ISBN 5-94052-066-0 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ. ПЕРЕСЛАВЛЬ-ЗАЛЕССКИЙ, 2004

УДК 519.68

С. М. Абрамов, А. Ю. Орлов, Л. В. Парменова,
С. М. Пономарева, А. Ф. Слепухин

Новый подход к реализации системы программирования Рефал Плюс

Аннотация. В работе описывается новый подход к реализации языка программирования Рефал Плюс на основе массивного представления объектных выражений и прямой компиляции в императивные языки. Даны описания разработанных формализмов, структуры библиотеки поддержки периода исполнения (run-time support library), приведены данные о сравнительном тестировании разработанного ПО и других реализаций языка Рефал Плюс.

Ключевые слова и фразы: Рефал, Рефал Плюс, компиляция, библиотека поддержки периода исполнения, система программирования.

1. Введение

Язык программирования Рефал был разработан в нашей стране в середине шестидесятых годов прошлого столетия [13]. Классическую авторскую характеристику языка процитируем из работы В. Ф. Турчина [17]:

> **РЕФАЛ** (**РЕ**курсивных **Ф**ункций **АЛ**горитмический язык) — это функциональный язык программирования, ориентированный на так называемые «символьные преобразования»: обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом. Функциональные языки программирования в настоящее время пользуются вполне заслуженной популярностью. Один из старейших членов этого семейства (впервые реализован в 1968 году в России, где широко используется и поныне), РЕФАЛ соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ.

Работа выполнена при поддержке Суперкомпьютерной программы «СКИФ» Союзного государства и Российского Фонда Фундаментальных Исследований, проект № 02–07–90126.

За годы своего существования и развития Рефал, среди прочего, использовался в следующих областях:

- *Манипуляции над программами.* На Рефале реализовано большое число различных макрогенераторов, трансляторов, интерпретаторов, систем анализа программ и их преобразования (в том числе средства метавычислений над программами).
- *Искусственный интеллект.* Рефал использовался для разработок систем обработки текстов на естественном языке и доказательства теорем.
- *Компьютерная алгебра.* Здесь речь идет и о разработанных на Рефале программах для проведения вычислений с абсолютной точностью (например, вычисления без округлений над целыми числами без ограничения на их длину, рациональными числами и квадратными корнями из них), и о многочисленных системах компьютерной алгебры.
- *Веб приложения, средства написания сценариев (scripts).* Ярким (но не единственным) образцом одного из таких применений Рефала является система, интегрирующая Рефал, PHP и редактор SciTE [18, 19].

За время становления и развития языка образовалось несколько диалектов языка Рефал, а реализации языка прошли эволюционный путь от интерпретирующих и полукомпилирующих[1] реализаций до систем, основанных на компиляции. В настоящее время наиболее популярными реализациями диалектов языка Рефал являются следующие системы:

- В системах программирования языка *Рефал-5* [17–19] используются классическое списковое представление [4] объектных выражений и полукомпилирующая схема [4, 9] реализации языка.
- В системе программирования *Рефал-6* [15] используется модифицированное списковое представление[2] объектных выражений и полукомпилирующая схема реализации языка.

[1] Рефал-программы преобразуются в так называемый язык сборки, который затем исполняется интерпретатором языка сборки [4, 9].

[2] Отличается от классического [4] более эффективным (более экономным по памяти) представлением термов, имеющих вид выражений в скобках — так называемые «подвешенные скобки со счетчиками ссылок».

- Предыдущая версия реализации *Рефала Плюс* основана на представлении объектных выражений массивами. В системе использована идея двухпроходной компиляции: рефал-программа преобразуется в виртуальный код [12], который затем отображается в исполняемый код целевого компьютера.
- В системе *Рефал-6/Java* [16] используется представление объектных выражений Java-массивами и прямая компиляцию рефал-программы в язык Java.

Данная работа посвящена новому подходу к реализации диалектов Рефала. При этом, целью являлось создание такой системы программирования, которая обладала бы следующими свойствами:

- *Использование представления объектных рефал-выражений в виде массивов.* Массивное представление рефал-выражений обладает рядом преимуществ перед другими способами. В том числе, оно позволяет эффективно в процессе выполнения рефал-программ выполнять операции вычисления длины рефал-выражения и построения подвыражения по длине и позиции расположения подвыражения в рефал-выражении. Эффективная реализация этих двух операций позволила разработать и использовать в системе *новый подход к эффективной реализации синтаксического отождествления* в языке Рефал [2].
- *Прямая компиляция рефал-программ в императивный язык* В системе реализована схема прямой компиляции рефал-программ в некий абстрактный императивный язык, который затем, отдельным проходом (*back-end* модулем), может быть легко отображен в любую целевую платформу. В настоящее время полностью реализован *back-end* модуль для языка, отображающий результат компиляции в C++, на стадии реализации — компиляция в T++.
- *Открытость, гибкость, продуманная модульность.* Система построена как набор отдельных модулей, с четко описанными интерфейсами между ними. Модули доступны в исходных текстах. Компилятор системы написан на Рефале Плюс, варианты библиотек поддержки периода исполнения (run-time support library, RSL) под различные платформы разрабатывается на языках высокого уровня (C++, Java,

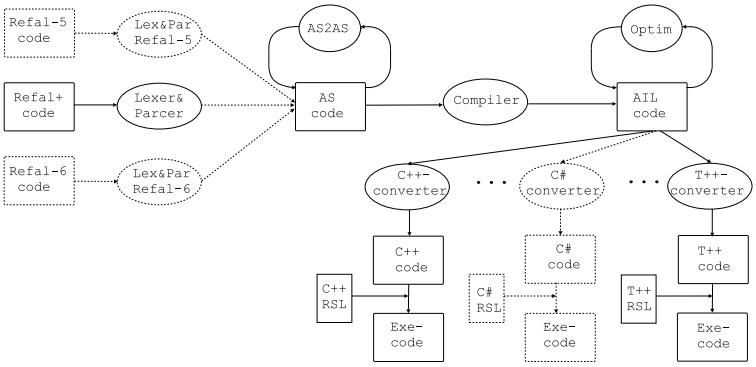

Рис. 1. Система программирования Рефал Плюс

C#, T++). Среди прочего, такой подход должен обеспечивать *высокий уровень переносимости системы.*

- *Поддержка возможных расширений.* За счет модульной структуры системы авторы старались поддержать легкость изменения ее функциональности, без необходимости модификации большей части кода и системы в целом. Речь идет о возможностях (см. рис. 1):
  - Добавления различных (в том числе оптимизирующих) преобразований рефал-программ в формате абстрактного синтаксиса («AS2AS»-преобразователи).
  - Добавления различных (в том числе оптимизирующих) преобразований результата компиляции в формате абстрактного императивного языка («Optim»).
  - Реализации в рамках системы поддержки разнообразных рефал-диалектов, как существующих (Рефал-2, -5. -6), так и будущих, за счет возможности добавления различных *front-end* («Lexer & Parser») модулей.
  - Реализации в рамках системы поддержки разнообразных целевых архитектур и платформ. Речь идет о возможности реализации разных версий *back-end* модулей и соответствующих библиотек периода исполнения (RSL) и за счет этого реализации системы как на базе привычных императивных языков (C++, Java, C#), так и на базе языков, поддерживающих параллельное

исполнение программ в мультипроцессорных системах (T++, OpenTS) [3].

В последующих разделах статьи будут рассмотрены более подробно различные аспекты реализации новой версии системы программирования Рефал Плюс.

2. Архитектура системы

На рис. 1 на стр. 376 показана общая архитектура системы программирования Рефал Плюс. В данной схеме сплошные линии отображают реализованные или находящиеся в стадии реализации модули системы, пунктир — то, что предполагается реализовать в перспективе.

Предложенная многопроходная схема реализации системы обеспечивает относительно простое расширение как набора входных языков другими диалектами Рефала, так и множества выходных императивных языков. Для реализации этого подхода в процессе компиляции рефал-программ выделены четыре основные этапа:

- Выполняемое блоком Lexer & Parcer преобразование рефал-программы в текст на промежуточном рефал-ориентированном абстрактном языке, называемом *Абстрактный синтаксис* (AS-code). Абстрактный синтаксис включает элементы, адекватно отображающие особенности всех диалектов Рефала, и подробно рассмотрен в разделе 3.
- Преобразование AS-кода в программу на *Абстрактном императивном языке* (AIL-code) с помощью модуля Compiler. Описание синтаксиса абстрактного императивного языка дано в разделе 4.
- Генерация модулем Converter выходного текста на императивном языке. Правила конвертирования, использованные для C++, сформулированы в разделе 5.
- Генерация исполняемого модуля на основе библиотеки поддержки периода исполнения, структура RSL для C++ описана в разделе 6.

Рассмотрим более подробно основные этапы компиляции.

3. Этап компиляции «Рефал Плюс — Абстрактный синтаксис»

При разработке *Абстрактного синтаксиса* (далее AS) главной была задача обеспечить возможность отображения в его конструкциях всех особенностей существующих диалектов Рефала, нивелируя различия, связанные с несовпадением семантики одних и тех же символов, которое наблюдается при сравнении, например, Рефала Плюс и Рефала-6. В результате было разработано следующее «рефальское» описание AS-языка:

```
t.Syntax          ::= t.Module | t.Interface
t.Module          ::= (MODULE t.ModName e.ModBody)
t.Interface       ::= (INTERFACE t.InterfaceName
                  e.InterfaceBody)
t.ModName         ::= t.Name
t.InterfaceName   ::= t.Name
e.ModBody         ::= t.Item | e.ModuleBody t.Item
t.Item            ::= t.Function | t.Object | t.Const
                  | t.Initializer | t.PragmaTrace
t.Function        ::= (s.IntLinkage s.FunctionTag
                  t.Pragma t.FunctionName t.InputParam
                  t.OutputParam e.Sentence)
                  | (s.ExtLinkage s.FunctionTag
                  t.Pragma t.FunctionName t.InputParam
                  t.OutputParam)
s.Linkage         ::= s.IntLinkage | s.ExtLinkage
s.IntLinkage      ::= EXPORT | LOCAL
s.ExtLinkage      ::= IMPORT
s.FunctionTag     ::= FUNC | FUNC?
t.FunctionName    ::= t.Name
t.InputParam      ::= (e.ListParam)
t.OutputParam     ::= (e.ListParam)
e.ListParam       ::= /*Empty*/ | e.ListParam t.Param
t.Param           ::= (SVAR) | (EVAR) | (TVAR)
                  | (VVAR) | (PAREN e.ListParam)
e.Sentence        ::= /*Empty*/
                  | t.Statement e.Sentence
t.Statement       ::= t.Action | t.Operator | t.Branch
                  | t.ComplexOperator | t.Pragma
```

```
                    | t.Format
t.Format            ::= (FORMAT t.Pragma e.HardExpr)
e.HardExpr          ::= e.HardTerms
                    | e.HardTerms t.EVar e.HardTerms
                    | e.HardTerms t.VVar e.HardTerms
e.HardTerms         ::= /*Empty*/
                    | t.HardTerm e.HardTerms
t.HardTerm          ::= t.Symbol | (PAREN e.HardExpr)
                    | t.SVar | t.TVar
t.Operator          ::= (CUT t.Pragma) | (ERROR t.Pragma)
                    | (STAKE t.Pragma) | NOFAIL
                    | (FAIL t.Pragma) | (CUTALL t.Pragma)
t.ComplexOperator   ::= (NOT t.Branch)
                    | (ITER t.IterBody t.IterVars
                    t.IterCond)
                    | (TRY t.TrySentence
                    e.Nofail t.CatchSentence)
t.IterBody          ::= t.branch
t.IterVars          ::= t.Format
t.IterCond          ::= t.branch
t.TrySentence       ::= t.branch
t.CatchSentence     ::= t.block | t.branch
t.Action            ::= t.Pattern | t.Result | t.Format
                    | t.Block
t.Block             ::= (BLOCK t.Pragma e.Branches)
                    | (BLOCK? t.Pragma e.Branches)
e.Branches          ::= /*Empty*/ | t.Branch e.Branches
t.Branch            ::= (BRANCH t.Pragma e.Sentence)
t.Pattern           ::= (LEFT t.Pragma e.PatternExpr)
                    | (RIGHT t.Pragma e.PatternExpr)
e.PatternExpr       ::= /*Empty*/
                    | t.PatternTerm  e.PatternExpr
t.PatternTerm       ::= t.Symbol | (PAREN e.PatternExpr)
                    | t.Variable
t.Variable          ::= t.EVar | t.VVar | t.TVar | t.Svar
t.EVar              ::= (EVAR t.Pragma t.Name)
t.VVar              ::= (VVAR t.Pragma t.Name)
t.TVar              ::= (TVAR t.Pragma t.Name)
t.SVar              ::= (SVAR t.Pragma t.Name)
```

```
t.Name              ::= (e.QualifiedName)
e.QualifiedName     ::= s.Word | e.QualifiedName  s.Word
t.Result            ::= (RESULT t.Pragma e.ResultExpr)
e.ResultExpr        ::= /*Empty*/
                    | t.ResultTerm e.ResultExpr
t.ResultTerm        ::= t.Symbol | t.Variable | t.block
                    | (PAREN e.ResultExpr)
                    | (CALL t.Pragma t.FunctionName
                    e.ResultExpr)
t.Object            ::= (s.Linkage s.ObjectType t.Pragma
                    t.ObjectName)
s.ObjectType        ::= TABLE | BOX | STRING | VECTOR
                    | CHANNEL
t.ObjectName        ::= t.Name
t.Initializer       ::= (EXEC e.ResultExpr)
t.Pragma            ::=
                    (PRAGMA t.PragmaFile t.PragmaLine)
t.PragmaFile        ::= (FILE e.FileName)
e.FileName          ::= s.Char | s.Char e.FileName
t.PragmaLine        ::= (LINE s.Line s.Column)
s.Line              ::= s.Number
s.Column            ::= s.Number
t.PragmaTrace       ::= (TRACE e.TraceNames)
e.TraceNames        ::= /*Empty*/
                    | t.TraceName e.TraceNames
t.TraceName         ::= t.Name
t.Const             ::= (s.IntLinkage CONST t.Pragma
                    t.ConstName e.ConstExpr)
                    | (s.ExtLinkage CONST t.Pragma
                    t.ConstName)
t.ConstName         ::= t.Name
e.ConstExpr         ::= /*Empty*/
                    | t.ConstTerm e.ConstExpr
t.ConstTerm         ::= t.Symbol | (PAREN e.ConstExpr)
t.Symbol            ::= s.StaticSymbol | (REF t.Name)
s.StaticSymbol      ::= s.Char | s.Word | s.Number
s.Char              ::= символ-литера (в смысле Рефал+)
s.Word              ::= символ-слово (в смысле Рефал+)
s.Number            ::= символ-число (в смысле Рефал+)
```

4. Этап компиляции «Абстрактный синтаксис — Абстрактный императивный язык»

Как говорилось выше, одна из важных особенностей данной реализации — использование на этапе компиляции в модуле Compiler преимуществ массивного представления данных для эффективной реализации синтаксического отождествления [2]. При этом в коде на императивном языке наиболее часто используются операции вычисления длины рефал-выражения и построения подвыражения по длине и позиции расположения подвыражения в рефал-выражении, т.е. действия, связанные с двумя типами данных — целыми (обозначаемые int) и классом Expr, реализующим рефал-выражения. Как следствие этого был сформирован набор конструкций *Абстрактного императивного языка* (AIL), который включает наряду с общими элементами императивных языков (вызов функций с параметрами, цикл, условный оператор, операторы перехода, арифметические операции и др.) набор методов класса Expr, обращение к которым обусловлено особенностями алгоритма компиляции.

Ниже приводится описание синтаксиса *Абстрактного императивного языка*, где в качестве комментариев даны пояснения некоторых конструкций AIL.

```
e.ASAIL       ::= /*empty*/ | t.Func e.ASAIL
t.Func        ::= (FUNC t.QualifName (e.param)
                    (e.param) e.body)
// Определение  функции: t.QualifName - имя функции,
// e.param - входные и выходные параметры, e.body -
// тело функции.
t.QualifName ::= (e.name)
e.name        ::= /*empty*/ | s.numb e.name
                | s.word e.name
e.param       ::= /*empty*/ | t.var-ref e.param
e.body        ::= /*empty*/ | t.statement e.body
t.statement   ::= t.call | t.tailcall | t.assign
                | t.if | t.for | t.continue | t.break
                |t.lsplit | t.rsplit | t.catch
                | t.try | t.drop | t.expr-obj | t.deref
                | t.subexpr | t.constexpr | t.blok-label
                | t.decl | t.decl-int | t.decl-func
                | t.decl-const | t.decl-obj | t.obj
```

```
                | t.error
                | RETURN
// Оператор успешного выхода из функции.
                | FATAL
// Оператор аварии, вырабатывающий исключение
// вида Unexpected fail.
                | RETFAIL
// Оператор неуспешного выхода из функции (fail).
t.call          ::= (CALL t.QualifName (e.args) (e.args))
// Вызов функции, где t.QualifName - имя функции,
// e.args - входные и выходные фактические
// параметры, передаваемые по значению.
e.args          :: = /*empty*/ | t.arg e.args
t.arg           ::= t.var | (e.expr)
t.var           ::=  t.var-ref | t.var-int
e.expr          ::=  e.expr-ref | e.expr-int
t.tailcall      ::= (TAILCALL t.QualifName (e.arg)(e.arg))
// <<Хвостовой>> вызов функции,
// где t.QualifName - имя функции,
// e.arg - входные и выходные фактические параметры,
// передаваемые по значению.
t.assign        ::= (ASSIGN t.var e.expr)
// Оператор присваивания, где t.var - имя переменной,
// e.expr - присваиваемое значение.
t.decl          ::= (DECL s.type t.var)
// Объявление переменной t.var типа s.type.
// Предполагается поддержка двух базовых типов:
// int (целые числа) и Expr (Рефал-выражения).
s.type          ::= "Expr" | "int"
t.decl-int      ::= (INT t.var e.expr-int)
// Объявление переменной t.var типа int
// и инициализация ее значением e.expr-int.
t.if            ::= (IF (e.cond) e.body)
// Условный оператор, где e.cond - условие,
// e.body - последовательность действий, выполняемых
// в случае его истинности.
e.cond          ::= /*empty*/ | t.cond-term e.cond
// Операнды t.cond-term, входящие в условие e.cond,
// соединяются AND.
```

```
t.cond-term  ::= t.call
             | (SYMBOL? e.expr-ref (e.pos))
// Проверка структуры элемента, занимающего в
// выражении e.expr-ref позицию e.pos;
// условие истинно, если этот элемент - символ.
             | (EQ e.expr-ref1 (e.expr-ref2) (e.pos))
// Проверка равенства выпажения e.expr-ref1
// подвыражению выражения e.expr-ref2, начинающемуся
// с позиции e.pos.
             | (TERM-EQ e.expr-ref1 (e.expr-ref2)
             (e.pos))
// Проверка равенства первого терма выражения
// e.expr-ref1 терму выражения e.expr-ref2,
// занимающему позицию e.pos.
             | (NOT e.cond)
// Логическое отрицание условия e.cond.
             | (CHECK-ITER e.expr-ref)
// Проверка успешности очередного шага итерации.
             | (INFIX s.op-bool e.args-log)
             | (INFIX s.op-eq e.args-int)
e.args-log   ::= /*empty*/ | (e.expr-log) e.args-log
e.pos        ::=e.expr-int
e.len        ::= e.expr-int
e.expr-log   ::= /*empty*/
             | t.cond-term e.expr-log
             | t.int-term e.expr-log
s.op-bool    ::= "&&" | "||"
s.op-eq      ::= "<" | ">" | "<=" | ">=" | "==" | "!="
t.for        ::= (FOR (e.cont-lab) (e.break-lab)
             (e.cond) (e.step) e.body)
// Цикл  FOR: e.cont-lab - empty или метка цикла
// (для использования в операторе CONTINUE),
// e.break-lab - empty или метка для оператора BREAK,
// e.cond - условие выполнения цикла,
// e.step  - оператор итерации, e.body - тело цикла.
e.cont-lab      ::= t.label | /*empty*/
e.break-lab  ::= t.label | /*empty*/
t.label      ::= t.QualifName
e.step       ::= (INC-ITER e.expr-ref)
```

```
                | (DEC-ITER e.expr-ref)
// Шаг цикла FOR - инкремент/декремент итератора
// переменной e.expr-ref, созданного ранее
// соответствующей функцией lsplit() или rsplit().
t.continue    ::= (CONTINUE t.label)
// Оператор перехода к следующей итерации цикла FOR,
// t.label - метка цикла.
t.break       ::= (BREAK t.label)
// Оператор выхода из цикла FOR,
// t.label - метка блока, содержащего цикл.
t.lsplit      ::= (LSPLIT e.expr-ref (e.min) t.l t.r)
t.rsplit      ::= (RSPLIT e.expr-ref (e.min) t.l t.r)
// Функции отщепления фрагментов Рефал-выражения
// (соответственно, слева и справа);
// e.expr-ref - исходное выражение;
// e.min - минимальная длина отщепляемого участка;
// t.l и t.r, соответственно, левая и правая части,
// на которые разделяется выражение.
t.l           ::= t.var-ref
t.r           ::= t.var-ref
e.min         ::= e.expr-int
t.try         ::= (TRY e.body)
// try-блок, где e.body - тело блока. Исключения из
// него перехватываются в соответствующем catch-блоке.
t.error       ::= (ERROR e.expr-ref)
// Оператор генерации исключения e.expr-ref.
t.catch       ::= (CATCH-ERROR e.body)
// Блок перехвата исключений, вырабатываемых
// try-блоком. Для любого try-блока существует только
// один catch-блок, условие которого всегда  истинно
// (вида Expr const& err), анализ  и обработка
// исключения проводится в теле блока e.body.
t.drop        ::= (DROP t.var-ref)
// Блок, сигнализирующий о возможности освобождения
// памяти, занимаемой объектом  t.var-ref,
// т.к. она далее не используется.
t.expr-obj    ::= (EXPR t.var-ref e.expr-ref)
// Конструктор объекта t.var-ref класса Expr
// со значением e.expr-ref.
```

```
t.deref       ::= (DEREF t.var-expr e.expr-ref (e.pos))
// Конструктор объекта t.var-ref класса Expr,
// значение которого - содержимое скобочного терма,
// занимающего в выражении e.expr-ref позицию e.pos.
t.subexpr     ::= (SUBEXPR t.var-ref e.expr-ref
              (e.pos) (e.len))
// Конструктор объекта t.var-ref класса Expr,
// который выделяет в выражении e.expr-ref,
// начиная с элемента e.pos, подвыражение длиной e.len.
e.pos         ::= e.expr-int
e.len         ::= e.expr-int
t.constexpr   ::= (CONSTEXPR s.linkage t.const-name
              (e.comment) e.const-expr)
// Конструктор константного объекта класса Expr.
s.linkage     ::= EXPORT | LOCAL
e.comment     ::= выражение (в смысле Рефал Плюс)
// e.comment - выражение в том виде, в котором оно
// встретилось в Рефал-программе.
t.const-name  ::= t.QualifName | (STATIC t.QualifName)
e.const-expr  ::= e.comp-expr | e.const-subexpr
e.comp-expr   ::= [empty] | t.const-term e.const-expr
// Термы, входящие в e.comp-expr, конкатенируются.
t.const-term  ::= s.symbol | (PAREN e.const-expr)
              | (REF t.QualifName)
              | (STATIC t.QualifName)
e.const-subexpr ::= (SUBEXPR t.const-name s.const-pos
              s.const-len)
e.const-pos   ::= s.numb
s.const-len   ::= s.numb
t.decl-obj    ::=
              (DECL-OBJ s.linkage s.tag t.QualifName)
// Объявление объектов, s.tag - тип объекта.
s.tag         ::= "Box" | "Table" | "Vector"
              | "Channel" | "String"
t.obj         ::= (OBJ s.linkage s.tag t.name)
// Конструктор произвольного объекта.
t.blok-label  ::= (LABEL (e.maybe-empty) e.body)
// Блок: e.body - тело блока,
// e.maybe-empty - метка или empty.
```

```
t.decl-func  ::= (DECL-FUNC  t.QualifName)
// Объявление функции с именем t.QualifName.
e.expr-ref   ::= /*empty*/ | t.term-ref e.expr-ref
// Термы в Рефал-выражении объединяются конкатенацией.
t.term-ref   :: = s.symbol | t.var-ref
             | (PAREN e.expr-ref)
// Оператор <<навешивания скобок>> для выражения
// e.expr-ref; результат типа Expr.
             | (DEREF e.expr-ref (e.pos))
// Извлечение содержимого скобочного терма,
// находящегося в e.expr-ref в позиции e.pos,
// результат типа Expr.
             | (SUBEXPR e.expr-ref (e.pos) (e.len))
// Выделение в выражении e.expr-ref подвыражения,
// начинающегося в e.pos и имеющего длину e.len;
// результат типа Expr.
e.expr-int   ::= /*empty*/ | t.int-term e.expr-int
// Термы в целочисленном выражении e.expr-int
// соединяются + (сложение); результат типа int.
t.int-term   ::= s.numb | t.var-int
             | (LENGTH e.expr-ref)
// Вычисление длины выражения e.expr-ref;
// результат типа int.
             | (MIN e.args-int) | (MAX e.args-int)
// Вычисление минимума (максимума) для аргументов типа
// int; результат типа int.
             | (INFIX s.op-int e.args-int)
// Арифметические операции (левоассоциативные) для
// аргументов типа int.
e.args-int   ::= /*empty*/ | (e.expr-int) e.args-int
s.op-int     ::= "+" | "-" | "*" | "/" | "%"
t.var-ref    ::= (STATIC (s.numb) )|(VAR t.QualifName)
             | (TVAR t.QualifName) |(VVAR t.QualifName)
             | (EVAR t.QualifName) |(SVAR t.QualifName)
t.var-int    ::= (VAR t.QualifName)
s.numb       ::= символ-число (в смысле Рефал Плюс)
s.word       ::= символ-слово (в смысле Рефал Плюс)
```

5. Этап компиляции «Абстрактный императивный язык — Си++»

5.1. Правила конвертирования. Основной тип объекта, над которым выполняются действия — это рефал-выражение, для его описания в C++-реализации определен класс Expr, методы работы которого позволяют эффективно выполнять действия, на которые ориентирован алгоритм, рассмотренный в [2]: оценка длины выражения, выбор терма и подвыражения, снятие скобок с выражения и т.д.

В модуле генерации программы на императивном языке выделяются правила, связанные с генерацией операторов и блоков, которым соответствуют стандартные средства императивных языков, и правила, предназначенные для генерации функций для работы с объектами класса Expr. Также отдельно выделены группы методов для вычисления значений целочисленных, логических (получаемых как промежуточные результаты) выражений и выражений типа Expr.

Рассмотрим использованные в модуле C++-Converter правила соответствия конструкций *Абстрактного императивного языка* выходному C++-коду. Этот набор правил отражает схему конвертирования, которая не требует существенных изменений при необходимости компиляции в другие объектно-ориентированные языки.

Будем преобразование AIL-выражения **X** обозначать конструкцией $[\![\,\mathbf{X}\,]\!]_{C++}$ и отделять от результата знаком $\Longrightarrow$.

5.1.1. *Генерация операторов и блоков.*

- Определение функции

$$
\begin{aligned}
&[\![\,(FUNC\ t.name\ (e.arg)\ (e.res)\ e.body)\,]\!]_{C++} \Longrightarrow \\
&\quad RF\_FUNC\ \ (\quad [\![\,t.name\,]\!]_{C++}\,, \\
&\qquad\qquad\qquad\quad (RF\_ARG\ [\![\,e.arg\,]\!]_{C++}\,), \\
&\qquad\qquad\qquad\quad (RF\_RES\ [\![\,e.res\,]\!]_{C++}\,) \\
&\qquad\qquad\quad) \\
&\qquad\qquad\quad [\![\,e.body\,]\!]_{C++} \\
&\quad RF\_END
\end{aligned}
$$

- Вызовы функции

$$
\begin{aligned}
&[\![\,(CALL\ t.name\ (e.arg_{in})\ (e.arg_{out}))\,]\!]_{C++} \Longrightarrow \\
&\quad RF\_CALL\ ([\![\,t.name\,]\!]_{C++}\,, \\
&\qquad\quad ([\![\,e.arg_{in}\,]\!]_{C++}\,),\ ([\![\,e.arg_{out}\,]\!]_{C++}\,)) \\
&[\![\,(TAILCALL\ t.name\ (e.arg_{in})\ (e.arg_{out}))\,]\!]_{C++} \Longrightarrow \\
&\quad RF\_TAILCALL\ ([\![\,t.name\,]\!]_{C++}\,,
\end{aligned}
$$

$$(\llbracket\, e.arg_{in} \,\rrbracket_{C++}),\ (\llbracket\, e.arg_{out} \,\rrbracket_{C++}))$$

- Оператор присваивания

$$\llbracket\, (ASSIGN\ t.var\ e.expr) \,\rrbracket_{C++} \Longrightarrow \llbracket\, t.var \,\rrbracket_{C++} = \llbracket\, e.expr \,\rrbracket_{C++}\ ;$$

- Объявление переменной

$$\llbracket\, (DECL\ s.type\ t.var) \,\rrbracket_{C++} \Longrightarrow \llbracket\, s.type \,\rrbracket_{C++}\ \ \llbracket\, t.var \,\rrbracket_{C++}\ ;$$

- Условный оператор

$$\begin{array}{l}\llbracket\, (IF\ (e.cond)\ e.body) \,\rrbracket_{C++} \Longrightarrow \\ \quad if\ (\llbracket\, e.cond \,\rrbracket_{C++}\,) \\ \quad \{ \\ \qquad \llbracket\, e.body \,\rrbracket_{C++} \\ \quad \}\end{array}$$

- Генерация помеченного блока

$$\begin{array}{l}\llbracket\, (LABEL\ (e.label)\ e.body\ ) \,\rrbracket_{C++} \Longrightarrow \\ \quad \{ \\ \qquad \llbracket\, e.body \,\rrbracket_{C++} \\ \quad \} \\ \quad \llbracket\, e.label \,\rrbracket_{C++} : \{\ \}\end{array}$$

- Цикл FOR

$$\begin{array}{l}\llbracket\, (FOR\ (e.c\text{-}lab)\ (e.b\text{-}lab)\ (e.cond) \\ \quad (INC\text{-}ITER\ t.var)\ e.body) \,\rrbracket_{C++} \Longrightarrow \\ \qquad for\ (\ ;\ \llbracket\, e.cond \,\rrbracket_{C++}\ ;\ iter(\llbracket\, t.var \,\rrbracket_{C++}\,)\text{++}) \\ \qquad \{ \\ \qquad\quad \llbracket\, e.body \,\rrbracket_{C++} \\ \qquad\quad \llbracket\, e.c\text{-}lab \,\rrbracket_{C++}\ : \{\ \} \\ \qquad \} \\ \qquad \llbracket\ e.b\text{-}lab \,\rrbracket_{C++} : \{\ \}\end{array}$$

- Оператор перехода к следующей итерации цикла for

$$\llbracket\, (CONTINUE\ t.label) \,\rrbracket_{C++} \Longrightarrow goto\ \llbracket\, t.label \,\rrbracket_{C++}\ ;$$

- Оператор выхода из цикла

$$\llbracket\, (BREAK\ t.label) \,\rrbracket_{C++} \Longrightarrow goto\ \llbracket\, t.label \,\rrbracket_{C++}\ ;$$

где t.label — метка блока, содержащего цикл.

- Функция отщепления слева фрагмента рефал-выражения

$[\![$ *(LSPLIT e.expr (e.min) t.l t.r)* $]\!]_{C++} \Longrightarrow$
lsplit($[\![$ *e.expr* $]\!]_{C++}$ *,* $[\![$ *e.min* $]\!]_{C++}$ *,* $[\![$ *t.l* $]\!]_{C++}$ *,*
$[\![$ *t.r* $]\!]_{C++}$ *);*

- Функция отщепления справа фрагмента рефал-выражения

$[\![$ *(RSPLIT e.expr (e.min) t.l t.r)* $]\!]_{C++} \Longrightarrow$
rsplit($[\![$ *e.expr* $]\!]_{C++}$ *,* $[\![$ *e.min* $]\!]_{C++}$ *,* $[\![$ *t.l* $]\!]_{C++}$ *,*
$[\![$ *t.r* $]\!]_{C++}$ *);*

- Оператор генерации исключения

$[\![$ *(ERROR e.expr)* $]\!]_{C++} \Longrightarrow$
error ($[\![$ *e.expr* $]\!]_{C++}$ *);*

- try-блок

$[\![$ *(TRY e.body)* $]\!]_{C++} \Longrightarrow$
trap
{
$[\![$ *e.body* $]\!]_{C++}$
}

- Блок перехвата исключений

$[\![$ *(CATCH-ERROR e.body)* $]\!]_{C++} \Longrightarrow$
with
{
$[\![$ *e.body* $]\!]_{C++}$
}

- Вызов метода освобождения памяти, занимаемой объектом t.var

$[\![$ *(DROP t.var)* $]\!]_{C++} \Longrightarrow$
$[\![$ *t.var* $]\!]_{C++}$ *.drop ();*

- Оператор успешного выхода из функции

$[\![$ *RETURN* $]\!]_{C++} \Longrightarrow$
return true;

- Оператор неуспешного выхода из функции

$[\![$ *RETFAIL* $]\!]_{C++} \Longrightarrow$
retfail;

- Оператор неуспешного выхода из функции

$[\![$ *FATAL* $]\!]_{C++} \Longrightarrow$
error ("Unexpected fail");

- Конструктор объекта типа Int

$$[\![\ (INT\ t.var\ e.expr\ )\ ]\!]_{C++} \Longrightarrow$$
$$\quad uintptr\_t\ [\![\ t.var\ ]\!]_{C++}\ =\ [\![\ e.expr\ ]\!]_{C++}\ ;$$

- Конструктор объекта класса Expr

$$[\![\ (EXPR\ t.var\ e.expr\ )\ ]\!]_{C++} \Longrightarrow$$
$$\quad Expr\ [\![\ t.var\ ]\!]_{C++}\ (\ [\![\ e.expr\ ]\!]_{C++}\ );$$

- Конструктор объекта класса Expr, значение которого — содержимое скобочного терма, занимающего в e.expr позицию e.pos

$$[\![\ (DEREF\ t.var\ e.expr\ (e.pos))\ ]\!]_{C++} \Longrightarrow$$
$$\quad Expr\ [\![\ t.var\ ]\!]_{C++}\ (\ [\![\ e.expr\ ]\!]_{C++}\ ,\ [\![\ e.pos\ ]\!]_{C++}\ );$$

- Конструктор объекта класса Expr, который является подвыражением выражения e.expr, начиная с элемента e.pos, длиной e.len

$$[\![\ (SUBEXPR\ t.v\ e.exp\ (e.pos)\ (e.len))\ ]\!]_{C++} \Longrightarrow$$
$$\quad Expr\ [\![\ t.v\ ]\!]_{C++}\ (\ [\![\ e.exp\ ]\!]_{C++}\ ,$$
$$\qquad\qquad [\![\ e.pos\ ]\!]_{C++}\ ,\ [\![\ e.len\ ]\!]_{C++}\ );$$

- Конструктор константного объекта класса Expr (термы, входящие в e.expr, объединены операцией конкатенации)

$$[\![\ (CONSTEXPR\ s.link\ t.name\ (e.comment)$$
$$\quad e.expr\ )\ ]\!]_{C++} \Longrightarrow$$
$$\qquad [\![\ s.link\ ]\!]_{C++}\ const\ Expr\ [\![\ t.name\ ]\!]_{C++}\ =$$
$$\qquad\quad [\![\ e.expr\ ]\!]_{C++}\ ;$$

- Объявление функции

$$[\![\ (DECL\text{-}FUNC\ t.name)\ ]\!]_{C++} \Longrightarrow$$
$$\quad RF\_DECL\ (\ [\![\ t.name\ ]\!]_{C++}\ );$$

- Объявление объекта типа Box, String, Vector, Table, Channel

$$[\![\ (DECL\text{-}OBJ\ t.name)\ ]\!]_{C++} \Longrightarrow$$
$$\quad extern\ const\ Expr\ [\![\ t.name\ ]\!]_{C++}\ ;$$

- Конструктор объекта типа Channel, String, Vector, Table, Box

$$[\![\ (OBJ\ s.linkage\ s.tag\ t.name)\ ]\!]_{C++} \Longrightarrow$$
$$\quad [\![\ s.linkage\ ]\!]_{C++}\ const\ Expr\ [\![\ t.name\ ]\!]_{C++}\ =$$
$$\qquad new\ rftype::[\![\ s.tag\ ]\!]_{C++}\ ();$$

5.1.2. *Генерация логических функций, результат которых — true или false.*

- Условие истинно, если элемент, занимающий в выражении e.expr позицию e.pos, является символом (в смысле Рефала Плюс)

$$[\![\ (SYMBOL?\ e.expr\ (e.pos))\ ]\!]_{C++} \Longrightarrow \\ [\![\ e.expr\ ]\!]_{C++}\ .symbol\_at(\ [\![\ e.pos\ ]\!]_{C++}\ )$$

- Условие истинно, если выражение e.1 равно подвыражению выражения e.2, начинающегося с позиции e.pos

$$[\![\ (EQ\ e.1\ (e.2)\ (e.pos))\ ]\!]_{C++} \Longrightarrow \\ [\![\ e.1\ ]\!]_{C++}\ .eq\ (\ [\![\ e.expr2\ ]\!]_{C++}\ ,\ [\![\ e.pos\ ]\!]_{C++}\ )$$

- Условие истинно, если первый терм выражения e.1 равен терму выражения e.2, который занимает позицию e.pos

$$[\![\ (TERM\text{-}EQ\ e.1\ (e.2)\ (e.pos))\ ]\!]_{C++} \Longrightarrow \\ [\![\ e.1\ ]\!]_{C++}\ .term\_eq\ (\ [\![\ e.2\ ]\!]_{C++}\ ,\ [\![\ e.pos\ ]\!]_{C++}\ )$$

- Логическое отрицание

$$[\![\ (NOT\ e.cond)\ ]\!]_{C++} \Longrightarrow \\ !\ [\![\ e.cond\ ]\!]_{C++}$$

- Логическая операция (левоассоциативная)

$$[\![\ (INFIX\ s.op\ (t.log_1\ \ldots\ t.log_n))\ ]\!]_{C++} \Longrightarrow \\ (\ [\![\ t.log_1\ ]\!]_{C++}\ s.op\ \ldots\ s.op\ [\![\ t.log_n\ ]\!]_{C++}\ )$$

5.1.3. *Генерация методов работы с объектами класса Expr, их результат — объект класса Expr.*

- Операция «навешивания скобок»

$$[\![\ (PAREN\ e.expr)\ ]\!]_{C++} \Longrightarrow \\ [\![\ e.expr\ ]\!]_{C++}\ ()$$

- Извлечение содержимого скобочного терма, занимающего в выражении e.expr позицию e.pos

$$[\![\ (DEREF\ e.expr\ (e.pos))\ ]\!]_{C++} \Longrightarrow \\ Expr\ (\ [\![\ e.expr\ ]\!]_{C++}\ ,\ [\![\ e.pos\ ]\!]_{C++}\ )$$

- Выделение в выражении e.expr подвыражения, начинающегося в e.pos и имеющего длину e.len

$$[\![\ (SUBEXPR\ e.expr\ (e.pos)(e.len))\ ]\!]_{C++} \Longrightarrow \\ Expr\ (\ [\![\ e.expr\ ]\!]_{C++}\ ,\ [\![\ e.pos\ ]\!]_{C++}\ ,\ [\![\ e.len\ ]\!]_{C++}\ )$$

5.1.4. *Генерация целочисленных операций.*

- Вычисление длины выражения

$$[\![\ (LENGTH\ e.expr)\ ]\!]_{C++} \Longrightarrow$$
$$(int)\ [\![\ e.expr\ ]\!]_{C++}\ .get\_len()$$

- Вычисление минимума целых аргументов

$$[\![\ (MIN\ e.args)\ ]\!]_{C++} \Longrightarrow$$
$$pxx\_min\ (\ [\![\ e.args\ ]\!]_{C++}\ )$$

- Вычисление максимума целых аргументов

$$[\![\ (MAX\ e.args)\ ]\!]_{C++} \Longrightarrow$$
$$pxx\_max\ (\ [\![\ e.args\ ]\!]_{C++}\ )$$

- Арифметическая операция (левоассоциативная)

$$[\![\ (INFIX\ s.op\ (t.log_1\ \ldots\ t.log_n))\ ]\!]_{C++} \Longrightarrow$$
$$(\ [\![\ t.log_1\ ]\!]_{C++}\ s.op\ \ldots\ s.op\ [\![\ t.log_n\ ]\!]_{C++}\ )$$

5.2. Пример компиляции «Рефал Плюс — C++». В качестве иллюстрации к вышесказанному, рассмотрим результаты конвертирования Рефал Плюс-программы в C++-код.

Входной текст на языке Рефал Плюс:

```
$use StdIO;
Main = <PrintLN 'Hello!'>;
```

Выходной C++-текст:

```
#include <rf_core.hh>
#include <refal/StdIO.hh>
namespace refal
{
using namespace rfrt;
namespace hello
{
 static const Expr _c_0 = Char::create_expr('Hello!');
 RF_FUNC(Main, (), (RF_RES _v_res1))
   RF_CALL(StdIO::PrintLN, (_c_0), ());
    _v_res1 = empty;
 RF_END
}
rfrt::Entry rf_entry(hello::Main);
}
```

6. C++-библиотека поддержки периода исполнения (RSL)

6.1. Архитектура run-time support library. На рис. 2 изображена схема C++-библиотеки поддержки периода исполнения.

Библиотека разделена на три каталога:

- Директория `library` включает набор поддиректорий, содержащих модули реализации библиотечных функций стандарта языка Рефал Плюс. Основная задача этих функций — организация на достаточно общем уровне набора действий по переходу от «рефальских» вызовов, сгенерированных конвертером, к низкоуровневым функциям работы с C++-объектами. Имена файлов соответствуют названиям библиотечных рефал-функций. Поддиректория include содержит каталог `refal`, где находятся заголовочные файлы для организации вызова функций.
- Каталог `libp++` содержит модули описания абстрактных классов наиболее общих механизмов C++-реализации, обеспечивающих распределение памяти, буферизацию, работу с некоторыми видами объектов (строками, векторами), обработку ошибок. Имена файлов этой директории имеют префикс `pxx_`.

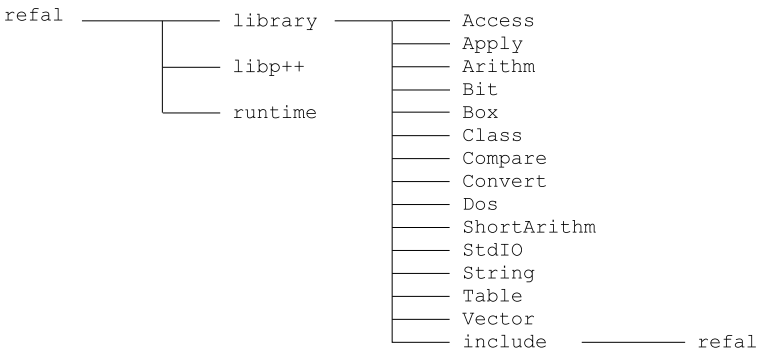

Рис. 2. Структура RSL

- В директорию `runtime` включены файлы с описаниями классов C++-объектов, таких как `Expr`, а также файлы реализации «низкоуровневых» функций поддержки для стандартных библиотечных функций из каталога `library`. Здесь имена файлов отличаются префиксом `rf_`.

6.2. Описание класса `Expr`. Основной класс для представления рефал-выражений — класс `Expr`. Рассмотрим его структуру и интерфейс, который используется в итоговом C++-коде:

```
class Expr
{
private:

  // указатель на первый терм
  Term* first;

  // указатель на следующий за последним термом
  Term* last;

  // указатель на область памяти, в которой находятся
  // термы выражения
  MemoryChunk* mem_chunk;

  // набор дополнительных флагов;
  // на данный момент используется только один, указывающий
  // на то, что в выражении имеются скобочные термы
  uintptr_t flags;

public:
  // конструктор пустого выражения
  inline Expr ();

  // конструктор, создающий выражение из содержимого скобок,
  // находящихся в выражении _expr в позиции _index
  inline Expr (Expr const& _expr, uintptr_t _index);

  // конструктор подвыражения
  inline Expr
    (Expr const& _expr, uintptr_t _index, uintptr_t _length);

  // конструктор копии
  inline Expr (Expr const& _expr);

  // конструктор, создающий выражение из одного терма —
  // ссылки на передаваемый объект
  inline Expr (Object* _obj);

  // деструктор
  inline ~Expr ();
```

```
  // оператор присваивания
  inline Expr& operator = (Expr const& _expr) ;

  // конкатенация выражений
  INLINE Expr operator + (Expr const& _expr) const ;

  // оператор взятия выражения в скобки
  inline Expr operator () () const ;

  // оператор, снимающий скобки с выражения, состоящего из одного
  // скобочного терма
  inline Expr operator * () const ;

  // проверка, является ли терм по указанному индексу символом
  inline bool symbol_at (uintptr_t _index) const ;

  // проверка, является ли выражение плоским
  inline bool is_flat () const ;

  // проверка, является ли выражение пустым
  inline bool is_empty () const ;

  // вычисление длины выражения
  inline uintptr_t get_len () const ;

  // сравнение с подвыражением, начинающимся в позиции _index
  // выражения _expr;
  inline bool eq (Expr const& _expr, uintptr_t _index) ;

  // сравнение первого терма с каким-либо термом другого выражения
  inline bool term_eq
    (Expr const& _expr, uintptr_t _index) const ;

  // проверка равенства выражений
  inline bool operator == (Expr& _expr) ;

  // проверка неравенства выражений
  inline bool operator != (Expr& _expr) ;

  // удаление выражения; используется в деструкторе и операторе
  // присваивания
  inline void drop () ;
};
```

7. Результаты сравнительного тестирования

В качестве тестов эффективности работы разработанного программного обеспечения использовались четыре рефал-программы решения следующей задачи: задана конечная последовательность конечных непустых множеств $S_1, S_2, \ldots S_N$ литер, необходимо построить последовательность элементов:

$$Select(S_1, S_2, \ldots S_N) = (x_1, x_2, \ldots x_N)$$

такую, что $x_i \in S_i$ для любого $i = 1, \ldots N$, и $x_i \neq x_j$ при всех $j \neq i$.

При решении задачи на Рефале будем представлять последовательности и множества литер строками из этих литер. Тем самым, требуется реализовать функцию `Select` со следующим форматом:

```
<Select ( ) (e.Set1) (e.Set2) \dots  (e.SetN)> ==> s.1 s.2 \dots  s.N
```

где `e.SetK` — непустые строки литер, `s.K` — литеры, отобранные (без повторов) по одной из `e.SetK`. Если невозможно отобрать `s.1 s.2 ...s.N` указанным способом, то функция `Select` возвращает `'*'`.

Для тестовых прогонов функция `Select` вызывалась на следующем аргументе:

```
<Select ( ) ('1234567890abc') ('1234567890ab') \dots  ('12') ('1')>
       ==> 'cba0987654321'
```

Такой вид начального вызова формировался с помощью вспомогательных функций:

```
Main = <MkArg '1234567890abc'> :: e.Prefixes,
  <PrintLN e.Prefixes>,
  <Select () e.Prefixes> :: eXs-or-sFail,
  <PrintLN eXs-or-sFail> ;

MkArg e.String =  /* e.Prefixes */
  e.String : {
    e.String-- s = (e.String) <MkArg e.String-->;
    /* empty */  = ;
  };
```

Функция `Select` была реализована четырьмя различными способами:

- `Select.rf` — предикат `OneOf` реализован в стиле языка Рефал-5 (возвращает либо `'T'`, либо `'F'`).

```
Select (e.Xs) e.Sets = /* (eXs) или '*' */
  e.Sets : {
    /*empty*/ = (e.Xs);
    (e.Set) e.Sets--,
      e.Set : e sX e,
      <OneOf (e.Xs) sX> : F,
      <Select (e.Xs sX) e.Sets--> : (e.Xs-) = (e.Xs-);
    e = '+';
  };

OneOf (e.Xs) s.X = /* 'T' или 'F' */
  e.Xs : {
    e sX e = T;
    e      = F;
  };
```

- `Select1.rf` — предикат `OneOf` реализован в стиле языка Рефал Плюс — как откатная функция.

```
Select (e.Xs) e.Sets = /* (eXs) или '*' */
  e.Sets : {
    /*empty*/ = (e.Xs);
    (e.Set) e.Sets--,
      e.Set : e sX e,
      #<OneOf (e.Xs) sX>,
      <Select (e.Xs sX) e.Sets--> : (e.Xs-) = (e.Xs-);
    e = '*';
   };

OneOf (e.Xs) s.X = /* пусто или $fail */
   e.Xs : e s.X e;
```

- `Select2.rf` — еще более в стиле языка Рефал Плюс: функция `Select` реализована как откатная фукция `Sel`, а тело предиката `OneOf` подставлено в единственную точку вызова.

```
Select (e.Xs) e.Sets = /* (eXs) или '*' */
  {
    <Sel (e.Xs) e.Sets>;
    '*';
  };

Sel (e.Xs) e.Sets = /* (eXs) или $fail; */
  e.Sets : \{
    /*empty*/ = (e.Xs);
    (e.Set) e.Sets--,
      e.Set : e sX e,
      # \{ e.Xs : e s.X e; },
      <Sel (e.Xs sX) e.Sets-->;
  };
```

- `Select3.rf` — отличается от `Select2.rf` тем, что первый аргумент `Sel` представлен не в виде строки литер (например,

'cba09'), а в виде выражения-стека
('c' ('b' ('a' ('0' ('9'))))),
с замыслом сэкономить на времени выполнения операций конкатенаций.

```
Select (e.Xs) e.Sets = /* (eXs) или '*' */
  {
    <Sel () e.Sets> :: (e.Xs) = (<Stack2Exp e.Xs>);
    '*';
  };

Sel (e.Xs) e.Sets =  /* (eXs) или $fail; */
  e.Sets : \{
    /*empty*/ = (e.Xs);
    (e.Set) e.Sets--,
      e.Set : e sX e,
      #<InStack e.Xs s.X>,
      <Sel (s.X (e.Xs)) e.Sets-->;
  };

InStack e.Xs s.X =  /* пусто или $fail; */
  e.Xs : s.Y (e.Xs--),
  \{
    s.Y : s.X;
    <InStack e.Xs-- s.X>;
  };

Stack2Exp e.Xs =    /* e.ExpXs */
  e.Xs : {
    /*empty*/ = /*empty*/;
    s.Y (e.Xs--) = <Stack2Exp e.Xs--> s.Y;
  };
```

В таблице 1 приведены временные характеристики выполнения этих программ для трех реализаций языка Рефал Плюс: v 1.8.00.8b, v 1.8.7 и обсуждаемой программной системы v 2.0, работающей под Linux. Тестирование проводилось на компьютере Pentium 4 2,0 GHz. Как и ожидалось, новые принципы реализации языка позволили получить на задаче `Select` ускорение (от 1,3 до 5,3 раз) счета. Аналогичные значения коэффициентов ускорения были получены и для других тестов.

Несомненно, улучшение скоростных характеристик было одной из задач данного проекта. Однако еще раз заметим, что главная цель данной разработки — в другом: выполнить реализацию семейства языков Рефал, обладающую хорошей структурой организации модулей, легко переносимую на различные платформы, в том числе с поддержкой параллельных архитектур, и гибкую — допускающую

Версии Рефала Плюс	Тесты			
	select	select1	select2	select3
refal-plus 1.8.00.8b (DOS)	252,5c	232,9c	225,3c	170,1c
refal-plus 1.8.7 (Linux)	45,5c	41,5c	39,3c	33,65c
refal-plus 2.0 (Linux)	17,9c	11,2c	7,45c	25,54c
ускорение (2.0 vs. 1.8.7)	2,54	3,71	5,28	1,32

Таблица 1. Результаты тестирования

модификацию функциональности системы без существенных изменений большей части ее кода.

8. Состояние разработки и ближайшие планы развития

Сегодня в основном завершена разработка всей «магистральной» линии системы, обеспечивающей реализацию языка Рефал Плюс на базе системы программирования C++:

- входной (front-end) модуль для языка Рефал Плюс;
- компилятор;
- базовый оптимизатор AIL-кода;
- выходной (back-end) модуль для языка C++;
- библиотека периода исполнения на базе языка C++.

Дистрибутив распространяется как архив «tar.gz», предназначенный для ОС Linux: http://skif.pereslavl.ru/refal . Ближайшие планы разработчиков:

- Реализация для ОС Windows дистрибутивного пакета системы «Рефал Плюс/C++».
- Реализация в системе эффективных и удобных средств отладки рефал-программ.
- Разработка и реализация в системе элементов поддержки (back-end и RSL) целевых языков C# (платформа .NET) и T++ (платформа OpenTS, поддерживающая автоматическое динамическое распараллеливание программ пользователя [3]).
- Разработка и реализация в системе элементов поддержки (front-end) существующих диалектов языка Рефал (Рефал-2, -5 и -6) в качестве входных языков.

Сегодняшние результаты по проекту позволяют надеяться, что в результате выполнения запланированных работ будет получена удобная, гибкая, эффективная и мобильная система программирования для семейства диалектов языка Рефал.

Благодарности. Авторы благодарны всему рефал-сообществу за содержательные и полезные обсуждения во время выполнения данного проекта.

Список литературы

[1] Абрамов С. М., Романенко С. А. Представление объектных выражений массивами при реализации языка Рефал: Препринт 186. — М.: ИПМ им. М.В.Келдыша, 1988, с. 27. ↑

[2] Абрамов С. М., Орлов А. Ю. *Компиляция в императивные языки синтаксического отождествления языка Рефал* // Международная конференция «Программные системы: теория и приложения». — Переславль-Залесский: Физматлит, 2004. ↑1, 4, 5.1

[3] Абрамов С. М., Адамович А. И., Позлевич Р. В. Т-система — среда программирования с поддержкой автоматического динамического распараллеливания программ // Программные системы: Теоретические основы и приложения: Сборник. — М.: Физматлит, 1999. ↑1, 8

[4] Базисный Рефал и его реализация на вычислительных машинах. — М.: ЦНИИПИАС, 1977, с. 258. ↑1, 1, 2

[5] Гурин Р. Ф., Романенко С. А. Язык программирования Рефал Плюс. — М.: Интертех, 1991, с. 183. ↑

[6] Климов Анд. В., Романенко С. А. Система программирования Рефал-2 для ЕС ЭВМ. Описание библиотеки функций: Препринт 200. — М.: ИПМ им. М.В.Келдыша, 1986, с. 38. ↑

[7] Климов Анд. В., Романенко С. А. Система программирования Рефал-2 для ЕС ЭВМ. Описание входного языка: Препринт. — М.: ИПМ им. М.В.Келдыша, 1987, с. 52. ↑

[8] Климов Анд. В., Романенко С. А., Турчин В. Ф. Компилятор с языка Рефал. — М.: ИПМ им. М.В.Келдыша, 1972, с. 74. ↑

[9] Романенко С. А. Реализация Рефала-2. — М.: ИПМ им. М.В.Келдыша, 1987, с. 191. ↑1, 1

[10] Романенко С. А. Рефал-4 — расширение Рефала-2, обеспечивающее выразимость результатов прогонки: Препринт 147. — М.: ИПМ им. М.В.Келдыша, 1987, с. 27. ↑

[11] Романенко С. А. Прогонка для программ на Рефале-4: Препринт 211. — М.: ИПМ им. М.В. Келдыша АН СССР, 1987, с. 19. ↑

[12] Романенко С. А. Компиляция Рефал-программ в виртуальный код, 1999, http://shade.msu.ru/refal.msu.ru/docs/rbvcomp.ps.gz. ↑1

[13] Турчин В.Ф. *Метаязык для формального описания алгоритмических языков* // Цифровая вычислительная техника и программирование: Сборник. — М.: Сов. Радио, 1966, с. 116–124. ↑1

[14] Турчин В.Ф. Программирование на языке Рефал: Препринт 41, 43, 44, 48, 49. — М.: ИПМ им. М.В.Келдыша, 1971. ↑

[15] Klimov Ark. V. Refal-6, 2003, http://www.refal.org/~arklimov/refal6. ↑1

[16] Klimov Ark. V., Klimov And. V. REFAL — JAVA. The programming language Refal implemented on top of the Java2 Platform, 2003, http://www.refal.org/~arklimov/refal6/refal-j.htm. ↑1

[17] Turchin V. F. Refal-5: Programming Guide and Reference Manual. — Holyoke: New England Publishing Co., 1989, http://shura.botik.ru/refal/book/html/, на русском языке: http://www.refal.ru/rf5_frm.htm. ↑1

[18] Белоус Л. Ф. *Refal-PHP: Универсальный инструмент Интернет-технологий* // Всероссийская научная конференция «Научный сервис в сети Интернет», г. Новороссийск, 23–28 сентября 2002 г. — М.: изд-во. МГУ, 2002, с. 44–47. ↑1

[19] Белоус Л. Ф. REFAL-SCITE: Интегрированная оболочка для разработки программ в интранет-среде: Электронный ресурс, 2003, http://www.refal.net/~belous/refscite.htm. ↑1

Институт программных систем РАН

Исследовательский центр мультипроцессорных систем ИПС РАН

Московский Государственный Университет им. М. В. Ломоносова

S. M. Abramov, A. Y. Orlov, L. V. Parmyonova, S. M. Ponomareva, A. F. Slepukhin. *Refal Plus implementation: new approach.* (in russian.)

Abstract. The article describes Refal Plus programming language implementation based on array refal-expression representation and direct compilation into imperative languages. The descriptions of developed formalisms, runtime support library structure are given. Testing results of developed software and other implementations of Refal Plus programming language are represented.