Program Recursive Forms and Programming Automatization for Functional Languages

N.ARCHVADZE
Department of Computer Sciences
Faculty of Exact and Natural Sciences
I. Javakhishvili Tbilisi State University
2, University st., 0143, Tbilisi
GEORGIA
natarchvadze@yahoo.com http://www.tsu.ge

M.PKHOVELISHVILI
Department of Programming
N.Muskhelishvili computing mathematic Institute
7, Akuri st., 0193, Tbilisi
GEORGIA
merab5@list.ru http://www.acnet.ge/icm

L.SHETSIRULI
Department of Mathematics and Computer Science
Shota Rustaveli State University
35, Ninoshvili st., 6010, Batumi
GEORGIA
lika77u@yahoo.com http://www.bsu.edu.ge

M.NIZHARADZE
Faculty of Informatics and Control Systems
Georgian Technical University
77 Kostava st., 0175, Tbilisi
GEORGIA
mziana60@mail.ru http://www.gtu.ge

Abstract: - The automatic programming system has been considered by means of which it becomes easier to carry out traditional programming stages. There is discussed both recursive forms: parallel, interrecursion and recursion of high level that exist for functional programming languages and induction methods for the purpose of their verification. The way how to present imperative languages easy and double cycles by means of recursion forms is shown, the possibility of verification has been studied for each recursion form.

Key-Words: - Functional Programming Languages, Recursive Forms, Programs Verification.

1. Automatic Programming System (APS)

1.1. Task of Automatic Programming

Development of computational technics fails to provide problem solving technology elaboration. First-hand user fails to draw make up a program independently depending on his/her knowledge (without transforming into formal algorithm language by the programmer). Because of this reason and the critic towards the possibilities of procedure programming the informatics intellectual, automatic programming direction is developing. We suggest the automatic programming system (APS) [1], that should automatize all the processes starting with writing the program until the eventual fixing up. APS concerns only functional programming languages. These languages reached in their

developing process such results that they could be considered as alternatives for traditional languages. First of all APS is drawn up for languages working with lists. As it is known the most practiced language in programming is LISP for working out lists. In this language application of recursion is allowed. By means of well-selected recursive functions any lists could be worked out. It is not difficult to prove that drawing up recursive functions representing lists in other lists are computable [2] ones. Hence our first problem concerns recursion in the functional languages. The second problem is verification that is one of the basic tasks of automatic, intellectual programming. Programmers know experience how difficult it is to make correct program, i.e. such one that does exactly what is required. In most bulky programs there errors consequences of which could be most various. The deeper penetration of computers into all spheres of life the more serious is the possible threat to the life op people as a result of program errors. Hardware is corroborated with similar problems. We consider exclusively software [3]. One of APS system application is possible in el teaching E-learning. Here programming languages are taught [4, 5].

1.2. APS structure

APS includes five modules: a) program synthesizer, b) tester, c) proof testimony to the correctness of program (program verification), d) optimizing processor, e) corrector. Each of these modules could be functioned independently; various combinations of these modules are allowed.

Automatic Synthesizer of programs should draw up programs according to target models. Models are given through examples or verbal description, or annotations. Annotation consists of three parts: input clause, output clause and cycle invariancy.

Tester is intended for semiautomatic testing of programs. On the one hand an attempt is being made proceeding from the program description to choose the required tests without the user, on the other hand to allow his/her help in a dialog regime.

Verification of the program tries to prove irregularity of the resulting program. If it works well then controlling goes to the Corrector. In case of fail the attempt to prove its correctness takes place; for that the method of induction is applied. If irregularity and non- irregularity of the program is not proved then processor requires additional descriptions in a dialog regime.

Optimizing Processor is intended for optimizing separate parts of the program. Here is included the transformation synthesis block. The basic function of which is translation of recursive programs into interactive ones. In optimizing processor clearing of cycles, joining cycles, etc are considered as well.

Complex of Program Production (CPP) is intended for functional languages. First of all it is drawn up for Lisp. For that basic forms of recursive functions are applied. We speak about them later.

The general scheme of the CPP is presented as follows:

1. The required program is synthesized (not necessarily correct);
2. By means of tester the errors are found, if failed then shift to 4;
3. Corrector corrects errors and shifts to 2;
4. Attempt to prove correctness of the program takes place. If irregularity is proved then shifting to 3;
5. Optimization of the program.

2. Recursive forms in the functional programming languages

2.1. Peculiarity of Functional Programming

The repeated calculation in the functional programming is made with the recursion help, which is not only the calculation organization base means, but it is the thinking shape in the functional programming and the methodology of the task decisions.

At the recursive description action should be payed attention to sharpening at the following: firstly the procedure must contain at least the one terminal branch and the finishing condition; secondly, when the procedure comes to the recursive branch, then the functional process is stopped and the same process starts again. The remembering of the interrupting process occupied, which will be done after the new process is finished. The new process may be stopped as well and will be have to wait for the other process execution and so on.

Thus the interrupted processes stack is made, from which the last process is realized for the time given. After the finishing the previous process is realized. Then, as soon as stack empties, the whole process is executed or all interrupted processes will be done.

A function is the general recursive, if its algorithmic description is impossible. Such function calculation is possible infinitely. The general recursive function example is *(f n m)* function, result of which is 1, if in π-number decimal record is a fragment with *m* ciphers and

n length found.

A recursion is simple, if the function is called in any branch only once.

The recursion differs by meaning and argument. It is considered that a recursion is meaning, when the call is by the image, which is presented IN the function result. It the function result as other function meaning is returned and the recursive calculation is in this function argument, then say, that the recursion is by argument. The recursive call argument may be again recursive call and so on. For example, will be define the function ***APPEND*** , with help of which is made the unification of two lists. One can to notice, that all functions will be presented in the functional programming language COMMON LISP.

```
(defun APPEND (x y)
  (cond((null x) y)
     (t(cons(car x)
          (APPEND (cdr x) y )))))
```

There is used the recursion by argument. ***APPEND*** may be definite by the recursion meaning:

```
(defun APPEND (x y)
  (cond((null x) y)
     (t(APPEND ( cdr x)
          (cons(car x) y )))))
```

This definition differs by this that a result is constructed direct in the second argument.

2.2. The recursions form

In the functional languages the following form recursions are determined [8]:

1) The parallel recursion, when the F-function definition includes the definite G-function call, one or some arguments of which recursively call F-function:

```
(defun f ...
  ...(g...(f...)...(f...)...)
  ...)
```

2) Inter-recursion, when in F-function definition the definite G-function IS CALLED, which includes the F-function call:

```
(defun f ...
  ...(g...)...)

(defun g ...
  ...(f...)...)
```

3) The recursion of high level, when the recursive call argument is the recursive call:

```
(defun f ...
  ...(f...(f...)...)
  ...)
```

2.2.1. Parallel recursion

A recursion is parallel, if it is in several arguments of the function simultaneously.

Let definite function ***COPY***, the result of which is the argument copy. Copy of the list is the primary list.

```
(defun COPY (l)
    (cond((null l) nil)
                    ; The finishing condition
       (t(cons(car l)   ; The recursion
          (COPY(cdr l))))))
```

This is the function with recursive argument since the recursive call is the function ***CONS*** argument. A conditional operator includes two branches: the branch with finishing condition and the one with recursion, by help of which a list passes, is copied and shortened in this process in the ***CDR***-direction.

The considered function ***COPY*** list is copied in the ***CDR*** direction only in the upper level. When it is necessary to copy list both in CDR and car directions, the recursion should spread on the sublists as well. So is obtained the ***COPY-TREE*** function:

```
(defun COPY-TREE(l)
  (cond((null l)nil)
     ((atom l)l)
     (t(cons
        (COPY-TREE
                    ; copy of the head
          (car l))
        (COPY-TREE    ; copy of the tail
           (cdr l))))))
```

In this function the recursion is used both in the list head and the list tail. Since could be called recursively two arguments one function (cons) of it is parallel recursion. The parallelism means only textual and not temporal, though parallel recursive functions may be very naturally treated on the multiprocessor computer, if the recursion each branch will be calculated on the separate processor.

2.2.2. Inter-recursion

The recursion between two or more functions is called inter-recursive (mutual), if they call each other. For example, let us discuss function ***REVERSE***, which turns over the list:

```
(defun REVERSE(l)
  (cond((atom l)l)
       ((t(REARRANGE l nil))))
(defun REARRANGE (l result)
  (cond((null l) result)
       (t(REARRANGE (cdr l)
            (cons(REVERSE(car l)) result)))))
```

Function ***REVERSE*** is used as the auxiliary function with additional parameters. At the turned list construction it cares of the sub lists to be turned. It does not do it itself, but instructs function REARRANGE. Besides ***REARRANGE*** participates in the inter-recursion, it is recursive itself.

2.2.3. High level recursion

Let us consider the nested cycles programming by the form, during which the function definition recursive call is the same function argument. At recursions of that type orders of various level, could be distinguished according to which level of recursion the call is at. This kind of recursion is called the one of higher level.. The considered so far functions were of zero level.

The high level recursion classic example is Akkerman's function, which recursively is determined for *m* and *n* numbers as follow way:

$$A(m,\ n) = \begin{cases} n+1, & m = 0; \\ A(m-1,\ 1), & m > 0,\ n = 0; \\ A(m-1,\ A(m,\ n-1)), & m > 0,\ n > 0. \end{cases}$$

Let us reduce Akkerman's determination to the LISP:

```
(defun AKKERMAN(m n)
  (cond((= m 0)(+ n 1))
       ((= n 0)(AKKERMAN (- m 1)1))
       (t(AKKERMAN (- m 1)
          (AKKERMAN m (- n 1))))))
```

Akkerman's function is recursive one of the first order. Its calculation is rather complicated, the calculation time grows very even for the small arguments.

The first order recursion another example is the function IN-ONE-LEVEL, which places the list's elements on one level:

```
(defun IN-ONE-LEVEL (l)
   (LEVEL l nil))
(defun LEVEL(l result)
   (cond((null l) result)
        ((atom l)(cons l result))
        (t(LEVEL (car l)
             (LEVEL (cdr l))))))
```

In this function the recursive call argument is the recursive call. By recursions of high order the function determination could be recorded more abstractly and shortly though it is rather difficult to present such work.

The function ***LEVEL*** works as follows: A result is composed in the list *RESULT*. If *l* is the list and its first element is an atom, then all is reduced to the recursion previous level, but in a situation, when the list *RESULT* contains the list tail, which in one level lined up is already.

In case, when the *l* list head is again a list firstly it reduced one level. This happens by recursive call, which lasts as long as an atom is found, which will be added to already lined up list.

The function ***REVERSE*** following explanation is an example of deeper recursion

```
(defun REVERSE(l)
  (cond
    ((null l)l)
    ((null(cdr l))l)
    (t(cons
        (car(REVERSE(cdr l)))
        (REVERSE(cons(car l)
           (REVERSE(cdr
           (REVERSE(cd r l)))))))
```

In the definition the second order recursion is used. To understand these calculations is very difficult. In general it is possible to avoid such calculations, if the definitions are divided in some parts and the corresponding parameters are used for the intermediate results keeping and passing.

3. Presentation of imperative languages cycle operators by functional programming languages recursion

3.1. Presentation of simple cycles

In imperative languages the cycle operators

belong to the ruling operators and they are used for the repeated activity presentation. Our aim is to present them by the recursive forms which are exercised in the functional programming. This allows to apply the function verification means not only for the functional languages, but also for the imperative type of languages.

A cycle is a group commands implementation of which is repeated as long as, the cycle continuation condition is true. The repetition takes place either by the special counter (there the repetition quantity is known) or the controlling meaning (the rate is unknown beforehand).

Simple recursion corresponds the imperative languages cycle operators. The examples on **C** are given. Using the counter by means of the ***while*** cycle operator the program is:

```
#include <stdio.h>
int main()
{ int counter=1;
  while (while counter<=10) {
       printf("%d \n",counter);
         ++counter;        }
  return 0;  }
```

Using the counter with ***for*** structure cycle will be:

```
#include <stdio.h>
int main()
{ int counter;
  for (counter=1;counter<=10;counter++)
      printf("%d \n",counter);
  return 0; }
```

Both of these cases are reduced to the shape after the simple recursion where ***l*** argument is a number (on ***LISP***):

```
(defun f(l)
   (cond((= l 1)(print l))
      (t(f(- l 1)))))D
```

In general,

```
(defun F(l)
   (cond((PR  l) (A l))
      (t(F(B l)))))
```

Where ***l*** is the counter with the given primary meaning, ***A*** - the function, which is implemented for the counter's given significance (or the action, which is repeated in the cycle), ***B*** is the function, which changes the counter's significance (for example, reduces by 1), but ***PR*** is the predicate, the true of which defines the cycle interrupting condition. So ***F*** function repeats on itself the recursive address by the argument, which means changed the significance by ***B*** function for a time, until the argument reaches the meaning when function PR is true.

It is clear, that the recursion of this form works not only for the numeral argument, but also for complicated data: for the massifs and lines. In this case ***1*** will be a list but function ***B*** will be ***CDR***-function:

```
(defun F(l)
   (cond((null l) (A l))
      (t(F(cdr l)))))
```

As to the cycle operator with the control significance (the function ***PR1***), it will be also presented by the simple recursion, where ***F1*** is the working function, on the argument that simplifies it

```
(defun F1(l)
    (cond((PR1 l)(A1 l))
       (t(F(F1 l)))))
```

3.2. Nested cycles presentation

The existed nested cycles in the imperative languages in the functional programming languages in general could be realized by means of two or more functions, out of which each corresponds the simple cycle. Such recursive function call will be the recursive call argument in the other function. It is natural, that in the function definition the recursive call argument may be the other recursive call. It is recursion of higher level.

First let us consider the nested cycles programming by means of two different functions (inter-recursion). The nested cycles may be expressed by the cycle sentences (***DO***, ***LOOP*** and etc.) or by the specialized repeatable functions (for example the function ***MAP***).

Let us consider the nested cycles programming on the example of the lists sorting. At first the function ***INSERT*** should be determined, which adds element ***a*** in the arranged list ***l*** so, that the distribution remained. At this time any two element row is defined by predicate ***EARLIER-P:***

```
(defun INSERT (a l order)
   (cond((null l)(list a))
```

```
((EARLIER-P a(car l) order)
   (cons a l))
 (t(cons (car l)
      INSERT a (cdr l) order)))))
```

The predicate ***EARLIER-P*** controls if ***a*** element is before ***b***-element according to their disposition in the list ***order***:

```
(defun EARLIER-P (a b order)
  (cond
    ((null order) nil)
    ((eq a(car order))t)      ; a earlier b
    ((eq  b(car order))nil)   ; b earlier a
    (t (EARLIER-P a b (cdr order)))))
```

Functions ***INSERT*** and ***EARLIER-P*** are two level nested iterative structures.

The list, which is not arranged, may be done by function ***SORT***, which recursively puts the list's first element in the corresponding place in the beforehand arranged list tail.

```
(defun SORT (l order)
  (cond((null l)nil)
    (t(INSERT (car l)
        (SORT (cdr l)order)order))))
```

Functions ***SORT, INSERT*** and ***EARLIER-P*** already make three level nested structure.

Thus the cycles may be presented by simple recursion, but the nested cycles by the inter-recursion or the recursion of higher level.

4. Creation of universal forms of recursive function presentations for functional programming languages

Our purpose is to create such forms of recursive function presentations that will be general, in particular, for functional programming languages Lisp functions.

4.1. Recursive function general forms for Lisp

In [8] the common forms of recursive functions are planned allowing working out the lists. Let us present these functions in the Lisp language. They will look as follows:

```
<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)
              g  f(CDR x>
```

Proceeding from this any recursive LISP function working out arbitrary x-list could be presented as follows:

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

or

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

In [6] recursive function forms are given; they are more limited than (1). For instance, in f=

$$\begin{cases} \text{if } x = \text{NIL}, \textit{then} \text{NIL} \\ \text{else } \text{CONS}(\alpha(\text{CAR}(x)), \text{F}(\beta(CDRx)) \end{cases}$$

G and A from (1) are more concrete:
G=CONS, A=NIL, a f=α и I=β.

Here, I is reflection
(DE I(x)x)

Let us consider the example describing REV functions it turns the list upside-down

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

Substituting AND for COND expressions we get:

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

If compared this with (1) form the discrepancies appear in constructions:

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

Here function APP is determined as follows:

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

It should be said that LIST1 and LIST2 forms do not exhaust all recursive functions of working out of lists. For that we consider PLIST

function which unites neighboring pairs of elements:

(PLIST '(x1 x2 x3 x4 ...x2n-1 x2n))⇒

((x1 x2)(x3 x4)...(x2n-1 x2n))

This function is as follows:

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

If compared this with (1) form the discrepancies appear in constructions:

F(CAR x)≠(LIST(CAR x)(CADR x))

(CDR x)≠ (CDDR x)

If PLIST is presented by means of MAPLIST function:

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

And MAPLIST could be presented:

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

4.2. Generalization of presentations

Previous descriptions prompt more perfect working out of lists for recursive functions:

```
<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>
<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>
```

By help of LIST11, PLIST could be described

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

where *<DE FF(x)(LIST(CAR x)(CADR x>*

Analogically MAP-type functions could be described:

```
(MAPCAR f Q x)=(LIST11 NIL
                CONS f(CAR x)Q x)
(MAPLIST f Q x)=(LIST11 NIL
                CONS F Q x)
```

where *F=f(w(x))*

In (2) and (3) review of list takes place by means of Q and elements for working out are getting ready by means of f. More handy is presence of the pair for Q function Q^0 that is determined as Q

If *Q=CDR*, then *Q0=CAR*

If *Q=CDDR*, then *Q0=LIST(CAR,CADR))*

If *Q=CDDR,*

then *Q0=LIST(CAR,CADR,CADDR)*

and etc.

Then new forms could be determined:

```
<DE LIST111(a g f Q x)
(COND((NULL x)NIL)
        (T(APPLY* g(APPLY
          f(APPLY Q0 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 Q0 x))a)
            g f (APPLY Q0 x>
```

4..3. Recursive forms for N arguments

Now we discuss recursive forms for N arguments working out forms. Each of them is a list. We discuss function TRARG that receives one list out of N arguments. They are further arguments for LIST1 and LIST2.

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

If is planned

(TRARG '(x1 x2 ...xn)'(y1 y2 ...yn)

'(z1 z2 ...zn)) ⇒

((x1 y1...z1)(x2 y2...z2)...(xn yn...zn))

Not always required "keeping" all elements of arguments; then TRARG* is applied.

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

where in RRN – numbers of necessary elements in the list are kept. The task is more complicated if not identical numbers of elements are required for lists working out.. If the conformity with law of element distinguish from lists is targeted F=(F1 F2 ... Fn), where F1 plays the part for Q for the first list, F2 – for the second list, etc. then the new function could be defined:

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

If more complicated argument is required for choosing required elements two ways are possible:

a) by means of number lists;

b) or by means of two F and F^0, where the first element of the list F acts on the first element as Q^0, as Q from LIST111 and LIST211. If it is required to choose from the first argument odd elements, from the second even numbers, then for (a) the list is required:

((1 3 5 ...)(2 4 6...)...)

for (b):

F=(CAR CADR ...) F^0=(CDDR CDDR...)

As it is seen (b) is more real as the quantity of arguments is not limited with it. For (b) the corresponding function is as follows:

```
<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))
```

Now new forms of recursiveness could be recorded for N arguments:

```
<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>
```

It is clear that we have to work up extra information since we work up one argument first from N arguments and then LIST11 or LIST21 work on this argument. Presence of new functions is more effective:

```
<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>
```

where

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

Thus each recursive function of list working up with N arguments could be presented as:

```
<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>
```

or

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

Here limit is imposed on F list which must contain at least one element being the CDR-type function.

Thus we have determined two universal forms of recursive functions for working up LISP lists.

5. POSSIBILITIES OF FUNCTIONS VERIFICATION PRESENTED BY RECURSION DIFFERENT FORMS IN THE FUNCTIONAL LANGUAGES

We consider that for program verification it is necessary to apply structural and transfinite methods of induction of proofs. The proof methods are applied for recursive functions which arguments are numbers according which changes take place.

Let us discuss how to prove recursive function verification of various type discussed by us.

5.1. Verification of Parallel Recursive Functions

In the programs verification proofs the transfinite induction and structural methods are considered. The proofs methods is used for the recursive functions, the arguments of which are numbers and by which are made the changes.

The parallel recursive functions verification may be realized by the structural verification method. It is meant that the recursive functions arguments are not numbers, but structures and an induction is realized by the list length. Such programs correctness may be proved as follows: a) let us prove that the program works corrects for simpler data (for the empty list). b) Let us prove, that the program works correctly for more complicated data (for N+1 length list) with the admittance, that it works correctly for comparatively simple data (for N length list)

5.2. Verification of interrecursive functions

The interrecursive functions verification may be realized by using transfinite induction method. The transfinite induction method is the proof method, which is the mathematical induction generalization for the parameters non-numerical signification. It is as follows: it is necessary to define the necessary for it dependence, and such, that passes ahead of previous calls before further recursive calls - this gives the algorithm finishing guarantee.

The interrecursive functions differ from some recursive functions, which call each other. The argument of each function is the number or a list. Let us define the size, which is equal to sum of all these function argument lengths. Let us consider this size as the transfinite induction measure. Then the induction method has following shape: we verify the interrecursive functions correctness for the arguments with the zero length. Later it is supposed, that the functions are correct, when arguments sum is N length and we try to prove for $N-1$ length, as when used with structural induction. The induction step decreases, if the lengths sum decreases.

5.3. Combined induction technique

Combined induction technique is used for verification for such complex functions whose arguments are again the recursive functions. For example, for the functions of the following type:

$\varphi (\varphi_1(), \varphi_2(), \dots , \varphi_N ( ))$, where each argument $\varphi_1(), \varphi_2(), \dots, \varphi_N ( )$ is the recursive function.

A special case is the cases with mixed version which are of great interest: a) the case when each function $\varphi_1()$, $\varphi_2()$, ..., $\varphi_N ( )$ is numerically stated and its recurrent values are also numbers, while the function φ is operable on lists, and b) the case when each function $\varphi_1()$, $\varphi_2()$, ..., $\varphi_N ( )$ is list-operable, while φ is operable on numbers. In this case, separate use of the proofs induction and structural induction techniques could cause errors; therefore, their combined techniques should be used. For example, we have a composition of functions $\varphi (\varphi_1(L), \varphi_2(L))$, where $\varphi_1(L)$ is the function that computes the maximum elements of L list, $\varphi_2(L)$ – computes the minimum, while φ circulates the simple mean of these arguments. In such a case, proof of $\varphi_1(L)$ and $\varphi_2(L)$ functions correctness is proved by using first the structural induction technique and then the proof-by-induction technique.

In the composition of functions, account should be taken that the computation of values of the functions takes place from left to right, when the S-expression is rather straight than parallel. This can cause a number of errors, for example, in case when each function in the composition is database-operable and replaces the data in the base.

5.4. Special program verification technique

The program verification history is known for the cases when special languages used to be created on purpose to simplify verification of the programs written in them. An alternative approach to this is the necessity that a generalized, abstract program for the given

programming language be created and verified its accuracy. It is necessary that the program needing verification be automatically transferred to the form of an abstract program. In this case, it will not be subjected to verification (because it will already be a particular form of a correct program).

The problem of the program verification process automation in programming has retained its urgency up to this day. In respect of the functional language LISP, this problem must be reduced to any program abstract form and then to its further automatic verification.

We offer the following verification algorithm:

a) To make the so-called universal function (or functions) for the given programming language of which will make it possible to present the recursive function of any type written in this language. After that the universal function verification should take place. For this purpose the above-mentioned induction techniques will be used, an attempt will be made to define the size (reflecting natural number arguments) finding algorithms, certainly, within the bounds of the given language.

b) To create construction transformers, i.e. the apparatus thanks to which constructions of one form, in particular, loops and conditional representations, will be transformed into recursive construction one. If the transformers are universal, then it will be possible to transfer loops and conditional representations to a recursive form for the non-functional languages which will, in their turn, make it possible to use the given algorithm for other languages as well.

c) If the given function that needs verification is recursive, then it will be transformed into a after which its accuracy will be proved automatically. If the given functions contain loops and conditional statements, then it will automatically transfer to the form of recursive and correspondingly universal functions form by means of the transformer described in paragraph (b) above. Thereafter, its accuracy will be proved.

6. Conclusion

Thus we have discussed recursive functions verification process automatization problems. We discussed functional programming language for LISP abstract program image and by applying proofs induction, structural and transfite induction methods to prove the recursive function correctness. We should say that it would be desirable to arrange a dialog rate in the process when the program adjusting to abstract program form takes place. Analogical forms could be determined for non-list expressions since imperative languages controlling operators (cycles) could be presented my means of discussed recursive forms.

References:

[1] Archvadze N., Pkhovelishvili M., Shetsiruli L. Complex of Production of programs for functional languages System Analysis and Information Technologies: Materials of the XI International Conference on Science and Technology. -ESC "LASA" NTUU "KRI", 2009, p.453.

[2] N. Katlend. Vichislimost. Vvedenie v teoriu rekursivnikh funkstii. M., “Mir”, 1983.

[3] J.Rutkowski, D.Use of Artificial Intelligence Techniques to Fault Diagnosis in Analog Systems. 2nd EUROPEAN COMPUTING CONFERENCE (ECC’08). Malta, September 11-13, 2008. pp. 267-274.

[4] E.Doicary, C.Dan. E-TLSS – A Powerful E-Learning Tool for Structural Synthesis of TL Circuits. 2nd EUROPEAN COMPUTING CONFERENCE (ECC’08). pp.173-178.

[5] F. De Arriaga, C. Gingell, A. Arriaga, J. Arriaga, F. Arriaga. A General Student’s Model Suitable for Intelligent E-Learning Systems. 2nd EUROPEAN COMPUTING CONFERENCE (ECC’08), pp. 167-172.

[6] Archvadze N., Pkhovelishvili M., Shetsiruli L. The Methods of the Effective Date Search for the Listed Structures. International Conference on Modern Problems in Applied Mathematics Dedicated to the 90th Anniversary of the Iv/Javakhishvili Tbilisi State University and 40th Anniversary of the I.Vekua Institute of Applyied Mathematics. Book of Abstracts. Tbilisi, 2008. p.16.

[7] V. Berdj. Metodi rekursivnogo programiro-vania. M., “Mashinostroenie”, 1983.

[8] 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.