случае первой достигается компонента C'. Аналогично предыдущему варианту в определённый момент времени обход в глубину достигает некую вершину $v \in C'$, при этом все остальные вершины компонент C и C' остаются непосещёнными. Из условия в графе конденсации существует ребро (C,C'), вследствие ацикличности графа конденсации, обратного пути $C' \not\rightsquigarrow C$ существовать не может. Таким образом, обход из вершины v не может достигнуть вершин C. Тогда вершины компоненты C будут посещены при обходе в глубину позже. Отсюда следует: $t_{out}(C) > t_{out}(C')$, что и требовалось доказать. ▷

Алгоритм поиска компонент сильной связности основан на выше доказанной теореме. Отсюда следует, что в графе конденсаций любое ребро (C,C') проходит из компоненты, которая имеет большую величину t_{out}, в компоненту, где эта величина меньше.

При сортировке вершин $v \in V$ по убыванию времени выхода $t_{out}(\mathrm{v})$, первой станет некоторая вершина u, относящаяся к «корневой» компоненте сильной связности, то есть той компоненте, в которую не направлено ни одно из ребер в графе конденсаций.

Следующим шагом необходимо запустить такой обход из полученной вершины u, который бы затронул только эту компоненту сильной связности и не посетил бы никакой другой. Осуществив такой проход, мы сможем поочередно выделить все компоненты сильной связности: запуская обход вершин снова и снова, раз за разом удаляя найденную вершину с наибольшей величиной времени выхода t_{out}.

Для нахождения принципа такого обхода рассмотрим транспонированный граф G^T, то есть граф, который получен из графа G путём изменения направления каждого ребра на противоположное. В полученном графе будут те же компоненты сильной связи, что и в первоначальном графе. Также граф конденсации транспонированного графа $(G^T)^{SCC}$ равен транспонированному графу конденсации исходного графа G^{SCC}. Таким образом, из рассматриваемой нами «корневой» компоненты не будут выходить рёбра в другие компоненты.

Получаем ситуацию, в которой для обхода всей «корневой» компоненты сильной связности, которая содержит некоторую вершину v, достаточно запустить обход из соответствующей вершины v в транспонированном графе G^T. Произведённый обход охватит все вершины данной компоненты сильной связности и никакие более. Как уже упоминалось, мы можем условно исключить эти вершины из графа и продолжить поиск следующей вершины, значение t_{out}(v) которой будет максимальным, и запустить из неё обход на транспонированном графе и повторять данную последовательность действий до достижения искомого результата.

Таким образом, мы выстроили алгоритм нахождения компонент сильной связности:

1. Запуск серии обходов в глубину графа G, которая даёт список вершин в порядке увеличении их времени выхода t_{out}, обозначим этот список $order$;
2. Построение транспонированного графа G^T путём зеркальной смены направлений рёбер исходного графа. Запуск серии обходов в глубину или ширину этого графа в порядке, определённом списком $order$, то есть в порядке уменьшения времени обхода, то есть в обратном порядке. Множество вершин, которые будут достигнуты во время очередного запуска обхода, и будут очередной полученной компонентой сильной связности.

Поскольку алгоритм включает всего 2 обхода в глубину или ширину, его асимптотика будет равняться $O(n+m)$.

Таким образом, всем функциям, которые представляют собой вершину графа конденсации или входят в компоненту, но вызываются внутри неё только один раз, то есть имеют только одно входящее ребро (что можно проверить с помощью приведённого алгоритма), безопасно добавлять в дереве метку `DRIVE`.

2.2.2 Упрощение алгоритма

Легко заметить, что поиск компонент сильной связности необходим только для того, чтобы найти «соединяющие» вершины, то есть вершины, которые имеют ребро, направленное из одной компоненты сильной связности в другую. Очевидно, что такая вершина при обходе графа в топологическом порядке будет обработана первой, следовательно, вызов соответствующей ей функции, происходящий в потомке, будет обработан позднее. Таким образом, все потомки обрабатываются после родителей и вызов, создающий рекурсию, легко может быть найдён как вызов уже обойдённой вершины. Также очевидно, что данное требование достигается базовым алгоритмом поиска в глубину.

Следовательно, алгоритм поиска вершин, прогонка которых невозможна, может быть упрощён. Рассмотрим этот алгоритм.

При обходе графа в глубину нужно запоминать те вершины, к которым ведут обратные ветки. Пусть у нас есть корень R и текущая вершина N. Предки вершины – это путь, по которому мы спустились от R к N, включая R и N:

$$R \equiv P_1 \to P_2 \to \ldots \to P_{k-1} \to P_k \equiv N$$

Если среди потомков N есть хотя бы одна P_i, то от N к P_i ведёт обратная ветка, а вершину P_i нельзя прогонять.

Например, рассмотрим такой граф (Рис. 2.1):

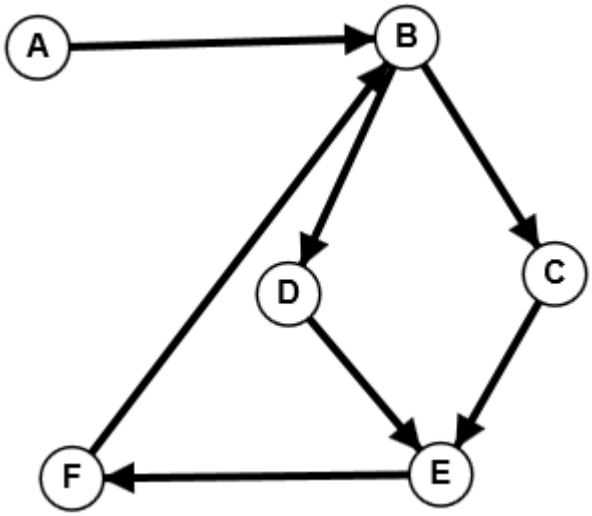

Рис 2.1 – Граф программы, состоящей из функций A, B, C, D, E, F

В данном графе обратное ребро будет только одно: $F \to B$, а значит, прогонять нельзя только вершину B, а остальные можно. Для поиска обратных рёбер достаточно будет только одного обхода графа.

Таким образом, всем функциям, к которым не ведёт обратное ребро при обходе в глубину, можно ставить метку `DRIVE`.

2.3 Реализация автоматической разметки специализируемых функций

В данный момент в компиляторе Рефала-5λ реализовано прерывание специализации. Без реализации такого прерывания, если поставить рекурсивной функции метку специализации, для данной функции будет построен специализированный вариант, содержащий вызов, для которого будет аналогично построен специализированный вариант и так далее – цикл может продолжаться бесконечно до истечения лимита итераций.

Благодаря наличию прерывания обеспечена безопасность специализации при написании пользователем программы, обработка которой должна будет выполняться вечно, и в результирующей программе не будет генерироваться россыпь специализированных функций, каждая из которых вызывает последующую.

Таким образом, благодаря наличию прерывания специализации, помечать можно как нерекурсивные, так и рекурсивные функции. Анализировать граф вызова функций для пометки функций как специализируемых не требуется. Однако провести некоторое исследование для расстановки меток специализации всё равно необходимо, ведь для каждой функции нужно найти шаблон специализации.

Для того чтобы обеспечить компилятору возможность специализации некоторой функции, оптимизатор должен получить спецификацию её формата в виде:

```
$SPEC Имя_Функции Жёсткое_Выражение;
```

Жёсткое_Выражение – это такой образец, который не содержит повторных переменных и не содержит двух e-переменных на одном скобочном уровне.

Вычисление подобного жёсткого выражения для всех функций, которым будет ставиться метка специализации, и является основной задачей дельнейшего исследования.

Для этого необходимо:

Шаг 1:

Вывести форматы функций.

Обозначим, что такое «формат функции». Все функции в Рефале должны содержать один аргумент и возвращать только одно значение, однако на практике не редки случаи, когда необходимо передавать в функцию и/или возвращать из неё несколько независимых значений. Форматом называют способ упаковки нескольких самостоятельных значений в одно объектное выражение. Формат функции – эта пара форматов, один из которых описывает формат аргумента, другой формат результата для данной функции [2] [3].

Сопоставление с образцом – это единственный способ анализа объектного выражения в языковой семье Рефал, поэтому формат должен быть записан в виде некоторого образцового выражения. Для записи формата будет использоваться формула:

```
<Имя_Функции Формат_Аргумента> == Формат_Результата
```

Переменные в форматах аргумента и результата будут соответствовать отдельным значениям, которые упакованы в одном выражении.

Для вывода форматов функций нам потребуется вычислять обобщение всех образцов. Обобщение образца P – это образец P_G такой, что существует подстановка, переводящая P_G в P. Обобщение нескольких образцов $P_1,...,P_n$ – такой образец P_G, что существуют подстановки, переводящие P_G в каждый из исходных образцов.

Для выполнения этой задачи в компиляторе существует функция `GlobalGen`, расположенная в одноимённом файле. Эта функция принимает несколько образцов и возвращает обобщение в виде жёсткого выражения. Функция `GlobalGen` возвращает глобальное сложнейшее обобщение (ГСО) – такое обобщение для нескольких образцов, в котором теряется минимум информации (любое обобщение – потеря информации). Недостаток функции `GlobalGen` – она возвращает выражение, где индексы переменных стёрты. Поэтому полученное выражение необходимо обойти и приписать переменным новые индексы.

Шаг 2:

В полученном обобщении образцов предложения нужно промаркировать статические и динамические параметры.

Напомним, что статические параметры – параметры, которые проецируются в каждом предложении на одну переменную того же типа, а динамические – все остальные. Для того чтобы понять, на что проецируется каждый параметр в обобщении, в составе компилятора есть функция `GenericMatch`, которая реализует алгоритм обобщённого сопоставления, первоначально описанный в работах Турчина [11] [12].

Например, возьмём функцию `Replace` (Листинг 2.1).

```
Replace {
  (e.From) (e.To) e.Text-B e.From e.Text-E
    = e.Text-B e.To <Replace (e.From) (e.To) e.Text-E>;

  (e.F) (e.T) e.Text = e.Text;
}
```

Листинг 2.1 – Функция `Replace`

Вычисляем ГСО для образцов обоих предложений с помощью `GlobalGen` и получаем

```
(e) (e) e
```

Припишем последовательные индексы –

```
(e.0) (e.1) e.2
```

Параметр `e.0` в обоих предложениях проецируется на e-переменную (`e.From` в первом, `e.F` – во втором). Следовательно, он является статическим. Параметр `e.1` по тем же соображениям тоже должен быть статическим. Параметр `e.2` хоть и проецируется в последнем предложении на e-переменную, в первом предложении он проецируется на выражение `e.Text-B e.From e.Text-E`, поэтому он динамический. Таким образом, шаблон специализации:

```
$SPEC Replace (e.STAT1) (e.STAT2) e.dyn3.
```

2.4 Разрешение конфликта пользовательской и автоматической разметки

Гипотетически возможна ситуация, когда метки оптимизации назначаются даже тем функциям, оптимизация которых небезопасна, это может произойти как по вине пользователя, так и за время других проходов компилятора. Так, к примеру, «обессахариватель» может самостоятельно назначать метки прогонки и специализации и присваивает их всем неявно создаваемым функциям: вложенным функциям, блокам и присваиваниям. Хотя на практике такая разметка работает удовлетворительно, в теории она небезопасная, чтобы избежать проблем с зацикливанием, необходимо очистить код от неправильно расставленных меток.

Также, как уже было сказано ранее, у функции не может быть двух меток одного типа или двух конфликтующих меток разных типов.

3 Технологический раздел

3.1 Описание разработанного программного обеспечения

В результате работы в компилятор Рефала-5λ была добавлена возможность автоматической разметки функций, которые могут быть оптимизированы. В итоге при запуске компиляции программы с соответствующим ключом компилятор автоматически добавляет метки `DRIVE` тем функциям, которые по выбранным критериям можно прогонять, и метки `SPEC` тем функциям, которые можно специализировать, и осуществляет их прогонку и специализацию соответственно. Тем самым уменьшается общее время, затрачиваемое на выполнение программы.

Для добавления возможности автоматической разметки оптимизированных функций, в исходный код компилятора были добавлены 5 модулей:

- OptTree-AutoMarkup-GraphExtractor (Приложение А), извлекающий все вызовы функций из строящий набор конструкций вида

  ```
  (Func (<Function>)
     Children ((<Child_1>)(<Child_2>) … (<Child_n>))
  ),
  ```

  где `<Function>` – имя некоторой функции, а `<Child_1> … <Child_n>` – имена всех функций, вызываемых исходной функцией, то есть её «потомков».

- OptTree-AutoMarkup-BasisVertexesExtractor (Приложение Б), обеспечивающий выбор функций, которым может быть добавлена метка `DRIVE` и которые, соответственно, могут быть оптимизированы, на основе выбранных критериев. Напомним, для расстановки меток `DRIVE` были выбран следующий критерий: функцию можно помечать

меткой `DRIVE`, если к ней не ведёт обратное ребро при обходе в глубину.

- OptTree-AutoMarkup-SpecializableExtractor (Приложение В), вычисляющий для каждой функции её шаблон специализации.
- OptTree-AutoMarkup-Common (Приложение Г), содержащий вспомогательные функции, такие как проверка, содержится ли элемент в некотором множестве, проверка двух элементов на равенство, нахождение разности двух множеств, проверка множества на пустоту, а также получение родителей функции на основе списка всех функций с их детьми.
- OptTree-AutoMarkup (Приложение Д), непосредственно добавляющий расставленные метки в дерево и удаляющий некорректные метки, расставленные пользователем.

3.2 Руководство пользователя

3.2.1 Установка и раскрутка самоприменимого компилятора

Выполнение указанных ниже действий потребует установки:

- Git;
- Любого компилятора C++;
- Редактора. Например, можно использовать IntelliJ IDEA, так как он имеет плагин подсветки кода, автоматическую подстановку при вводе и валидацию кода до его компиляции. Это добавляет удобство пользователю, но не критично использовать и любой другой редактор.

Исходники для сборки самоприменимого компилятора можно выгрузить из репозитория github командой:

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

В терминале корневой директории компилятора нужно выполнить команду:

Для раскрутки на Windows:

```
bootstrap.bat --no-tests
```

Для раскрутки на Unix-like системах:

```
./bootstrap.sh --no-tests
```

В результате этих действий генерируется скрипт `c-plus-plus.conf.bat`, в котором нужно приписать путь к установленному компилятору C++. Затем нужно повторно запустить скрипт `bootstrap.bat` (или `bootstrap.sh`).

Последний пункт подготовки: приписать путь к папке `bin` с исполняемыми файлами компилятора к списку каталогов переменной среды `PATH`, а путь к папке `lib` – в `RL_MODULE_PATH`. [1] [2]

3.2.2 Компиляция программы с оптимизацией и логирование компиляции

Для компиляции программы с автоматической расстановкой оптимизируемых функций необходимо указать ключ компиляции `-OA`, с прогонкой `-OD`, со специализацией `-OS` [2]. Ключ можно указать через опции командной строки или через переменные окружения `RLMAKE_FLAGS`, `SREFC_FLAGS`. Ключи можно комбинировать: `-OADS`.

Для отладки программ удобно использовать логирование, которое вызывается после основных стадий компиляций. Для того чтобы получить лог, нужно установить значение соответствующего ключа компиляции – имя файла,

куда сохранится лог. Например, через опции командной строки `--log=log.txt`.

Таким образом, запуск через командную строку компиляции некоторой программы из директории, где она расположена, может выглядеть следующим образом:

```
rlc <имя_программы>.ref --log=log.txt -OADS
```

3.2.3 Самоприменение компилятора

Для самоприменения с оптимизацией и логом нужно установить переменную среды `RLMAKE_FLAGS=-X--log=log.txt` и запустить `makeself` с ключом `-OADS`:

Для Windows:

```
set RLMAKE_FLAGS=-X--log=log.txt -X-OADS makeself.bat
```

Для UNIX:

```
RLMAKE_FLAGS='-X--log=log.txt -X-OADS' ./makeself.sh
```

При этом стоит учитывать, что в тексте программы компилятора уже стоят некоторые вручную расставленные метки. Но автоматическая расстановка меток позволяет найти и другие функции, которые также могут быть оптимизированы, что в теории может дать еще больший прирост производительности.

4 Тестирование

4.1 Тестирование на пользовательской программе

Протестируем работу обновлённого компилятора на примере данного урезанного интерпретатора стекового языка программирования (Листинг 4.1).

```
$ENTRY Go {
  = <Prout '6! = ' <Fact 6>>
}

Fact {
  s.N
    = <Int
        /* пустой стек */

        (
          q s.N   /* ... N */
          q 1     /* ... N 1 */
          swap    /* ... 1 N */
          while (
                  /* ... P N, N ≠ 0 */
            swap  /* ... N P */
            over  /* ... N P N */
            '*'   /* ... N P' */
            swap  /* ... P' N */
            q 1   /* ... P' N 1 */
            '-'   /* ... P' N' */
            )
                  /* ... P 0 */
          drop    /* ... P */
        )
      >
}

$SPEC Int e.stack (e.CODE);

Int {
  e.Stack (e.Code)
    = <Int-Drive e.Stack (e.Code)>;
}

$DRIVE Int-Drive;
```

```
Int-Drive {
  /* примитивные операции */
  e.Stack s.X s.Y (swap e.Code)
    = <Int e.Stack s.Y s.X (e.Code)>;
  e.Stack s.X s.Y (over e.Code)
    = <Int e.Stack s.X s.Y s.X (e.Code)>;
  e.Stack s.X s.Y ('*' e.Code)
    = <Int e.Stack <* s.X s.Y> (e.Code)>;
  e.Stack s.X s.Y ('-' e.Code)
    = <Int e.Stack <- s.X s.Y> (e.Code)>;
  e.Stack s.X (drop e.Code)
    = <Int e.Stack (e.Code)>;
  e.Stack (q s.Num e.Code)
    = <Int e.Stack s.Num (e.Code)>;

  /* выполняем цикл только если на верхушке не ноль */
  e.Stack 0 (while (e.Loop) e.Code)
    = <Int e.Stack 0 (e.Code)>;

  e.Stack s.X (while (e.Loop) e.Code)
    = <Int e.Stack s.X (e.Loop)> : e.Stack^
    = <Int e.Stack (while (e.Loop) e.Code)>;

  e.Stack (/* пусто */)
    = e.Stack;
}
```

Листинг 4.1 – Интерпретатор

У интерпретатора есть две области памяти – стек и память программы. Память программы состоит из команд, из них одна составная – оператор цикла:

- q N – кладёт на вершину число N;
- swap – меняет местами вершину и подвершину;
- over – кладёт на стек подвершину;
- арифметические операции – снимают со стека вершину и подвершину и выполняют с ними арифметическое действие;
- drop – снимает элемент с верхушки стека;
- while (тело цикла) – выполняет тело цикла, пока на верхушке не ноль.

Все команды модифицируют стек.

Попробуем скомпилировать данный интерпретатор с оптимизацией и получим специализированную версию интерпретатора (Листинг 4.2).

```
$ENTRY Go {
  /* empty */ = <Prout '6! = ' <Fact 6>>;
}

Fact {
  s.N#1 = <Int@1 s.N#1>;
}

Int@1 {
  e.Stack#1 s.N#1 = <Int@2 e.Stack#1 s.N#1>;
}

Int@2 {
  e.Stack#1 = <Int@3 e.Stack#1 1>;
}

Int@3 {
  e.0#0 s.X#1 s.Y#1 = <Int@4 e.0#0 s.Y#1 s.X#1>;
}

Int@4 {
  e.#0 0 = <Int@5 e.#0 0>;

  e.#0 s.X#1 = <Int-Drive$8=1@1 <Int@6 e.#0 s.X#1>>;
}

Int@5 {
  e.#0 s.X#1 = <Int@7 e.#0>;
}

Int@6 {
  e.0#0 s.X#1 s.Y#1 = <Int@8 e.0#0 s.Y#1 s.X#1>;
}

Int-Drive$8=1@1 {
  e.Stack#2 = <Int@4 e.Stack#2>;
}
```

```
Int@7 {
  e.Stack#1 = e.Stack#1;
}

Int@8 {
  e.0#0 s.X#1 s.Y#1 = <Int@9 e.0#0 s.X#1 s.Y#1 s.X#1>;
}

Int@9 {
  e.0#0 s.X#1 s.Y#1 = <Int@10 e.0#0 <Mul s.X#1 s.Y#1>>;
}

Int@10 {
  e.0#0 s.X#1 s.Y#1 = <Int@11 e.0#0 s.Y#1 s.X#1>;
}

Int@11 {
  e.Stack#1 = <Int@12 e.Stack#1 1>;
}

Int@12 {
  e.0#0 s.X#1 s.Y#1 = <Int@7 e.0#0 <Sub s.X#1 s.Y#1>>;
}
```

Листинг 4.2 – Специализированная версия интерпретатора

Как видно из результата, для каждого шага вычисления функции факториала была построена специализированная версия интерпретатора – функции с именами `Int@1`, `Int@2`, `Int@3` и так далее. То есть «интерпретатор» при оптимизации полностью растворяется. По сути, была построена первая проекция Футамуры-Турчина, которая заключается в том, что имея специализатор и интерпретатор, мы можем компилировать программы для интерпретатора в целевой язык специализатора [13].

Большинство функций `Int@n` можно прогнать в точки их вызова. Посмотрим результат следующих шагов работы компилятора (Листинг 4.3):

```
$ENTRY Go {
  /* empty */ = <Prout '6! = ' <Int@4 1 6>>;
}
```

```
Int@4 {
  e.#0 0 = e.#0;

  e.0#0 s.X0#1 s.X#1
    = <Int-Drive$8=1@1 <Int@10 e.0#0 s.X#1 <Mul s.X0#1
s.X#1>>>;
}

Int-Drive$8=1@1 {
  e.Stack#2 = <Int@4 e.Stack#2>;
}

Int@7 {
  e.Stack#1 = e.Stack#1;
}

Int@10 {
  e.0#0 s.X#1 s.Y#1 = <Int@7 e.0#0 s.Y#1 <Sub s.X#1 1>>;
}
```

Листинг 4.3 – Результат выполнения прогонки

Как видно, многие функции `Int@n` прогнались. Вызовы `Int-Drive` и `Int@7` не оптимизировались, так как прогонка с активными аргументами на данный момент в компиляторе не поддерживается, но и без этого результирующая программа стала проще и эффективнее. При этом важно отметить, что это стало возможным именно благодаря наличию автоматической оптимизации: вручную поставить метки `DRIVE` функциям `Int@n` было невозможно, так как они создаются в процессе компиляции.

Рассмотрим, как оптимизации повлияли на время работы итоговой программы. Для этого нужно последовательно запустить сборку программы с разными ключами оптимизации и осуществить замер времени работы.

Тестирование проводилось при следующих условиях:

- Тип процессора: Intel Core i3-2370M 2.40 ГГц 2.40 ГГц;
- Объем оперативной памяти: 6 ГБ;

- Операционная система: Windows 7 Home Premium x64;
- Компилятор языка C++: компилятор Visual C++.

Для каждого случая было проведено 13 замеров, определено «типичное» значение, то есть медиана, и вычислен доверительный интервал, определяющий погрешность измерений и представляющий собой интервал между значениями на границах верхнего и нижнего квартилей (Таблица 4.1).

Параметры оптимизации	Медианное значение	Доверительный интервал	
		Левое значение	Правое значение
_	25.164	25.087	25.278
`ODS`	10.933	10.651	11.159
`OADS`	6.892	6.793	7.127

Таблица 4.1 – Результаты тестирования

Сборка с параметрами `OADS` дала уменьшение во времени почти на 37% по сравнению со сборкой с параметрами `ODS`, то есть эффективность итоговой программы возросла.

4.2 Тестирование на самоприменимом компиляторе

Проведём аналогичное тестирование на самоприменимом компиляторе на примере специализации, чтобы убедиться, что и в компиляторе будут найдены новые функции, которые можно оптимизировать, помимо уже размеченных.

Для этого нужно последовательно запустить сборку компилятора со следующими параметрами (Таблица 4.2) и осуществить замер времени работы.

RLMAKE_FLAGS=	Оптимизации отключены.
RLMAKE_FLAGS=-OS	Включена оптимизация специализации, то есть только те функции, у которых уже есть вручную проставленные метки, будут специализированы.
RLMAKE_FLAGS=-OAS	Включена оптимизация специализации, включена автоматическая разметка оптимизируемых функций.

Таблица 4.2 – Ключи компиляции с их значением

Тестирование производилось с помощью специальной утилиты, которая пересобирает компилятор и запускает его компилировать свои исходники (без фазы компоновки), замеряя время исполнения задачи. Для каждого случая было проведено 13 замеров и были получены следующие результаты (Таблица 4.3):

Параметры сборки	Медианное значение	Доверительный интервал	
		Левое значение	Правое значение
RLMAKE_FLAGS=	43.53200	42.91400	44.26500
RLMAKE_FLAGS=-OS	42.23700	41.78200	42.29900
RLMAKE_FLAGS=-OAS	39.32200	39.26300	39.47800

Таблица 4.3 – Результаты тестирования

В компиляторе уже стояли проставленные вручную метки оптимизаций для некоторых функций, что видно из результатов сборки с параметром RLMAKE_FLAGS=-OS. Сборка с параметром RLMAKE_FLAGS=-OAS дала уменьшение во времени почти на 7%. То есть написанный в ходе данной работы модуль автоматической разметки смог помимо уже размеченных найти и другие функции, которые можно оптимизировать.

ЗАКЛЮЧЕНИЕ

В результате работы был реализован алгоритм автоматической разметки прогоняемых и специализируемых функций в самоприменимом компиляторе языка Рефал-5λ. При этом полученная разметка удовлетворяет критериям безопасности, то есть не приводит к зацикливанию компилятора.

Теперь программисту больше не нужно тратить время на осуществление данных оптимизаций вручную, и он сможет больше фокусироваться на своих задачах.

Наиболее перспективными направлениями дальнейшего исследования представляются:

- Выбор альтернативных критериев проведения автоматической разметки и исследование повышения или понижения времени выполнения программ с использованием альтернативных критериев;
- Расширение возможностей прогонки и специализации;
- Добавление автоматической разметки оптимизируемых функций для других видов оптимизации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рефал-5λ [Электронный ресурс] – Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/README.md. Дата обращения: 10.05.2020.
2. Рефал-5λ | Компилятор Рефала-5λ [Электронный ресурс] – Режим доступа: https://bmstu-iu9.github.io/refal-5-lambda. Дата обращения: 10.05.2020.
3. Турчин В.Ф. Руководство по программированию и справочник по языку РЕФАЛ-5 [Электронный ресурс] – Режим доступа: http://www.refal.ru/rf5_frm.htm. Дата обращения: 10.05.2020.
4. Turchin V.F. REFAL-5 programming guide & reference manual [Электронный ресурс] – Режим доступа: http://refal.botik.ru/book/html. Дата обращения: 10.05.2020.
5. Немытых А.П. Лекции по языку программирования Рефал. Сборник трудов по функциональному языку программирования Рефал. Том №1 — Переславль-Залесский: Сборник, 2014.
6. Интерфейсы между отдельными проходами компиляции в компиляторе Рефала-5 λ [Электронный ресурс] – Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/src/compiler/README.md. Дата обращения: 10.05.2020.
7. Романенко С.А. Прогонка для программ на РЕФАЛе-4. Препринт №211 – Институт Прикладной Математики АН СССР, 1987.
8. Климов А.В. Введение в метавычисления и суперкомпиляцию [Раздел книги] // Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям. – Москва: КомКнига, 2008.
9. Климов А.В., Романенко С.А. Метавычислитель для языка Рефал. Основные понятия и примеры [Книга]. – Москва: ИПМ им.М.В.Келдыша АН СССР, 1987.

10. Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий – 2 изд. – Москва: Вильямс, 2008.
11. Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛе – ЦНИПИАС, 1974.
12. Турчин В.Ф. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ – Киев-Алушта, 1972.
13. Абрамов С. М., Пармёнова Л. В. Метавычисления. Часть II. – Переславль-Залесский: «Университет города Переславля», 2016.

ПРИЛОЖЕНИЕ А

Модуль OptTree-AutoMarkup-GraphExtractor

```
$INCLUDE "LibraryEx";

*$FROM DisplayName
$EXTERN DisplayName;

*$FROM OptTree-AutoMarkup-Common
$EXTERN OptTree-AutoMarkup-Contains,
        OptTree-AutoMarkup-GetElemByKey;

/**
  <OptTree-AutoMarkup-ExtractGraph e.AST> ==
e.FunctionCallsGraph
*/

$ENTRY OptTree-AutoMarkup-ExtractFunctionCallsGraph {
  e.AST
    = <MapAccum
        {
          ((e.GraphNodes) (e.IndirectChildren))
          e.AstNode
            = <OptTree-AutoMarkup-GetFuncs e.AstNode>
              :
                {
                  (e.Node) (e.NodeIndirectChildren)
                    = (
                        (e.GraphNodes (e.Node))
                        (e.IndirectChildren
e.NodeIndirectChildren)
                      );

                  e.Other
                    = ((e.GraphNodes)
(e.IndirectChildren));
                }

        }
        ((/*graph nodes*/) (/*indirect children*/)) e.AST
      >
      : ((e.ExtractedFuncs) (e.IndirectChildren))
     = <Map &OptTree-AutoMarkup-GetMetatablesNames e.AST>
      : e.MetatablesNames
    = <Unique e.IndirectChildren>
```

```
      : e.UniqueIndirectChildren
    = <Map
        {
          (e.FuncName)
            , e.MetatablesNames : e.MetatableHead
(e.FuncName) e.MetatableTail
            = ;

          e.Other
            = e.Other;
        }
        e.UniqueIndirectChildren
      >
      : e.ClearedUniqueIndirectChildren
    = (Func (Indirect function for implicit calls)
      Children (e.ClearedUniqueIndirectChildren))
      : e.IndirectFuncNode
    = <Map &OptTree-AutoMarkup-GetMetatablesNames e.AST>
      : e.MetatablesNames
    = <Map &OptTree-AutoMarkup-GetMetatables e.AST>
      : e.ExtractedMetaTables
    = <OptTree-AutoMarkup-InlineFuncCallsWithMetatable
        (e.ExtractedFuncs)
        (e.ExtractedMetaTables)
      >
      : e.InlinedGraph
    = <Map &OptTree-AutoMarkup-GetEntryPoints e.AST>
      : e.EntryPoints
    = e.EntryPoints e.UniqueIndirectChildren
      : e.EntryPoints^
    = <OptTree-AutoMarkup-GetClearedByEntryPointsGraph
        (e.InlinedGraph)
        (e.EntryPoints)
      >
      : ((e.ResultGraph) (e.InlinedGraph^))
    = <Unique e.IndirectFuncNode e.ResultGraph>;
}

OptTree-AutoMarkup-GetEntryPoints {
  (Function GN-Entry (e.FuncName) Sentences e.Body)
    = (e.FuncName);

  (Function s.Scope (e.FuncName) Sentences e.Body)
    ,<OneOf (e.FuncName) (INIT) (FINAL)> : True
    = (e.FuncName);
```

```
  e.Smth
    = ;
}

OptTree-AutoMarkup-GetClearedByEntryPointsGraph {
  (e.Graph) (e.EntryPoints)
    = <OptTree-AutoMarkup-GetClearedByEntryPointsGraph
        (e.Graph)
        ()
        (e.EntryPoints)
      >;

  (e.Graph) (e.ProccessedNodes) (e.EntryPoints)
    = <MapAccum
        {
          ((e.ProcessedNodes) (e.Result) (e.Graph^))
          (e.EntryPoint)
            , <OptTree-AutoMarkup-Contains
                (e.EntryPoint)
                (e.ProcessedNodes)
              > : False
            = (e.ProcessedNodes (e.EntryPoint))
              : (e.NewProcessedNodes)
            = <OptTree-AutoMarkup-GetElemByKey
                (e.EntryPoint)
                (e.Graph)
              >
              : e.Node (e.Graph^)
            = e.Node
              : {
                  Func (e.FuncName) Children (e.Children)
                    = <OptTree-AutoMarkup-
GetClearedByEntryPointsGraph
                        (e.Graph)
                        (e.NewProcessedNodes)
                        (e.Children)
                      >
                      : ((e.SubTreeResult) (e.Graph^))
                    = (
                        (e.NewProcessedNodes)
                        (
                          e.Result
                          (Func (e.FuncName) Children
(e.Children))
```

```
                            e.SubTreeResult
                          )
                          (e.Graph)
                        );

                  e._
                    = (
                        (e.NewProcessedNodes)
                        (e.Result)
                        (e.Graph)
                        );
                };

          ((e.ProcessedNodes) (e.Result) (e.Graph^))
e.Other
            = ((e.ProcessedNodes) (e.Result) (e.Graph));
        }
        ((e.ProccessedNodes) (/*result*/) (e.Graph))
e.EntryPoints
      >
      : ((e.ProccessedNodes^) (e.Result) (e.Graph^))
    = ((e.Result) (e.Graph))
}

OptTree-AutoMarkup-GetMetatables {
  (Function s.ScopeClass (e.Name) Metatable e.Metatable)
    = <Map
        &OptTree-AutoMarkup-
ExtractFunctionCallsFromMetatable
        e.Metatable
      >
      : e.FunctionCalls
    = (e.Name (<Unique e.FunctionCalls>));

  e.Smth
    = ;
}

OptTree-AutoMarkup-GetMetatablesNames {
  (Function s.ScopeClass (e.Name) Metatable e.Metatable)
    = (e.Name);

  e.Smth
    = ;
}
```

```
OptTree-AutoMarkup-GetFuncs {
  (Function s.ScopeClass (e.Name) Sentences e.Body)
    = <MapAccum
        {

((e.DirectCalls)(e.ImplicitCalls)(e.IndirectChildren))
          (e.Sentence)
            = <OptTree-AutoMarkup-
ExtractFunctionCallsFromSentencesBody
                (e.Sentence)
              >
              : (e.SentenceDirectCalls)
                (e.SentecnseImplicitCalls)
                (e.SentecnseIndirectChildren)
            = (
                (e.DirectCalls e.SentenceDirectCalls)
                (e.ImplicitCalls
e.SentecnseImplicitCalls)
                (e.IndirectChildren
e.SentecnseIndirectChildren)
              )
        }
        ((/*direct calls*/) (/*implicit calls*/)
(/*indirect children*/))
        e.Body
      >
      : ((e.DirectCalls) (e.ImplicitCalls)
(e.IndirectChildren))
    = <Unique e.DirectCalls> : e.UniqueDirectCalls
    = (e.ImplicitCalls)
      : {
          ()
            = ;

          (e._)
            = (Indirect function for implicit calls)
        }
        : e.IndirectChild
    = (Func (e.Name) Children (e.UniqueDirectCalls
e.IndirectChild))
      (e.IndirectChildren);

  e.Smth
    = ;
```

```
}

OptTree-AutoMarkup-ExtractFunctionCallsFromMetatable {
  ((Symbol Identifier e.Ident) (Symbol Name e.Name))
    = (e.Name);
}

OptTree-AutoMarkup-ExtractFunctionCallsFromSentencesBody
{
  ((e.Pattern) e.Conditions (e.Result))
    = <OptTree-AutoMarkup-ExtractExpressionName
e.Pattern>
      : (e.PatternNames) (e.PatternIndirectCalls)
(e.PatternIndirectChildren)
    = <MapAccum
        {

((e.DirectCalls)(e.IndirectCalls)(e.IndirectChildren))
          (Condition (e.Name) (e.ConditionResult)
(e.ConditionPattern))
              = <OptTree-AutoMarkup-ExtractExpressionName
                  e.ConditionResult
                >
                : (e.ConditionResultCalls)
                  (e.ConditionResultIndirectCalls)
                  (e.ConditionResultIndirectChildren)
              = <OptTree-AutoMarkup-ExtractExpressionName
                  e.ConditionPattern
                >
                : (e.ConditionPatternCalls)
                  (e.ConditionPatternIndirectCalls)
                  (e.ConditionPatternIndirectChildren)
              = (
                  (
                    e.DirectCalls
                    e.ConditionResultCalls
                    e.ConditionPatternCalls
                  )
                  (
                    e.IndirectCalls
                    e.ConditionResultIndirectCalls
                    e.ConditionPatternIndirectCalls
                  )
                  (
                    e.IndirectChildren
```

```
                    e.ConditionResultIndirectChildren
                    e.ConditionPatternIndirectChildren
                  )
                );
        }
        ((/*direct calls*/) (/*indirect calls*/)
(/*indirect children*/))
        e.Conditions
      >
      : (
          (e.ConditionNames)
          (e.ConditionIndirectCalls)
          (e.ConditionIndirectChildren)
        )
    = <OptTree-AutoMarkup-ExtractExpressionName e.Result>
      : (e.ResultNames) (e.ResultIndirectCalls)
(e.ResultIndirectChildren)
    = (
        e.PatternNames
        e.ConditionNames
        e.ResultNames
      )
      (
        e.PatternIndirectCalls
        e.ConditionIndirectCalls
        e.ResultIndirectCalls
      )
      (
        e.PatternIndirectCalls
        e.ConditionIndirectChildren
        e.ResultIndirectChildren
      );
}

OptTree-AutoMarkup-InnerExtractExpressionName {
  (e.Expr) (e.DirectCalls) (e.IndirectDetected)
(e.IndirectChildren)
    = <MapAccum
        {
          ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
          (Symbol Name e.Name)
            = (
                (e.DirectCalls (e.Name))
                (e.IndirectDetected)
```

```
                (e.IndirectChildren)
              );

          ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
          (t.Marker e.Nested)
            , <OneOf
                (t.Marker)
                (Brackets)
                (ADT-Brackets)
                (ClosureBrackets)
              > : True
            = <OptTree-AutoMarkup-
InnerExtractExpressionName
                (e.Nested)
                (e.DirectCalls)
                (e.IndirectDetected)
                (e.IndirectChildren)
              >;

          ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
          (CallBrackets
            (Symbol Name e.Name)
            e.NestedHead
            (Symbol Name e.IndirectChild)
            e.NestedTail
          )
            = (e.IndirectChild)
              <Map
                {
                  (Symbol Name e.NextInderectChild)
                    = (e.NextInderectChild);

                  e.Other
                    = ;
                }
                e.NestedTail
              >
              : e.AdditionaIndirectChildren
            = <Map
                {
                  (Symbol Name e._)
                    = ;
```

```
                  e.Other
                    = e.Other;
                }
                e.NestedHead e.NestedTail
            >
            : e.Nested^
          = <OptTree-AutoMarkup-
InnerExtractExpressionName
              (e.Nested)
              (e.DirectCalls)
              (e.IndirectDetected)
              (e.IndirectChildren)
            >
            : (
                (e.DirectCalls^)
                (e.IndirectDetected^)
                (e.IndirectChildren^)
              )
          = (
              (e.DirectCalls (e.Name))
              (e.IndirectDetected)
              (e.IndirectChildren
e.AdditionaIndirectChildren)
            );

        ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
        (CallBrackets (Symbol Name e.Name) e.Nested)
          = <OptTree-AutoMarkup-
InnerExtractExpressionName
              (e.Nested)
              (e.DirectCalls)
              (e.IndirectDetected)
              (e.IndirectChildren)
            >
            : (
                (e.DirectCalls^)
                (e.IndirectDetected^)
                (e.IndirectChildren^)
              )
          = (
              (e.DirectCalls (e.Name))
              (e.IndirectDetected)
              (e.IndirectChildren)
            );
```

```
          ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
          (CallBrackets e.Nested)
            = <OptTree-AutoMarkup-
InnerExtractExpressionName
                (e.Nested)
                (e.DirectCalls)
                (e.IndirectDetected)
                (e.IndirectChildren)
              >
              : (
                  (e.DirectCalls^)
                  (e.IndirectDetected^)
                  (e.IndirectChildren^)
                )
              = (
                  (e.DirectCalls)
                  (Indirect detected)
                  (e.IndirectChildren)
                );

          ((e.DirectCalls^) (e.IndirectDetected^)
(e.IndirectChildren^))
          (t.Marker e.Smth)
            = ((e.DirectCalls) (e.IndirectDetected)
(e.IndirectChildren));
        }
        ((e.DirectCalls) (e.IndirectDetected)
(e.IndirectChildren))
        e.Expr
      >;
}

OptTree-AutoMarkup-ExtractExpressionName {
  e.Expr
    = <OptTree-AutoMarkup-InnerExtractExpressionName
(e.Expr) () () ()>
      : ((e.DirectCalls) (e.IndirectDetected)
(e.IndirectChildren))
    = (e.DirectCalls) (e.IndirectDetected)
(e.IndirectChildren)
}

OptTree-AutoMarkup-InlineFuncCallsWithMetatable {
```

```
  (e.ExtractedFuncs) (e.ExtractedMetaTables)
    = <Map
        {
          e.Elem
            = <OptTree-AutoMarkup-
InlineOneFuncChildrenWithMetatable
                e.Elem
                (e.ExtractedMetaTables)
              >;
        }
        e.ExtractedFuncs
      >;
}

OptTree-AutoMarkup-InlineOneFuncChildrenWithMetatable {
  (Func (e.Name) Children (e.UniqueFunctionCalls))
(e.ExtractedMetaTables)
      = <Map
          {
            e.Elem
              = <OptTree-AutoMarkup-
GetCallsFromMetaTables
                  e.Elem
                  (e.ExtractedMetaTables)
                >;
          }
          e.UniqueFunctionCalls
        >
        : e.ReplacedChildren
      = <Unique e.ReplacedChildren>
        : e.UniqueReplacedChildren
      = (Func (e.Name) Children
(e.UniqueReplacedChildren));
}

OptTree-AutoMarkup-GetCallsFromMetaTables {
  (e.FuncName) (e.ExtractedMetaTables)
    = (<Map
        {
          (e.Key (e.Value))
            , e.Key : e.FuncName
            = e.Value;

          (e.Other)
            = ;
```

```
          }
          e.ExtractedMetaTables
        >)
        : e.ReplaceValue
    = <Fetch
        e.ReplaceValue
        {
          ()
            = (e.FuncName);

          (e.FuncSet)
            = e.FuncSet;
        }
      >;
}
```

ПРИЛОЖЕНИЕ Б

Модуль OptTree-AutoMarkup-BasisVertexesExtractor

```
$INCLUDE "LibraryEx";

*$FROM OptTree-AutoMarkup-Common
$EXTERN OptTree-AutoMarkup-GetSetDifference,
        OptTree-AutoMarkup-Contains,
        OptTree-AutoMarkup-GetElemByKey,
        OptTree-AutoMarkup-GetParents,
        OptTree-AutoMarkup-Equal;

/**
    <OptTree-AutoMarkup-ExtractBasisVertexes
e.ExtractedGraph>
    = (e.BasisVertex)* (e.Graph)

    e.BasisVertex = e.FuncName
*/

$ENTRY OptTree-AutoMarkup-ExtractBasisVertexes {
  e.Graph
    = <OptTree-AutoMarkup-GetGraphRoots (e.Graph)>
      : (e.GraphRoots)
    = <OptTree-AutoMarkup-GetBasisVertexes
        (e.GraphRoots)
        e.Graph
      >
      : (
          (e.CurrentPath)
          (e.Proccessed)
          (e.BasisVertexes)
          (e.Graph^)
        )
    = (e.BasisVertexes) (e.Graph);
}

OptTree-AutoMarkup-GetGraphRoots {
  (e.Graph)
    = <MapAccum
        {
          ((e.Result) (e.Graph^))
          (Func (Indirect function for implicit calls)
Children (e.Children))
            = ((e.Result) (e.Graph));
```

```
          ((e.Result) (e.Graph^))
          (Func (e.FuncName) Children (e.Children))
            , e.Graph
              : e.GraphHead
                (
                  Func (e.ParentName)
                  Children (e.ChildrenHead (e.FuncName)
e.ChildrenTail)
                )
                e.GraphTail
            , <OptTree-AutoMarkup-Equal (e.ParentName)
(e.FuncName)> : False
            = ((e.Result) (e.Graph));

          ((e.Result) (e.Graph^))
          (Func (e.FuncName) Children (e.Children))
            = ((e.Result (e.FuncName)) (e.Graph))
        }
        ((/*result*/) (e.Graph)) e.Graph
      >
      : ((e.Roots) (e.Graph^))
    = (e.Roots);
}

OptTree-AutoMarkup-GetBasisVertexes-GetResultByNode {
  (Func (e.Name) Children (e.Children))
  ((e.CurrentPath) (e.Proccessed) (e.BasisVertexes)
(e.Graph))
    = <MapAccum
        {
          ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
          (e.ChildFunc)
            , e.BasisVertexes : e._ (e.ChildFunc) e._
            = (
                (e.CurrentPath)
                (e.Proccessed)
                (e.BasisVertexes)
                (e.Graph)
              );

          ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
          (e.Name)
```

```
            , e.CurrentPath : e._ (e.ChildFunc) e._
            = (
                (e.CurrentPath)
                (e.Proccessed)
                (e.BasisVertexes (e.Name))
                (e.Graph)
              );

          ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
          (e.ChildFunc)
            , e.CurrentPath : e._ (e.ChildFunc) e._
            = (
                (e.CurrentPath)
                (e.Proccessed)
                (e.BasisVertexes (e.ChildFunc))
                (e.Graph)
              );

          ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
          (e.ChildFunc)
            , e.Proccessed : e._ (e.ChildFunc) e._
            = (
                (e.CurrentPath)
                (e.Proccessed)
                (e.BasisVertexes)
                (e.Graph)
              );

          ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
          (e.ChildFunc)
            = e.CurrentPath (e.ChildFunc) :
e.NewCurrentPath
            = <OptTree-AutoMarkup-GetBasisVertexes
                ((e.ChildFunc))
                (
                  (e.NewCurrentPath)
                  (e.Proccessed)
                  (e.BasisVertexes)
                  (e.Graph)
                )
              >
              : (
```

```
                    (e.NewCurrentPath^)
                    (e.Proccessed^)
                    (e.BasisVertexes^)
                    (e.Graph^)
                  )
              = (
                  (e.CurrentPath)
                  (e.Proccessed)
                  (e.BasisVertexes)
                  (e.Graph)
                );
          }
          ((e.CurrentPath) (e.Proccessed) (e.BasisVertexes)
(e.Graph))
          e.Children
        >
        : ((e.CurrentPath^) (e.Proccessed^)
(e.BasisVertexes^) (e.Graph^))
        = e.Proccessed (e.Name) : e.NewProccessed
        = (
            (e.CurrentPath)
            (e.NewProccessed)
            (e.BasisVertexes)
            (e.Graph)
          );

  (e.Other)
  ((e.CurrentPath) (e.Proccessed) (e.BasisVertexes)
(e.Graph))
    = ((e.CurrentPath) (e.Proccessed) (e.BasisVertexes)
(e.Graph));
}


OptTree-AutoMarkup-GetBasisVertexes {
  (e.NextGraphNodes)
  (
    (e.CurrentPath)
    (e.Proccessed)
    (e.BasisVertexes)
    (e.Graph)
  )
    = <MapAccum
        {
```

```
((e.CurrentPath^)(e.Proccessed^)(e.BasisVertexes^)(e.Grap
h^))
           (e.FuncName)
             = <OptTree-AutoMarkup-GetElemByKey
                 (e.FuncName)
                 (e.Graph)
               >
               : e.Node (e.Graph^)
             = <OptTree-AutoMarkup-GetBasisVertexes-
GetResultByNode
                 (e.Node)

((e.CurrentPath)(e.Proccessed)(e.BasisVertexes)(e.Graph))
               >
         }
         (
           (e.CurrentPath)
           (e.Proccessed)
           (e.BasisVertexes)
           (e.Graph)
         )
         e.NextGraphNodes
       >;

  (e.RootNodes)
  e.Graph
    = <OptTree-AutoMarkup-GetBasisVertexes
         (e.RootNodes)
         (
           ()
           ()
           ()
           (e.Graph)
         )
       >;
}
```

ПРИЛОЖЕНИЕ В

Модуль OptTree-AutoMarkup-SpecializableExtractor

```
$INCLUDE "LibraryEx";

*$FROM GlobalGen
$EXTERN GlobalGen;

*$FROM GenericMatch
$EXTERN GenericMatch;

*$FROM Log
$EXTERN Log-PutLine;

/**
  <OptTree-AutoMarkup-GetSpecializableDict (e.AST)>
    == ((e.FuncName) '->' ((e.TypedVariable)+))

  e.TypedVariable = {s.VariableType}
                    {STAT | dyn}
                    {index}
*/

$ENTRY OptTree-AutoMarkup-GetSpecializableDict {
  (e.AST)
    = <Map
        {
          (Function s.ScopeClass (e.Name) Sentences
e.Body)
            = <Map
                {
                  ((e.Pattern) e.Conditions (e.Result))
                    = (e.Pattern);

                  e.Other
                    = ;
                }
                e.Body
              >
              : e.ExtractedPatterns
            = <OptTree-AutoMarkup-ContainsDuplications
                (e.ExtractedPatterns)
              >
              : {
                  True
```

```
                    = (NOTSPECIALIZABLE (e.Name));

                  False
                    = <OptTree-AutoMarkup-
GetSpecifiedPattern
                        (e.ExtractedPatterns)
                      >
                      : e.SpecifiedPattern
                    = <OptTree-AutoMarkup-
GetSpecializableDictNode
                        (e.Name)
                        (e.SpecifiedPattern)
                      >
                      : e.DictNode
                    = e.DictNode;
              };

          e.Other
            = ;
        }
        e.AST
      >;

  e.Other
    = ;
}

OptTree-AutoMarkup-GetSpecializableDictNode {
  (e.FuncName) (e.SpecifiedPattern)
    , e.SpecifiedPattern : e.Head ('e.STAT' e.Suffix)
e.Tail
    = (SPECIALIZABLE (e.FuncName) '->'
(e.SpecifiedPattern));

  (e.FuncName) e.Other
    = (NOTSPECIALIZABLE (e.FuncName));
}

OptTree-AutoMarkup-ContainsDuplications {
  (e.ExtractedPatterns)
    = <Map
        {
          (e.Patterns)
            = <Map
                &OptTree-AutoMarkup-GetPatternVariables
```

```
                e.Patterns
              >
              : e.Stm
            = e.Stm
              : {
                  e.Head (e.Variable) e._ (e.Variable)
e.Tail
                    = (True);

                  e.Other
                    = (False);
                };
        }
        e.ExtractedPatterns
      >
      : {
          e.Head (True) e.Tail
            = True;

          e.Other
            = False;
        }
}

OptTree-AutoMarkup-GetPatternVariables {
  (TkVariable e.Name)
    = (e.Name);

  (Symbol Identifier VAR e.Name)
    = (e.Name);

  (Brackets e.Nested)
    = <Map
        {
          (e.SubPattern)
            = <OptTree-AutoMarkup-GetPatternVariables
(e.SubPattern)>;
        }
        e.Nested
      >;

  (e.Other)
    = ;
}
```

```
OptTree-AutoMarkup-GetSpecifiedPattern {
  (e.ExtractedPatterns)
    = <OptTree-AutoMarkup-GetIndexedGlobalGens
(e.ExtractedPatterns)>
      : e.IndexedGlobalGens
    = <Map
        {
          (e.ConcretePattern)
            = <GenericMatch
                (e.ConcretePattern)
                (e.IndexedGlobalGens)
              >
              : e.GenericMatch
            = (e.GenericMatch);
        }
        e.ExtractedPatterns
      >
      : e.GenericMatches
    = <OptTree-AutoMarkup-GetVariableMatches
        (e.GenericMatches)
        (e.IndexedGlobalGens)
      >
      : e.VariableMatchesDict
    = <OptTree-AutoMarkup-GetVariableSpecType
(e.VariableMatchesDict)>
      : e.TypedVariables
    = e.TypedVariables;
}

OptTree-AutoMarkup-GetIndexedGlobalGens {
  ()
    = ();

  (e.Patterns)
    = <GlobalGen e.Patterns>
      : e.GlobalGens
    = <OptTree-AutoMarkup-FillIndexes (e.GlobalGens) 't'>
      : e.Result
    = <OptTree-AutoMarkup-FillIndexes (e.Result) 'e'>
      : e.Result^
    = <OptTree-AutoMarkup-FillIndexes (e.Result) 's'>
      : e.Result^
    = e.Result;

  e.Other
```

```
    = ;
}

OptTree-AutoMarkup-FillIndexes {
  (e.GlobalGens) s.Type
    = <OptTree-AutoMarkup-FillIndexes (0) (e.GlobalGens)
s.Type>
      : (e._) e.IndexedTypeGlobalGens
    = e.IndexedTypeGlobalGens;

  (e.StartIndex) (e.GlobalGens) s.Type
    = <MapAccum
        {
          (e.CurrentIndex) (TkVariable s.Type)
            = TkVariable s.Type '.' e.CurrentIndex
              : e.IndexedVariable
            = <Add e.CurrentIndex 1>
              : e.NewIndex
            = (e.NewIndex) (e.IndexedVariable);

          (e.CurrentIndex) (TkVariable e.Other)
            = (e.CurrentIndex) (TkVariable e.Other);

          (e.CurrentIndex) (Brackets e.Nested)
            = <OptTree-AutoMarkup-FillIndexes
                (e.CurrentIndex)
                (e.Nested)
                s.Type
              >
              : (e.NextIndex) e.IndexedTypeGlobalGens
            = (e.NextIndex) (Brackets
e.IndexedTypeGlobalGens);

          (e.CurrentIndex) (e.Other)
            = (e.CurrentIndex) (e.Other);
        }
        (e.StartIndex) e.GlobalGens
      >
}

OptTree-AutoMarkup-GetVariableMatchesByIndexedGlobalGen {
  TkVariable e.VariableName (e.GenericMatches)
    = <Map
        {
          (Clear e.Dict)
```

```
            = <OptTree-AutoMarkup-
GetMatchesByVariableName
                (e.VariableName)
                (e.Dict)
              >;

          e.Other
            = ;
        }
        e.GenericMatches
      >
      : e.CurrentVariableRes
    = <OptTree-AutoMarkup-GetVariablesTypes
e.CurrentVariableRes>
      : e.CurrentVariableTypeRes
    = (e.VariableName ':' e.CurrentVariableTypeRes);

  Brackets e.Nested (e.GenericMatches)
    = <OptTree-AutoMarkup-
GetVariableMatchesByIndexedGlobalGen
        e.Nested
        (e.GenericMatches)
      >;

  (e.Nested) (e.GenericMatches)
    = <OptTree-AutoMarkup-
GetVariableMatchesByIndexedGlobalGen
        e.Nested
        (e.GenericMatches)
      >;

  e.Other
    = ;
}

OptTree-AutoMarkup-GetVariableMatches {
  (e.GenericMatches) (e.IndexedGlobalGens)
    = <Map
        {
          e.IndexedGlobalGenNode
            = <OptTree-AutoMarkup-
GetVariableMatchesByIndexedGlobalGen
                (e.IndexedGlobalGenNode)
                (e.GenericMatches)
              >;
```

```
        }
        e.IndexedGlobalGens
      >
}

OptTree-AutoMarkup-GetVariablesTypes{
  ()
    = ();

  TkVariable 's' e._
    = TkVariable 's';
  TkVariable 't' e._
    = TkVariable 't';
  TkVariable 'e' e._
    = TkVariable 'e';

  Symbol e.Tail
    = Symbol e.Tail;

  Symbol Char e.Tail
    = Symbol Char e.Tail;

  Brackets e.Nested
    = <Map
        &OptTree-AutoMarkup-GetVariablesTypes
        e.Nested
      >
      : e.NestedRes
    = Brackets e.NestedRes;

  (e.Nested) e.Tail
    = (<OptTree-AutoMarkup-GetVariablesTypes e.Nested>)
      <OptTree-AutoMarkup-GetVariablesTypes e.Tail>;

  e.Other
    = e.Other;
}

OptTree-AutoMarkup-IsAllEqual {
  (e.Single)
    = True;

  (e.Value) (e.Value)
    = True;
```

```
  (e.First) (e.Next) e.Tail
    , e.Next : e.First
    = <OptTree-AutoMarkup-IsAllEqual (e.First) e.Tail>;

  e._
    = False;
}

OptTree-AutoMarkup-GetMatchesByVariableName {
  (e.Key) (e.Head (e.Value ':' (e.Key)) e.Tail)
    = (e.Value);

  (e.Key) e.Other
    = ;
}

OptTree-AutoMarkup-GetVariableSpecType {
  (e.VariableMatchesDict)
    = <Map
        {
          (t.VariableType '.' t.VariableIndex ':'
e.Matches)
            = <OptTree-AutoMarkup-IsAllEqual e.Matches>
              : {
                  True
                    = (t.VariableType '.STAT'
t.VariableIndex);

                  False
                    = (t.VariableType '.dyn'
t.VariableIndex);
                }
        }
        e.VariableMatchesDict
      >
}
```

ПРИЛОЖЕНИЕ Г

Модуль OptTree-AutoMarkup-Common

```
$INCLUDE "LibraryEx";

/**
    <OptTree-AutoMarkup-GetSetDifference (e.FirstSet)
(e.SecondSet)> ==
  (e.FirstSet\e.SecondSet)
*/

$ENTRY OptTree-AutoMarkup-GetSetDifference {
  (e.FirstSet) (e.SecondSet)
    = <Map
        {
          e.Node, <OptTree-AutoMarkup-Contains e.Node
(e.SecondSet)> : False
            = e.Node;

          e.Other
            = ;
        }
        e.FirstSet
      >
      : e.Res
    = (e.Res);
}

/**
    <OptTree-AutoMarkup-Contains (e.Elem) (e.Array)> ==
True|False
*/

$ENTRY OptTree-AutoMarkup-Contains {
  (e.Elem) (e.ArrayHead (e.Elem) e.ArrayTail)
    = True;

  (e.Elem) (e.Array)
    =  False;
}

/**
  <OptTree-AutoMarkup-Equal (e.First) (e.Second)> ==
True|False
*/
```

```
$ENTRY OptTree-AutoMarkup-Equal {
  (e.X) (e.X)
    = True;

  (e.X) (e.Y)
    = False;
}

/**
    <OptTree-AutoMarkup-GetElemByKey (e.Key) (e.Graph)>
    == e.GraphNode (e.Graph)

    e.GraphNode = (Func (e.Key) Children (e.Children))
*/

$ENTRY OptTree-AutoMarkup-GetElemByKey {
  (e.Key) (e.GraphHead (Func (e.Key) Children
(e.Children)) e.GraphTail)
    = Func (e.Key) Children (e.Children)
      (e.GraphHead (Func (e.Key) Children (e.Children))
e.GraphTail);

  (e.Key) (e.Graph)
    = /* пусто */ (e.Graph);
}

/**
    <OptTree-AutoMarkup-GetParents (e.ChildName)
(e.Graph)> == e.GraphNode*

    e.GraphNode = (Func (e.Key) Children (e.Children))

    e.Children = (e._ (e.ChildName) e._)
*/

$ENTRY OptTree-AutoMarkup-GetParents {
  (e.ChildName)
  (e.Graph-B (Func (e.FuncName) Children (e._
(e.ChildName) e._)) e.Graph-E)
    = (e.FuncName)
      <OptTree-AutoMarkup-GetParents (e.ChildName)
(e.Graph-E)>;

  (e.ChildName) (e._)
```

```
    = ;
}

$ENTRY OptTree-AutoMarkup-IsEmpty {
  ()
    = True;

  e.Other
    = False;
}
```

ПРИЛОЖЕНИЕ Д

Модуль OptTree-AutoMarkup

```
$INCLUDE "LibraryEx";

*$FROM Log
$EXTERN Log-PutLine, Log-AST;

*$FROM DisplayName
$EXTERN DisplayName;

*$FROM OptTree-AutoMarkup-GraphExtractor
$EXTERN OptTree-AutoMarkup-ExtractFunctionCallsGraph;

*$FROM OptTree-AutoMarkup-SpecializableExtractor
$EXTERN OptTree-AutoMarkup-GetSpecializableDict;

*$FROM OptTree-AutoMarkup-BasisVertexesExtractor
$EXTERN OptTree-AutoMarkup-ExtractBasisVertexes;

*$FROM OptTree-AutoMarkup-Common
$EXTERN OptTree-AutoMarkup-Contains,
        OptTree-AutoMarkup-GetSetDifference,
        OptTree-AutoMarkup-IsEmpty;

/**
  <OptTree-AutoMarkup s.OptAutoMarkup e.AST> == e.AST
*/

$ENTRY OptTree-AutoMarkup {
  NoOpt e.AST
    = <Log-PutLine 'AutoMarkup is disabled'> :
    = e.AST;

  OptAutoMarkup e.AST
    = <OptTree-AutoMarkup-GetSpecializableDict (e.AST)>
      : e.SpecializableDict
    = <OptTree-AutoMarkup-
GetForbiddenToSpecializeFunctionNames (e.AST)>
      : e.ForbiddenToSpecializeFunctionNames
    = <OptTree-AutoMarkup-GetSpecializePatterns
        (e.SpecializableDict)
        (e.ForbiddenToSpecializeFunctionNames)
      >
      : e.SpecPatterns
```

```
    = <OptTree-AutoMarkup-ExtractFunctionCallsGraph
e.AST>
      : e.ExtractedGraph
    = <OptTree-AutoMarkup-ExtractBasisVertexes
e.ExtractedGraph>
      : (e.BasisVertexes) (e.ExtractedGraph^)
    = <OptTree-AutoMarkup-GetDrivenFunctions
        (e.ExtractedGraph)
        (e.BasisVertexes)
      >
      : e.DrivenFunctions
    = <OptTree-AutoMarkup-GetForbiddenToDriveFunctions
e.AST>
      : e.ForbiddenToDriveFuncs
    = <OptTree-AutoMarkup-GetSetDifference
        (e.DrivenFunctions)
        (e.ForbiddenToDriveFuncs)
      >
      : (e.FunctionToDrive)
    = <OptTree-AutoMarkup-
GetUpdatedWithDrivenFunctionsAst
        (e.AST)
        (e.FunctionToDrive)
        (e.BasisVertexes)
      >
      : e.DrivedAst
    = e.DrivedAst e.SpecPatterns;
}

OptTree-AutoMarkup-GetForbiddenToSpecializeFunctionNames
{
  (e.AST)
    = <Map
        {
          (Spec (e.Name) e.SpecPattern)
            = (e.Name);

          (Function s.ScopeClass (e.FuncName) Sentences
e.Body)
            , (e.FuncName) : (e.FuncName '@'
e.FuncNameTail)
            = (e.FuncName);

          (Function s.ScopeClass (e.FuncName) Sentences
e.Body)
```

```
            , (e.FuncName) : (e.FuncNameMain SUF
e.FuncNameTail)
            = (e.FuncName);

          e.Other
            = ;
        }
        e.AST
      >
}

OptTree-AutoMarkup-GetSpecializePatterns {
  (e.SpecializableDict) (e.Filter)
    = <Map
        {
          (SPECIALIZABLE (e.FuncName) '->'
(e.SpecifiedPattern))
            , <OptTree-AutoMarkup-Contains
(e.FuncName)(e.Filter)> : False
            , <OptTree-AutoMarkup-IsEmpty
(e.SpecifiedPattern)> : False
            = <Map
                {
                  (e.Variable)
                    = (TkVariable e.Variable)
                }
                e.SpecifiedPattern
              >
              : e.MarkedSpecPattern
            = (Spec (e.FuncName) e.MarkedSpecPattern);

          e.Other
            = ;
        }
        e.SpecializableDict
      >
}

OptTree-AutoMarkup-GetForbiddenToDriveFunctions {
  e.AST
    = (INIT) (FINAL)
      <Map
        {
          (s.Label e.Name)
            , <OptTree-AutoMarkup-Contains
```

```
                (s.Label)
                ((Drive)(Inline)(Intrinsic))
              > : True
            = (e.Name);

          (Declaration GN-Entry e.FuncName)
            = (e.FuncName);

          (Function GN-Entry (e.FuncName) Sentences
e.Body)
            = (e.FuncName);

          e.Other
            = ;
        }
        e.AST
      >
}

OptTree-AutoMarkup-GetDrivenFunctions {
  (e.ExtractedGraph) (e.BasisVertexes)
    = <Map
        {
          (Func (Indirect function for implicit calls)
Children (e.Children))
            = ;

          (Func (e.FuncName) Children (e.Children))
          , <OptTree-AutoMarkup-Contains
(e.FuncName)(e.BasisVertexes)> : False
            = (e.FuncName);

          e.Other
            = ;
        }
        e.ExtractedGraph
      >
}

OptTree-AutoMarkup-GetUpdatedWithDrivenFunctionsAst {
  (e.AST) (e.FunctionsToDrive) (e.BasisVertexes)
    = <Map
        {
          (e.FuncName)
            = (Drive e.FuncName);
```

```
          }
          e.FunctionsToDrive
        >
        : e.DrivedFunctions
    = <Map
          {
            (Drive e.Name)
              , e.BasisVertexes : e._ (e.Name) e._
              = ;

            t.Other
              = t.Other;
          }
          e.AST
        >
        : e.ClearAST
    = e.ClearAST e.DrivedFunctions;
}
```