A recursion forms and their verification by using the inductive methods

N.Archvadze
Department of Computer Sciences
Faculty of Exact and Natural Sciences
I. Javakhishvili Tbilisi State University
2 University st., 0143, Tbilisi
GEORGIA
nataarchvadze@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.Sheetsiruli
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: It is considered both the recursion forms which exist for the functional languages, and the inductive methods with the goal of their verification. It is shown how the simple and double cycles of imperative languages may be presented by the recursive forms; the verification possibility is studied for each recursive forms.

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

1 Recursion forms in the functional programming languages

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

The action’s recursive description needs the attention to be paid on the following: first, the procedure must contain in extreme case one terminal branch and the finishing condition; second, when the procedure comes to the recursive branch, then the functional process is stopped and such process again is started. The interrupted process is remembered which will be carried out after the new process is finished. Also the new process may be stopped and will wait for the other process execution and so on.

The interrupted processes stack is created this way, from which the last interrupted process for that time is realized. After the finishing its previous process is realized. The whole process is executed then, when stack is emptied, or the all interrupted process will be made.
In a [1] some explanation is introduced. A function is the general recursive if its algorithmic description is impossible. Such function’s calculation is possible infinitely. The general recursive function example is \((f \ n \ m)\) function with result of 1 if in \(\pi\)-number decimal record will be found a fragment with \(m\) ciphers and \(n\) length.

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

Recursion is differentiated based on the meaning and the argument. It is considered that a recursion is with the meaning when the calling is by the image which represents the function result. If the returned function result is the meaning of other function and the recursive calculation is in this function argument, then they say that the recursion is by argument. The recursive call argument may be again recursive call and so on. For example, let’s define the function \(APPEND\) with help of which is made the unification of two lists. One can notice that all functions will be presented in the functional programming language COMMON LISP.

\(\text{(defun APPEND} (x \ y))\)
\(\text{(cond ((null x) y)}\)
\(\text{t(cons(car x) \ (APPEND} \text{ (cdr x) y))))\)

Here the recursion by argument is used. \(APPEND\) may be definite by the recursion meaning:

\(\text{(defun APPEND} (x \ y))\)
\(\text{(cond ((null x) y)}\)
\(\text{t(APPEND (cdr x) \ (cons(car x) y))))\)

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

In the functional languages the following form recursions are determined [1]:

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

\(\text{(defun f} \ldots\)
\(\ldots(g\ldots(f\ldots)\ldots)\)
\(\ldots)\)

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

\(\text{(defun f} \ldots\)
\(\ldots(g\ldots)\)
\(\text{(defun g} \ldots\)
\(\ldots(f\ldots)\)

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

\(\text{(defun f} \ldots\)
\(\ldots(f\ldots(f\ldots)\ldots)\)
\(\ldots)\)

1.1 A parallel recursion

A recursion is parallel if it is met simultaneously in the function's same arguments.

Let’s define the function \(COPY\), the result of which is the argument's copy. The copy of list is the primary list.

\(\text{(defun COPY} (l))\)
\(\text{(cond ((null l) nil)}\)
\(\text{t(cons(car l) \ (COPY (cdr l))))}\)

This is the function with recursive argument, so far as the recursive call is the function \(CONS\) argument. A conditional operator includes two branches: the branch with finishing condition and the branch with recursion, with help of which a list is passed, copied and shortened is this process in the \(CDR\)-direction.

The considered function \(COPY\) list is copied in the \(CDR\) direction only on the high level. When it is necessary to copy the list to \(CDR\) and \(CAR\) directions, the recursion should spread on the sublists. The \(COPY-TREE\) function is derived this way. The word \(TREE\) in the function title is connected with the fact that in the function determination the list is pointed pair-binary tree, the left sub tree of which corresponds to the list head, the right-to the tail:

- \(\text{tree} \rightarrow \text{nil}\) ; empty tree
- \(\text{tree} \rightarrow \text{atom}\) ; tree's leaf
- \(\text{tree} \rightarrow (\text{tree. tree})\) ; pointed pair-tree

\(\text{(defun COPY-TREE} (l))\)
\(\text{(cond ((null l) nil)}\)
\(\text{t(cons \ (COPY-TREE} (\text{car l})\)
\(\text{(COPY-TREE} (\text{cdr l))))}\)

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

1.2 Inter-recursion

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

```lisp
(defun REVERSE (l)
  (cond ((atom l) l)
        ((tt (REARRANGE l nil))))
)
```

The function REVERSE is used as the auxiliary function with the additional parameters. At the turned list construction it troubles about that the sub lists will be turned. It does not make this itself, but instructs to the function REARRANGE. Besides the fact that REARRANGE participates in the inter-recursion, itself it is recursive.

1.3 The high level recursion

Let’s consider the introduced cycles programming by such form, during which at the function definition recursive call is the same function argument. For such type recursions we can separate different levels of orders depending on which level of recursion the call is made. Such type of recursion is higher level's recursion. The functions considered till now were zero level functions.

The high level recursion’s classic example is Akkerman's function, which is recursively determined for m and n numbers in the following 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’s put Akkerman's determination on the LISP:

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

The Akkerman's function is recursive function of the first order. Its calculation is sufficiently complicated; the calculation time very quickly grows even for the small arguments.

The first order recursion's other example is the function IN-ONE-LEVEL which places the list's elements on the same levels:

```lisp
(defun IN-ONE-LEVEL (l)
  (LEVEL l nil))
```

In this function the recursive call argument is the recursive call. With the us of he high order recursions, the function determination may be written more abstractly and shortly, although presentation of such function’s work is very complicated.

The function LEVEL works so: A result composes in the list RESULT. If l list and its first element is an atom, then all is reduced on the recursion previous level, but in such situation, when in the one level is lined up.

In such case, when the l list head is again list then firstly it tallies to one level. This is made by recursive call, which is prolonged so long time, till is found an atom, which will be added to already lined up list.

The deeper example of the recursion is the function REVERSE following explanation:

```lisp
(defun REVERSE (l)
  (cond ((null l) l)
        ((null (cdr l)))
        (t (REVERSE (car l)) (REVERSE (cdr l))))
```
several parts and use the corresponding parameters for the intermediate results’ keeping and passing.

2 The presentation of the imperative languages cycle operators by the functional programming languages recursion

In the imperative languages the cycle operators belong to the ruled operators and they are used for the repeated actions presentation. Our aim is to present them by the recursive forms which are received in the functional programming. This gives the ability to use the functions verification tools for not only for the functional languages, but also for the imperative type languages.

A cycle- it is the commands group, the implementation of which is repeated until the cycle continuation condition is true. The repetition is managed by either the special counter (there it is known the repetition quantity) or the significance (in advance we do not know how much will be executed.)

Cycle operators correspond to the imperative languages’ simple recursion. The examples on C are given. By using the counter, by the means of the while cycle operator the program has shape:

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

By using the counter by the means the for cycle operator the program has shape:

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

These two case are reduced after the simple recursion to the shape where \( l \) argument is a number (on LISP):

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

In general,

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

Where \( l \) is the counter with the given primarily significance, \( A \) the function which is implemented for the counter's given value (or the action which is repeated in the cycle), \( B \) is the function which changes the counter's value (for example, by 1 decrease), and \( PR \) is the predictor the trueness of which defines the interrupt condition. So \( F \) function repeats on itself the recursive address by argument, which by means of \( B \) function changes the significance until the value of this argument will be such that \( PR \) function will be true.

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

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

As for the cycle operator with the control value (the function \( PRT \)), it will also be presented similarly by the simple recursion, where working on the argument is \( F1 \) function which simplifies argument:

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

Nested cycles, existing in the imperative languages, in the functional programming languages may be realized in general form with two or more functions, out of which each corresponds to simple cycle. Such recursive function’s call will be in the other function recursive call argument. It is natural, that in the function’s call the recursive call argument may be the other recursive call. It is the higher level recursion.
At first let’s 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 other) or by the specialized repeated functions (for example, the function MAP).

Let’s consider the nested cycles programming on the example of the lists sorting. At first should be determined the function INSERT, which adds element a in the distributed list l so that a distribution remains. In this case, any two elements’ row is defined by the predicate EARLIER-P:

```lisp
(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 predictor EARLIER-P checks whether the a element is before b-element or not, depending how they are disposed in the list order:

```lisp
(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))))))
```

The functions INSERT and EARLIER-P are the two levels nested iterative structure.

The list, which is not tidied, may be regulated by the function SORT, which recursively puts the list’s first element on the corresponding place in the list tail that is put in order beforehand.

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

The functions SORT, INSERT and EARLIER-P make the three-level nested iterative structure.

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

3. THE POSSIBILITIES OF THE FUNCTIONS WHICH ARE PRESENTED BY THE DIFFERENT FORMS OF RECURSION, IN THE FUNCTIONAL LANGUAGES VERIFICATION

[2] discusses the structural and transfinite methods of program verification proofs. The proof methods are used for the recursive functions, the arguments of which are numbers and according to which are made the changes.

The parallel recursive functions’ verification may be realized by the method of the structural verification. It is implied that the recursive functions arguments are not numbers, but the structures and an induction by the list length is realized. Such programs’ truthiness may be proved by following: a) let’s prove that the program works correctly for the more simple data (for the empty list); b) let’s prove, that the program works correctly for the more complicated data (for N+1 length list) with the admittance that it works correctly for the comparatively simple data (for the N length list).

The inter-recursive functions’ verification may be realized by using of the 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 in following: it is necessary to definite for its need dependence, and such, that next recursive calls passes ahead previous calls - this gives the algorithm finishing guarantee.

The inter-recursive functions are defined by some recursive functions that call each other. The argument of each function is the number or list. Let’s define the size which is equal to the sum of all this function's argument length. Let consider this size as the transfinite induction measure. Then the induction method receives the following shape: to verify the inter-recursive functions’ truth for the arguments with the zero length. Later it is supposed that the functions are true when the arguments sum is N length, and we try to prove for the N-1 length, as with using the structural induction. The induction step decreases if the sum of lengths decreases.

References: