УДК 681.3

ПРИМЕНЕНИЕ СИСТЕМ ПЕРЕПИСЫВАНИЯ ТЕРМОВ К АНАЛИЗУ ИСХОДНОГО ПРОГРАММНОГО КОДА

Р.С. Шевченко

ООО ГрадСофт, ruslan@shevchenko.kiev.ua

А.Е. Дорошенко

НТУ Украины «КПИ» и Институт программных систем НАН Украины,
03187, Киев, проспект Академика Глушкова, 40.
Тел.: 526 1538, dor@isofts.kiev.ua

Описана система переписывания термов TermWare и ее применение к задачам анализа и преобразования исходного кода. Особенностью подхода является комбинирование декларативного и императивного стилей программирования и построение слабых аппроксимаций вычислительной семантики, которые отображают определенные аспекты поведения программы, вместо полной формальной модели. Описаны два практических приложения системы TermWare – статический анализатор исходного кода для Java и система частичного выполнения Java-програм.

The term-rewriting system TermWare and its application to the analysis and transformation of source code are described. The feature of the approach is combining declarative and imperative styles and construction of weak approximations of program computational semantics, representing certain aspects of program behavior, instead of complete formal model. Two practical applications of TermWare system are described – static analyzer of source code for Java and system of partial evaluation of the Java-programs.

Введение

Переписывающие правила являются достаточно известной технологией с широким спектром применения – от символьных вычислений до программной инженерии [1]. Существуют полнофункциональные среды программирования, основанные на правилах переписывания, такие как Maude [2], Stratego [3] или APS [4], представляющие собой альтернативу логическому программированию. Их общая черта – представление объектов предметной области как алгебраических термов и представления вычислений как трансформации этих термов некоторым набором правил в соответствии с дополнительно заданной последовательностью применения правил (т.е. стратегией переписывания). Система TermWare [5] также относится к полнофункциональным системам переписывания, но отличается от большинства систем как семантикой, так и технологией реализации. Использование TermWare не предполагает канонизацию набора термов с помощью систем переписывания в “замкнутом” мире, где операции ввода-вывода являются единственным способом взаимодействия с окружением, а собственно поведения является побочным эффектом, а рассматривает решение задачи как описание взаимодействий системы с динамическим окружением в “открытом мире” с помощью переписывающих правил. Само динамическое окружение представляет собой объектную модель и описывается на языке Java. TermWare не является самостоятельной программной системой, а представляет собой библиотеку функций, которая стандартными способами встраивается в среду Java.

На сегодня существует довольно много переписывающих систем для погружения в среду Java, таких как Jess [6], в которой используются правила переписывания для построения экспертных систем и ASF+SDF [7], для преобразования синтаксических деревьев. Система Tom [8], в отличие от TermWare не встроена в Java, а представляет собой расширение Java для манипулирования древовидными структурами. Система TermWare уже применялась к различным аспектам программной инженерии [9–11]. В данной работе описывается семантическая модель переписывающих правил с действиями и ее применение в задачах анализа исходного кода.

1. TermWare: Описание системы

Модель TermWare будем строить как алгебру термов, состоящую из выражений вида $f(x_1..x_N)$ с переменными и структурами данных, типичными для бестипового функционального языка. Единственное отличие от таких языков состоит в том, что выделен конструктор упорядоченного множества и упорядоченные множества представляют собой отдельный тип термов. Вычисления определяются набором переписывающих правил вместе со стратегией им применения, также как и в других системах, таких как APS [4], Stratego [3] и Maude [2], однако семантика вычислений описывается в терминах поведения программы, т.е. наборов входных

воздействий и реакцией на эти воздействия. Если обычно в подобных системах вычисления это преобразования входных термов к некоторой канонической форме с помощью набора правил вида *f:x->y*, где *x* – входной терм, *y* соответственно выходной, то в TermWare шаг вычислений описывается с помощью правила *f:* (*x*,*s*) -> (*s'*,*y*). где *x* – входной терм, *y* – выходной; а *s* и *s'* – внутренние состояния системы. В целях экономии места подробности синтаксиса системы здесь не рассматриваются, детали могут быть найдены в предыдущих статьях авторов [9-11] и на веб-сайте TermWare [5].

1.1. Основы языка

Алфавит языка состоит из множества констант c_i примитивных типов языка Java и специального типа ATOM (множеставо атомарных неинтерпретируемых значений, включающее в себя выделенный атом NIL); множества терминальных символов t_i, пропозициональных переменных x_i, скобок '(' и ')', запятой ',' символа окружения *S* и множества символов чтения информации из окружения in_i и символов воздействия на окружения out_i. Термы языка конструируются из символов с помощью следующей схемы. Сначала определяется множество конкретных термов T_c:

- $c_i \in T_c$
- если $a_1 \in T_c, ..., a_N \in T_c$, и f_i – функциональный символ арности *N*, тогда $f_i(a_{1,...} a_N) \in T_c$.

Множество подстановочных символов T_v определяется следующим образом:

- $v_i \in T_v$;
- если $a \in T_c$, то $a \in T_v$;
- если $a_1 \in T_v$, ... $a_N \in T_v$ и f_i – функциональный символ арности *N*, то $f_i(a_1,...., a_N) \in T_v$.

Терм, принадлежащий разности множеств T_v/T_c , назовем термом со свободными переменными. Будем обозначать *v*(*t*) множество свободных переменных, содержащихся в *t*.

Теперь определим такие синтаксические функции:

- *name*: $T_v \rightarrow String$. – возвращает имя головного терма;
- *arity*: $T_v \rightarrow Integer$ – возвращает арность терма;
- *subterm*(*i*,*t*): *Integer* • $T_v \rightarrow T_v$ – вычисляет *i*-й подтерм терма *t*, если $i \le arity(t)$, или NIL, если $i > arity(t)$;
- *free_equal*(t_1,t_2): T_v • $T_v \rightarrow Boolean$, возвращает значение *true*, если t_1 и t_2 эквиваленты с точностью до переименования свободных переменных, и значение *false* в противном случае;
- *bound_equal*(t_1,t_2): T_v • $T_v \rightarrow Boolean$, возвращает значение *true* если t_1 и t_2 эквиваленты с учетом нумерации свободных переменных; и значение *false* в противном случае.
- *term_less*(t_1,t_2): T_v • $T_v \rightarrow Boolean$ – определяет полное упорядочивание термов; это позволяет ввести конструктор множество *set*(t_1,t_2,..,t_n), с тем ограничением, что *set*(t_1,t_2...t_n) имеет значение тогда и только тогда, когда из $i<j$ следует *term_less*(t_i , t_j).

Выражение *subst*(*t*,*x*,*s*) обозначает подстановку *s* в *t* вместо свободной переменной *x*. Для этого будем также использовать эквивалентное обозначение *t*[*x*,*s*]. Терм *bound unify*(t_1; t_2) обозначает операцию унификации термов t_1 и t_2 по множеству свободных переменных этих термов. Терм *free_unify*(t_1,t_2): T_v • $T_v \rightarrow T_v$ обозначает операцию унификации термов t_1 и t_2 с точностью до переименования свободных переменных.

Семантика переписывающих правил без взаимодействия с окружением определяется выражением *apply*(*t*,*x*→*y*)=*subst*(*y*,*free_unify*(*x*,*t*)), где *x*→*y* – правило переписывания *x* в *y*. Динамика взаимодействия со средой вводится через понятие окружения *S* и пару функций: S*in: S* • $T_c \rightarrow T_c$ для чтения информации из *S* и S*out: S* • $T_c \rightarrow S$ для записи. Главным объектом нашего рассмотрения, система термов, может быть представлена как пара (*R*,*S*), где *R* – это набор правил а *S* – состояние. Правило из *R* – это терм с четырьма аргументами *rule*(*x*,*in*,*y*,*out*), который мы также будем обозначать в виде *x*[*in*] → *y*[*out*] и которое интерпретируется как выполнение переписывания *x* в *y* при условии *in* с последующим совершением действия *out*. Семантику преобразования можно записать в виде соотношения *apply*((*s*,*t*),*x*[*in*]→*y*[*out*])=*Sout*(*s*,*free_unify*(*y'*,*out*),*x''*), где *x''*=*subst*(*x'*,*Sin*(*s*,*v*),*v*) для всех пропозициональных переменных *v* из *in*.. *Sin* и *Sout* в TermWare являются вызовами методов Java или их комбинацией. Еще два элемента, которые надо добавить к паре (*R*,*S*) – это имя и стратегия. То есть, термальная система представляется в виде *System* (*name*,*ruleset*,*facts*,*strategy*) где *name* – это обозначение в иерархическом пространстве имен (интерпретатор TermWare может загружать определения по требованию из соответсвующего файла в иерархии каталогов файловой системы). Стратегия *strategy* – это последовательность выбора термов и правил для редукции. Термы и правила также определяются по имени и могут быть встроенны в TermWare franework как Java классы.

В качестве простого примера рассмотрим несколько предопределенных стратегий: *FirstTop* стратегия находит первое сопоставление (сверху-слева), редуцирует найденный подтерм и снова начинает сканировать самый верхний терм, *BottomUp* работает наоборот – всегда пытается редуцировать подтерм максимальной

вложенности; Top – нерекурсивная стратегия, которая работает тогда и только тогда, когда удалось сопоставление самого верхнего подтерма с образцом.

К примеру, пусть имеем следующий набор правил:

```
ruleset(
  p(y,q) → y
  p($x,q)  →  q,
  p(q) → y
)
```

который требуется применить к терму *f*(*p*(*p*(*q*),*q*)). Тогда стратегия *FirstTop* будет иметь результатом *f*(*q*) по второму правилу; стратегия *BottomTop* сначала редуцирует *p(q*) к *y* по третьему правилу с получением *f*(*p*(*y*,*q*)), а затем *f*(*y*) по первому. Стратегия *Top* не редуцирует ничего и оставляет исходный терм в неизменном состоянии.

Введением еще одной функции *reduce*(*s*,*t*) (*s* – имя системы, а *t* – терм), где значением этой функции является редукция *t* в *s* путем последовательного применения правил до тех пор, пока это возможно, допускается возможность вызывать одну систему из другой и выражать ограничения на последовательность применения правил, независимых от стратегии. Таким образом, TermWare определяет прямое отображение логической модели правил со взаимодействиями в язык программирования, поддерживающий наследование и иерархическую систему имен.

2. Ситаксический подход: манипуляции с AST

В задачах программной инженерии самый простой, но практичный подход состоит в прямой записи абстрактных синтаксических деревьев (AST) как алгебраических термов и применении к ним переписывающих правил. К примеру следующая java программа:

```
package x;
class X
{      int x() { return 1; }}
```

может быть записана в виде термального дерева следующим образом:

```
CompilationUnit(
  PackageDeclaration(Name(cons(Identifier("x"),NIL))),
  TypeDeclaration(Modifiers(1,NIL),
     ClassOrInterfaceDeclaration(class
       ,Identifier("X")
       ,NIL, ExtendsList(NIL), ImplementsList(NIL),
       ClassOrInterfaceBody(
          cons(
           ClassOrInterfaceBodyDeclaration(
             Modifiers(0),
             MethodDeclaration(NIL,
               int
               MethodDeclarator(Identifier("x"),FormalParameters(NIL))
              ,NIL,
               Block(cons(ReturnStatement(IntegerLiteral("1")),NIL)))
             ),
             NIL
           )))))
```

Такими термами манипулируют с помощью обычных правила переписывания. Например, следующее правило из проекта JavaChecker [10] предупреждает о пустых блоках перехвата исключений:

```
Catch($x,Block(NIL)) -> true
     [ violationDiscovered(EmptyCatchBlock,"empty catch block",$x) ].
```

Большинство средств для статического анализа исходного кода, начиная с Lint [12], используют сопоставление со статическим образцом в качечтве основного метода реализации детектора дефектов [13,14]. Переписывающие правила могут дать нам чуть больше, чем просто сопоставления – мы можем производить дополнительный анализ подтерма, выбранного с помощью сопоставления. Например, рассмотрим следующий фрагмент кода:

```
switch(x) {
     case 1:
```

```
        case 2:
              System.out.println("x<3");
              break;
        case 3:
              System.out.println("x<4");
              throw new Exception("qqq");
        case 4:
               System.out.println("x<5");
        case 5:
               System.out.println("x<6");
               return;
        default:
               System.out.println("x>=6");
    }
```

В данном примере можно увидеть потенциальный дефект программы в том, что блок обработки случая с меткой 4 не заканчивается выражением, прерывающим выполнение блока switch. Выявить этот случай с помощью только сопоставления с образцом невозможно, но с помощью систем переписывания определение не представляет труда.

3. От синтаксиса к семантике – модельные термы

Как можно видеть из предыдущих примеров, манипуляции с синтаксическими деревьями имеют довольно ограниченное значение: Во-первых, вычисления всегда “локальны” т.е. системы правил могут обрабатывать только отношения между сущностями, находящимися недалеко друг от друга в дереве анализируемого кода. И во-вторых, отсутствует категоризация символов: к примеру, в выражении *this.x=x* невозможно отличить *x* как член класса в левой части от *x* как параметра или локальной переменной в правой части.

В TermWare эти ограничения преодолеваются с помощью выбора “правильного” представления анализируемого кода, включающего в себя не только синтаксическое дерево, но и некоторый объект (маркер), несущий семантический контекст, с помощью которого можно осуществлять доступ к глобальному контексту и навигацию по семантическим связям. Существуют и другие подходы, к примеру в системе Stratego [3] предлагается использовать так называемые динамические правила [15], которые могут быть созданы и удалены во время выполнения в зависимости от уровня вложенности обрабатываемой конструкции. Это решает проблему доступа к локальному контексту (и частично – навигации), но при этом делает довольно сложной обработку нетривиальных конструкций. TermWare предоставляет вместо AST-термов так называемые “модельные термы”, где к синтаксическому дереву прибавлен еще один элемент – контекст, предоставляющий метаинфрмацию о текущем выражении (подобно API JSR 269 [16]). Так как TermWare определяет отображение Java классов в термы, то можно использовать API данных классов и одновременно сохранить декларативный стиль правил.

Итак, для каждого AST-терма *t* в определенном контексте можно построить модельный терм *model(t*) таким образом:

- если $t=f(x_1 \ldots x_N)$, то $model(t)=f_m(model(x_1),\ldots model(x_N),ctx)$,где f_m – соответствующий модельный символ, и *ctx* – специальный объект, предоставляющий API для разрешения доступа и предоставления семантического контекста выражения;
- если *t* – литерал, то $m(t)=t$.

Например, модельный терм для X.java, приведенный в начале предыдущего раздела работы, будет выглядеть так:

```
Modifiers(1),class,Identifier("X"),NIL,NIL,NIL,
ClassOrInterfaceBody(
 cons(MethodModel(Modifiers(0),  NIL,
  TypeRef(int,

jobject(ua.gradsoft.javachecker.models.JavaPrimitiveTypeModel@179dce4)
   ),
   Identifier("x"),
   FormalParameters(NIL),
   NIL,
   cons(
    ReturnStatementModel(
     IntegerLiteral("1"),

jobject(ua.gradsoft.javachecker.models.JavaPlaceContext@19bb25a)),
     NIL),
```

```
            jobject(ua.gradsoft.javachecker.models.JavaPlaceContext@1e58cb8)),
          NIL)),
         jobject(ua.gradsoft.javachecker.models.JavaPlaceContext@179935d)
        )
```

где JavaPlaceContext предоставляет API для разрешения доступа к семантической модели программы. Это делает окружение разработки достаточно богатым для построения строгих алгоритмов анализа, таких как абстрактная интерпритация или частичные вычисления.

В качестве иллюстрации применения такого подхода разберем фрагмент задачи об определении “утечки ресурсов”, которая происходит при выполнении программы тогда, когда программа запрашивает какой-то ресурс у операционной системы или промежуточного програмного обеспечения, но своевременно его не возвращает. Рассмотрим следующий фрагмент кода:

```
public static void main(String[] args) {
    FileReader reader = null;
    try {
       reader = new FileReader(args[1]);
    }catch(FileNotFoundException ex){
       ex.printStackTrace();
       return;
    }
    for(int i=0; i<10; ++i) {
          int ch=0;
          try {
            ch=reader.read();
          }catch(IOException ex){
            ex.printStackTrace();
            return;
          }
          System.out.print(ch);
    }
    if (reader!=null) {
      try {
         reader.close();
      }catch(IOException ex){
      }
    }
}
```

Если исключение возникнет внутри цикла, то FileReader reader останется незакрытым. Может случится, если нужно обработать несколько сотен файлов в секунду, то программа исчерпает ресурсы операционной системы раньше, чем запустится сборщик мусора. Представим абстрактное состояние программы как набор открытых файловых переменных и для каждой языковой конструкции либо генерировать набор состояний, либо редуцировать к известным операциям на этих состояниях. При этом возможны такие состояния:

- `OPENED(expr,fileAndLine)` – обозначающее, что выражение `expr` открыто, но не закрыт; ex`pr` представляет собой модельное выражение (обычно локальная переменная или аргумент), а `fileAndLine` – место в коде, где `expr` инициализировано;
- `cons(s1,s2),` где `s1` и `s2` – состояния; это означает, что `s1` возникло после `s2`;
- `RETURN(state),` обозначающе, что выполнение программы прерывается в состоянии `state;`
- `cond(v,s)`, которое обозначает, что если условие `v` выполняется, то возможно состояние `s;`
- `s1||s2` – обозначает, что имеет место одно из двух состояний, `s1` или `s2`, но мы не знаем, какое именно;
- `s1&&s2` – обозначает, что имеет место как `s1,` так и `s2`.

Финальное состояние корректно (т.е. анализ не нашел случаев утечек ресурсов), если состояние не содержит вхождений OPENED.

Теперь отобразим модель Java-класса на вышеописанную абстрактную семантику. Класс корректен, если корректны все его инициализаторы и методы. Инициализаторы и методы корректны, если корректен блок тела метода. Блок – последовательность предложений, и настоящая работа начинается на уровне отдельных предложений и выражений. Среди всех выражений особую роль в анализе утечек памяти играют выражения создания нового объекта (AllocationExpression) и вызов метода – именно при них могут возникнуть ситуации создания нового объекта, который надо закрыть. Для вызова метода это происходит, если метод обозначен как FactoryMetod. С выражениями создания нового объекта ситуация более сложная, ибо в общем случае нельзя сказать, что любая операция создания объекта, реализующего интерфейс Closeable порождает ресурс, которые необходимо закрыть – могут существовать объекты, поддерживающие этот интерфейс но “неопасные” с точки

зрения неообходимости закрытия (такие, как StringWriter). Еще одна проблема – это так называемые опосредованные размещения ProxyAllocation, когда фрагмент кода:

```
PrintStream x = new PrintStream(new FileStream(fname));
```

может вызвать утечку ресурсов, а фрагмент кода

```
StringOutputStream ss = new StringOutputStream();
PrintStream x = new PrintStream(ss);
```

не может.

Очевидно, что для решения подобных проблем информации, находящейся в исходном коде, недостаточно. Необходимо знать, что определенные классы могут вести себя как прокси, а некоторые – нет. В JavaChecker [10] был разработан механизм внешних аннотаций для создания такой метаинформации.

Теперь посмотрим на правило обработки выражения создания объекта:

```
      CheckExpression($var,

AllocationExpressionModel(TypeRef($tname,$type),$arguments,$ctx),$state)
         [
           $ctx.subtypeOrSame($type,
                              $ctx.resolveFullClassName("java.io.Closeable"))
           &&
              ! ($type.getAttribute("NotCloseable")==true)
         ]-> CheckAllocationNotProxy($var,$type,$arguments,$ctx,$state)
         !-> CheckExpressions($arguments,$state),


      CheckAllocationNotProxy($var,$type,$arguments,$ctx,$state)
          ->
            ( (
              !($type.getAttribute("CloseableProxy")==true)
              ||
                apply(openclose::CloseableArguments,
                      apply(NormalizeExpressions,$arguments))
             ) )
                  ?
                    cons(OPEN($var,$ctx.getFileAndLine()),$state)
                   :CheckExpressions($var,$arguments,$state),
```

Сначала смотрим на условие – вызов метода `subtypeOrSame`. Здесь определяеся, действительно ли созданный объект наследует описание `java.lang.Closeable`. Следующее условие – это проверка того, что внешний атрибут "NotCloseable" не установлен. Если эти условия выполняются, то происходит проверка выполнения условия `checkAllocationNotProxy`, в противном случае собственно `allocation`-выражение исключается из рассмотрения и остается рассмотреть только аргументы с помощью выражения `checkExpressions`. Например, в выражении:

```
   CheckExpressions([$x,$y],$state) ->

CheckExpression($x,cons(CheckExpressions($y,$state)),
   CheckExpressions(NIL,$state) -> $state,
```

`CheckAllocationNotProxy` проверяет внешний атрибут, и если он не установлен, добавляет состояние OPEN; в противном случае – проверяет дополнительно условие – а является ли "настоящим" состояние `Closeable` хотя бы одного из аргументов конструктора.

Теперь посмотрим на правило другого рода, обрабатывающее if-предложение:

```
      CheckStatement(IfStatementModel($expr,$s1,$s2,$ctx),$state) ->
         let ($evstate <- $state )
            cond($expr,
CheckStatement($s1,CheckExpression(NONE,$expr,$evstate))
           &&
             cond(not($expr), CheckStatement($s2,
                               CheckExpression(NONE,$expr,$evstate)))).
```

Это правило больше похоже на часть окружения интерпретации. На некотором уровне абстракции,

отслеживания закрытия ресурсов, действительно можно описать как абстрактную интерпретацию.

Еще одно приложение – специализация Java-программ с помощью частичной интерпретации [17]. Для его реализации потребовался только один новый элемент по сравнению с описанной функциональностью – вывод AST деревьев в его оригинальном виде. Специализация с помощью частичных вычислений может быть описана как процесс отображения Java-программы *A* и множества констант *C* на такую программу *mix*(*A*,*C*), что ее поведение на множествах данных, включающих *C*, – такое же, как и у оригинальной программы. Метод решения – простой рекурсивный спуск маркера частичных вычислений с помощью правил, похожих на нижеследующее:

```
JPE(ClassOrInterfaceModel(
     $modifiers,
     $type,
     $name,
     $typeParameters,
     $extendsList,
     $implementsList,
     $body,
    $context))
      ->
    ClassOrInterfaceModel(
          $modifiers,
          $type,
          $name,
          $typeParameters,
          $extendsList,
          $implementsList,
          JPE($body),
          $context),
JPE(ClassOrInterfaceBody($x)) -> ClassOrInterfaceBody(JPE($x)),
JPE([]) -> [],
JPE(cons($x,$y)) -> cons(JPE($x),JPE($y)),
```

На уровне выражения появляются также правила для выполнения константных вычислений:

```
JPE(AndExpressionModel($x,$y,$ctx)) ->

AndExpressionModel(JPE($x),JPE($y),$ctx),
JPE(AndExpressionModel(BooleanLiteral(false),$y,$ctx)
                                    -> BooleanLiteral(false),
JPE(AndExpressionModel($x,BooleanLiteral(false),$ctx)
                                    -> BooleanLiteral(false),
JPE(AndExpressionModel(BooleanLiteral(true),$y,$ctx) -> $y,
JPE(AndExpressionModel($x,BooleanLiteral(true),$ctx) -> $x,
```

Также мы должны подставить в переменные из *C* их значения:

```
JPE(FieldModel($obj,$id,$fm,$ctx))
   [ isJPEField($fm,$y) ]  ->  $y
   |
   [ isStaticFinalLiteralField($fm,$y) ] -> $y
                     !>  FieldModel($obj,$id,$fm,$ctx),
```

В процессе частичных вычислений, правила TermWare выполняют упрощение модельных термов и их преобразования в AST терм. Процесс частичных вычислений тоже может быть представлен как абстрактная интерпретация, где абстрактное состояние – это специализированный исходный код. Весь процесс частичных вычислений описывается с помощью 200 правил, занимающих менее 1000 строк кода. Практическое использование частичных вычислений состоит не только в специализации, но и оптимизирующих трансформациях, которые могут быть построены на основании того факта, что программа выполняется в специализированном окружении. Две основных трансформации – это девиртуализация вызовов методов и элиминация недостижимого кода.

Элиминация недостижимого кода включает в себя анализ достижимости и удаления класса или метода, недостижимого из точки входа приложения. Эта оптимизация реализована как императивный алгоритм, работающий после фазы специализации. Девиртуализация вызовов методов – это замена виртуального вызова метода обычным, если конкретный тип объекта статически известен. В некоторых случаях это преобразование может принести ощутимое повышение скорости выполнения программы. Естественно, девиртуализация может

использоваться не полностью. К примеру, одним из упрощенных вариантов реализации может быть просто приписывание модификатора `final` всем классам, у которых нет потомков.

Заметим, что ни одна из этих оптимизаций не может быть произведена ни виртуальной машиной JVM, ни компилятором, так как требуют глобального анализа.

Заключение

Описаны применения среды переписывания термов TermWare к задачам анализа и преобразования исходного кода. Приведены примеры типичных образцов использования в реальных индустриальных задачах, которые показывают что комбинация декларативного и императивного подходов – более мощное и удобное средство, чем императивный или декларативный подходы в отдельности. Полагаем, что интеграция переписывающих систем в мультипарадигмальные среды программирование является перспективным направлением развития с возможностью индустриального применения. В дальнейших работах будет уделено больше внимания объектным моделям с типовыми комбинациями из нескольких применяющийся языков (как программирования, так и моделирования).

1. *Baader F. and Nipkow T.* Term Rewriting and All That. 1998. Cambridge University Press.
2. *Winkler T.* Programming in OBJ and Maude, in Functional Programming, Concurrency, Simulation and Automated Reasoning, International Lecture Series 1991--1992, McMaster University, Hamilton, Ontario, Canada, by ed. Peter Lauer, p. 229-277, Springer Verland. LNCS.
3. *Visser E.* Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. Rewriting Techniques and Applications (RTA 01). 2001, Springer-Verland, LNCS 2051 p. 357 – 361.
4. *Kapitonova V, Letichevsky A.A., L'vov M.S. and Volkov V. A.* Tools for solving problems in the scope of algebraic programming. 1995. Springer-Verlag LNCS, 958.
5. Termware: http://www.gradsoft.ua/products/termware_eng.html
6. *E. Friedman-Hill,* Jess in Action, Manning Piblications Co, 2003.
7. *M. van den Brand and H. de Jong and P. Klint and P. Olivier,* Efficient annotated terms, Software|Practice & Experience, 30,:259-291, 2000.
8. *Balland E. Brauner P. Kopetz R., Moreau P.-E., Reilles A.* Tom: Piggybacking Rewriting on Java, Proceedings of the 18th Conference on Rewriting Techniques and Applications Springer-Verlag, Lect. Notes in Comp. Sci., 2007.
9. *Doroshenko A., Shevchenko R.* A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundam. Inform. 72 1-3, 2006. – P. 95–108.
10. *Javachecker*: http://www.gradsoft.kiev.ua/products/javachecker_eng.html
11. *Shevchenko R., Doroshenko A.* Managing Business Logic with Symbolic Computations. Lecture Notes in Informatics Proceeding of Information Systems Technology and its Applications. – 2003. – V.30. – P. 143–152.
12. *Johnson* S. Lint – a C program checker. Bell Laboratories. Computer Science Technical Report. 65 1977.
13. *Copeland T.* PMD Applied. Centennial Books. 2005.
14. *Nelson G.* Extended Static Checking for Java. MPC. 2004.
15. *Bravenboer M., A. van Dam, Olmos K. , Visser E.* . Program Transformation with Scoped Dynamic Rewrite Rules. Fundam. Inf. 69, 2006. – P. 123–176.
16. *Java Community Process:* JSR269. Pluggable Annotation Processing API http://jcp.org/en/jsr/detail?id=269
17. *GradSoft* JPE home page. http://www.gradsoft.ua/products/jpe eng.html.