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:

Расширенный алгоритм обобщённого сопоставления в компиляторе Рефала-5λ

Студент ИУ9-82Б (Группа)	(Подпись, дата)	В. Е. Пичугин (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)

2021 г.

АННОТАЦИЯ

Темой данной работы является «Расширенный алгоритм обобщённого сопоставления в компиляторе Рефала-5λ». Объём дипломной работы составляет 56 страниц. Для её написания было использовано 16 источников. В работе содержится 11 листингов и 1 рисунок.

Объектом исследования настоящей ВКР являются алгоритмы обобщённого сопоставления с образцом и расширенной специализации функций в Рефале-5λ. Цель работы – реализовать их в компиляторе.

Дипломная работа состоит из четырёх глав. В первой главе даётся краткий обзор языка Рефал-5λ, архитектуры компилятора, оптимизации специализации, а также вводится основная терминология, связанная с алгоритмом обобщённого сопоставления. Во второй главе обсуждаются новые алгоритмы обобщённого сопоставления и специализации функций. В третьей главе излагаются детали и особенности реализации рассмотренных алгоритмов. В четвёртой главе рассказывается о способах тестирования реализованных алгоритмов.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Центральное место в языке программирования Рефал-5λ занимает механизм сопоставления с образцом (англ. pattern matching), использующийся для анализа структур данных. Сопоставление с образцом используется во время выполнения любой программы на Рефале-5λ, а также в некоторых оптимизациях, происходящих на этапе компиляции программы (например, в таких важных оптимизациях как прогонка и специализация функций) [1].

В общем случае алгоритм сопоставления имеет дело с уравнениями вида `E : P`, где `E` и `P` могут быть любыми образцовыми выражениями. Есть три граничных случая, когда решение уравнения можно записать в виде набора наборов сужений и присваиваний:

1. `E` – произвольное выражение, `P` – L-выражение;
2. `E` – объектное выражение, `P` – произвольный образец;
3. `E` – e-переменная `e.X`, `P` – произвольный образец.

Задача, рассмотренная в рамках дипломной работы, заключалась в реализации расширенного алгоритма обобщённого сопоставления, который позволял бы единообразно получать решение для этих граничных случаев, а также для различных «пограничных» ситуаций. Кроме того, алгоритм должен поддерживать механизм динамического обобщения, необходимый для реализации расширенной специализации функций.

Новая версия алгоритма обобщённого сопоставления необходима, потому что текущая реализация учитывает только первый граничный случай (когда `P` является L-выражением), а также не поддерживает динамическое обобщение.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Обзор языка Рефал-5λ

1.1.1. Введение в язык

РЕФАЛ – РЕкурсивный Функциональный АЛгоритмический язык [2], язык функционального программирования, ориентированный на символьные вычисления, обработку и преобразование текстов.

В отличие от большинства функциональных языков, где основной структурой данных является однонаправленный список, РЕФАЛ использует особую структуру данных – объектное выражение (двунаправленную последовательность). Другой важной особенностью РЕФАЛа является сопоставление с образцом (англ. pattern matching). РЕФАЛ был одним из первых языков программирования, использующим этот механизм для анализа структур данных.

Рассматриваемый в настоящей работе язык Рефал-5λ является одним из множества диалектов языка РЕФАЛ. Он разрабатывается на кафедре ИУ9 МГТУ им. Баумана с 2016 года. Его исходный код хранится в публичном репозитории сервиса GitHub. Кроме непосредственно кода компилятора в репозитории также можно найти большое количество автоматических тестов и обширную документацию по языку [3]. Стоит отметить, что Рефал-5λ является, во-первых, точным надмножеством диалекта РЕФАЛ-5, а это значит, что любая программа, которая работала на классической реализации, останется корректной и в Рефале-5λ. Во-вторых, он является расширением РЕФАЛа-5, которое включает в себя функции высшего порядка, в том числе и вложенные функции, а также различный «синтаксический сахар» (например, присваивания и блоки). В-третьих, актуальная реализация Рефала-5λ не замкнута, в отличие от многих других реализаций РЕФАЛа. Это означает, что программист не ограничен набором встроенных функций языка, т.к. компилятор имеет удобный встроенный интерфейс с языком C++. Актуальная реализация может

компилировать программы как в промежуточный интерпретируемый код, так и в код на C++ [1].

1.1.2. Основы программирования на Рефале-5λ

Любая программа на Рефале-5λ представляет собой набор *функций*. Точкой входа является функция `Go` (предваренная директивой `$ENTRY`) с пустым аргументом. Определение функции записывается как имя функции, за которым следует блок – тело функции, ограниченное фигурными скобками. Тело функции в общем случае состоит из нескольких *предложений*, разделенных точкой с запятой. Предложение – это правило, определяющее, как построить значение функции на некотором подмножестве её аргументов. Любое предложение состоит из двух частей – левой части, *образца*, описывающей подмножество значений аргумента функции, на котором это предложение применимо, и правой части, *результата*, описывающей значение функции на этом подмножестве. Левая и правая части разделяются знаком равенства (см. Листинг 1) [4].

Листинг 1. Общий вид функции.

```
ИмяФункции {
  образец1 = результат1;
  образец2 = результат2;
  …
  образецN = результатN;
}
```

В предложениях разрешается использовать *символы* – объекты, которые при сопоставлении невозможно разложить на более мелкие фрагменты. В Рефале имеются символы-литеры, символы-числа и символы-слова. Помимо этого разрешается использовать *переменные* – фрагменты выражений, которые могут заменяться на произвольные значения в соответствии с их типом (см. пример на Листинге 2). В Рефале есть три типа переменных: s-, t- и e-переменные. Значением s-переменной или переменной символа может быть любой одиночный символ. Значением e-переменной или переменной

выражения может быть любой фрагмент аргумента функции, в том числе пустой. О t-переменных будет сказано чуть позже. Переменная записывается как признак типа (`s`, `t`, `e`), за которым следует знак «.» («точка») и имя переменной – некоторая последовательность букв и цифр (так называемый, *индекс* переменной). Переменные могут встречаться как в левой части предложения, так и в правой. При этом в правой части предложения могут использоваться только те переменные, которые есть в левой. Если в выражении переменная встречается несколько раз, то она называется *повторной*, все её вхождения должны иметь одинаковое значение [4].

Листинг 2. Пример использования переменных.

```
/* проверяет два символа на равенство */
EqSymb {
  s.X s.X = True;
  e.Other = False;
}
```

Для того, чтобы в Рефале работать с выражением как с единым объектом, его заключают в круглые скобки, которые называют *структурными* скобками. Такой объект (его называют скобочный терм) сам может быть частью другого выражения, которое, в свою очередь, тоже может быть заключено в круглые скобки. Так в Рефале строятся иерархические вложенные данные. Символы, которые были рассмотрены до этого, тоже являются термами. Таким образом, выражение на Рефале состоит из *термов*, каждый из которых может быть либо символом, либо скобочным термом, который внутри содержит другое выражение на Рефале. Также существуют и именованные структурные скобки, использующиеся для построения абстрактных типов данных (отсюда и их второе название – *АТД-скобки*). Они представляют из себя выражение в квадратных скобках, первым символом которых является идентификатор, именующий абстрактный тип данных, представляющий из себя имя функции в текущей области видимости [4]. Примеры скобочных выражений:

```
(A (B (C D)) E)
```

```
[Person 'Name' 'Age']
```

Таким образом, можно уточнить смысл переменных:

- e-переменные могут принимать произвольную последовательность термов, т.е. значением e-переменной может быть только выражение с правильной скобочной структурой;
- значением t-переменных может быть любой одиночный терм – как символ, так и выражение в скобках.

Также стоит отметить, что при сопоставлении с образцом может возникнуть ситуация, когда для e-переменной будет существовать несколько решений. Такие e-переменные называются *открытыми*.

Вызовы функций на Рефале, в отличие от математической нотации, оформляются при помощи *угловых* скобок «<» и «>» (знаков «меньше» и «больше»), при этом имя функции пишется не перед открывающей скобкой, а после неё.

Для удобства дальнейшего изложения стоит привести классификацию выражений в Рефале. *Объектным выражением* называется выражение языка Рефал, которое может содержать только символы и круглые скобки. Соответственно, термы, из которых составлено объектное выражение, называются объектными термами. Аргументом функции может быть только объектное выражение. *Активным выражением* называется выражение, которое содержит символы, круглые и угловые скобки. *Образцовым выражением* или *образцом* называется выражение, составленное из символов, структурных скобок и переменных. Левая часть предложения является образцовым выражением. *Результатным выражением* или *результатом* называется выражение, содержащее символы, круглые и угловые скобки и переменные. Правые части предложений являются результатными выражениями [4].

Ранее описанная структура предложения является неполной. В общем случае левая часть состоит из образца, за которым может следовать ноль или более *условий*. Образец описывает шаблон – выражение с переменными. Сопоставление некоторого выражения с образцом является успешным, если

удалось найти подстановку переменных образца, переводящую его в аргумент в данное выражение. Образец левой части сопоставляется с аргументом функции. Условие описывает дополнительные ограничения. Оно начинается на знак «,» и состоит из пары – расширенного результатного выражения (левой части условия) и образца (правой части условия), разделенных знаком «:». Во время проверки условия вычисляется значение его левой части и затем оно сопоставляется с образцом. Условие выполняется, если удалось сопоставить значение левой части с образцом. Выражение успешно сопоставилось с левой частью, если оно сопоставилось с образцом и выполнились все условия [5].

Правая часть состоит из нуля или более *присваиваний*, за которыми следует либо знак «=» и расширенное результатное выражение, или знак «,», за которым следует *классический блок*. При выполнении правой части сначала последовательно выполняются присваивания, после чего вычисляется значение расширенного результатного выражения или классического блока. Присваивание начинается на знак «=», за которым следуют расширенное результатное выражение и левая часть, разделённые знаком «:». Расширенное результатное выражение вычисляется и его значение сопоставляется с левой частью. Сопоставление должно быть успешным. При неуспешном сопоставлении с левой частью программа аварийно завершается. Присваивания используются для того, чтобы вычислить некоторое значение и связать его с переменными, также иногда используются ради побочного эффекта. *Расширенное результатное выражение* состоит из результатного выражения, за которым следует ноль или более тел функций, предваренных знаками «:». Значение результатного выражения последовательно преобразуется путём вызова вложенных функций, описанных их телами. Классический блок состоит из результатного выражения и тела функции. Его семантика эквивалентна семантике расширенного результатного выражения с одним телом функции [5]. На Листинге 3 приведен пример функции, использующей множество синтаксических возможностей Рефала-5λ.

Листинг 3. Пример сложной функции.

```
Func {
  (s.A t.B) e.Rest
    , <Cond1 s.A> : s.C
    , <Cond2 t.B> : True
    = <Foo s.C> : e.D
    = <Bar (e.D) e.Rest>
    : {
        Success e.Solutions = e.Solutions;
        Failure = Failure;
      };

  e.Other = <Baz e.Other>;
}
```

Как упоминалось ранее, ключевыми особенностями диалекта Рефал-5λ являются функции высшего порядка, в том числе и *вложенные функции*, записываемые как тело функции в фигурных скобках. На Листинге 4 в качестве примера приведена функция `Map`, принимающая в качестве аргументов замыкание и список термов, и применяющая замыкание к термам слева-направо, формируя тем самым результирующий список.

Листинг 4. Пример использования вложенной функции.

```
<Map
  {
    (s.Label e.Name)
      = <BaseName e.Name> : e.BaseName s.Num
      = (s.Label e.BaseName);
  }
  e.OptNames
>
```

1.1.3. Архитектура компилятора

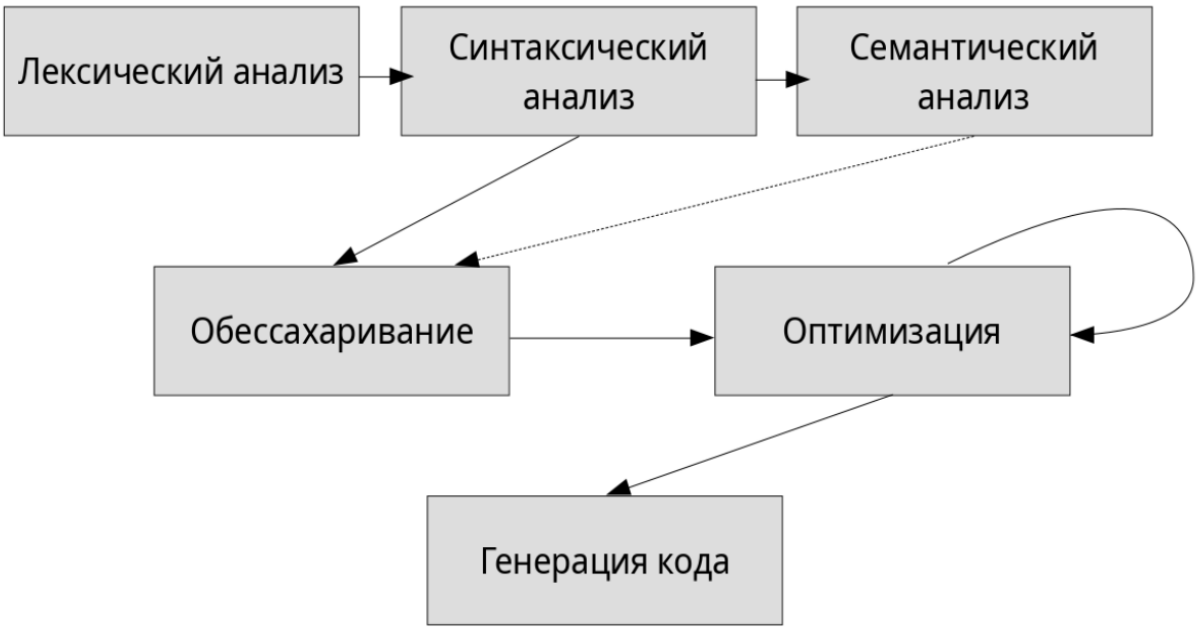

Рисунок 1. Архитектура компилятора Рефала-5λ.

Компилятор языка Рефал-5λ является самоприменимым. Процесс компиляции программы состоит из нескольких этапов (см. Рисунок 1).

1. *Лексический анализ*. Из входного потока символов формируется выходной поток лексем языка.
2. *Синтаксический анализ*. На основе грамматики языка из потока лексем строится абстрактное синтаксическое дерево [6].
3. *Семантический анализ*. Происходит проверка семантики построенного синтаксического дерева. Пример: поиск объявлений функций без их определения и т.п.
4. *Обессахаривание*. Некоторые синтаксические конструкции, являющиеся «синтаксическим сахаром», преобразуются, и формируется упрощенное синтаксическое дерево.
5. *Оптимизация.* Синтаксическое дерево обходится несколько раз, с применением различных алгоритмов оптимизации.
6. *Генерация кода* состоит из более мелких этапов. На первом этапе на основе синтаксического дерева порождается символический ассемблер.

На втором этапе из промежуточного кода порождается двоичный код или код на языке C++.

1.1.4. Промежуточный язык

Как было сказано выше, на этапе обессахаривания устраняется «синтаксический сахар». На входе этого прохода имеется синтаксическое дерево, полученное от парсера, на выходе – программа на промежуточном языке Высокоуровневые оптимизации выполняют эквивалентные преобразования также на промежуточном языке.

Программа в промежуточном языке представляет собой последовательность глобальных функций. Предложения в функциях состоят из образца, результата и нуля или нескольких условий между ними. В образцовых выражениях допустимы привычные термы Рефала: символы, переменные, круглые и квадратные скобки. В результатных выражениях допустимы те же термы, что и в образцовых, плюс скобки активации <...> и, так называемые, *конструкторы замыканий* `{{ ... }}`.

Для каждой вложенной функции создаётся глобальная функция с почти тем же телом, что и исходная. А на место вложенной функции помещается конструкция построения замыкания. Вложенные функции могут содержать переменные из окружающей области видимости. А значит, для вычисления функции эти переменные где-то должны храниться. Хранятся они в объекте замыкания, который в образце может быть сопоставлен только с s-переменной. Вообще, объект замыкания хранит две вещи: указатель на функцию, реализующую логику замыкания и связанные переменные контекста. Обессахариватель просто берёт тело вложенной функции и в начало образца каждого предложения добавляет имена захваченных переменных, причём e-переменные заключаются в скобки. На место самой вложенной функции помещается *конструктор замыкания*, содержащий указатель на новую функцию и значения тех же переменных [5]. Блоки и присваивания сначала

преобразуются в вызовы вложенных функций, а затем те – во вспомогательные функции. Например, следующая программа

```
CartProd {
  (e.X) (e.Y)
    = <Map
        {
          t.X
            = <Map
                { t.Y = (t.X t.Y) } e.Y
              >
        }
        e.X
      >
}
```

будет транслирована в такой промежуточный код

```
CartProd {
  (e.X) (e.Y)
    = <Map {{ &CartProd\1 (e.Y) }} e.X>;
}
CartProd\1 {
  (e.Y) t.X
    = <Map {{ &CartProd\1\1 t.X }} e.Y>;
}
CartProd\1\1 {
  t.X t.Y = (t.X t.Y);
}
```

1.1.5. Оптимизация специализации

Одной из основных высокоуровневых оптимизаций в компиляторе является оптимизация специализации функций [7][8].

Будем говорить, что данные известны *статически*, если они известны на стадии компиляции. Если данные известны только во время выполнения, будем говорить, что они известны *динамически*.

Пусть имеется вызов `<F ARG>`. Преобразованием специализации назовём замену этого вызова на вызов `<F' ARG'>` и построение новой функции `F'` на основе `F` такое, что тело функции `F'` учитывает статически известную информацию из исходного аргумента `ARG`, а новый аргумент `ARG'` эту статически известную информацию не содержит. Функцию `F'` будем называть *экземпляром* функции `F`. *Сигнатурой* экземпляра будем называть информацию из `ARG`, учтённую при построении экземпляра. Разные вызовы с одной сигнатурой будут вызывать один и тот же экземпляр.

Простейший случай специализации – вызываемая функция имеет несколько аргументов, один из аргументов в вызове является статическим. В теле экземпляра все вхождения этого параметра заменены на соответствующие константы.

Особенности текущей реализации:

- Для специализируемых функций необходимо явно задавать входной формат и обозначать в нём статические и динамические параметры. Индексы статических параметров именуются заглавными буквами, динамических – строчными. Например, `$SPEC Map s.FUNC e.arg.` Образцы предложений функции должны соответствовать формату. При этом статические параметры могут отображаться в образцах только на переменные того же типа.
- Специализация ведётся только по статическим параметрам.
- Экземпляры специализированных функций получают суффиксы `@n`: `Map@1, Map@2` и т.д.
- Безымянные функции неявно специализируются по контексту.

1.2. Задача расширенного алгоритма обобщённого сопоставления

1.2.1. Терминология

Требуется решить уравнение `E : P`, где `E` – *параметризованное* пассивное выражение, `P` – образец. Выражение `E` построено из символов, скобок (круглых и абстрактных), переменных и конструктора замыкания. Конструктор замыкания и отличает его от образца (в образце он запрещён).

Будем обозначать `T` и `Pt`, соответственно, параметризованный терм и образец терма, `Sym` и `Psym` – параметризованный символ и образец символа.

Также будем использовать следующую терминологию, пришедшую из суперкомпиляции [9]: переменные в `E` будем называть *параметрами*, переменные в `P` – *переменными*.

Подстановка – это отображение некоторого множества переменных или параметров в некоторые выражения. Подстановку, отображающую параметры из `E` в некоторые образцы, будем называть *сужениями*, обозначать `Ct` и записывать как `par → P'`. Подстановку, отображающую переменные из `P` в некоторые параметризованные выражения, будем называть *присваиваниями*, обозначать `As` и записывать как `E' ← var`. Применение подстановки к выражению будем обозначать косой чертой: `E / Ct`, `P / As`. Композицию подстановок будем обозначать как `S1 · S2`:

`E / (S1 · S2) = E / S1 / S2`

Частным решением уравнения `E : P` будем называть пару `(Ct, As)`, такую что `E / Ct = P / As`. *Полным решением* уравнения `E : P` будем называть конечный набор непересекающихся частных решений (Ct_1, As_1), …, (Ct_N, As_N), такой, что для любого частного решения уравнения `(Ct', As')` найдётся такое `j` и такая подстановка `S`, что

$E / Ct_j / S = E / Ct' = P / As' = P / As_j / S$

Иначе говоря:

$Ct' = Ct_j \cdot S,\ As' = As_j \cdot S$

т.е. любое частное решение уравнения будет частным случаем одного из частных решений полного решения уравнения. Частные решения полного решения не должны пересекаться, т.е. для любых i, j, $i \neq j$, верно, что

$E / Ct_i \cap E / Ct_j = \varnothing$

или, что тоже самое

$P / As_i \cap P / As_j = \varnothing$

Полное решение будем обозначать как Sol. Заметим, что не для всех уравнений оно существует.

Обобщением выражения (параметризованного выражения или образца) E назовём такое выражение E', что существует подстановка S, такая что E' / S = E. *Уточнением* выражения E назовём выражение E', такое что E – обобщение E'.

Если для подстановки S существует обратная подстановка S*, такая что для любого E: E / S / S* = E, то S – подстановка-переименование. Подстановка-переименование согласовано переименовывает переменные в E.

Если обобщение E для E' переводится в E' не подстановкой-переименованием, то такое обобщение будем называть *нетривиальным*. Любое нетривиальное обобщение приводит к потере информации: множество объектных выражений в результате расширяется.

Для уравнения E : P *динамическим обобщением* будем называть пару из параметризованного выражения E' и подстановки S такую, что E = E' / S и уравнение E' : P имеет полное решение. Если уравнение E : P имеет полное решение, то достаточно пустой подстановки S. Интерес возникает, если исходное уравнение не имело полного решения. Динамическое обобщение существует для любого уравнения. Доказательство: E' = e.X и S = { E ← e.X } является динамическим обобщением, т.к. уравнение e.X : P имеет решение As = ∅, Ct = { e.X → P }. Такое динамическое обобщение будем называть *тривиальным*.

Если уравнение `E : P` не имеет решения, то в динамическом обобщении `E'`, `S` обобщение `E'` будет нетривиальным. Т.к. обобщение есть потеря информации, то имеет смысл искать такое обобщение `E'`, которое стирает минимум информации. Например, если есть два динамических обобщения `E1`, `S1` и `E2`, `S2` и `E2` есть обобщение `E1`, то, очевидно, `E1` теряет меньше информации и более предпочтительно. Однако задача определения объёма информации в произвольном выражении весьма нетривиальна.

1.2.2. Постановка задачи

Требуется разработать и реализовать в компиляторе алгоритм, который для данного уравнения `E : P` или находит полное решение, или находит приемлемое динамическое обобщение и строит полное решение для него.

При этом, алгоритм должен давать решение, пригодное для выполнения расширенной специализации, т.е. не нарушать семантику программы. Из этого следует, что полное решение должно представлять собой упорядоченный набор частных решений.

2. РАЗРАБОТКА

2.1. Расширенный алгоритм обобщённого сопоставления

2.1.1. Состояние решателя и обзор алгоритма

Перед выполнением основной части алгоритма необходимо несколько модифицировать исходное уравнение, а именно добавить в его левую часть метки координат в начало, конец и между токенами. Например,

$_1$(`s.A` $_3$ `e.B` $_4$) $_5$ ($_6$`e.B` $_7$ `s.A` $_8$)$_9$ `:` `(e.X) (e.X)`

Эта процедура необходима для осуществления динамического обобщения левой части. Смысл меток координат в том, что пара меток ограничивает некоторый участок, который при необходимости можно обобщить. В исходном уравнении метки и токены чередуются. Однако, в процессе решения к параметризованному выражению будут применяться подстановки и чередование будет нарушаться. Например, при подстановке `e.X` → `e.Y t.Z` появится пара смежных переменных без метки посередине, а стирающая подстановка `e.X` → ε может привести к тому, что две метки окажутся по соседству. Буквой `E` (иногда с номерами) будут обозначаться выражения, которые помимо термов могут содержать координаты. Буквой `E*` (иногда с номерами: `E1*, E2*`…) будут обозначаться выражения, на верхнем уровне которых (вне скобок) нет координатных меток.

В процессе решения будут рассматриваться два вида уравнений. *Клэшем* (или *асимметричным клэшем*) будем называть уравнение вида

$_m$ `E` $_n$ `:` `P`

В процессе решения будут возникать новые клэши. Их набор следует хранить упорядоченным по возрастанию левой координаты – порядок будет важен при анализе открытых переменных. *Симметричным клэшем* будем называть уравнение вида

$_k$ `E1` $_l$ `=` $_m$ `E2` $_n$

Для их решения будут применяться основы теории уравнений в словах [10].

Основная часть алгоритма выполняется в два этапа:

1. Сопоставление выражения с образцом без учёта повторных переменных.
2. Разрешение повторных переменных.

На первом этапе разрешаются асимметричные клэши, на втором – формируются и решаются симметричные клэши.

Состояние алгоритма на первом этапе содержит следующие значения:

- текущий набор сужений `Ct`
- систему клэшей
- текущий набор присваиваний `As`, при этом присваивания являются мультисловарём – одному имени переменной может соответствовать несколько значений

Присваивания содержат выражения с координатами.

Состояние алгоритма на втором этапе содержит:

- текущий набор сужений `Ct`
- систему симметричных клэшей
- текущий набор присваиваний `As`, который уже является обычным словарём

В процессе решения алгоритм ветвится – строится упорядоченный набор ветвей. Их порядок важен для правильного анализа сопоставлений с открытыми переменными. Каждая ветвь может завершиться одной из трёх ситуаций:

- успешное решение – даёт пару `(Ct, As)` полного решения
- отсутствие решения – данная ветвь решений не имеет
- запрос на обобщение – указывает координаты обобщаемого участка параметризованного выражения `E`

Если хотя бы одна из ветвей решения вернула запрос на обобщение – `E` обобщается. Если разные ветви предлагают обобщить разные участки аргумента, выбирается один из вариантов, он применяется и делается новая попытка решения. Ветки с отсутствием решений усекаются в процессе решения. Может так оказаться, что все ветки оказались усечены. Это значит, что решений нет.

2.1.2. Преобразования системы клэшей

При преобразовании системы асимметричных клэшей могут формироваться новые сужения и присваивания. Если формируется новое сужение, то оно применяется ко всему состоянию решателя: и к набору сужений, и к клэшам (левым частям), и к левым частям присваиваний. Если формируется новое присваивание, то оно просто добавляется к набору `As`. Применение сужения к набору сужений сводится к двум операциям: применение сужения к правым частям `Ct` и добавление сужения к набору.

При преобразовании симметричных клэшей могут формироваться только сужения. Они применяются к `Ct`, симметричным клэшам (обеим частям) и присваиваниям.

При генерации сужений часто требуются новые переменные. Эти переменные в дальнейшем будут обозначаться при помощи индекса `.NEW`. Если требуется несколько новых переменных, будут использоваться индексы `.NEW1`, `.NEW2` и т.д.

Также в ходе преобразований клэшей (как симметричных, так и асимметричных), могут возникать избыточные метки координат, не несущие смысловой нагрузки. Например, при подстановке сужений некоторые параметры могут редуцироваться до ε, из-за чего возникает скопление координат. Такие координаты нужно упрощать по следующим правилам:

```
k m E n        ↦   m E n
    k E m n    ↦   k E m
m n            ↦   ε
E^1k m nE^2    ↦   E^1k nE^2
```

Здесь `E^1, E^2` – части записи `E`, которые, могут не быть правильными выражениями Рефала. В частности, в них могут быть незакрытые и неоткрытые скобки. Третье правило говорит о том, что у пустого выражения нет координат. Смысл координат – указание точек для динамического обобщения, а пустоту нигде обобщать не требуется.

Упрощение координат выполняется в клэшах (любых) и присваиваниях после каждой подстановки сужения, а также после некоторых операций преобразования уравнений.

2.1.3. Сопоставления с L-образцами

Существует подмножество образцов – так называемые L-образцы, описанные В.Ф. Турчиным [11], для которых уравнение `E : P` всегда разрешимо (при этом в `E` должны отсутствовать конструкторы замыканий).

L-образцы по Турчину запрещают любые t-переменные, а также открытые и повторные e-переменные. Однако, его подход несложно расширить и на неповторные t-переменные. Таким образом, далее L-образцом будем называть образцовое выражение следующего вида:

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

Первый этап алгоритма начинается с применения нижеописанных операций преобразования системы асимметричных клэшей, которые будем называть L-операциями. Они выполняются в указанном ниже порядке до тех пор, пока можно применить хотя бы одну из них.

Сначала происходит попытка применить операции для клэшей типа «терм : терм»:

```
 m T n  : t.X   ↦  m T n ← t.X
m Sym n : s.X   ↦  m Sym n ← s.X
m (E) n : (P)   ↦  m E n : P
m [X E] n : [X P]   ↦  m E n : P
m [X E] n : [Y P]   ↦  нет решений
m [E] n : (P)   ↦  нет решений
m (E) n : [P]   ↦  нет решений
m (E) n : Psym  ↦  нет решений
```

```
m Sym n : (P)    ↦  нет решений
m Sym n : [P]    ↦  нет решений
m t.X n : (P)    ↦  t.X → (e.NEW)
m t.X n : [P]    ↦  t.X → [e.NEW]
m t.X n : Psym   ↦  t.X → s.NEW
m s.X n :  X     ↦  s.X → X
 m X n  :  X     ↦  стираем
 m X n  :  Y     ↦  нет решений
```

Стоит пояснить введенные обозначения, которые будут использоваться и далее:

- Если справа от ↦ находится присваивание, значит, клэш стирается, а присваивание добавляется к набору присваиваний.
- Если справа от ↦ находится новый клэш, то он подменяет собой исходный. Если несколько новых клэшей, разделённых знаком `&&` (дальше будет), то они подменяют собой исходный в указанном порядке, т.к. порядок клэшей важен.
- Если справа от ↦ находится сужение, то оно применяется к набору сужений, клэшей и присваиваний, включая текущий клэш. В результате применения сужения клэш может быть разрешён на следующем шаге. Несколько сужений могут быть разделены знаком `||` (дальше будет) – это означает порождение новых веток решений в указанном порядке (порядок важен).
- Если справа находится фраза `стираем`, клэш удаляется как разрешённый.
- Фраза `нет решений` говорит об усечении текущей ветки решений.
- Знаки `X` и `Y` обозначают произвольные символы, при этом `X` ≠ `Y`. Предпоследнее правило говорит о том, что слева и справа находится одинаковый символ, такой клэш тождественно истинен и поэтому стирается. Последнее – что клэш противоречив.

Далее происходит попытка выполнить операции отделения термов, для удобства определения которых были введены две вспомогательные функции. Функция TERM_LEFT(E) = T, E' отделяет терм слева от выражения.

TERM_LEFT($_a$ T $_b$ E) = $_a$ T $_b$, $_b$ E

TERM_LEFT($_a$ T E1* $_b$ E2) = $_a$ T $_b$, $_a$ E1* $_b$ E2

Функция TERM_RIGHT(E) = E', T определяется аналогично.

TERM_RIGHT(E $_a$ T $_b$) = E $_a$, $_a$ T $_b$

TERM_RIGHT(E1 $_a$ E2* T $_b$) = E1 $_a$ E2* $_b$, $_a$ T $_b$

Сами правила отделения термов принимают следующий вид:

$_m$ T E : Pt P ↦ $_a$ T $_b$: Pt && $_c$ E : P

где $_a$ T $_b$, $_c$ E := TERM_LEFT($_m$ T E)

E T $_m$: P Pt ↦ E $_c$: P && $_a$ T $_b$: Pt

где E $_c$, $_a$ T $_b$:= TERM_RIGHT(E T $_m$)

Затем наступает анализ e-параметров, где впервые происходит ветвление:

$_m$ e.X E $_n$: Pt P ↦ e.X → t.NEW1 e.NEW2 || e.X → ε

$_m$ E e.X $_n$: P Pt ↦ e.X → e.NEW1 t.NEW2 || e.X → ε

Порядок ветвей здесь важен, иначе сопоставления с открытыми переменными отработают неправильно. Также имеются следующие L-операции, которые пытаются примениться в случае, если другие операции применить не получается:

E : e.X ↦ E ← e.X

$_m$ e.X E $_n$: ε ↦ e.X → ε

$_m$ T E $_n$: ε ↦ решений нет

ε : E1 Pt E2 ↦ решений нет

ε : ε ↦ стираем

2.1.4. Сопоставления с открытыми переменными

Если к системе клэшей нельзя применить ни одну из вышеописанных операций, значит возможны два варианта. Либо клэши закончились и нужно переходить ко второму этапу алгоритма, либо все клэши имеют вид

E : e.X P e.Y

Таким образом, необходимо решать задачу сопоставления с открытыми переменными, общий принцип работы с которыми описан в препринте С.А. Романенко [12]. Стоит отметить, что сопоставления с открытыми переменными, в отличие от предыдущих операций, могут привести к динамическому обобщению.

Итак, если имеется клэш вида

```
E : e.X P e.Y
```

то нужно рассмотреть различные разбиения `E` на две части. Левая часть будет присвоена переменной `e.X`, правая – сопоставлена с `P e.Y`. Точки разбиения выбираются по следующему принципу:

- Если `E` начинается на терм, то точка разбиения располагается перед первым термом.
- Точки разбиения добавляются между двумя смежными термами.
- Точки разбиения находятся «внутри» e-параметров.
- Если `E` заканчивается на терм, то точка разбиения добавляется в конец.

Если точка разбиения располагается между термами, в начале или в конце, применяются следующие правила:

```
m E n : e.X P e.Y  ↦  ε ← e.X,  m E n : P e.Y
↑

k E1 T2 m T3 E4 n : e.X P e.Y  ↦  k E1 T2 m ← e.X,
        ↑                          m T3 E4 n : P e.Y

E1 m E2* T3.T4 E5* n E6 : e.X P e.Y ↦ E1 m E2* T3 n ← e.X,
          ↑                           m T4 E5* n E6 : P e.Y

m E n : e.X P e.Y  ↦  m E n ← e.X,  ε : P e.Y
  ↑

ε : e.X P e.Y  ↦  ε ← e.X, ε : P e.Y
↑
```

Стрелками показаны точки разбиения. Если в точке разбиения находится координата, то стрелка показывает на неё. В случае, когда между термами координаты нет, точка разбиения обозначена точкой.

Если точка разбиения находится внутри e-параметра, то для параметра строится сужение с открытой переменной. При этом, если в левой части находится несколько смежных e-параметров, то последний из них подвергается сужению `e.X → e.1 e.2`, а все предшествующие – `e.X → e.1 t.2 e.3`. Формально:

```
E1 e.X m e.Y E2 : e.L P e.R  ↦  e.X → e.NEW1! t.NEW2 e.NEW3,
    ↑                             E1 e.NEW1! m ← e.L,
                                 n t.NEW2 e.NEW3 m e.Y E2 : P e.R
                где E1 = E3 n E4*
E1 e.X E2 : e.L P e.R  ↦  e.X → e.NEW1! e.NEW2,
    ↑                       E1 e.NEW1! m ← e.L,
                            n e.NEW2 E2 : P e.R
                где E1 = E3 n E4*
                    E2 = E5* m E6
```

Каждая точка разбиения по сути формирует новый клэш, который подлежит решению. Также имеется нюанс работы с открытыми переменными. Если в системе есть несколько клэшей на открытые переменные, то первый клэш может иметь слева произвольное выражение, а остальные должны иметь тривиальную левую часть, т.е. являться клэшами вида

```
m e.X n : e.L P e.R
ε : e.L P e.R
```

Клэши, не удовлетворяющие этому условию приводят к запросу на динамическое обобщение соответствующего участка.

2.1.5. Решение симметричных клэшей

У симметричных клэшей есть два отличия от обычных клэшей:

- их решениями являются только сужения,
- их решение в общем случае может зацикливаться.

Решение асимметричных клэшей зациклиться не может, т.к. длина правой части на каждом шаге либо уменьшается (отщепляется один элемент), либо подготавливается к отщеплению. Для уравнений в словах (частных случаев симметричных клэшей) это неверно: при применении сужений длина уравнения может и сохраняться, и расти. В общем случае, решение уравнений в словах требует построения графа суперкомпиляции с распознаванием зацикливаний, решение будет иметь вид не конечного набора сужений, а набора функций, проверяющих некоторое условие. Исследованию этого вопроса, например, посвящён модельный суперкомпилятор MSCP-A [13].

Однако задачей настоящей работы является решение симметричных клэшей только для частных случаев, которые можно выразить в виде конечного набора сужений. Либо можно отказаться решать симметричный клэш, обобщив соответствующие участки исходного выражения до новых e-переменных. Поэтому правила решения симметричных клэшей должны быть сформулированы таким образом, чтобы решать настолько, насколько возможно, но при этом не зацикливаться.

Симметричные клэши возникают в результате кратных вхождений переменных в правую часть исходного уравнения. Пусть имеется переменная `v.X` (далее литерал `v` будет использоваться для обозначения переменной произвольного типа) и в процессе решения было получено несколько присваиваний:

$E_1 \leftarrow$ `v.X`, …, $E_k \leftarrow$ `v.X`, …, $E_m \leftarrow$ `v.X`

Из этих присваиваний в качестве результата нужно оставить одно (не важно какое, для определённости пусть будет первое, $E_1 \leftarrow$ `v.X`), а для остальных необходимо построить уравнения $E_i = E_j$ (каждый с каждым). Для двух присваиваний – одно уравнение ($E_1=E_2$), для трёх – 3 ($E_1=E_2$, $E_1=E_3$, $E_2=E_3$), для четырёх – 6 и т.д. Для N, соответственно, $N \times (N - 1) / 2$ уравнений.

В процессе решения в системе симметричных клэшей могут возникать уравнения, у которых левая и правая части будут одинаковы с точностью до

координат. Такие клэши будем называть *тавтологиями.* Для них введём следующее правило:

```
E1 = E2  ↦  стираем, если CLEAR(E1) ≡ CLEAR(E2)
```

где `CLEAR(E)`– вспомогательная функция, удаляющая все координаты из выражения. Например, если сужение изменило клэш, то его нужно проверить на тавтологию и стереть при необходимости.

Как и на первом этапе алгоритма, на втором этапе к системе симметричных клэшей применяются различные операции. Сначала происходит попытка применить операции для клэшей типа «переменная = переменная»:

```
a e.X b = c e.Y d  ↦  e.X → e.NEW, e.Y → e.NEW
a e.X b = c t.Y d  ↦  e.X → t.NEW, t.Y → t.NEW
a e.X b = c s.Y d  ↦  e.X → s.NEW, s.Y → s.NEW
a t.X b = c t.Y d  ↦  t.X → t.NEW, t.Y → t.NEW
a t.X b = c s.Y d  ↦  t.X → s.NEW, s.Y → s.NEW
a s.X b = c s.Y d  ↦  s.X → s.NEW, s.Y → s.NEW
```

Т.е. если слева и справа находится переменная, то выбирается наименьший вид переменной, генерируется новая переменная этого типа и порождаются два сужения. Клэш стирается, потому что он станет тавтологией. Также стоит отметить, что здесь и далее уравнения описаны с точностью до симметрии. Т.е. случаи, когда левая и правая части поменяны местами, не написаны, но подразумеваются. Одинаковая переменная с обеих сторон здесь появиться не может, т.к. это уравнение будет тавтологией и сотрётся раньше. Далее происходит попытка применить правила сопоставления с пустотой:

```
ε = m e.X E n  ↦  e.X → ε && ε = m E n
ε = m T E n  ↦  решений нет
```

Следом происходят сопоставления типа «терм = терм»:

```
a t.X b = c {{ &F e.X }} d  ↦  обобщаем {c-d}
a s.X b = c {{ &F e.X }} d  ↦  обобщаем {c-d}
a  X  b = c {{ &F e.X }} d  ↦  решений нет
a (E) b = c {{ &F e.X }} d  ↦  решений нет
```

```
a [E] b = c {{ &F e.X }} d  ↦  решений нет
a [E1] b = c (E2) d  ↦  решений нет
a t.X b  = c X d  ↦  t.X → X
a s.X b  = c X d  ↦  s.X → X
a X b    = c Y d  ↦  решений нет
a (E1) b = c (E2) d  ↦  a E1 b = c E2 d
a [X E1] b = c [X E2] d  ↦  a E1 b = c E2 d
a [X E1] b = c [Y E2] d  ↦  решений нет
a (b E c) d = e t.X f  ↦  t.X → (e.NEW)
a (E) d = e t.X f  ↦  обобщаем {a-d}, {e-f}
a [b E c] d = e t.X f  ↦  t.X → [e.NEW]
a [E] d = e t.X f  ↦  обобщаем {a-d}, {e-f}
a (E) b = c Sym d  ↦  решений нет
a [E] b = c Sym d  ↦  решений нет
```

Затем выполняются отделения термов:

```
a T1 E1 = b T2 E2  ↦  c T1 d = e T2 f && g E1′ = h E2′
  где c T1 d, g E1′ := TERM_LEFT(a T1 E1)
      e T2 f, h E2′ := TERM_LEFT(b T2 E2)

E1 T1 a = E2 T2 b  ↦  E1′ c = E2′ d && e T1 f = g T2 h
  где E1′ c, e T1 f := TERM_RIGHT(E1 T1 a)
      E2′ d, g T2 h := TERM_RIGHT(E2 T2 b)
```

Наконец, анализируются e-параметры:

```
a T b E1 = c e.X E2 ↦ e.X → t.NEW1 e.NEW2 || e.X → ε
E1 a T b = E2 e.X c ↦ e.X → e.NEW1 t.NEW2 || e.X → ε
```

Здесь, как и в случае скобок, требуется, чтобы терм был окружён координатами. Иначе применение данного правила приведёт к зацикливанию. Если остались какие-либо симметричные клэши, к которым не применимо ни одно из вышеперечисленных правил, то берётся произвольный из них:

```
a E1 b = c E2 d
```

и объявляется запрос на обобщение для диапазонов `{a-b}` и `{c-d}`.

В конечном итоге для каждой ветки либо должна опустеть система симметричных клэшей, либо ветка должна усечься. В первом случае возвращается сформированный набор сужений и присваиваний `(Ct, As)` как часть полного решения уравнения (возможно, для этого придется сделать несколько динамических обобщений). Во втором случае решений у ветки нет.

Также стоит отметить, что уравнения в словах – очень широкая тема и для их решения можно придумать ещё много вспомогательных правил. Например, такие:

$$_a\ \texttt{e.X E1}\ _b = {}_c\ \texttt{e.X E2}\ _d \mapsto {}_a\ \texttt{E1}\ _b = {}_c\ \texttt{E2}\ _d$$

$$_a\ \texttt{E1 e.X}\ _b = {}_c\ \texttt{E2 e.X}\ _d \mapsto {}_a\ \texttt{E1}\ _b = {}_c\ \texttt{E2}\ _d$$

Т.е. если обе части уравнения начинаются или заканчиваются на одну и ту же e-переменную, то её можно стереть.

2.2. Алгоритм расширенной специализации

В главе «Обзор предметной области» было приведено краткое описание оптимизации специализации функций. Исходя из него, можно сразу выделить следующие ограничения текущей реализации специализации:

- Необходимо задавать шаблон функции.
- Статические переменные должны отображаться на переменные в предложениях.

Отсюда возникает очевидное направление развития специализации – специализация без шаблона.

Итак, пусть имеется специализируемая функция `S` и некоторый её вызов `<S ARG>` (см. Листинг 5).

Листинг 5. Исходные данные специализации.

```
F {
  … <S ARG> …
}
```

```
S {
  Pat1 Tail1;
  …
  PatN TailN;
}
```

Функции F и S – это функции в обессахаренном синтаксическом дереве, т.е. не содержат блоков и присваиваний, и соответственно, в аргументе ARG будут находиться не вложенные функции, а замыкания. Pat_i – это образцовое выражение в начале i-го предложения, $Tail_i$ – «остаток» предложения, содержащий условия и правую часть. Для простоты будем считать, что ARG пассивный (иначе заменим в нём вызовы функций на новые e-переменные) и множество переменных из тела S не пересекается со множеством переменных из ARG (иначе переименуем переменные в S).

Необходимо построить специализированный экземпляр S′ для данного вызова, учитывающий всю статически известную информацию о вызове. В предлагаемой реализации вся статически известная информация о вызове – это выражение ARG. А динамические данные – это значения переменных из ARG. Таким образом, вызов будет иметь вид <S′ wrap(vars(ARG))>, где vars(•) – это функция, возвращающая множество переменных из ARG, перечисленных в порядке их первого появления в ARG (упорядочивание гарантирует, что новые переменные, соответствующие вызовам, сохранят свой порядок, и что если какие-то переменные сужаются в открытые переменные в S/S′, то их порядок тоже сохранится), а wrap(•) – это функция, которая строит из переменных форматное выражение – s- и t-переменные перечисляет как есть, e-переменные заворачивает в скобки кроме последней.

Далее рассмотрим уравнение ARG : Pat_i, где Pat_i – i-й образец функции S (i = 1…N). Его решением будет набор пар (C_{i1}, A_{i1}), …, (C_{ij}, A_{ij}), …, (C_{iK_i}, A_{iK_i}), где K_i – число решений этого уравнения. Компоненты C_{ij} будут представлять собой сужения – подстановки для vars(ARG),

компоненты A_{ij} являются присваиваниями – подстановками для $vars(Pat_i)$. Тогда вид функции S′ можно будет описать следующим образом:

```
S′ {
  …
  wrap(vars(ARG)) / Cij  Taili / Aij;
  …
}
```

Однако, как было сказано ранее, не для любого уравнения вида ARG : Pat существует решение в виде конечного набора пар сужений и присваиваний. В этом случае может помочь введённое ранее динамическое обобщение. Если в функции S существуют такие образцы Pat_i, что уравнение ARG : Pat_i неразрешимо, то нужно обобщить ARG. Под задачей динамического обобщения в данном случае будет подразумеваться поиск такого обобщения (ARG′, S_g), что для всех образцов Pat_i функции S уравнение ARG′ : Pat_i разрешимо – его решение представимо в виде конечного набора пар сужений и присваиваний. Таким образом, при использовании динамического обобщения вызов функции S будет специализироваться по следующей схеме:

```
F {
  … <S′ wrap(vars(ARG′)) / Sg> …
}

S′ {
  …
  wrap(vars(ARG′)) / Cij  Taili / Aij;
  …
}
```

Рассмотрим пример, иллюстрирующий новый подход к специализации функций. Пусть имеется специализируемая функция Rot и некоторый ее вызов (см. Листинг 6).

Листинг 6. Пример исходных данных специализации.

```
… <Rot 'A' e.X e.Y> …

$SPEC Rot;

Rot {
 e.1 s.2 = s.2 e.1;
}
```

В этом случае

```
wrap(vars('A' e.X e.Y)) = (e.X) e.Y
```

и новый вызов, соответственно, будет иметь вид

```
<Rot@1 (e.X) e.Y>
```

При построении нового экземпляра `Rot@1` рассматривается уравнение

```
'A' e.X e.Y : e.1 s.2,
```

которое легко решается без необходимости динамического обобщения и даже без разрешения симметричных клэшей и сопоставления с открытыми переменными (используются исключительно правила для L-образцов). Решений у такого уравнения три, и формируют они следующие наборы сужений и присваиваний `(Ct, As)`:

```
(
  (e.Y → e.N1 s.N6, t.N2 → s.N6),
  (s.N6 ← s.2, 'A' e.X e.N1 ← e.1)
)
(
  (e.Y → ε, e.X → e.N3 s.N5, t.N4 → s.N5),
  ('A' e.N3 ← e.1, s.N5 ← s.2)
)
(
  (e.Y → ε, e.X → ε),
  (ε ← e.1, 'A' ← s.2)
```

```
)
```

Таким образом, функция `Rot@1` примет следующий вид:

```
Rot@1{
  (e.X) e.N1 s.N6 = s.N6 'A' e.X e.N1;

  (e.N3 s.N5) /*пусто*/ = s.N5 'A' e.N3;

  (/*пусто*/) /*пусто*/ = 'A';
}
```

3. РЕАЛИЗАЦИЯ

3.1. Реализация расширенного алгоритма обобщённого сопоставления

Все изменения вносились в исходный код компилятора Рефала-5λ в файл GenericMatch.ref (писалось все на самом Рефале-5λ, т.к. компилятор языка является самоприменимым) [3].

Изначально в рабочем файле находилась функция `Solve-Drive`, используемая при оптимизации прогонки функций. В новой реализации к ней добавилась функция `Solve-Spec`, использующаяся, соответственно, при специализации. Её исходный текст приведён на Листинге 7.

Листинг 7. Исходный код функции `Solve-Spec`.

```
/**
  <Solve-Spec (e.UsedVars) (e.Left) (e.Right)>
    == Success ((t.Contr*) (t.Assign*))* (t.Assign*) (e.Left^)
    |  Failure

  t.Contr ::= (t.Var ':' e.Val)
  t.Assign ::= (e.Val ':' t.Var)
*/

$ENTRY Solve-Spec {
  (e.UsedVars) (e.Left) (e.Right)
    = <AddCoordinateLabels e.Left> : e.Left^
    = <Solve-Spec-Aux
        (e.UsedVars)
        ((e.Left) ':' (e.Right))
        ()
      >;
}

Solve-Spec-Aux {
  (e.UsedVars) ((e.L) ':' (e.R)) (e.GenAssigns)
```

```
    = <Solve-Clashes (e.UsedVars) (None) ((e.L) ':' (e.R)) ()>
    : {
        e.Begin Generalize (e.Intervals) e.End
          = <DoGeneralize
              (e.UsedVars) (e.Intervals)
              (e.L) (e.GenAssigns)
            >
          : (e.UsedVars^) (e.L^) (e.GenAssigns^)
          = <Solve-Spec-Aux
              (e.UsedVars) ((e.L) ':' (e.R)) (e.GenAssigns)
            >;

        /* пусто */ = Failure;

        e.Success
          = Success
            <CombineResults () e.Success>
            (e.GenAssigns)
            (<ClearCoordinates e.L>);
      }
}
```

Она принимает на вход список всех используемых в выражениях уравнения переменных и собственно обе части исходного уравнения. Вернуть она может либо слово `Failure`, если уравнение неразрешимо (усеклись все ветки), либо полное решение уравнения вместе с динамическим обобщением.

В самом начале `Solve-Spec` вызывает вспомогательную функцию `AddCoordinateLabels`, которая добавляет в левую часть уравнения метки координат. Далее начинается итеративный процесс. Производится попытка разрешить исходное уравнение (вызывается функция `Solve-Clashes`), далее в блоке обрабатываются результаты решения. В случае успеха формируется итоговый ответ, в случае отсутствия решений – слово `Failure`. А если хотя бы одна из ветвей вернула запрос на обобщение, то вызывается вспомогательная функция `DoGeneralize`, которая находит в левой части исходного уравнения

обобщаемый участок с помощью координатных меток и заменяет его содержимое по следующему правилу:

- Замыкание обобщается до новой s-переменной.
- Всё остальное обобщается до новой e-переменной.

После осуществления обобщения наступает следующая итерация цикла, производится попытка решения обобщённого уравнения.

Изучая листинг, можно обратить внимание на необычное представление набора сужений. Вместе со списком подстановок он хранит также специальный идентификатор, показывающий, было ли уже добавлено к состоянию решателя сужение `e.X → e.Y e.Z`. Сделано так было из-за вышеописанного нюанса работы с открытыми e-переменными. Дойдя до момента, когда система клэшей состоит только из клэшей с открытыми e-переменными, решатель действует следующим образом: берёт очередной клэш из системы, если он тривиальный , то разрешает его обычным способом (формирует разбиение левой части, и для каждой точки разбиения формирует новую ветку в решении), но если клэш нетривиален, и среди сужений ранее было сужение вида `e.X → e.Y e.Z`, то для текущего клэша формируется запрос на обобщение.

Тело функции `Solve-Drive` также претерпело изменения в связи с переходом на новый алгоритм обобщённого сопоставления. Теперь она внутри себя также вызывает `Solve-Clashes`, но обрабатывает её результат отличным от `Solve-Spec` образом: в случае пустоты также возвращается `Failure`, при наличии хотя бы одного запроса на обобщение возвращается `Undefined`, а при успехе сначала проверяется флаг в сужениях. Если он поднят, то возвращается `Undefined`, иначе возвращается искомое полное решение.

Функция `Solve-Clashes` строго придерживается алгоритма, описанного в главе «Разработка». Каждое её предложение – это по сути перенос правила, описанного ранее на псевдокоде, на Рефал-5λ. Например, код анализа e-параметра для L-образца приведён на Листинге 8.

Листинг 8. Анализ e-параметра для L-образца.

```
/* {m} e.X E {n} : Pt P   ↦  e.X → t.NEW1 e.NEW2  ||  e.X → ε */
(e.UsedVars) (e.Contrs) e.ClashesStart
((('{'s.M'}') t.X e.E ('{'s.N'}')) ':' (t.Pt e.P))
e.ClashesEnd (e.Assigns)
  , t.X : (Var 'e' e.XIndex)
  , <IsTerm t.Pt> : True
  = <NewVarName (e.UsedVars) 't' e.XIndex> : t.NewVars1 't' e.New1
  = <NewVarName t.NewVars1 'e' e.XIndex> : t.NewVars2 'e' e.New2
  = <AddContraction-Spec
      (t.X ':' (Var 't' e.New1) (Var 'e' e.New2))
      (e.Contrs) e.ClashesStart
      ((('{'s.M'}') t.X e.E ('{'s.N'}')) ':' (t.Pt e.P))
      e.ClashesEnd (e.Assigns)
    >
  : e.Branch1
  = <AddContraction-Spec
      (t.X ':' /* пусто */)
      (e.Contrs) e.ClashesStart
      ((('{'s.M'}') t.X e.E ('{'s.N'}')) ':' (t.Pt e.P))
      e.ClashesEnd (e.Assigns)
    >
  : e.Branch2
  = <Solve-Clashes t.NewVars2 e.Branch1>
    <Solve-Clashes (e.UsedVars) e.Branch2>;
```

Здесь на входе принимается состояние решателя, затем проверяются необходимые условия для термов клэша. В случае их успешного прохождения, алгоритм ветвится, в состояние для каждой ветки добавляются новые сужения, а в случае первой ветки – и новые переменные.

При реализации обоих этапов алгоритма также создавалось много вспомогательных функций. Например, вот некоторые из них:

- `DoGeneralize` – функция, выполняющая динамическое обобщение;
- `AddCoordinateLabels` – функция, добавляющая в левую часть уравнения метки координат;

- `SimplifyCoordinates` – функция, выполняющая упрощение координат;
- `SplitLeftPart` – функция, разбивающая левую часть клэша на список выражений, каждое из которых содержит одну точку разбиения;
- `SplitSolve` – функция, разрешающая систему клэшей для одной точки разбиения;
- `CreateSymmClashes` – функция, создающая систему симметричных клэшей по списку присваиваний;
- `Solve-SymmClashes` – функция, разрешающая систему симметричных клэшей (вызывается из `Solve-Clashes` при наступлении второго этапа алгоритма).

3.2. Реализация алгоритма расширенной специализации

Все изменения, связанные непосредственно с расширенной специализацией, вносились в файл OptTree-Spec.ref. Новая реализация алгоритма оказалась существенно проще и короче старой. Отчасти из-за того, что при новом подходе были упрощены структуры данных, отвечающие за представление сигнатуры и истории сигнатур (история нужна для проверки отношения Хигмана-Крускала в целях предотвращения зацикливания специализации [14]):

```
-   t.Signature ::= ((e.InstanceName) t.StaticVarVals*)
-   t.StaticVarVals ::= (e.Expression)
+   t.Signature ::= ((e.InstanceName) e.Expression)
-   e.History ::= ((e.FuncName) t.StaticVarVals*)*
+   e.History ::= ((e.FuncName) e.Expression)*
```

Здесь знаком «-» отмечены старые грамматические правила, а знаком «+» – новые. Тело сигнатуры теперь является выражением, а не последовательностью нескольких выражений, заключенных в скобки, в силу того, что в новой реализации, как было описано выше, сигнатурой (статической информацией, учтенной при построении экземпляра) является выражение `ARG`.

Самым существенным изменениям подверглась функция SpecCall, выполняющая специализацию для данного вызова специализируемой функции. На вход она принимает вызов исходной функции, её тело и текущую историю сигнатур. Выходными данными являются новый вызов, новый экземпляр специализируемой функции и модифицированная история сигнатур. Причём, последние два значения могут отсутствовать.

Саму реализацию алгоритма можно описать следующим образом (исходный код приведён в Приложении А). Пусть на вход поступили следующие данные:

<S ARG>, S { Pat_1 $Tail_1$; …; Pat_N $Tail_N$; }, History

Тогда сначала происходит извлечение вызовов функций из ARG и их замена на новые e-переменные. При этом формируется новое значение аргумента ARG* и набор присваиваний S_g. Далее следовало бы рассматривать уравнения ARG* : Pat_i, но здесь имеется один нюанс. Дело в том, что необходимо следить за тем, чтобы образцы условий в $Tail_i$ в конечном итоге не содержали объектов замыканий, т.к. это синтаксически и семантически недопустимо. Для этого нужно сформировать список всех переменных из образцов условий (назовём его condvars), и в дальнейшем рассматривать уравнения вида

ARG* (e.NEW) : Pat_i (wrap(condvars)),

поскольку оно разрешимо, если переменным из condvars не присваиваются значения с замыканиями. И если уравнение неразрешимо, то эти замыкания будут обобщены. Для таких уравнений происходит вызов Solve-Spec. Причём при получении решения с динамическим обобщением (expr, S_g'), новое параметризованное выражение становится новым ARG*, S_g модифицируется по правилу

S_g := S_g ∪ { (E / S_g ← v) | (E ← v) ∈ S_g' },

полученные ранее решения стираются и цикл начинается сначала. При получении решения без динамического обобщения, оно просто добавляется к набору решений. Таким образом, после осуществления данного цикла должно

сформироваться такое обобщение (ARG*, S_g), что все уравнения ARG* : Pat_i разрешимы. Далее алгоритм уходит еще в один цикл, на каждой итерации которого происходит следующее. Если в результате обобщений получилась тривиальная сигнатура (ARG* == e.X), то возвращается ‹<S ARG>, ∅›. Иначе происходит формирование канонической формы сигнатуры sig (переменные согласованно переименовываются) и её поиск среди сигнатур готовых специализированных функций. Если сигнатура обнаружена, то в качестве ответа возвращается результат найденной специализации. Иначе происходит проверка отношения Хигмана-Крускала, чтобы исключить зацикливание специализации. В случае наличия отношения между sig и некоторой сигнатурой из истории sig', формируются их обобщение gen_sig и соответствующие присваивания S_g', связывающие ARG* и gen_sig, которая становится новым аргументом. Набор присваиваний S_g модифицируется по правилу:

S_g := { (E / S_g ← v) | (E ← v) ∈ S_g' }

Для нового ARG* происходит новое формирование решений с помощью Solve-Spec по принципу, описанному немного выше. После этого цикл запускается по новой. Если же отношение не установлено, то формируются предложения для нового экземпляра специализируемой функции по следующему принципу. Для каждого Pat_i $Tail_i$ и для каждого соответствующего ему решения (Ct, As) создается новое предложение

wrap(vars(ARG*)) / Ct $Tail_i$ / As;

После этого возвращается результат:

```
‹
  <S@NEW wrap(vars(ARG*)) / Sg>,
  (S@NEW {new_sentences}, History ∪ sig)
›
```

4. ТЕСТИРОВАНИЕ

Чтобы провести тестирование работоспособности новых алгоритмов, следует проделать следующие действия. Сначала нужно получить исходный код компилятора Рефала-5λ [3] с помощью системы контроля версий Git:

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

Компилятор является самоприменимым, поэтому одним из показательных тестов можно считать раскрутку компилятора. Для ее осуществления потребуется наличие на компьютере любого компилятора языка C++. Итак, для проведения раскрутки нужно перейти в директорию `src/compiler` и выполнить следующие команды (для Windows исполняемые файлы именуются так же, но с расширением `bat`):

```
RLMAKEFLAGS='-X-OiADPRS' ./makeself-s.sh
RLMAKEFLAGS='-X-OiADPRS' ./makeself.sh
```

Первая из них возьмёт стабильную версию компилятора и соберет ей новую. Вторая команда соберёт новую версию ей же самой. Передаваемые флаги указывают компилятору на необходимость использования алгоритмов оптимизации. В частности, будут выполняться оптимизации прогонки и специализации.

В репозитории компилятора имеется большое количество автотестов, поэтому для того, чтобы убедиться в том, что новые алгоритмы не ухудшают старое поведение, можно эти тесты запустить (после проведения раскрутки). Для этого нужно перейти в директорию `autotests` и выполнить команду `./run.sh`. Однако стоит отметить, что данная команда выполняется чрезвычайно долго, поэтому в целях экономии времени можно запустить тесты только для древесной оптимизации `./run.sh opt-tree*.ref`

Хотя вышеперечисленные способы тестирования позволяют в некоторой мере убедиться в том, что новые алгоритмы не портят старое поведение, они не позволяют убедиться в том, что новые алгоритмы корректно решают свои задачи.

Чтобы проверить работу только алгоритма обобщённого сопоставления, можно, например, создать в директории `build` следующий файл `test.ref`:

```
*$FROM TreeUtils
$EXTERN ExtractVariables;

*$FROM GenericMatch
$EXTERN Solve-Spec;

$ENTRY Go {
* (e.X) (e.X) : (e.1 '@' e.2) (e.3 '#' e.4)
  = ((Brackets (Var 'e' 'X')) (Brackets (Var 'e' 'X'))) : (e.Left)
  = (
      (Brackets (Var 'e' '1') (Symbol Char '@') (Var 'e' '2'))
      (Brackets (Var 'e' '3') (Symbol Char '#') (Var 'e' '4'))
    )
  : (e.Right)
  = (<ExtractVariables ((e.Left e.Right) ())>) : (e.UsedVars)
  = <Solve-Spec (e.UsedVars) (e.Left) (e.Right)>
  : Success (
      (
        ((Var 'eX')  ':' (Var 'eX0')(Symbol Char '@')(Var 'eX2'))
        ((Var 'eX1') ':' (Symbol Char '@')(Var 'eX2'))
        ((Var 'tX1') ':' (Symbol Char '@'))
        ((Var 'sX1') ':' (Symbol Char '@'))
        ((Var 'e')   ':' (Var 'e0')(Symbol Char '#')(Var 'e6'))
        ((Var 'e5')  ':' (Symbol Char '#')(Var 'e6'))
        ((Var 't5')  ':' (Symbol Char '#'))
        ((Var 's5')  ':' (Symbol Char '#'))
      )
      (
        ((Var 'eX0') ':' (Var 'e1'))
        ((Var 'eX2') ':' (Var 'e2'))
        ((Var 'e0')  ':' (Var 'e3'))
        ((Var 'e6')  ':' (Var 'e4'))
      )
```

```
    )
    (
      ((Var 'eX') ':' (Var 'e'))
    )
    (
      (Brackets (Var 'eX'))
      (Brackets (Var 'e'))
    )
  = /* пусто */;

* (t.X) t.X : t.Eq t.Eq
  = ((Brackets (Var 't' 'X')) (Var 't' 'X')) : (e.Left)
  = ((Var 't' 'Eq') (Var 't' 'Eq')) : (e.Right)
  = (<ExtractVariables ((e.Left e.Right) ())>) : (e.UsedVars)
  = <Solve-Spec (e.UsedVars) (e.Left) (e.Right)>
  : Success (
      (
        ((Var 'e0') ':' (Brackets (Var 'e3')))
        ((Var 'e1') ':' )
        ((Var 't0') ':' (Brackets (Var 'e3')))
        ((Var 'e')  ':' (Var 'e3'))
        ((Var 'e2') ':' (Var 'e3'))
      )
      (
        ((Brackets (Var 'e3')) ':' (Var 'tEq'))
      )
    )
    (
      ((Var 'tX') ':' (Var 'e'))
      ((Var 'tX') ':' (Var 'e0'))
    )
    (
      (Brackets (Var 'e'))
      (Var 'e0')
    )
  = /* пусто */;
```

```
}
```

В нём содержатся тесты для двух уравнений, покрывающих все этапы реализованного алгоритма. Это позволит в какой-то мере убедиться в том, что алгоритм способен решать сложные уравнения и выполнять динамическое обобщение. Запустить данный тестовый файл можно с помощью следующей команды:

```
rlmake --rich test.ref --dir ../src/compiler && ./test
```

Если во время выполнения этого кода не произошло ни одной ошибки, значит тесты прошли успешно.

Для тестирования корректности работы нового алгоритма специализации можно воспользоваться, например, встроенной в компилятор возможностью формирования лога состояния программы на момент различных проходов компилятора [5]. С помощью лога можно отследить, каким образом изменялся код программы после специализации функций. Например, выведя в лог результат специализации программы, представленной на Листинге 9, можно убедиться, что он совпадёт с тем, что было описано в примере специализации из главы «Разработка».

Листинг 9. Пример для тестирования специализации.

```
$SPEC Rot e.dyn;

Rot {
  e.1 s.2 = s.2 e.1;
}

Example {
  (e.X) (e.Y) = <Rot 'A' e.X e.Y>;
}

$ENTRY Go {
 = <Example (1 'a' 2) (3 'b' 4)>;
}
```

В качестве еще одного примера для демонстрации возможностей нового алгоритма специализации можно рассмотреть следующую программу:

```
$ENTRY AllA {
  e.X = <Eq (e.X A) (A e.X)>;
}

$SPEC Eq e.arg;

Eq {
  t.X t.X = True;
  t._ t._ = False;
}
```

Функция `AllA` возвращает `True`, если её аргумент состоит из нуля или более символов `A`. Данный пример интересен тем, что при специализации `Eq` будут разрешаться симметричные клэши, а также будет производиться динамическое обобщение. В итоге, в логе компиляции можно увидеть следующий результат специализации:

```
$ENTRY AllA {
  e.X = <Eq@1 (e.X) e.X>;
}

Eq@1 {
  (A e.3) e.3 A = True;

  (/* пусто */) /* пусто */ = True;

  (e.1) e.2 = False;
}
```

И наконец, довольно показательным примером является программа, рассмотренная на Листинге 10. Содержательной её частью является интерпретатор стекового языка программирования.

Листинг 10. Демонстрационная программа.

```
$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′,   P′ = P×N  */
            swap  /* ... P′ N              */
            q 1   /* ... P′ N 1            */
            '-'   /* ... P′ N′,  N′ = N−1  */
          )
                  /* ... P 0               */
          drop    /* ... P                 */
        )
      >
}

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

Int {
 /* 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;
}
```

Данный пример ранее уже рассматривался в рамках одной из дипломных работ [15], но с одним отличием. Дело в том, что предыдущая версия компилятора не способна оптимизировать функцию `Int` в том виде, в каком она представлена выше. Поэтому приходилось делать следующий некрасивый трюк:

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

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

```
$DRIVE Int-Drive;

Int-Drive {
  …
}
```

Новая же версия специализатора таких трюков не требует и позволяет выполнять оптимизацию для функции `Int`, записанной в «естественном» виде. Компиляция данной программы с ключами `-OADS` интересна тем, что в остаточной программе интерпретируемого кода и интерпретатора не останется, код на интерпретируемом языке полностью скомпилируется в Рефал. То есть построится первая проекция Футамуры-Турчина, которая заключается в том, что имея специализатор и интерпретатор, можно компилировать программы для интерпретатора в целевой язык специализатора [16]. Таким образом, Рефал-5λ позволяет встраивать в программу простые интерпретаторы, которые при компиляции «растворяются». А новый алгоритм специализации позволяет писать такие интерпретаторы проще.

ЗАКЛЮЧЕНИЕ

В результате выполнения настоящей работы был реализован и отлажен новый алгоритм обобщённого сопоставления, способный получать решения для более широкого класса уравнений, чем его предшественник, а также поддерживающий механизм динамического обобщения.

Создание такого алгоритма открыло возможности для расширенной специализации функций (специализации без шаблона). Данный алгоритм также был реализован и отлажен в рамках текущего дипломного проекта. Таким образом, теперь возможно специализировать любой вызов любой функции, возможно с обобщением. Это позволяет программисту писать выразительный код без ущерба для производительности.

В качестве направления дальнейшего развития алгоритма обобщённого сопоставления можно выделить добавление новых ad hoc правил разрешения симметричных клэшей, которые позволят проводить более глубокий анализ уравнений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рефал-5λ. Введение в язык. Режим доступа: https://bmstu-iu9.github.io/refal-5-lambda/2-intro.html (дата обращения 08.06.2021)
2. А. П. Немытых, Лекции по языку программирования Рефал. Сборник трудов по функциональному языку программирования Рефал, том I // Под редакцией А. П. Немытых. — Переславль-Залесский: Издательство «СБОРНИК», 2014, 194 с. — ISBN 978-5-9905410-1-6 — стр. 120.
3. Репозиторий компилятора Рефала-5λ. Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda (дата обращения 08.06.2021)
4. Рефал-5λ. Основы программирования на Рефале. Режим доступа: https://bmstu-iu9.github.io/refal-5-lambda/3-basics.html (дата обращения 08.06.2021)
5. Рефал-5λ. Приложение B. Краткий справочник. Режим доступа: https://bmstu-iu9.github.io/refal-5-lambda/B-reference.html (дата обращения 08.06.2021)
6. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М.: Изд. дом Вильямс, 2008. – 1184 с.
7. Сухомлинова Д.П. ВКР на тему «Специализация функций в Рефале-5λ». Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/РПЗ_Сухомлинова_Специализация_функций_2019.pdf (дата обращения 08.06.2021)
8. Климов Андрей Викторович Введение в метавычисления и суперкомпиляцию [Раздел книги] // Будущее прикладной математики: Лекции для молодых исследователей. От идей к технологиям. - Москва : КомКнига, 2008. - ISBN 978-5-484-01028-8.
9. А. П. Немытых. Суперкомпилятор SCP4: общая структура. Монография, М: Издательство URSS, 2007, ISBN 978-5-382-00365-8. — 152 с.

10. Г. С. Маканин. «Проблема разрешимости уравнений в свободной полугруппе», Матем. сб., 103(145):2(6) (1977), с. 147–236.
11. В.Ф.Турчин. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ – Киев-Алушта, 1972.
12. С.А.Романенко. Прогонка для программ на РЕФАЛе-4. Препринт №211 – Институт Прикладной Математики АН СССР, 1987
13. Суперкомпилятор MSCP-A. Режим доступа: https://github.com/TonitaN/MSCP-A (дата обращения 08.06.2021)
14. Кошелев А.А. ВКР на тему «Использование отношения Хигмана-Крускала для прерывания рекурсивной специализации функций». Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/РПЗ_Кошелев_Отношение_Хигмана-Крускала_2020.pdf (дата обращения 08.06.2021)
15. Калинина Е.А. ВКР на тему «Автоматическая разметка оптимизируемых функций в компиляторе Рефала-5λ». Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/РПЗ_Калинина_Автоматическая%20разметка%20оптимизируемых_функций_2020.pdf (дата обращения 08.06.2021)
16. Абрамов С. М., Пармёнова Л. В. Метавычисления. Часть II. – ПереславльЗалесский: «Университет города Переславля», 2016.

ПРИЛОЖЕНИЕ А

Листинг А.1. Исходный код нового алгоритма специализации.

```
SpecCall {
  (e.Name) (e.SpecPattern) (e.Body)
  s.NextNumber e.Signatures
  (e.Argument) (e.History)
    = <ExtractVariables ((e.Argument) (/* пусто */))>
    : e.UsedVars
    = <ExtractCalls-Expr (/* пусто */ (e.UsedVars)) e.Argument>
    : (e.ExtractedCalls (e.UsedVars^)) e.NewArgument
    /* ищем динамическое обобщение для аргумента и образцов */
    = <DynGenArg
        (e.NewArgument) (e.ExtractedCalls) (/* пусто */)
        (/* пусто */) e.Body
      >
    : (e.NewArgument^) (e.Sg) (e.Solutions)
    = <SpecCall-Aux
        (e.Name) (e.SpecPattern) (e.Body)
        s.NextNumber e.Signatures
        (e.Argument) (e.NewArgument) (e.Sg)
        (e.Solutions) (e.History)
        True
      >;
}

DynGenArg {
  (e.Arg) (e.Sg) (e.Sol) (e.Begin)
  ((e.Pat) e.Tail) e.Sentences
    = <ExtractVariables ((e.Arg e.Pat) e.Tail)> : e.UsedVars
    = <NewVarName (e.UsedVars) 'eNew'> : (e.UsedVars^) e.eNew
    = e.Arg (Brackets (Var e.eNew)) : e.Arg^
    = <WrappedCondVars e.Tail> : e.WrappedCondVars
    = e.Pat (Brackets e.WrappedCondVars) : e.Pat^
    = <Solve-Spec (e.UsedVars) (e.Arg) (e.Pat)>
    : {
        Success e.Solutions (e.SgNew) (e.ArgNew)
          , e.SgNew : /* пусто */
          = e.ArgNew : e.ArgNew^ (Brackets e._)
```

```
          = e.Pat : e.Pat^ (Brackets e._)
          = <DynGenArg
              (e.ArgNew) (e.Sg) (e.Sol (e.Solutions))
              (e.Begin ((e.Pat) e.Tail)) e.Sentences
            >;

        Success e.Solutions (e.SgNew) (e.ArgNew)
          = e.ArgNew : e.ArgNew^ (Brackets e._)
          = e.Pat : e.Pat^ (Brackets e._)
          = <DynGenArg
              (e.ArgNew) (e.Sg <ApplySubst-Subst (e.Sg) e.SgNew>)
              (/* пусто */) (/* пусто */)
              e.Begin ((e.Pat) e.Tail) e.Sentences
            >;

        Failure
          = e.Arg : e.Arg^ (Brackets e._)
          = e.Pat : e.Pat^ (Brackets e._)
          = <DynGenArg
              (e.Arg) (e.Sg) (e.Sol (Failure))
              (e.Begin ((e.Pat) e.Tail)) e.Sentences
            >;
      };

  (e.Arg) (e.Sg) (e.Sol) (e.Begin) /* пусто */
    = (e.Arg) (e.Sg) (e.Sol);
}

SpecCall-Aux {
  /* тривиальная сигнатура */
  (e.Name) (e.SpecPattern) (e.Body) s.NextNumber e.Signatures
  (e.OldArg) (e.NewArg) (e.Sg) (e.Solutions) (e.History)
  s.NeedRelationCheck
    , <IsTrivialSignature e.NewArg> : True
    = ((e.SpecPattern) (e.Body) s.NextNumber e.Signatures)
      (CallBrackets (Symbol Name e.Name) e.OldArg)
      /* пусто */ (/* пусто */);

  /* проверяем, известна ли сигнатура */
```

```
(e.Name) (e.SpecPattern) (e.Body) s.NextNumber e.Signatures
(e.OldArg) (e.NewArg) (e.Sg) (e.Solutions) (e.History)
s.NeedRelationCheck
  = <RenameSignatureVars-Expr (/* пусто */ 0) e.NewArg>
  : t._ e.Signature
  = <Spec-FindInSignatures (e.Signature) e.Signatures>
  : {
      Found e.InstanceName (e.Signatures^)
        = ((e.SpecPattern) (e.Body) s.NextNumber e.Signatures)
          (CallBrackets
            (Symbol Name e.InstanceName)
            <ApplySubst-Expr (e.Sg) <WrapVars e.NewArg>>
          )
          /* пусто */ (/* пусто */);

      /* проверяем на зацикливание */
      NotFound e.Signatures^
        = s.NeedRelationCheck
        : {
            True
              = <HasHigmanKruskalRelation
                  (e.Name) (<OptTree-CanonizeExpr e.Signature>)
                  e.History
                >;

            False = False;
          }
        : {
            True e.HistorySignature
              /* Получает обобщённую сигнатуру для двух сигнатур */
              = <GlobalGen (e.Signature) (e.HistorySignature)>
              : e.GenSignature
              = <ExtractVariables ((e.NewArg) (/* пусто */))>
              : e.ArgVars
              = <NameSignatureVars-Expr (e.ArgVars) e.GenSignature>
              : t._ e.GenSignature^
              = <GenericMatch (e.NewArg) (e.GenSignature)>
              : Clear e.NewSg
              = <IsTrivialSubstitutions e.NewSg>
```

```
    : {
        /*
          Если подстановка e.NewSg является тривиальной,
          проверку отношения Хигмана-Крускала
          в рекурсивном вызове не выполняем
        */
        True = False;

        False = True;
      }
    : s.NeedRelationCheck^
    = <DynGenArg
        (e.GenSignature)
        (<ApplySubst-Subst (e.Sg) e.NewSg>)
        (/* пусто */) (/* пусто */) e.Body
      >
    : (e.NewArg^) (e.Sg^) (e.Solutions^)
    = <SpecCall-Aux
        (e.Name) (e.SpecPattern) (e.Body) s.NextNumber
        e.Signatures (e.OldArg) (e.NewArg)
        (e.Sg) (e.Solutions) (e.History)
        s.NeedRelationCheck
      >;

  False
    = <CreateNewSentences
        (e.NewArg) (/* пусто */)
        (e.Solutions) (e.Body)
      >
    : e.NewSentences
    = <AddSuffix e.Name ('@' s.NextNumber)>
    : e.InstanceName
    = (
        (e.SpecPattern) (e.Body) <Inc s.NextNumber>
        e.Signatures ((e.InstanceName) e.Signature)
      )
      (CallBrackets
        (Symbol Name e.InstanceName)
        <ApplySubst-Expr (e.Sg) <WrapVars e.NewArg>>
```

```
                    )
                    (Function
                      GN-Local (e.InstanceName)
                      Sentences e.NewSentences
                    )
                    (
                      (
                        (e.InstanceName) e.History
                        ((e.InstanceName) e.Signature)
                      )
                    );
            }
        };
}
```