Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования Московский
государственный технический университет имени Н.Э. Баумана

Курсовая работа
«Вычислитель и верификатор типов функций для языка РЕФАЛ-5»
по курсу
«Конструирование компиляторов»

Студент группы ИУ9-72 Иванов Г.М.

Руководитель Коновалов А.В.

Москва, 2018

Содержание

1 Введение

Функциональное программирование — это парадигма программирования, в которой вычисление понимается как вычисление значений функций в математическом понимании. Функция является базовым элементом, которая используется для расчётов вычислений. Даже переменные заменяются функциями. В функциональном программировании переменные — это просто синонимы (alias) для выражений, для того чтобы нам не пришлось писать всё в одну строку. Они неизменяемы. Существует много функциональных языков, и большинство из них делают одни схожие вещи по-разному: LISP (Clojure), Haskell, Scala, R.

РЕФАЛ (РЕкурсивных Функций АЛгоритмический язык) — это один из старейших функциональных языков программирования, который ориентирован на символьные преобразования: обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой. РЕФАЛ был впервые реализован в 1968 году в России Валентином Турчиным, где широко используется и поныне, который соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ. В 1970-1990 были реализованы несколько различных диалектов этого языка. В МГТУ имени Н.Э. Баумана на кафедре ИУ9 компилятор этого языка используется в качестве учебного полигона для курсовых и дипломных работ. [2]

Основа РЕФАЛа состоит в сопоставлении с образцом, подстановке и рекурсивном вызове, поэтому легко реализовать символьную обработку. Программа на РЕФАЛе вдвое-втрое короче по объему, чем аналогичная программа на языке LISP, и гораздо более читаема. В базисном Рефале выражаются через вспомогательные функции конструкции вида: if, while, for, switch.

Синтаксис РЕФАЛа не позволяет принимать функциям более одного аргумента — объектное выражение. Если в функцию передать множество аргументов, то тогда из них будет формироваться одно единое объектное выражение, которое в теле функции будет разбираться, соответствующее входным аргументам. Исходя из этого в РЕФАЛ могут возникать ошибки с типом аргументов — это передача в функцию данных, не входящих в область определения функции — в таких случаях происходит аварийный останов программы.

Таким образом, вытекает актуальность моей работы. Она заключается в определении области определения функции (множество объектных выражений, на которых функция не падает) и множество значений (все значения функции на области определения).

Целью данной курсовой работы является разработка верификатора типов в РЕФАЛ, который будет проверять корректность вызовов функций в программе, т.е. аргумент в любом вызове функции должен быть совместим с форматом аргумента. При этом пользователь может псевдокомментариями подсказывать верификатору форматы некоторых функций в сложных или неоднозначных случаях, поэтому такие подсказки должны проверяться на непротиворечивость. Благодаря верификатору производится проверка корректности программ, а также повышение местности и ко-местности функций, что полезно при векторном [8] и векторно-списковом [9] представлении данных. Форматы встроенных функций будут вшиты в верификатор.

Входной язык верификатора будет является классический РЕФАЛ-5. Почему не Простой Рефал? В Простом Рефале есть вложенные функции, которые серьёзно усложнят анализ. Почему не Рефал-6, Рефал-7? В них есть неуспехи, которые тоже усложнят анализ. Почему не Рефал+? В нём уже есть явная статическая типизация — типы всех функций должны быть явно описаны пользователем. Да-

лее будет понятно, что мы хотим неявную статическую типизацию для РЕФАЛа-5.

В ходе работы должны быть выполнены следующие задачи:

- Обзор предметной области, разработка языка описания типов;
- Реализация лексического и синтаксического анализа для РЕФАЛ-5 и языка описания типов;
- Реализация семантического анализа. Разработка и реализация алгоритмов вывода и проверки типов;
- Тестирование работоспособности приложения.

2 Обзор предметной области, разработка языка описания типов

В настоящее время основными диалектами языка являются РЕФАЛ-2 (1970-е), РЕФАЛ-5 (1985) и РЕФАЛ+ (1990), отличающиеся друг от друга деталями синтаксиса и набором «дополнительных средств», расширяющих первоначальный вариант. В документации диалекта РЕФАЛ-5 определен набор базовых понятий (так же называемых понятиями Базисного Рефала), использующихся во многих других диалектах языка, поэтому общие понятия можно рассмотреть на примере этого диалекта.

Изначально, язык Рефал работает с символьными данными. Поэтому основными объектами являются символы. Символы бывают простые и составные. Простой символ — это обычный символ. При записи программы простой символ обрамляется апострофами, за исключением самого апострофа. Составной символ — это либо составной символ-метка (детерминатив, имя функции), либо составной символ-число — макроцифра. Составной символ-метка — это последовательность букв, цифр и знаков -, _. Длина составного символа-метки не ограничивается. Составной символ-число (макроцифра) — это последовательность цифр, число. Число записывается в обычной десятичной системе счисления. «Длинная» арифметика реализована в РЕФАЛ, а это означает, что большие числа трактуются как последовательность цифр, если число превышает 2^{32}. 1 2 означает число $1 * 2^{32} + 2$. Последовательность символов — это несколько символов, идущих подряд. При записи последовательности простых символов в программе они разделяются пробелами.

Объектное выражение — это выражение, состоящее из символов, а также из левых и правых структурных скобок «(» и «)». Скобки должны образовывать правильную скобочную структуру. Выражение может быть, в частности, и пустым. Образцовое выражение — это выражение с переменными, но без вызовов функций, задаёт

некоторое множество объектных выражений (элементы множества получаются путём подстановки на место переменных допустимых объектных выражений)

Переменная символа записывается так: знак «s», символ «точка», за которым следует индекс. Переменная выражения записывается так: знак «e», символ «точка», и сразу за ним — индекс переменной (любая непустая последовательность из латинских букв, цифр и знаков - (дефис) и _ (прочерк), как и в случае с именами, последние два символа эквивалентны, индексы чувствительны к регистру). Переменная терма записывается так: знак «t», символ «точка», сразу за ним — индекс переменной. Переменная символа, выражения, терма являются свободными переменные. Две переменные называются повторными, если их типы и индексы совпадают.

Переменная символа может принимать значение одного символа (любого). Переменная выражения может принимать значение любого объектного выражения. Переменная терма может принимать значение любого терма, т. е. либо одного символа, либо одного объектного выражения, заключенного в структурные скобки.

Семантика языка РЕФАЛ описывается в терминах виртуальной машины, называемой «РЕФАЛ-машина» или «РЕФАЛ-автомат».

Конкретизация (вызов функции) определяется при помощи двух скобок, заголовка и аргументов функции. В процессе работы РЕФАЛ-машины выполняется процедура конкретизации, которая обозначается угловыми скобками. После открывающейся угловой скобки следуют заголовок функции, ее аргументы и закрывающаяся угловая скобка. В результате работы конкретизации, функция возвращает в ходе вычисления результат, который заменяет терм конкретизации в поле зрения. Если скобки конкретизации вложены друг в друга, РЕФАЛ-машина вызывает их поочерёдно, слева направо, начиная с самой внутренней.

Программа на Рефале состоит из последовательности программных элементов — определений функций и ссылок на функции, определённые в других единицах трансляции («$EXTERN»). Переводы

строк приравнены к обычным пробельным символам (пробел или табуляция). Пробельные символы можно вставлять между двумя любыми лексемами языка. Пробелы обязательны в том случае, когда две лексемы, записываемые подряд, могут интерпретироваться как одна сплошная лексема (например, два числа, записанные слитно, будут интерпретироваться как одно число и т.д.). Пробелы недопустимы внутри идентификаторов, чисел и директив. Пробелы внутри цепочек литер интерпретируются как образы литер со значением «пробел».

Глобальная регулярная функция представляет собой именованный блок из набора предложений, который может начинаться с ключевого слова «$ENTRY». Сама Рефал-функция представляет собой упорядоченный набор предложений, состоящих из образца и шаблона, разделённых символом «;». Опишем понятие процедур сопоставления с образцом. При вызове функции, выполняется попытка сопоставить входное выражение с левой частью первого предложения. Если сопоставление удаётся, то возвращается правая часть, иначе происходит переход к следующему предложению и так далее. Если ни одно предложение не может быть сопоставлено с аргументом функции, то программа завершает свою работу с ошибкой. Во избежание остановки в конце функции помещают предложение, с образцом которого можно сопоставить произвольное выражение.

Приведем простой пример программы на РЕФАЛ-5 (функция, подсчитывающая количество букв A во входном выражении)
Листинг 1. Программа на РЕФАЛ-5

```
1  F {
2      s.Count e.Items A =
3          <F <Add s.Count 1> e.Items>;
4      s.Count e.Items s.Other =
5          <F s.Count e.Items>;
6      s.Count /* пусть */ = s.Count;
7  }
```

Если говорить формально, то все функции в языке РЕФАЛ име-

ют ровно один аргумент и всегда выдают ровно один результат. Однако этот единственный аргумент состоит из частей, в действительности который является подаргументам. В определении функции производится объединение этих аргументов каким-либо образом, чтобы образовать единственный формальный аргумент. Это задает формат функций.

В РЕФАЛе существует единственный способ анализа объектного выражения — сопоставление с образцом. Поэтому, чтобы извлечь из одного объектного выражения его составные части (отдельные значения), его нужно сопоставить с образцом. В следствие этого формат мы будем записывать в виде некоторого образцового выражения. Для разделения двух подвыражений, требуется только одна пара круглых скобок, поэтому имеется очевидная свобода в выборе форматов:

$$\begin{cases} < F(e1)e2 > \\ < Fe1(e2) > \\ < F(e1)(e2) > \end{cases} \tag{1}$$

Продолжим рассматривать форматы функций на основе РЕФАЛ+. В этом диалекте РЕФАЛ уже есть явная статическая типизация — типы всех функций должны быть явно описаны пользователем. Рассмотрим её более подробно. Если говорить формально, то все функции в языке РЕФАЛ+ имеют ровно один аргумент и всегда выдают ровно один результат. Но чаще всего мы заранее знаем, какую структуру должны иметь аргумент и результат, например, функция Add всегда должна получать на вход объектное выражение, состоящее из двух символов, и всегда выдавать результат в виде объектного выражения, состоящее из одного символа. Ограничения, накладываемые на структуру аргументов и результатов функций, описываются с помощью объявлений спецификации функций. Так, например, объявление функции Add имеет вид:

$$\$func\ Add\ sX\ sY\ = sZ; \tag{2}$$

В общем случае объявление функции F имеет вид:

$$\$func\ F\ F_{in}\ = F_{out}, \tag{3}$$

где F_{in} — входной формат функции, а F_{out} — ее выходной формат. Форматы функции могут содержать символы, круглые скобки и переменные. Индексы переменных никакой информации не несут, используются в качестве комментариев и могут опускаться.

Определим еще несколько понятий:

- Жёсткое выражение — это образцовое выражение без открытых и повторных переменных, т. е. на одном уровне скобок может находиться не более чем одна e-переменная.
- Жёсткое выражение P является уточнение жёсткого выражения Q — P получается из Q путём подстановки на место переменных допустимых жёстких выражений.
- Q обобщает P — отношение, обратное уточнению.
- Аппроксимация множества объектных выражений жёстким выражением — такое жёсткое выражение P, которое включает в себя все выражения из множества, но при этом не существует такого выражения R, уточняющего P, что R также будет включать в себя все выражения из множества.

Все образцы и результаты функции в РЕФАЛ+ должны иметь структуру, предписанную ее объявлением спецификации. Объявление спецификации функции должно предшествовать ее первому использованию в каком-либо результатном выражении в программе. При этом, если функция определена в самой программе, ее объявление появляется в тексте программы явно.

Во время компиляции программы производится проверка того, что во всех вызовах функции структура ее аргумента соответствует ее входному формату. Например, результатное выражение <Add 2

<Add sX sY> > построено правильно. Для того чтобы проверить корректность внешнего вызова, уже требуется привлечь информацию о структуре результата функции Add. А именно, заменяем <Add sX sY> на выходной формат функции Add, в результате чего получается <Add 2 sZ>. Отсюда видно, что аргумент внешнего вызова соответствует входному формату функции Add. С другой стороны, если мы напишем в программе результатное выражение <Add 2 <Add sX sY> 3> оно будет расцениваться компилятором как ошибочное, поскольку аргумент наружного вызова функции Add состоит из трех символов, а не из двух, как предписано ее входным форматом.

Таким образом, информация о входных и выходных форматах функций позволяет находить многие ошибки в программе еще на стадии компиляции.

Изучив статическую типизацию РЕФАЛ+ [6], базисом для собственного языка типов в динамическом языке РЕФАЛ-5 будем иметь ввиду статическую типизацию в РЕФАЛ+, а именно для определения типа функции для верификатора спецификация функции будет иметь вид:

$$func\_name\ format_{in}\ = format_{out} \tag{4}$$

3 Реализация лексического и синтаксического анализа для РЕФАЛ-5 и языка описания типов

Для начала определим основные определения для лексического анализа.

- Лексический анализ — это фаза компиляции, объединяющая последовательно идущие во входном потоке кодовые точки в группы, называемые лексемами исходного языка.
- Лексический домен — это заданное некоторым формальным способом множество строк.Лексический домен может быть задан с помощью порождающей грамматики. На практике лексический домен часто является регулярным языком.
- Лексема — это фрагмент текста исходной программы, принадлежащий некоторому лексическому домену.
- Токен — это описатель лексемы, представляющий собой кортеж вида $\langle$ domain, coords, attr $\rangle$, где domain — лексический домен, которому принадлежит лексема, coords — координаты лексемы в тексте программы, а attr — атрибут токена.

Опишем основные лексические домены языка РЕФАЛ-5 (в виде регулярных выражений), исходя из выше упомянутых определений базисного РЕФАЛа.

```
1  EXTERN_KEYWORD=\$EXTERN|\$EXTRN|\$EXTERNAL|\$ENTRY
2  IDENT=[A-Za-z][-A-Za-z_0-9]*
3  MACROGIDIT=[0-9]*
4  VARIABLE=[set]\.[-A-Za-z_0-9]+
5  COMPOSITE\_SYMBOLS=IDENT|([^\\"]|\\([\\nrt"'()<>]|x[0-9a-fA-F]{2}))*
6  CHARS=([^\\']|\\([\\nrt"'()<>]|x[0-9a-fA-F]{2}))*
7  MARK_SIGN=[()<>=;:,{}]
8  LEFT_CALL_BRACKET=<[+-*/\%?]
```

Сам лексический анализатор напишем по UML-диаграмме, представленной на рисунке 1.

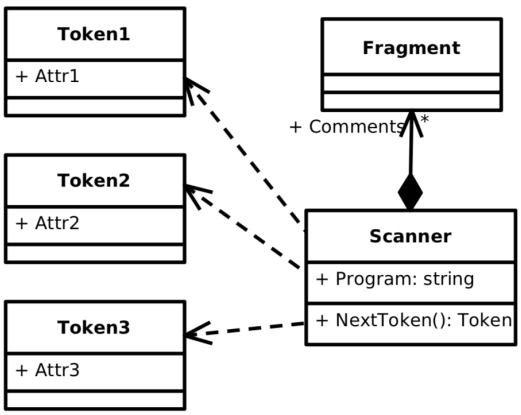

Рис. 1: Рисунок 1. UML-диаграмма лексического анализатора

Приведем пример реализации лексического анализатора на Python.
Листинг 2. Лексический анализатор, функция next_token.

```
def next_token(self):
    tok = UnknownToken(self.cur.letter(), Fragment(self.cur, self.cur))
    while tok.tag == DomainTag.Unknown:
        while self.cur.is_white_space():
            self.cur = next(self.cur, self.cur)
        if self.cur.letter() == "/" and self.cur != next_or_current(self.cur)
        ↪  and \
                next_or_current(self.cur).letter() == "*":
            self.read_many_line_comment()
            continue
        if self.cur.letter() == "*":
            self.read_comment()
            continue
        if self.cur.is_eof():
            tok = EopToken(Fragment(self.cur, self.cur))
        else:

```

```
            if self.cur.letter() == "$":
                read_tok = self.read_keyword()
            ...
            else:
                read_tok = UnknownToken(self.cur.letter(), Fragment(self.cur,
                ↪  self.cur))
            if read_tok.tag == DomainTag.Unknown:
                sys.stderr.write("Token[" + str(read_tok) + "]: unrecognized
                ↪  token\n")
                self.cur = next(self.cur, self.cur)
            else:
                self.cur = read_tok.coords.following
                self.cur = next_or_current(self.cur)
                tok = read_tok
    return tok
```

После выделения лексем происходит синтаксический анализ. Синтаксический анализ — это фаза компиляции, группирующая лексемы, порождаемые на фазе лексического анализа, в синтаксические структуры. Задача синтаксического анализа — проверить является ли последовательность лексем предложением в грамматике G. В случае положительного ответа на этот вопрос следует построить абстрактное дерево вывода последовательности лексем в грамматике G.

Определим BNF грамматику РЕФАЛ-5.

```
Program ::= Global*;
Global ::= Externs | Function | ';';
Externs ::= ExternKeyword 'Name' (',' 'Name')* ';';
ExternKeyword ::= '$EXTERN' | '$EXTRN' | '$EXTERNAL';
Function ::= ('$ENTRY')? 'Name' Body;
Body ::= '{' Sentences '}';
Sentences ::= Sentence (';' Sentences?)?;
Sentence ::= Pattern ( ('=' Result) | (',' Result ':' (Sentence | Body)) );
Pattern ::= PatternTerm*;
PatternTerm ::= Common | '(' Pattern ')';
Common ::= 'Name' | ''chars'' | '123' | 's.Var' | 't.Var' | 'e.Var';
Result ::= ResultTerm*;
ResultTerm ::= Common | '(' Result ')' | '<Name' Result '>';
```

Почти аналогично РЕФАЛ+, попробуем определить BNF грамматику языка описания типов функций РЕФАЛ-5.

```
File ::= Function*
Function ::= 'Name' Format '=' Format ';'
Format ::= Common ('e.Var' Common)?
Common ::= ('Name' | ''chars'' | '123' | 't.Var' | 's.Var' | '(' Format ')')*
```

Для реализации синтаксического анализатора будем использовать метод рекурсивного спуска. В синтаксическом анализаторе, написанном методом рекурсивного спуска, каждому нетерминалу грамматики соответствует отдельная функция, в которой закодирован эффект применения соответствующего нетерминалу правила (мы будем считать, что такое правило единственно). Цель этой функции — анализ последовательности токенов, которые по её запросу выдаёт лексический анализатор, и проверка соответствия этой последовательности правилу грамматики.

Функция, соответствующая нетерминалу X грамматики, составляется с учётом того, что:

- существует глобальная переменная Sym, в которую ДО вызова функции помещается токен, соответствующий первому символу цепочки, в которую раскрывается нетерминал X;
- функция должна либо полностью потребить последовательность токенов, в которую раскрывается нетерминал X, либо сгенерировать сообщение об ошибке;
- после завершения функции переменная Sym должна содержать токен, непосредственно следующий за раскрытым нетерминалом X.

Приведем пример реализации синтаксического анализатора на Python.

Листинг 3. Синтаксический анализатор языка описания типов функций РЕФАЛ-5

```
# Common ::= ('Name' | ''chars'' | '123' | 's.Var' | 't.Var' | '(' Format ')')* |
def parse_common(self):
    common_term = []
    while self.cur_token.tag == DomainTag.Ident or self.cur_token.tag == DomainTag.Number or \
            self.cur_token.tag == DomainTag.Characters or self.cur_token.tag == DomainTag.Composite_symbol \
            or (self.cur_token.tag == DomainTag.Variable and (self.cur_token.value[0] == "s" or self.cur_token.value[0] == "t")) \
            or (self.cur_token.tag == DomainTag.Mark_sign and self.cur_token.value == "("):
        if self.cur_token.tag == DomainTag.Mark_sign and self.cur_token.value == "(":
            self.cur_token = next(self.iteratorTokens)
            common_term.append(StructuralBrackets(self.parse_format().terms))
            if self.cur_token.tag == DomainTag.Mark_sign and self.cur_token.value == ")":
                self.cur_token = next(self.iteratorTokens)
            else:
                sys.stderr.write("Expected \")\" after declaring pattern\n")
                self.isError = True
        else:
            if self.cur_token.tag == DomainTag.Ident:
                token = self.cur_token
                self.cur_token = next(self.iteratorTokens)
                common_term.append(CompoundSymbol(token.value))
            ...
    return common_term
```

Для упрощения дальнейшнего написания семантического анализа синтаксическое дерево вывода будем хранить в виде структур классов. Обычно для каждого символа грамматики (терминального и нетерминального) придумывают свой класс.

Листинг 4. Структура абстрактного синтаксического дерева РЕФАЛ-5.

```
class AST(object):
    def __init__(self, functions):
        self.functions = [*default_functions, *functions]

```

```
class Function(ABC):
    def __init__(self, name, pos):
        self.name = name
        self.pos = pos

class Extern(Function):
    def __init__(self, name, pos=None):
        super(Extern, self).__init__(name, pos)
...
class Definition(Function):
    def __init__(self, name, pos, is_entry=False, sentences=None):
        super(Definition, self).__init__(name, pos)
        self.is_entry = is_entry
        self.sentences = sentences
...
class Expression(object):
    def __init__(self, terms):
        self.terms = terms

class Term(ABC):
    def __init__(self, value):
        self.value = value
...
class Variable(Term):
    def __init__(self, value, type_variable, pos, index=-1,
    ↪ sentence_index=-1):
        super(Variable, self).__init__(value)
        self.type_variable = type_variable
        self.index = index
        self.pos = pos
        self.sentence_index = sentence_index
```

4 Реализация семантического анализа. Разработка и реализация алгоритмов вывода и проверки типов

Семантический анализ — это фаза компиляции, выполняющая проверку синтаксического дерева на соответствие его компонентов контекстным ограничениям. Под контекстными ограничениями мы будем понимать такие вещи, как правила видимости идентификаторов, проверку типов выражений и т.п.

Определим основные проверки семантическего анализа в тексте на Рефале-5:

- Все имена функций после < или встроенные, или описанные в текущем файле, или объявленные как $EXTERN.
- Для каждой переменной в результатном выражении должно быть её вхождение в образцовом выражении в предшествующей части предложения. Т.е. при чтении предложения образцовые части пополняют область видимости новыми именами переменных (включая тип, s.Var и t.Var — две разные переменные), в результатных частях могут быть переменные только из области видимости.
- Функция в исходном тексте должна быть определена один раз — или как функция, или как $EXTERN.

В тексте спецификации типов семантический анализатор должен только проверять, что имена функций не повторяются — нет двух разных определений типов для одной функции. Больше ничего проверять не нужно.

В паре файлов Рефал-5 и спецификация типов семантический анализ должен проверять, что все функции, для которых заданы спецификации типов, определены в исходном файле.

Приведем пример реализации семантического анализа на Python.

Листинг 5. Семантический анализ языка РЕФАЛ-5

```
    def semantics(self):
        names = set()
        for function in self.ast.functions:
            if function.name in names:
                sys.stderr.write("Error. Function %s already defined\n" %
                ↪  function.name)
                self.isError = True
            else:
                names.add(function.name)

        if not self.isError:
            for function in self.ast.functions:
                if isinstance(function, Definition):
                    for sentence in function.sentences:
                        for term in sentence.result.terms:
                            if isinstance(term, CallBrackets):
                                if term.value not in names:
                                    sys.stderr.write("Error. Function %s
                                    ↪  isn't defined" % term.value)
                                    self.isError = True
        if not self.isError:
            for function in self.ast.functions:
                if isinstance(function, Definition):
                    self.semantics_variable(function.sentences)
```

Для описания алгоритма вывода типа функции в РЕФАЛ-5 требуется ввести еще одно дополнительное определение:

Сложность образца — числовая характеристика образца, определяемая по формуле:

$$C(P) = n_t + 2n_s + 3n_X + 3n_{()} - n_e + 1, \tag{5}$$

где n_t, n_s, n_e — количество, соответственно, t-, s-, e- переменных, $n_{()}$ — количество пар структурных скобок, n_X — количество атомов.

Опишем алгоритм вывода типов для базисного РЕФАЛа.

1. Выполняем сквозную нумерацию всех предложений. Ко всем индексам переменных приписываем номер предложения.

2. Для каждой правой части строится набор уравнений. Извлекается один из вызовов функции, например F. Терм конкретизации заменяется на out(F), аргумент сопоставляется с in(F). Например, для правой части:

$$EA\ <F\ EB\ <G\ EC>\ ED\ <H\ EF>\ EG>\ EH \quad (6)$$

строится система уравнений:

$$\begin{cases} EB\ out(G)\ ED\ out(H)\ EG : in(F) \\ EC : in(G) \\ EF : in(H) \end{cases} \quad (7)$$

3. Начальным форматом всех функций полагаем $in(F) = out(F) = \perp$, где $\perp$ - некоторое специальное значение (функция которая не возвращает никакого значения). Для $EXTERN - форматы беруется из подсказок (для встроенных функций — тоже).

4. Обозначим $Pat(f, i)$ — i-ая левая часть функции f, $Res(f, i)$ — i-я правая часть f, $\overline{Res}(f, i)$ — i-я правая часть f с заменой вызовов функций на out(g).

5. Пытаемся решить систему уравнений. Если у уравнения нет решений — сообщить об ошибке. Если система имеет единственное решение — берём для переменных соотвтествующее значение. Если несколько, то тогда берём первое попавшееся.

6. Уточняем форматы. Если у функции подсказки нет, строим обобщение:

$$\begin{cases} in(f) = Gen(Pat(f, i)) \\ out(f) = Gen(\overline{Res}(f, i)) \end{cases} \quad (8)$$

с подстановками значений переменных. Если у функции есть подсказка:

$$\begin{cases} in(f) = Gen(Pat(f, i), helpin(f)) \\ out(f) = Gen(\overline{Res}(f, i), helpout(f)) \end{cases} \quad (9)$$

7. Повторяем решение системы уравнений до тех пор, пока in(f) и out(f) не перестанут меняться.

Разберём более подробно про значение $\bot$.

Основные свойства:

- P - образцовое выражение, существует подстановка вида: $e.X \to \bot \Rightarrow P = \bot$
- $P\bot = \bot P = (\bot) = \bot$ (конкатенация)
- Сложность $\bot$=∞
- $Gen(P, \bot) = Gen(\bot, P) = P$

Первые два свойства похожи на свойства 0 при умножении ($X * 0 = 0 * X = 0$), а последние — свойства 0 при сложении ($X + 0 = 0 + X = 0$).

Полный алгоритм обобщённого сопоставления описан в работах Турчина [2]. Но неполный алгоритм является упрощением полного — в нём не надо рассматривать повторные s-переменные, которые дают «чужие» переменные и расщепления e-переменных, а также введением t-переменным. Простыми словами алгоритм можно описать так.

Вот есть сопоставление выражения общего вида E (которое может включать даже вызовы функций) и жёсткого выражения He:
$E \ : \ He$

Жёсткое выражение имеет вид $Ht_1^{`}...Ht_N^{`}e.XHt_M^{``}...Ht_1^{``}$,
либо $Ht_1...Ht_N$, где Ht — жёсткие термы, e.X — e-переменная. Т.е. надо рассмотреть четыре случая:

- жёсткое выражение начинается на жёсткий терм,
- жёсткое выражение кончается на жёсткий терм,

- жёсткое выражение состоит из одной e-переменной,
- жёсткое выражение пустое.

Жёсткое выражение имеет вид $Ht\ He`$, где Ht — жёсткий терм.

Если сопоставляемое выражение имеет вид $T\ E`$, где T — некий терм (символ, выражение в скобках, переменные s или t), то выполняем сопоставления соответственно $T\ :\ Ht$ и $E`\ :\ He`$. Иначе, если выражение E начинается на e-переменную или на вызов функции, имеем неудачу сопоставления.

Жёсткое выражение имеет вид $He`Ht$, где Ht — жёсткий терм, значит сопоставляемое выражение тоже должно заканчиваться на терм.

Жёсткое выражение состоит из e-переменной e.X, поэтому присваиваем этой переменной сопоставляемое выражение $E \leftarrow e.X$. Всегда успешно.

Жёсткое выражение пустое, поэтому если сопоставляемое выражение пустое, то сопоставление успешно, иначе неудача.

Как сопоставляются термы $(T : Ht)$?

Если Ht является t-переменной t.X, то строим присваивание $T \leftarrow t.X$. Если Ht является s-переменной, то T может быть только символом, s-переменной. В случае, если T — скобочный терм (Brackets), имеем неудачу сопоставления. Если Ht является символом (идентификатором, именем функции, числом или литерой), то T может быть только той же литерой. Иначе неудача. Если Ht является выражением в круглых скобках $(He`)$, то T может быть только выражением в круглых скобках $(E`)$. Соответственно, сопоставление выполняется рекурсивно $E` : He`$.

Если уравнение имеет вид $e.X\ E\ :\ Ht\ He$, где Ht — не e-переменная, то решаем две системы. В первом делаем подстановку $e.X \rightarrow \epsilon$ (и уравнение обращается в $E\ :\ Ht\ He$), во втором — $e.X \rightarrow t.i\ e.j$ (т.е. уравнение обращается в $t.i\ e.j\ E\ :\ Ht\ He$, откуда

$t.i \ : \ Ht$ и $e.j \ E \ : \ He$). Здесь t.i и e.j — переменные с новыми индексами.

Случай $E \ e.X \ : \ He \ Ht$ решается аналогично предыдущему (две системы с подстановками $e.X \to \epsilon$ и $e.X \to e.it.j$).

Остались случаи $\epsilon : He$, $E : \epsilon$ и $E \ : \ e.X$.

Для уравнения $\epsilon : He$ возможны три варианта. $\epsilon : \epsilon$ — тождество, тогда отбрасываем. $\epsilon : e.X$ — присваивание $\epsilon \leftarrow e.X$. $\epsilon : Ht \ He$ или $\epsilon : He \ Ht$ — решений нет.

В случае $E : \epsilon$ если E имеет вид $e.1 \ e.2 \ ...e.N$, то получаем подстановки $e.1 \to \epsilon, ..., e.N \to \epsilon$. В противном случае — решений нет.

В случае $E \ : \ e.X$ уравнение стираем, к решению добавляем присваивание $E \leftarrow e.X$. Но, поскольку нас присваивания переменным в правой части не интересуют (это нужно при прогонке), мы его даже не запоминаем, просто отбрасываем уравнение.

Приведем пример реализации алгоритма сопоставления на Python.

Листинг 6. Алгоритм сопоставления

```
def calculate_equation(self, equation, substitution, system):
    if eq.type_equation == EqType.Term:
        term_left = eq.left_part.terms[0]
        term_right = eq.right_part.terms[0]
        if isinstance(term_right, Variable) and isinstance(term_left,
        ↪  Variable) and term_left == term_right:
            return "Success", substitution, system, eq
        if is_symbol(term_left) and is_symbol(term_right) and term_left ==
        ↪  term_right:
            return "Success", substitution, system, eq
        elif isinstance(term_right, Variable) and term_right.type_variable ==
        ↪  Type.t:
            if term_right.index != -1:
                substitution.append(Substitution(Expression([term_left]),
                ↪  Expression([term_right])))
            return "Success", substitution, system, eq
        elif (isinstance(term_right, Variable) and term_right.type_variable
        ↪  == Type.s) or is_symbol(term_right):
            if (isinstance(term_right, Variable) and term_right.index != -1)
            ↪  or not isinstance(term_right, Variable):
                substitution.append(Substitution(Expression([term_left]),
                ↪  Expression([term_right])))
```

```
16              return "Success", substitution, system, eq
17          elif isinstance(term_left, StructuralBrackets) and
            ↪  isinstance(term_right, StructuralBrackets):
18              eq.type_equation = EqType.Expr
19              eq.left_part.terms = Expression(term_left.value).terms
20              eq.right_part.terms = Expression(term_right.value).terms
21              return self.calculate_equation(eq, substitution, system)
22          else:
23              return "Failure", [], system, eq
```

Далее разберём алгорим обобщения. Определяется образец общего вида, анализируя внешний вид отдельных образцов. К этому способу можно отнести стратегию, примененную в суперкомпиляторе SCP4 и стратегию построения ГСО, реализованную ранее в Рефале-5λ. Стратегию, примененную в Рефале-5λ, не используем из-за сложности реализации, поэтому используем стратегию в суперкомпиляторе SCP4. Данный подход описан в книге А. П. Немытых «Суперкомпилятор SCP4: Общая структура». Если у нас несколько (жёстких) образцов, то смотрим на левый и правый край каждого из них:

- если все левые края описывают термы (т.е. являются символами, скобками, s- или t-переменными) и все правые края описывают термы:
  - обобщаются все первые термы, обобщаются все последние термы,
  - выбирается, с какой стороны сложность больше,
  - если больше слева — выписываем обобщение левых термов как очередной слева терм обобщения, у всех образцов срезается первый терм,
  - если справа — симметрично;
- если слева у хотя бы одного из образцов есть e-переменная, а справа все края описывают термы:
  - обобщаем все последние термы,

– результат обобщения пишем как правый край результата,

– обрезаем у всех образцов правый терм;

- если все левые края описывают термы, а справа хотя бы у одного из образцов есть e-переменная:

  – симметрично предыдущему случаю;

- если слева есть хотя бы у одного образца e-переменная и справа есть хотя бы одна e-переменная:

  – результат обобщения — e;

- если все образцы пустые:

  – результат обобщения — ϵ;

- если есть хотя бы один пустой:

  – результат обобщения — ϵ.

Обобщения пары термов по таблице:

P_1/P_2	X	$Y \neq X$	s	t	(P_1)
X	X	s	s	t	t
s	s	s	s	t	t
t	t	t	t	t	t
(P_2)	t	t	t	t	$(GCG(P_1, P_2))$

Таблица 1. Обобщение термов.

Приведем пример реализации алгоритма обобщения термов на Python.

Листинг 7. Алгоритм обобщения

```
1  def generalization_term_rec(self, term_left, term_right):
2      if is_symbol(term_left) and is_symbol(term_right):
3          if term_left == term_right:
```

```
            return term_left
        else:
            index = generate_index()
            return Variable("generated%d".format(index), Type.s, None, index)
    if (isinstance(term_left, Variable) and term_left.type_variable ==
    ↪  Type.s) and is_symbol(term_right):
        return term_left
    if (isinstance(term_right, Variable) and term_right.type_variable ==
    ↪  Type.s) and is_symbol(term_left):
        return term_right
    if (isinstance(term_right, Variable) and term_right.type_variable ==
    ↪  Type.s) and \
            (isinstance(term_left, Variable) and term_left.type_variable ==
            ↪  Type.s):
        return term_left
    if isinstance(term_left, Variable) and term_left.type_variable == Type.t:
        return term_left
    if isinstance(term_right, Variable) and term_right.type_variable ==
    ↪  Type.t:
        return term_right
    if isinstance(term_left, StructuralBrackets) and isinstance(term_right,
    ↪  StructuralBrackets):
        result_terms = self.generalization([Expression(term_left.value),
        ↪  Expression(term_right.value)]).terms
        if result_terms == [SpecialVariable("@",SpecialType.none)]:
            return SpecialVariable("@",SpecialType.none)
        else:
            return StructuralBrackets(result_terms)
    if isinstance(term_left, StructuralBrackets):
        index = generate_index()
        return Variable("generated%d".format(index), Type.t, None, index)
    if isinstance(term_right, StructuralBrackets):
        index = generate_index()
        return Variable("generated%d".format(index), Type.t, None, index)
```

Рассмотрим на примере данный алгоритм вывода типов функций на РЕФАЛ-5.

Листинг 8. Пример функции на РЕФАЛ-5

```
F {
    ((((s.X)))) = s.X;
    (e.X) = <F e.X>;
}
```

Предположим, $in(F) = out(F) = \bot$. На первой итерации получается уравнение вида $e.X : in(F)$, следовательно с заменой $e.X : \bot$, а значит в итоге выводится подстановка вида $e.X \to \bot$

Теперь формат образца имеет вид при обобщении:

$$\begin{cases} ((((s)))) \\ (\bot) \equiv \bot \end{cases}, \tag{10}$$

а значит получается $in(F) = ((((s))))$ Формат результата:

$$\begin{cases} s \\ \bot \end{cases}, \tag{11}$$

а значит получается $out(F) = s$

На второй итерации $e.X : ((((s))))$, а значит в итоге выводится подстановка вида $e.X \to ((((s))))$

Теперь при обобщении формат образца имеет вид:

$$\begin{cases} ((((s)))) \\ (((((s))))) \end{cases}, \tag{12}$$

а значит получается $in(F) = ((((t))))$ Формат результата:

$$\begin{cases} s \\ s \end{cases}, \tag{13}$$

а значит получается $out(F) = s$

Третья итерация — неподвижная точка. Итоговый формат функции: $F((((t)))) = s$;

У Романенко, как и у нас, вводится полурешётка — множество с вершиной и дном. На каждой итерации типы функций уточняются. Также описано, что всем функциям, кроме первой типы параметров задаются как «дно», первой функции — «верхушка», any. Если параметр функции в неподвижной точке остался дном, то эта функция никогда не вызывается. Тип выводится не от семантики самой функции, а только от её использования. Условно, если функция всегда получает вторым параметром список из трёх элементов, а третьим — константу (одну и ту же во всех вызовах), то второй параметр можно заменить тремя параметрами, а третий вообще выкинуть.

Такой повышатель местности возник в контексте частичных вычислений — в остаточной программе оставалась функция из исходной программы, но она всегда вызывалась с одним и тем же аргументом по структуре и эту структуру разбирала. Поэтому логично было разделить параметр на несколько. [3]

5 Тестирование

Проверим работоспособность на разных программах РЕФАЛ-5 и также фрагментах кода компилятора РЕФАЛ-05.
Листинг 9. Пример программы на РЕФАЛ-5

```
1  F {
2    e.X = <G e.X> <H e.X>;
3  }
4
5  G { s.Y e.Z = s.Y }
6  H { e.Y (e.Z) = e.Z }
```

Получаем систему:

$$\begin{cases} e.X \ : \ s.1 \ e.2 \\ e.X \ : \ e.3 \ (e.4) \end{cases} \tag{14}$$

Первое уравнение имеет вид $e.X \ E \ : \ Ht \ He$, строим две подстановки: $e.X \to \epsilon$ и $e.X \to t.5 \ e.6$:

При $e.X \to \epsilon$:

$$\begin{cases} \epsilon : s.1 \ e.2 \\ \epsilon : e.3 \ (e.4) \end{cases}, \tag{15}$$

а при $e.X \to t.5 \ e.6$

$$\begin{cases} t.5 \ e.6 \ : \ s.1 \ e.2 \\ t.5 \ e.6 \ : \ e.3 \ (e.4) \end{cases} \tag{16}$$

Первая система несовместна, поскольку оба уравнения решений не имеют. Отбрасываем её. Для первого предложения второй системы есть правило $T \ E \ : \ Ht \ He$:

$$\begin{cases} t.5 : s.1 \\ e.6 : e.2 \\ t.5 \ e.6 \ : \ e.3 \ (e.4) \end{cases} \tag{17}$$

Первое уравнение даёт подстановку $t.5 \to s.7$, подставляем её:

$$\begin{cases} s.7 \ : \ s.1 \\ e.6 \ : \ e.2 \\ s.7 \ e.6 \ : \ e.3 \ (e.4) \end{cases} \tag{18}$$

Присваивания нам не нужны, поскольку с ними ничего не делаем, их отбрасываем. Первые два уравнения разрешаются в присваивания:

$$\begin{cases} s.7 \ e.6 \ : e.3 \ (e.4) \end{cases} \tag{19}$$

Последнее (и единственное) уравнение имеет вид $E \ e.6 \ : \ He \ Ht$, рассматриваем две подстановки: $e.6 \to \epsilon$ и $e.6 \to e.8 \ t.9$:

При $e.6 \to \epsilon$:

$$\begin{cases} s.7 : e.3(e.4) \end{cases} , \tag{20}$$

а при $e.6 \to e.8 \ t.9$:

$$\begin{cases} s.7 \ e.8 \ t.9 \ : \ e.3 \ (e.4) \end{cases} \tag{21}$$

Уравнения обоих систем имеют вид $E \ T \ : \ He \ Ht$, разделяем каждое на два $T : Ht$ и $E : He$:

При $e.6 \to \epsilon$:

$$\begin{cases} s.7 : (e.4) \\ \epsilon : e.3 \end{cases} , \tag{22}$$

а при $e.6 \to e.8 \ t.9$:

$$\begin{cases} t.9 : (e.4) \\ s.7 \ e.8 : e.3 \end{cases} \tag{23}$$

Первая система несовместна, поскольку уравнение $s.7 : (e.4)$ не имеет решений, отбрасываем систему. Первое уравнение второй системы даёт сужение t.9 $\to$ (e.10), подставляем его:

$$\begin{cases} (e.10) : (e.4) \\ s.7\ e.8\ :\ e.3 \end{cases} \tag{24}$$

Первое уравнение преобразуем по правилу $(E) : (He) \to E : He$:

$$\begin{cases} e.10 : e.4 \\ s.7\ e.8\ :\ e.3 \end{cases} \tag{25}$$

Два последних уравнения всегда успешны, разрешаются в присваивания, а поскольку присваивания нам не нужны, мы их просто отбрасываем. Получаем решение:

$$\begin{bmatrix} e.X \to t.5\ e.6 \\ \quad t.5 \to s.7 \\ e.6 \to e.8\ t.9 \\ t.9 \to (e.10) \end{bmatrix}$$

Если эти подстановки мы последовательно применим к левой части (которая имеет вид e.X), получим формат левой части $s.7\ e.8\ (e.10)$

Итоговый формат функции: $F\ s\ e\ (e) = s\ e$

А вот теперь рассмотрим функцию с ошибкой.

Листинг 10. Пример ошибочной программы на РЕФАЛ-5

```
1  F {
2    A A A A e.X = <F A A e.X>;
3    A = A;
4  }
```

На первой итерации $F\bot = \bot$. Для первых двух предложений находим, что $\begin{bmatrix} e.X \to \bot \\ e.Y \to \bot \end{bmatrix}$, поэтому они не участвуют в выводе формата.

Выводим формат по последнему предложению, получается A = A. На второй итерации решаем первое уравнение: $A\ A\ e.X\ :\ A$

Решений нет! Потому что первое уравнение несовместно. Оно же является неподвижной точкой. На стадии верификации первое уравнение не будет иметь решений — верификатор обнаружит ошибку в функции.

Теперь проверим работоспособность верификатора на «боевых» программах. На файле Parser.ref [10] верификатор работал около 2 минут, верно вычислил форматы почти всем функциям. Получается, что из 29 функций неточно вычислен формат только для трёх! Ещё у многих функций формат заканчивается на ... t, потому что они последним термом принимают и возвращают список ошибок. В идеале он тоже должен быть (ErrorList e). Неправильно выведенных форматов нет. Есть только неточный вывод, то есть формат получился более общим, чем должен быть.

6 Заключение

В рамках данной курсовой работы был изучен язык Рефал-5. А также были реализованы алгоритмы сопоставления, обобщения подстановок, а также разработан и реалиизован алгоритм вывода типа функций. Требования, предъявленные к разрабатываемому алгоритму, были выполнены. Данный верификатор был протестирован на более 30 программ, написанных на Базисном РЕФАЛе. При этом в дальнейшем планируется модификация верификатора для вывода более точных типов функций. Данный верификатор можно использовать в реальных программах.

Список литературы

[1] Документация языка Рефал-5λ, `http://github.com/bmstu-iu9/refal-5-lambda`

[2] Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ. В.Ф. Турчин, Киев-Алушта, 1972.

[3] Arity Raiser and its Use in Program Specialization. *3rd European Symposium on Programming* S.A. Romanenko, London, May 15 — 18, 1990.

[4] Суперкомпилятор SCP4: Общая структура А.П. Немытых, Издательство ЛКИ, 2007.

[5] Алгоритмический язык рекурсивных функций (РЕФАЛ) В.Ф. Турчин, ИПМ АН СССР, 1968.

[6] Язык программирования Рефал Плюс *Курс лекций. Учебное пособие для студентов университета города Переславля.* Р. Гурин, С. Романенко, Университет города Переславля» им.А.К.Айламазяна: 2006.

[7] Эквивалентные преобразования программ на РЕФАЛе *Труды ЦНИПИАСС "Автоматизированная система управления строительством", выпуск 6* В.Ф. Турчин, 1974.

[8] Представление объектных выражений массивами при реализации языка Рефал. С.М.Абрамов, С.А.Романенко. М.:ИПМ им.М.В.Келдыша АН СССР, 1988, препринт N 186.

[9] Реализация Рефал-компилятора. Представление данных и язык сборки. Скоробогатов С. Ю. Москва, 2008.

[10] Исходный код компилятора РЕФАЛ-05 версии 3.1, `https://github.com/Mazdaywik/Refal-05/tree/3.1`