А. В. Столяров

Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования

Настоящий документ представляет собой черновую версию статьи, опубликованной в сборнике трудов ф-та ВМиК МГУ в 2001 году. Просьба при цитировании ссылаться на опубликованную версию с использованием следующей библиографической информации:

А. В. Столяров. Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования. В кн. *Программные системы и инструменты. Тематический сборник*, под ред. Л.Н.Королева, том 2, стр. 184–195. Издательский отдел факультета ВМиК МГУ, Москва, 2001.

Для пользователей LaTeX приводится библиографическая информация в формате `bibtex`:

```
@INCOLLECTION{stolyarov:oorefal,
        AUTHOR={{А.~В.~Столяров}},
        TITLE="Расширенный функциональный аналог
               языка Рефал для мультипарадигмального
               программирования",
        BOOKTITLE = "Программные системы и инструменты.
                     Тематический сборник",
        VOLUME = "2",
        EDITOR = "{{Л.~Н.~Королев}}",
        PUBLISHER = "Издательский отдел факультета ВМиК МГУ",
        YEAR = "2001",
        PAGES = "184--195",
        ADDRESS = "Москва"
}
```

Расширенный функциональный аналог языка Рефал для мультипарадигмального программирования

А. В. Столяров

Аннотация

В статье рассматривается объектно-ориентированная модель рефал-машины с расширяемыми возможностями, предназначенная для практического применения мультипарадигмального программирования в проектах на языке Си++. Дается описание возможностей библиотеки классов Си++ InteLib, реализующей эту модель.

1 Введение

Язык Рефал был предложен В.Ф.Турчиным в 1966 году[8]. В.Ш.Кауфман отмечает сходство изобразительных средств Рефала с нормальными алгоритмами Маркова[1]. Парадигму языка Рефал Кауфман называет *ситуационным программированием*, выделяя две ключевые абстракции - *анализ* исходной структуры данных и *синтез* результирующей структуры.

Рефал чрезвычайно удобен при решении задач, связанных с преобразованиями символьных данных. В.Ф.Турчин изначально позиционировал Рефал как *метаалгоритмический* язык[9] и позднее использовал его, в числе прочего, для изучения эквивалентных преобразований программ[10]. В.И.Сердобольский отметил удобство языка для работы с алгебраическими формулами[3]. С середины восьмидесятых годов и до настоящего времени Рефал успешно применяется в исследованиях, связанных с суперкомпиляцией и частичными вычислениями[11].

К настоящему времени существует несколько диалектов языка Рефал, среди которых наиболее динамично развиваются Рефал-5[12] и Рефал Плюс. Разработаны методы эффективной реализации языка; созданы системы программирования для различных программно-аппаратных платформ.

Несмотря на это, Рефал крайне редко применяется в индустриальном программировании (под индустриальным программированием мы понимаем прежде всего изготовление программных продуктов для конечного пользователя, находящегося за пределами компьютерной индустрии). При этом Рефал нельзя назвать неизвестным языком; многие профессиональные программисты с этим языком знакомы. По-видимому, основной причиной слабой востребованности языка является трудность его интеграции с другими языками (в частности, с языками, представляющими наиболее популярный в современной индустрии гибрид императивного и объектно-ориентированного программирования).

В статье [5] можно найти более подробное обсуждение проблемы межъязыковой (мультипарадигмальной) интеграции и предварительное сообщение о методе, основанном на моделировании альтернативных парадигм как предметных областей средствами базового объектно-ориентированного языка (на примере языков Си++[7] и Лисп). Детальное описание экспериментальной реализации метода (библиотеки классов Си++ InteLib, моделирующей алгебру S-выражений языка Лисп) дается в работе [6]. Там же приводятся соображения по поводу моделирования средствами языка Си++ парадигмы языка Рефал.

Целью настоящей статьи является описание результатов экспериментальной реализации объектно-ориентированной модели парадигмы языка Рефал.

2 Краткий обзор метода. Библиотека InteLib

Библиотека InteLib, первоначально ориентированная на импорт в Си++-проекты парадигм языка Лисп, основана на моделировании алгебры S-выражений, представленной в языке Лисп, с помощью иерархии полиморфных классов, для которых переопределяются некоторые стандартные операции Си++ таким образом, чтобы обеспечить удобное для программиста отображение привычных Лисп-конструкций на синтаксис, допустимый в Си++-коде.

Все S-выражения представляются в InteLib объектами некоторых классов, являющихся потомками общего базового класса LTerm (это имя возникло на ранней стадии проектирования библиотеки и сохранено в силу исторических причин). Так, S-выражение типа «строковая константа» представляется объектом класса LTermString, лисповский символ (в терминологии, традиционной для стандарта Common Lisp[4]) или идентификатор (в терминах, принятых в языке Scheme[2]) моделируется объектом класса LTermSymbol, точечная пара - объектом класса LDotPair и т.п. Для доступа к объектам иерархии LTerm, существующим всегда в динамической памяти, используются объекты класса LReference. Класс LReference по своим свойствам аналогичен простому указателю, в дополнение к чему исполняет функции сборщика мусора.

Для построения лисповских списков, состоящих из точечных пар, используются две базовые операции - создание списка из одного (первого) элемента и добавление нового элемента в конец. Для введения первой операции используется класс LListConstructor с перекрытой операцией `|`; в качестве символа для обозначения второй операции из соображений наглядности выбрана двухместная операция «запятая», переопределенная в классе LReference. Класс LReference имеет конструкторы преобразования для всех стандартных встроенных типов языка Си++, что, в сочетании с механизмом неявных преобразований, дает возможность использования, например, следующих конструкций:

```
(L| 25, 36, 49)                    // (25 36 49)
(L| "I am the walrus", 1965)       // ("I am the walrus" 1965)
(L| 1, 2, (L| 3, 4), 5, 6)         // (1 2 (3 4) 5 6)
(L| (L| 1, 2), 3, 4)               // ((1 2) 3 4)
```

Для удобства построения точечных пар и точечных списков применяется операция `||`, заменяющая метку конца списка (символ NIL) в левом операнде на правый операнд. В роли лисповского сокращения для формы `(QUOTE A)` (апостроф) используется операция `~` (тильда). Приведем еще несколько примеров:

```
(L| 1 || 2)                        // (1 . 2)
((L| 1, 2, 3)|| 4)                 // (1 2 3 . 4)
(L| MEMBER, 1, ~(L| 1, 3, 5))      // (member 1 '(1 3 5))
```

Необходимо пояснить, что `L` - это имя объекта класса LListConstructor, `MEMBER` - это имя объекта LReference, ссылающегося на объект класса LTermSymbol, и т.д.

Наконец, для прямого построения точечной пары из левого и правого значения используется операция `^`.

Класс LTerm имеет виртуальную функцию Evaluate(), производящую вычисление соответствующего выражения в смысле Лиспа. Для констант (числовых, строковых и константных символов) метод Evaluate возвращает ссылку на исходное выражение. Для классов, наследуемых от LTerm и моделирующих другие (неконстантные) типы S-выражений, функция Evaluate() соответствующим образом перекрывается.

Рассмотрим для примера функцию на языке Лисп, определяющую, имеют ли два дерева изоморфную структуру (два дерева считаются изоморфными тогда и только тогда, когда они отличаются только значениями в листьях):

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1) (car tree2))
               (isomorphic (cdr tree1) (cdr tree2))))))
```

Если считать, что символы, соответствующие стандартным функциям Лиспа, уже описаны соответствующими объектами класса LSymbol, то можно записать на Си++ код, эквивалентный вышеприведенному:

```
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
  static LSymbol TREE1("TREE1");
  static LSymbol TREE2("TREE2");
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND, (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
             (L|(L|ATOM, TREE2), NIL),
             (L|T, (L|AND,
             (L|ISOMORPHIC, (L|CAR, TREE1), (L|CAR, TREE2)),
             (L|ISOMORPHIC, (L|CDR, TREE1), (L|CDR, TREE2))))
  )).Evaluate();
}
```

Следует особо отметить, что данный фрагмент кода является кодом на языке C++, не требующим какого-либо дополнительного препроцессирования. Таким образом, не выходя за пределы языка C++, мы получаем возможность применения парадигм языка Lisp в C++-проекте.

3 Моделирование рефал-выражений с помощью лисповских списков

Древовидные данные, состоящие из строк терминальных символов, символов-меток, символов-чисел и структурных скобок (то есть выражения, допустимые на входе в рефал-функцию), очевидным образом изоморфны лисповским деревьям, состоящим из строковых констант, символов и чисел. Например, рефальское выражение

```
'abc' symb ('def' 25)
```

соответствует в смысле рассматриваемого изоморфизма лисповскому

```
("abc" symb ("def" 25))
```

Необходимо отметить, что изоморфизм является достаточно условным. На самом деле, рефальские объектные выражения изоморфны подмножеству лисповских S-выражений, а именно – множеству списков, в которых нигде не встречается двух символьных строк подряд. В реализованной версии библиотеки атомарные S-выражения и точечные списки считаются не соответствующими никаким выражениям языка Рефал и при попытке использования в качестве таковых вызывают ошибку. Что касается списков с двумя и более строковыми константами подряд, то такие списки считаются эквивалентными спискам, в которых следующие подряд константы заменены на их конкатенацию.

Рассмотренный изоморфизм позволяет использовать для представления рефальских выражений уже существующие средства, разработанные для Лиспа. Приведенное выше выражение можно представить на Си++ как

```
(L|"abc", SYMB, (L|"def", 25))
```

Для представления рефальских угловых скобок, обозначающих вызовы функций (так называемые «активные списки»), введем новый класс RfCall, наследуемый от LTerm и инкапсулирующий обычный список. Для построения таких активных списков введем класс, аналогичный описанному ранее LListConstructor. Назовем его, например, RfListConstructor и опишем его единственный экземпляр с именем R. Также введем унаследованный от LReference класс RfCallReference, предназначенный для работы с активными списками. Основное отличие RfCallReference от LReference состоит в измененной семантике операции «запятая», которая в данном случае предполагает, что левым операндом является не списочное S-выражение, а выражение типа RfCall. Соответственно, добавление нового элемента производится к инкапсулированному в него списку.

Теперь рефальское `<fun1 'a' sym1 'b'>` может быть представлено в виде `(R|FUN1,"a",SYM1,"b")`.

Для представления рефальских переменных опишем специальный класс RfVariable, также наследуемый от LTerm. Строго говоря, никакими свойствами терма рефал-переменная не обладает, и такое наследование преследует единственную цель - позволить использовать для построения правил преобразования уже существующие средства.

Класс RfVariable имеет методы, используемые в процессе сопоставления выражений с образцами. Сам класс RfVariable является абстрактным, а переменные конкретных типов описываются с помощью наследуемых от него подклассов.

Путем введения дополнительных классов – наследников LReference и перекрытия для этих классов операций `()` (штатно обозначает вызов функции) и `[]` (штатно используется для адресации в массиве) можно дать достаточно наглядные средства для описания рефал-функций. Например, функция «палиндром», имеющая на Рефале-5 вид

```
Pal {            = True;
      s.1        = True;
      s.1 e.2 s.1 = <Pal e.2>;
      e.1        = False;  }
```

может быть представлена на Си++ как

```
LSymbol PAL("PAL");
LSymbol True("True");
```

```
LSymbol False("False");
RfVariable_E e_1("e.1"), e_2("e.2");
RfVariable_S s_1("s.1");
// ...
RFUNC(PAL) [(L)                ^ (L| True)]
           [(L| s_1)           ^ (L| True)]
           [(L| s_1, e_2, s_1) ^ (L| (R| PAL, e_2))]
           [(L| e_1)           ^ (L| False)] ;
```

Стратегия вычисления рефал-выражений существенно отличается от стратегии вычисления S-выражений в смысле Лиспа. Например, результатом вычисления простого списка в смысле Лиспа будет применение функции, заданной первым элементом списка, к остатку списка как к списку фактических параметров (обычно подлежащих, в свою очередь, вычислению перед передачей в функцию), тогда как результатом вычисления того же списка в смысле Рефала будет простая конкатенация результатов вычисления всех элементов списка (опять таки, в смысле Рефала). Поэтому для вычисления S-выражений в смысле Рефала применяется не метод Evaluate(), введенный в классе LTerm, а внешняя по отношению к иерархии LTerm функция, названная RefalMachineCall.

Кроме того, пользователю доступны функции RefalMatch (простое рефальское сопоставление с образцом), RefalReplaceVars (подстановка значений вместо переменных в выражении) и RefalSimplifyView (нормализация рефал-выражения). Под нормализацией понимается замена строковых констант, идущих в списке подряд, на их конкатенацию, так что в результирующем выражении нигде не встречается подряд двух строковых констант. В ситуации мультипарадигмального программирования, т.е. в условиях, в которых рефал-конструкции являются одним из возможных, но не единственным способом обработки данных, применение таких функций, реализующих частные возможности рефал-машины, зачастую бывает очень удобно.

4 Рефал-переменные. Возможность расширения их функциональности

Под переменной в Рефале понимается специфическая атомарная составляющая выражения-образца, которой может ставиться в соответствие любое рефал-выражение из некоторого множества.

В Рефале-5 существуют переменные трех типов (S-переменные, T-переменные и E-переменные). S-переменной может быть поставлен в соответствие любой символ (в том числе символ-метка или символ-число). Если воспользоваться введенным выше изоморфизмом, можно говорить, что S-переменная ставится в соответствие символу внутри строковой константы, лисп-символу или числовой константе. T-переменной может быть поставлено в соответствие, кроме вышеперечисленного, любое выражение, заключенное в скобки. Наконец, E-переменная может соответствовать любому выражению, в котором соблюден баланс скобок, в том числе пустому. Несложно видеть, что при выполнении операции сопоставления выражения с образцом (например, слева направо, как это делается в Рефале-5), для встреченных на очередном шаге S- и T-переменных можно сразу определить, возможно ли сопоставление вообще и если да, то какое именно выражение должно быть сопоставлено данной переменной. Если же в выражении встречается переменная типа E, в общем случае необходимо последовательно рассмотреть разные варианты сопоставления, для чего может понадобиться алгоритм поиска решения с возвратами (backtracking).

В более ранних версиях Рефала можно было указать ограничения на выражения, которые могут быть сопоставлены данной переменной (в данном образце). Так, образцу `'ab' s('cd')1 'ef'` могли ставиться в соответствие только две строки - `'abcef'` и `'abdef'`. В современные версии Рефала эти возможности не вошли, видимо, из соображений элегантности языка. Дополнительные условия на значения переменных в Рефале-5 могут быть заданы с помощью так называемых «*где*-предложений» (*where* clauses), что по своим возможностям превосходит механизм спецификаторов ранних версий Рефала, т.к. позволяет накладывать на переменные совершенно произвольные ограничения. Каково бы ни было ограничение на переменные `v.1, v.2, ..., v.n`, всегда возможно описать на Рефале характеристическую функцию-предикат вида `<my_fpred (v.1)(v.2)...(v.n)>`, которая возвращала бы, например, значение `true` для допустимой комбинации значений переменных и `false` в противном случае. Тогда конструкция типа where вида `,<my_fpred (v.1)(v.2)...(v.n)> : true` введет в рефал-предложение требуемое ограничение. Универсальность механизма следует из алгоритмической полноты языка Рефал.

Для программ, составляемых целиком на Рефале, такой механизм, безусловно, предпочтительнее. Однако в ситуации, когда основным языком проекта является Си++ или другой объектно-ориентированный язык, а от Рефала используются только изобразительные парадигмы, возможность введения дополнительных типов рефал-переменных может повысить наглядность, а в некоторых случаях - и эффективность составляемых программ.

В связи с этим библиотека InteLib делает минимально возможное количество предположений относительно выражений, которые могут быть сопоставлены рефал-переменной. Переменные делятся на две категории - однозначные и многозначные. В базовой библиотеке к первой категории относятся S- и T-переменные, ко второй - E-переменные. Обработка переменных второй категории отличается тем, что в ситуации, когда такая переменная встречается в открытой позиции (то есть за ней не следует непосредственно конец выражения или закрывающая скобка), процедура сопоставления организует перебор вариантов. Количество возможных вариантов, как и сами эти варианты, полностью определяется виртуальными функциями класса `RfVariable_E` (потомка класса RfVariable, представляющего E-переменные).

Таким образом, при необходимости можно описать дополнительные классы рефал-переменных. Например, можно ввести переменные, в соответствие которым ставится любая цепочка символов, допустимая в качестве имени идентификатора (в виде многозначной рефал-переменной). Более того, в случае, если в анализируемом языке имеется требование, что в качестве имени идентификатора всегда берется самая длинная из допустимых цепочек (в большинстве современных языков программирования это так и есть), то грамматическое понятие «идентификатор» может быть описано в виде однозначной переменной (то есть переменной, не допускающей вариантного сопоставления), что, безусловно, повысит эффективность программы и упростит реализацию лексического анализа.

5 Конструкции расширенного Рефала-5 и их моделирование

В описании Рефала-5 [12] под «средствами расширения» языка понимаются уже упоминавшаяся where-конструкция, или *условие*, а также with-конструкция, или *блок*.

При интерпретации where-конструкции производится вычисление некоторого рефал-выражения и попытка сопоставления результата с заданным образцом, которая может окончится успехом или неудачей. Возможно, что некоторые не связанные до выполнения конструкции переменные получат значения в процессе ее выполнения. В случае успеха производися обычная подстановка переменных в правой части предложения. В случае неудачи производится откат до ближайшей точки ветвления, если такая есть, или фиксируется неудача сопоставления для всего предложения.

With-конструкция фактически заменяет собой правую часть. При успешном сопоставлении аргумента функции с образцом предложения, содержащего with-конструкцию, вместо обычной подстановки производится вычисление аргумента with-конструкции; результат вычисления передается в безымянную рефал-функцию, заданную телом with-конструкции в виде обычного блока, состоящего из одного и более рефал-предложений.

В рассматриваемой объектно-ориентированной модели Рефала обе эти конструкции реализованы как частные случаи обобщенных возможностей, реализованных, соответственно, абстрактными классами RfLeftSubclause и RfRightSubclause.

Один или несколько объектов класса RfLeftSubclause могут быть помещены в конец образца, т.е. левой части рефал-предложения, что является оправданием названия класса. Когда в процесе сопоставления обнаруживается объект этого класса, считается, что образец закончен (то есть остаток сопоставляемого выражения должен быть к этому моменту пуст). Если сопоставление было успешным, производится вызов метода RfLeftSubclause::Process, которому передается текущий контекст сопоставления (совокупность связей между рефал-переменными и их значениями). Метод возвращает true для индикации успеха, false - для индикации неудачи.

Объект класса RfRightSubclause может быть помещен в рефал-предложении на место правой части. В этом случае вместо обычной процедуры подстановки значений переменных происходит вызов метода RfRightSubclause::Process. Метод возвращает рефал-выражение, которое и используется в качестве результата работы всей функции.

Собственно конструкции where и with реализованы в виде классов RfWhereClause и RfWithClause, наследуемых от RfLeftSubclause и RfRightSubclause соответственно.

Следует заметить, что обычную процедуру подстановки значений переменных базисного Рефала тоже можно реализовать в виде наследника класса RfRightSubclause. Этим возможности обобщенного механизма, разумеется, не исчерпываются.

Введенные обобщения позволяют существенно расширить изобразительные возможности рассматриваемой модели по сравнению с исходным языком Рефал-5.

6 Лисп-функции и их вызов из рефал-функций

Реализация функционального аналога языка Рефал в виде надстройки над функциональным аналогом языка Лисп логично приводит к мысли об использовании уже наработанных для лисповской части InteLib библиотечных функций, соответствующих встроенным функциям языка Лисп.

При этом следует помнить о различии между множеством всех S-выражений и множеством S-выражений, допустимых как представление рефал-выражения. Кроме того, необходимо учитывать различие семантик двух языков. Проиллюстрируем возникающую проблему на примере. Рассмотрим рефал-выражение

```
<CAR (1 2 3)><CAR (10 20 30)><CAR (100 200 300)>
```

где CAR обозначает стандартную Лисп-функцию. Логично ожидать, что результатом вычисления такой конструкции должно стать рефал-выражение `1 10 100`, или, иначе говоря, S-выражение `(1 10 100)`. Однако не все так просто. Первый активный вызов, если не принять специальных мер, вернет S-выражение `1`, второй - `10`, третий - `100`. Все три, следует отметить, не являются допустимыми в качестве рефал-выражения, что уже означает возникновение ошибочной ситуации. Но даже если бы все три результата случайно оказались принадлежащими допустимому множеству, - например, если бы исходный вызов выглядел как

```
<CAR ((1) 2 3)><CAR ((10) 20 30)><CAR ((100) 200 300)>
```

то даже в этом случае полученный результат несколько отличался бы от того, чего мы ожидали (`(1 10 100)` вместо `((1)(10)(100))`.

Причина этого в семантическом различии интерпретаций двух языков. В Рефале, в отличие от Лиспа, результат вычисления конкатенации нескольких выражений есть *конкатенация* результатов, то есть, в терминах Лиспа, результат применения функции `APPEND` к полученным результатам, тогда как в Лиспе в аналогичной ситуации традиции подсказывают ожидать чего-то похожего на результат функции `MAP` (список, построенный из результатов).

Преодолеть проблему можно, введя возможность помечать особым образом те активные (построенные с помощью класса RfListConstructor) списки, которые вызывают лисповскую функцию. При вычислении такого списка полученный результат, прежде чем быть возвращенным вызывающей функции, инкапсулируется в список из одного элемента. Для этого в классе RfListConstructor предусмотрен дополнительный оператор `||`, который делает то же, что и введенный ранее оператор `|`, отличаясь от него только выставлением соответствующего признака у порождаемого активного списка.

7 Непосредственное сочетание языков Лисп и Рефал

Описанная выше объектно-ориентированная модель рефал-машины в сочетании с реализованной ранее моделью алгебры лисповских S-выражений приводит к идее создания гибрида языков Лисп и Рефал.

В работах [5, 6] описан транслятор усеченного диалекта языка Лисп, названного InteLib Lisp. Транслятор генерирует на выходе модуль, написанный на языке Си++, предназначенный для использования с библиотекой InteLib. Отмечается, что устройство этого транслятора чрезвычайно примитивно в силу подобия получаемых конструкций Си++ и исходного кода на Лиспе.

Аналогичный транслятор может быть создан для языка Рефал (например, для диалекта Рефал-5, все возможности которого имеют адекватную модель в библиотеке). Очевидно, при построении такого транслятора необходимо предусмотреть возможность использования обобщенных рефал-переменных, а также обобщений конструкций with и where (естественно предположить, что сами обобщения реализуются на Си++, и в конструкциях Рефала должен быть предусмотрен только некоторый интерфейс к таким обобщениям). Следует ожидать, что сложность построения такого транслятора также будет невысока, что позволяет рассматривать его как побочный продукт разработки библиотеки InteLib.

Наконец, для построения транслятора для гибрида Лиспа и Рефала достаточно разработать синтаксические соглашения. Можно ожидать, что построение транслятора с такого гибридного языка также не составит проблемы и не окажется ресурсоемким.

8 Заключение

Предложенная в статье объектно-ориентированная модель расширенной рефал-машины позволяет, не выходя за рамки языка Си++ и используя, соответственно, существующие системы программирования без специальных модификаций, применять в проектах изобразительные парадигмы, наработанные в языке Рефал, а при необходимости и расширять возможности этих парадигм.

В сочетании с разработанной ранее моделью алгебры S-выражений языка Лисп, предложенная модель порождает среду для мультипарадигмального программирования в рамках проектов, основным языком которых является Си++.

Поскольку применение предложенных методов не требует ни модификации существующих систем программирования, ни внедрения новых, барьер внедрения для предложенной методологии оказывается чрезвычайно низок и сводится к освоению библиотеки классов InteLib, имеющей достаточно простую структуру. Это позволяет ожидать, что предложенный метод может найти применение в индустриальном программировании.

Список литературы

[1] Кауфман В.Ш. Языки программирования. Концепции и принципы. М.: Радио и Связь, 1993.

[2] Kelsey R. at al. Revised^5 report on Algorithmic Language Scheme.

[3] Сердобольский В.И. Язык РЕФАЛ и его использование для преобразования алгебраических выражений. // Кибернетика. 1969. N 3. Стр. 45-51.

[4] Steele G. L. Common Lisp the Language, 2nd edition. Digital Press, 1990.

[5] Stolyarov A., Bolshakova E., Building Functional Techniques into an Object-oriented System. // Knowledge-Based Software Engineering (proceedings of the 4th JCKBSE, Brno, Czech Republic, 2000), IOS Press, 2000. p.101-106

[6] Столяров А. Интеграция изобразительных средств альтернативных языков программирования в проекты на C++. Москва, 2001. *Рукопись деп. в ВИНИТИ РАН*

[7] Stroustrup B. The C++ Programming Language, 3rd edition. Addison-Wesley. 1997.

[8] Турчин В.Ф. Метаязык для формального описания алгоритмических языков. // Цифровая вычислительная техника и программирование. М.: Сов. радио, 1966. Стр. 116-124

[9] Турчин В.Ф. Метаалгоритмический язык. // Кибернетика. 1968. N 4. Стр. 45-54.

[10] Турчин В.Ф. Эквивалентные преобразования программ на Рефале // Автоматизированная система управления строительством. М.: ЦНИПИ-АСС, 1974. Стр. 36-68.

[11] Turchin V.F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. 1986. Vol. 8, N 3. Pg. 292-325.

[12] Turchin V. REFAL-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989.