ISSN 1512-1232

ПОСТРОЕНИЕ ОБОБЩЕННЫХ РЕКУРСИВНЫХ ФОРМ ДЛЯ ФУНКЦИОНАЛЬНЫХ ЯЗЫКОВ И ИХ ПРИМЕНЕНИЕ В ЗАДАЧАХ ВЕРИФИКАЦИИ ПРОГРАММ

Арчвадзе Н.Н.[1], Пховелишвили М.Г.[2], Шецирули Л.Д.[3]

[1]Тбилисский государственный университет им. И.Джавахишвили, Грузия, Тбилиси, 0143, ул. Университетская 2.
(natarchvadze@yahoo.com http://www.tsu.ge)
[2] Институт вычислительной математики им. Н.Мусхелишвили, Грузия, Тбилиси, 0193, ул. Акурская 7.
(merab5@list.ru http://www.acnet.ge/icm)
[3] Батумский государственный университет им.Шота Руставели. Грузия, Батуми, 6010, ул. Ниношвили 35.
(lika77u@yahoo.com http://www.bsu.edu.ge)

Аннотация

Предлагаются обобщенные формы для представления рекурсивных функций, которые используются в Комплексе Производства Программ (КПП) для решения задач автоматического синтеза и верификации программ. КПП-эта система интеллектуального, автоматического программирования, которая должна автоматизировать все процессы, начиная с написания программ до конечной отладки. Она предназначена для функциональных языков, конкретно для LISP -подобных систем. В статье для обработки LISP-овских списков были определенны две универсальные формы рекурсивных функций. Эти формы были обобщенны для N-аргументных функций. Они применяются в задачах верификаций функций, конкретно рассмотренны вопросы автоматического верификаций LISP- программ.

Ключевые слова*: функциональное программирование, рекурсивные функции, верификация программ.*

1. Введение.

Развитие средств вычислительной техники гарантирует ежегодное увеличение ресурсов памяти и быстродействию в два раза. Развитие информатики успевает только за техническими средствами ввода и вывода материала, особенно графического и звукового. Быстро развивается машинная графика и аудиообщение. Разработка сетей вычислительных машин привела к внедрению широких информационных служб, например, Интернета. Но технология решения задач до сих пор находится на прежнем уровне. Здесь главная фигура - программист. Прямой пользователь не имеет средств самостоятельной разработки программ на основе собственных знаний (без перевода их на формальный алгоритмический язык при помощи программиста). Этот недостаток, а также критика возможностей процедурного программирования, выявили и направление в информатике, которое именуется интеллектуальным, автоматическим программированием.

Одна из главных задач автоматического программирования - ***верификация программ*** (проверка правильности программ). Программисты не раз убеждались на своем горьком опыте как бывает сложно написать корректную программу, то есть такую, которая делает в точности то, что требуется. В большинстве объемных программ есть ошибки, последствия которых могут быть самыми различными. Некоторые ошибки безобидны, другие - раздражают, а некоторые - смертельно опасны. К примеру, программное обеспечение электрокардиостимуляторов, автопилотов, систем управления двигателями, антиблокировочных тормозных систем, приборов радиационной терапии, систем управления ядерных реакторов критично при наличии ошибок. Ошибки, допущенные при разработке таких программ, могут повлечь за собой массовые человеческие жертвы. Чем глубже

ISSN 1512-1232

проникновение компьютеров во все сферы жизни человечества, тем серьезнее возможная угроза жизни людей, порождаемая ошибками в программах. Аппаратное обеспечение подтверждено схожими проблемами, но мы будем рассматривать исключительно программное обеспечение.

По данным, опубликованным в [1], ежегодно ошибки по программному обеспечению в США обходятся в 60 млрд. долларов. Для преодоления этих проблем американские специалисты и специалисты из европейских стран по формальным методам и спецификациям программ приняли решение поставить теоретические достижения в этой области на производственную основу. В 2005 г. был сформулирован международный проект "целостного автоматизированного набора инструментов для проверки корректности ПС" сроком на 15 лет по формальной верификации программного обеспечения [2].

2. Основное назначение системы «Комплекс Производства Программ»

Предлагаемый нами Комплекс Производства Программ (КПП)[3,4] должен автоматизировать все процессы, начиная с написания программ до конечной отладки. КПП предназначен для функциональных языков, так как функциональные языки в своем развитии достигли результатов, которые позволяют рассматривать их в качестве реальной альтернативы для традиционных языков программирования. В первую очередь КПП строится для LISP[5] -подобных систем.

КПП включает в себя 5 модулей: а) синтезатор программ, б) тестатор, в) доказательство правильности программ (верификатор программ), г) оптимизирующий процессор, д) корректор. Каждый из этих модулей можно функционировать самостоятельно, допустимы разные комбинации этих модулей.

Автоматический синтезатор основан на теории индуктивных выводов [6].

Индуктивный вывод –это вывод из имеющихся данных объясняющего их общего правила. Например, допустим, что имеется некоторый многочлен от одной переменной. Посмотрим, как выводится f(x), если последовательно задаются в качестве данных пары значений (0,f(0)),(1,f(1)), . . . Вначале задается (0,1) и естественно, что есть смысл вывести постоянную функцию f(x)==1. Затем задается (1,1), эта пара удовлетворяет предложенной функции f(x)==1. Следовательно в этот момент нет необходимости менять вывод. Наконец, задается (2,3), что плохо согласуется с выводом, поэтому откажемся от него и выведем новую функцию $f(x)==x^2-x+1$, которая удовлетворяет фактам (3,7)(4,13)(5,21)... поэтому нет необходимости менять вывод.

Таким образом, из последовательности пар переменней-функции можно вывести многочлен второй степени $f(x)==x^2-x+1$. Такой метод вывода называют индуктивным[6].

Как видно из примера, при выводе в каждый момент времени объясняются все данные, полученные до этого момента. Разумеется, данные, полученные позже, уже могут и не удовлетворять этому выводу. В таких случаях приходится менять вывод. Следовательно, в общем случае индуктивный вывод – это неограниченно долгий процесс.

Для точного определения индуктивного вывода необходимо уточнить: 1) множество правил-объектов вывода, 2) метод представления правил, 3) способ показа примеров, 4) метод вывода и 5) критерию правильности вывода.

Для показа примеров функций f можно использовать последовательность пар (x,f(x)) входящих и выходящих значений так, как указанно выше, последовательность действий машины Тьюринга, вычисляющей f и другие данные. Задание машине выводов пары входных и выходных значений функции f соответствует заданию системе автоматического синтеза программ входных значений x и выходных значений f(x),

ISSN 1512-1232

которые должны быть получены программой вычисления f в ответ на x. В этом смысле автоматический синтез программ на примерах также можно считать индуктивным выводом функций f.

Индуктивный вывод – LISP функций.

Автоматическим синтезом программ по примерам называется процедура создания программ по примерам входных и выходных выражений и процесса выполнения (трассировки). Примеры, которые задаются такой процедуре в качестве входных данных, называют спецификацией. В общем случае примеры содержат только «частично ограниченную информацию» о целевой программе.

В проблеме автоматического синтеза LISP-ских программ примеры имеют структуру данных, которая называется S выражением. Наша цель – изучать методологии индуктивных выводов по таким примерам .

Рассмотрим пример: допустим существует такая пара <X,Y> S выражений, что X-->Y. Пусть (A B C D) --> ((A)(B)(C)(D))

являются парами S-выражений. Тогда рассмотрим задачу вывода "?", если в паре S-выражений (A B C D) --> ?

неизвестно выходное выражение. С Харди создал компьютерную модель GAP (обобщенный автоматический программист), предназначенную для подобных выводов, т.е. для решения задачи, с чем ассоциировать "?", после просмотра нескольких примеров. В общем случае для индуктивных выводов необходимо иметь две процедуры: процедуру генерации гипотезы из множества примеров, являющихся предпосылками вывода, и процедуру проверки, выводятся ли эти предпосылки логически из гипотезы.

Прежде всего GAP используя эвристический метод, генерирует из пары заданных S выражений в качестве программы-гипотезы скиммер, ассоциирующийся с некоторой «подсказкой». При этом предполагается, что подсказки - это априорные знания об S выражениях и LISP программах, которые предварительно накапливаются в некоторой базе данных. Затем благодаря проверке этой программы-гипотезы с помощью дедуктивной LISP системы она преобразуется в окончательную LISP программу.

Инициируется подсказка, которая обращает внимание на то, что длины списков (A B C D) и ((A)(B)(C)(D)) в паре

```
(A B C D) --> ((A)( B)( C)( D) )
```

равны и генерирует программный скиммер

```
(LABEL SELF (LAMBDA (X)
       (COND((ATOM(PARTOF X))NIL)
            (T(CONS(FORM X)(SELF(CDR X))))  )))
```

Здесь SELF–имя целевой программы, а (FORM X),(PARTOF X) –неизвестные части. Используя дедуктивную программу, GAP оценивает этот скиммер для конкретного S-выражения (A B C D). Обнаруженые неизвестных символов, скажем (FORM X), с помощью традиционной LISP системы заканчивается неудачей или ошибкой, а в LISP системе GAP эта операция активно используется, при этом приобретается информация о неопределенных частях. например, для

```
X==(A B C D)
```

путем сопоставления с образцом уточняется, что

```
<FORM  X>==(A),
(SELF '(B C D))=((B)(C)(D))
```

Продолжая генерацию скимеров на основе подобных подсказок до тех пор, пока не будут оценены неопределенные части, получаем окончательную LISP программу. В данном примере

```
(LABEL SELF(LAMBDA(X)
       (COND((ATOM  X)NIL)
```

ISSN 1512-1232

```
(T(CONS(LIST(CAR X))(SELF(CDR X))))  )))
```

Применив эту программу к еще одному S-выражению (A B C D E),GAP возвращает результат вывода

```
? =  ( (A)(B) (C) (D) (E)).
```

Возможности системы GAP главным образом зависят от набора подсказок. Однако в отличие от традиционных индуктивных выводов эта система может рассматривать подсказки, тесно связанные с конкретной предметной областью, например S-выражениями, в чем ее главная особенность. В этом смысле является системой индуктивных выводов, использующей специфичные знания предметной области.

Тестатор предназначен для полуавтоматического тестирования программ. С одной стороны, производится попытка - выходя из описания программ, подобрать нужные тесты без пользователя, с другой стороны, на некоторых этапах допускать его помощь в диалоговом режиме.

Верификатор программ старается доказать неправильность результирующей программы . Если это удается, то управление передается ***Корректору.*** В случае неуспеха происходит попытка доказательства правильности, для чего используется метод индукции. Если не доказана неправильность и не неправильность программы, то в диалоговом режиме процессор требует дополнительных описаний.

Оптимизирующий процессор предназначен для оптимизации отдельных частей программ. Здесь входит блок трансформационного синтеза, основное назначение которого - перевод рекурсивных программ на итеративных. В оптимизирующем процессоре предусмотрена также чистка циклов, вынесение переменных из циклов, объединение циклов и т.д.

КПП предназначен для функциональных языков. В первую очередь он строится для LISP, для чего используются основные формы рекурсивных функций, о которых мы будем разговаривать далее.

Общая схема работы КПП представляется в следующем:

1. Синтезируется нужная программа (не обязательно правильно);
2. С помощью тестатора находится ошибки, если их не нашли, то переход на 4;
3. Корректор исправляет ошибки и переход на 2;
4. Производится попытка доказать правильность программ, если доказана неправильность, то переход на 3;
5. Оптимизация программы.

3. Построение обобщенных рекурсивных форм для языка LISP.

3.1. Обобщение формы для LISP.

Как известно, самым распространенным языком программирования для обработки списков является LISP. На этом языке разрешено применение рекурсий. С помощью хорошо выбранных рекурсивных функций можно обработать любые списки. Не трудно доказать, что построение рекурсивных функций отображающих списки в другие списки, являются вычислимым [7].

В [8] задается общие формы рекурсивных функций, позволяющие обработку списков. Эти функций на языке LISP будут иметь вид:

```
<DE LIST1(a g f x)(COND((NULL x)a)
                       (T(APPLY* g(APPLY f(CAR X))
                                 (LIST1 a g f (CDR x>
<DE LIST2(a g f x)(COND((NULL x)a)
                       (T(LIST2(APPLY* g(APPLY f(CAR x))a)
```

ISSN 1512-1232

```
                                        g f(CDR x>
```

Выходя из этого, любую рекурсивную LISP -овскую функцию, обрабатывающую произвольный x−список, можно представить так:

```
<DE FUN(x)(COND((NULL x)a)
                (T(G(F(CAR x))(FUN(CDR x>                (1)
```

или

```
<DE FUN(A x)(COND((NULL x)A)
                  (T(FUN(G(F(CAR x))A)(CDR x>
```

В [8] даны формы рекурсивных функций, которые являются более ограниченными, чем (1). Например, в форме

$$f=\begin{cases}\text{усли } x = \text{NIL}, \ mo\,\text{NIL}\\ \text{иначе } \text{CONS}(\alpha(\text{CAR}(x)),\ \text{F}(\beta(CDRx))\end{cases}$$

более конкретизированы G и A из (1):

G=CONS, A=NIL, a f=α и I=β.

Здесь, I отображение на себя:

```
(DE I(x)x)
```

Рассмотрим пример описания функций REV, которая переворачивает список:

```
<DE REV(x)(AND x(APPEND(REV(CDR x))(LIST(CAR x>
```

Заменяя AND на COND выражения, получим:

```
<DE REV(x)(COND((NULL x) nil)
                (T(APPEND(REV(CDR x)))(LIST(CAR x>
```

Если запишем это в (1) форме:

```
<DE REV(x)(COND((NULL x) NIL)
                (T(APP(LIST(CAR x))(REV(CDR x>
```

Здесь функция APP определяется как:

```
<DE APP(x,y)(APPEND y x>
```

Надо сказать, что LIST1 и LIST2 формы не исчерпывают всех рекурсивных функций обработки списков. Для этого рассмотрим PLIST функцию, которая объединяет соседние пары элементов:

(PLIST '(x_1 x_2 x_3 x_4 ...x_{2n-1} x_{2n}))⇒((x_1 x_2)(x_3 x_4)...(x_{2n-1} x_{2n}))

Эта функция имеет вид:

```
<DE PLIST(x)(COND((NULL x)NIL)
                  (T(CONS(LIST(CAR x)(CADR x))(PLIST(CDDR x>
```

Если сопоставить это с (1) формой, то несоответствия выявляются в конструкциях:

F(CAR x)≠(LIST(CAR x)(CADR x))

(CDR x)≠ (CDDR x)

Если PLIST представить с помощью MAPLIST функцией:

```
<DE PLIST(x)(MAPLIST x
                   '(LAMBDA(Y)(LIST(CAR y)(CADR y)))'CDDR>
```

А со своей стороны MAPLIST можно представить:

```
<DE MAPLIST(f Q x)(COND((NULL x)NIL)
                        (T(CONS(APPLY f x)
                                (MAPLIST f Q(APPLY Q x>
```

3.2. Обобщение представлений.

Эти описания нам диктуют более совершенные для рекурсивных функций обработки списков:

```
<DE LIST11(a g f Q x)(COND((NULL x)a)                    (2)
                          (T(APPLY* g (APPLY f x)
                              (LIST11 a g f (APPLY Q x>
```

ISSN 1512-1232

```
<DE LIST21(a g f Q x)(COND((NULL x)a)                    (3)
                          (T(LIST21(APPLY* g(APPLY f x)a)
                               g f (APPLY Q x>
```

С помощью LIST11, PLIST можно описать:

(PLIST x)=(LIST11 NIL CONS FF CDDR x)

где <DE FF(x)(LIST(CAR x)(CADR x>

Аналогично можно описать функции типа MAP:

(MAPCAR f Q x)=(LIST11 NIL CONS f(CAR x)Q x)

(MAPLIST f Q x)=(LIST11 NIL CONS F Q x)

где F=f(w(x))

В (2) и (3) с помощью Q происходит просмотр списка нужными шагами, а элементы для обработки подготавливаются с помощью f. Более удобным представляется наличие пары для Q функций Q^0, которая определена как: Q

Если Q=CDR, то Q^0=CAR

Если Q=CDDR, то Q^0=LIST(CAR,CADR))

Если Q=CDDR, то Q^0=LIST(CAR,CADR,CADDR) и т.д.

Тогда можно определить новые формы:

```
<DE LIST111(a g f Q x)(COND((NULL x)NIL)
                          (T(APPLY* g(APPLY f(APPLY Q^0 x))
                                   (LIST111 a g f(APPLY Q x>
<DE LIST211(a g f Q x)
       (COND((NULL x)NIL)
            (T(LIST211(APPLY* g(APPLY f(APPLY Q^0 x))a)
                              g f (APPLY Q^0 x>
```

3.3. рекурсивные формы для N аргументов.

Теперь рассмотрим рекурсивные формы обработки N аргументов, каждый из которых является списком. Рассмотрим функцию TRARG, которая из N аргументов получает один список, являющийся дальнейшем аргументом для LIST1 и LIST2.

```
<DE TRARG L(SETQ RN O)
     (MAPLIST(CAR L)'(LAMBDA L)(SETQ RN(ADD1 RN))
                      (MAPCAR L '(LAMBDA(H)(CAR NTN H RN>
```

Если задан

(TRARG '(x_1 x_2 ...x_n)'(y_1 y_2 ...y_n)'(z_1 z_2 ...z_n)) ⇒

((x_1 y_1...z_1)(x_2 y_2...z_2)...(x_n y_n...z_n))

Но не всегда требуется "хранить" все элементы аргументов, тогда используется TRARG*.

```
<DE TRARG* L(MAPLIST RRN
                '(LAMBDA(x)(MAPCAR L '(LAMBDA(H)(CAR NTH H x>
```

где в RRN -хранит номера нужных элементов в списке. Более осложняется задача, если для обработки списков требуется не одинаковые номера элементов. Если задана закономерность выделения элементов из списков F=(F1 F2 ... Fn), где F1 играет роль Q для первого списка, F2 – для второго списка и т.д., тогда можно определить новую функцию:

```
<DE TRARG(F.L)
      (APPLY 'TRARG(MAPCAR L
                     '(LAMBDA(x)(PROGN(MAPLIST x 'CAR(CAR F))
                                      (SETQ F(CDR F>
```

ISSN 1512-1232

Если для выбора нужных элементов требуется более сложный аргумент, то это можно задать двумя способами:

(а) с помощью списков номеров;

(б) или с помощью двух F и F^0, где первый элемент F списка действует над первым элементом, как Q^0, как Q из LIST111 и LIST211.

Если требуется выбрать из первого аргумента нечетные элементы, из второго четные номера, то тогда для (а) требуется список: ((1 3 5 ...)(2 4 6...)...)

для (б):F=(CAR CADR ...) F^0=(CDDR CDDR...)

Как видно (б) более реально, так как этим не ограничиваем количество аргументов. Для (б) соответствующая функция будет иметь вид:

```
<DE TRARGEF (F F0 L)
     (APPLY 'TRARG (MAPCAR '(LAMBDA(x)
                        PROGN(MAPLIST x(CAR F)(CAR F0))
                   (AND(SETQ F(CDR F))(SETQ F0(CDR F0))
```

Теперь можно написать новые формы рекурсивности для N аргументов:

```
<DE LISTN1(a g f F F0 L)
      (LIST11 a g f Q (TRANGEF F F0 L>
<DE LISTN2(a g f F F0 L)
      (LIST21(a g f Q(TRARGEF F F0.L>
```

Ясно, что тут нам приходится обрабатывать лишнюю информацию, так как из N аргументов вначале вырабатываем один аргумент, а потом над этим аргументом работают LIST11 или LIST21. Более эффективным представляется наличие новых функций:

```
<DE LISTN11(a g f F F0.L)
          (COND((MEMBER NIL L)A)
               (T(APPLY* g(APPLY f(M F L))
                         (LIST11 a g f F F0.(M F0 L>
<DE LIST21(a g f F F0.L)
          (COND((MEMBER NIL L)a)
               (T(LISTN21(APPLY* g(APPLY f(M F L))a)
                                 g f F F0.(M F0 L>
```

где

```
<DE M(F L)(AND L(CONS(APPLY*(CAR F)(CAR L))(M(CDR F)(CDR L>
```

Таким образом, каждую с N-аргументами рекурсивную функцию обработки списков можно представить, как:

```
<DE FUN(F F0.L)
    (COND((MEMBER NIL L)a)
         (T(g(f(M F L))
              (APPLY 'FUN(CONS 'F(CONS 'F0(M F0 L>
```

или

```
   <DE FUN(A F F0 L)
          COND((MEMBER NIL L)A)
              (T(FUN(g (f(M F Ш) a)(F F0.(M F0 L>
```

Здесь, ограничение налагается на F-список, который должен обязательно содержать хотя бы один элемент, являющийся функцией CDR –типа.

4. Применение в задачах верификации программ

Мы определили две универсальные формы рекурсивных функций для обработки LISP - овских списков. Они были предложены для синтеза программ, но можно их также использовать для верификации программ.

ISSN 1512-1232

Современные направления в области верификации - формальные спецификации и методы доказательства их правильности. Для доказательства того, что спецификация программы задает правильное решение некоторой задачи, для которой она разработана, привлекается математический аппарат.

Языки спецификаций, используемые для формального описания свойств программ, более высокого уровня, чем языки программирования. Их можно классифицировать по таким категориям: универсальные языки с общематематической основой (например, RAISE, Z, API, VDM и др.); языки спецификации проблемных областей (например, ЯП, языки спецификаций ПрО или доменов - DSL и др.); специализированные языки спецификации (например, языки таблиц, логики, равенств и подстановок и др.); языки, ориентированные на спецификацию параллельных процессов (например, CIP-L, Ada-68 Concurrent Pascal и др.).

Альтернативный подход этого – а) создание для каждого конкретного языка программирования образца обобщенной, абстрактной программы и б) утверждение ее правильности. Тогда ту программу, для которого нужна верификация, нужно перевести до такого образца. В этом случае уже не требуется верифицировать эту программу, так как она представляет частный случай правильной программы.

Мы предлагаем следующий алгоритм для функционального языка LISP:

А). Для данного языка программирования создать т.н. образец универсальной функции (или функций), который даст нам возможность представить на нем все виды рекурсивных функции. После этого нужно совершить верификацию самой универсальной функции. Для этого нужно использовать методы индукций [9,10,11] - нужно определить мерку, которая отражает аргументы на натуральные числа (в пределах данного языка).

Б) Нужно создать преобразователь конструкций, т.е. аппарат, с помощью которого конструкций одного вида, в частности циклы и условные выражения перейдут в рекурсивные конструкции. Если преобразователь конструкции будут универсальными, то мы можем использовать их и для нефункциональных языков, что, со своей стороны станет возможным использовать данный алгоритм и для других языков (для языков другого типа).

Г) Если та функция, для которой нужна верификация, является рекурсивной, то она будет представлена в обобщенную форму и ее верификация произойдет автоматический. Если данная функция содержит циклы или условных операторов, то она перейдет с помощью преобразователя конструкции (описанный в пункте 2) в рекурсивную и следовательно в обобщенную форму и проверку ее правильности. В [10]-описана реализация этого алгоритма. Мы думаем, что в этом процессе допустимо осуществить диалог с целью получения дополнительной информации. В [11] представлены циклы с помощью простой рекурсии, а встроенные циклы представлены межрекурсивом и рекурсивом высокой степени.

Мы рассмотрели для функционального языка LISP абстрактную форму рекурсивной функций и вопросы ее верификации. Отметим, что будет полезным использовать диалоговый режим в процессе, когда происходит преобразование программы на абстрактную форму.

ISSN 1512-1232

Литература

1. Вудкок Д. Первые шаги к решению проблемы верификации программ. Открытые системы . – 2006 . – № 8. – С. 36-43.
2. Методы и средства инженерии программного обеспечения. Интернет-Университет Информационных Технологий. http://www.INTUIT.ru
3. Арчвадзе Н.Н. Пховелишвили М.Г. Шецирули Л.Д. Комплекс Производства Программ для функциональных языков. System Analysis and Information Technologies: Materials of the XI International Conference on Science and Technology. -ESC "LASA" NTUU "KRI", 2009, p.453.
4. Арчвадзе Н.Н. Пховелишвили М.Г. Шецирули Л.Д. Комплекс Производства Программ для функциональных языков. s4102.pdf — PDF document, 196Kb. Презентация доклада. http://sait.org.ua/eproc/2009/4/s4102.pdf/view
5. ნ.არჩვაძე, ლ.შეწირული. პროგრამირების ენა LISP (სახელმძღვანელო). "უნივერსალი", თბილისი, ISBN 978-9941-12-258-3, 2008, გვ.145.
6. Приобратение знаний. Под ред. С.Осуги, Ю.Саэли. Москва, «Мир», 1990, с.303.
7. Н. Катленд. Вычислимость. Введение в теорию рекурсивных функций. М."Мир",1983.
8. В. Бердж. Методы рекурсивного программирования. М., "Машиностроение", 1983.
9. Archvadze N., Pkhovelishvili M., Shetsiruli L . Problems of Verification of Functional Programs. Bulletin of the Georgian Academy Sciences. Bulleten of the Georgian National Academy of Sciences. vol.3. no.3, 2009. pp 16-19
10. Archvadze N., Pkhovelishvili M., Shetsiruli L., Nizharadze. A recursion forms and their verification by using the undictive methods. Computing and Computational Intelligence. Proceeding of the 3nd European Computing Conference (ECC'09), Tbilisi, 2009, pp.357-361. http://www.wseas.org/conferences/2009/tbilisi/Program.pdf
11. N. Archvadze, M. Pkhovelishvili, L. Shetsiruli, M. Nizharadze. Program Recursive Forms and Programming Automatization for Functional Languages. WSEAS TRANSACTIONS on COMPUTERS. Volume 8, 2009. ISSN: 1109-2750. pp. 1256-1265. http://www.wseas.us/e-library/transactions/computers/2009/29-531.pdf

Статья получена: 2010-02-13