SEVERAL ISSUES OF PROGRAM VERIFICATION

Natela Archvadze[1], Givi Silagadze[2], Merab Pkhovelishvili[3], and Lia Shetsiruly[4]

[1]I.Javakhishvili Tbilisi State University, Tbilisi, Georgia
[2,3]N.Muskhelishvili Computing Mathematic Institute, Tbilisi, Georgia
[4]Shota Rustaveli State University, Batumi, Georgia
[1]*natarchvadze@yahoo.com*, [2,3]*merab5@list.ru*, [4]*lika77u@yahoo.com*

The program verification test is deemed to be one of the most important tasks of the automated programming. The solution of such problem yet remains the actual topic since the programming software error worldwide leads to several billion dollars of expenses annually [1].

Formal specifications and evidence methods are the modern trends of verification. The mathematical apparatus is applied to verify the fact that the program specifications produce the proper solution for the task the program has been developed for.

The alternative attitude is the creation of the generalized, abstract program sample for each specific language programming and verification of the accuracy. Afterwards, the program which requires verification should be translated to such sample. In this case, there is no need to verify such program since the program is referred as the private case of the accurate program.

For the functional languages for whom the repeated calculations are executed through the recursion, we can determine the general forms of the recursive functions those enabling processing of lists. The recursion within the functional programming is deemed to be not only the major mean for calculation but the thinking mean and calculation method as well. The recursive forms and the possibility of the verification are thoroughly described in [2-4].

As for the verification of the programs of the imperative paradigm, it is worthy to note that the precisely determined representation directions for the program verification do not yet exist. The method of verification offered by us is mainly applied for the functional languages. However, considering that the majority constructions of these languages are reducible for the constructions of the functional languages, the same methods can be applicable for the verification of the imperative languages.

We face the tasks as follows: a) to create the mechanism by which the cycle operators (procedural languages) shall be translated into the recursion functions (functional languages); b) to establish the correspondence of the recursion functions with the common recursion forms (on the Lisp language); c) to confirm the correctness of the recursion form by the induction methods to verify the initial, original function.

While the description of the actions through the recursions, the attention should be focused to the trends as follows: first of all, the procedure should contain at least one terminal branch and condition of completion; second, as soon as the procedure reaches the terminal branch, the process of functioning is being interrupted and the new, same process is being restarted. In addition, the interrupted process is being saved, which will be launched upon the completion of the new process. Besides, the new process might be as well interrupted thus waiting for the completion of another process and so on.

The stack of the interrupted processes is being developed out of which the last process is being processed in the given moment. Upon its completion, the penultimate process is being processed. The whole process will be completed when the stack empties, i.e. all the interrupted processes are completed.

Pursuant to this and other suggestions formulated in [5, 6]. B [2] the general forms of the recursive functions enabling list processing are stipulated. They are given at the Lisp language:

```
<DE LIST1(a g f x)(COND((NULL x)a)
                       (T(APPLY* g(APPLY f(CAR X))        (1)
                                  (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>
```

We focus the major attention to the opportunity of translation of the cycles into the recursive forms. As an example we take the language C, being one of the most popular languages of the programming, and Lisp. It should be also noted that our task is not the creation of the translator for the whole language C into the language Lisp but only the C language subset. The subset is included into the arithmetical and logical images, operations `+, -, *, /, %, ^, ++, --, ==, >=, <=,` operator-assigning, operators of input-output, functions, operations access functions, operators `return, for, do, while, if.`

The program translation is executed through the support of the special package of Lisp-functions, including "pre-processor", "translator" and "kit of supplementary functions".

"Pre-processor" translates the C program to S representation of List. For this purpose, C program dividers are translated into the spaces, and the expression itself is being put into the brackets. In such way, the C program obtains the view of correct S expression, thus enabling its processing by the List functions. "Translator" takes S expression as argument and through "the kit of the supplementary functions" translates it into Lisp format. The text on Lisp in collected into the file. Upon completion of the translation, the direction on the function main () is added into the developed file and the file is closing. Hence, the translation process is being finishing and the developed file can be implemented over the Lisp with the help of "load".

There are three types of the cycle in the language C. We will represent them through the recursion forms.

1. Translation of cycle "for"

The translation scheme of the operator for language C into the recursion Lisp form is as follows:

```
for( <parameter>=<expression>; <term>; <operator>)<body>
```

is translated in function `Gnnnn` (the name is being generated)

```
(defun Gnnnn (<parameter >)(if(not<condition on List >;)nil      ;output nil
 (progn <body of Lisp>
     <operator on Lisp>                              ;translation of parameter
     (Gnnnn <parameter>)) ))                          ; recursion over own
 (setq <parameter> <expression on Lisp>
(Gnnnn <parameter>)                                   ;new function access
```

The program translation is detailed below:

```
(defun cfor(org)(progn
    (print "cfor is started" ou3)      ; translates into the recursive function
    ( princ opr ou3)                                        ; this is temporary
    (princ"(defun Gnnnn(j" ou1)            ; defun recursive function
    (let((topr(blsia(cadr opr))))   ; presentation of the cycle title by the list
        (let((opr1(car topr))                          ; opr1 initial meaning
            (gam2(cadr topr))                     ;conditional endings gam 2
            (opr3(caddr topr)) )                 ;changing parameter opr3
     (print "\n topr " ou3)(princ topr ou3)          ; this is temporary
     (print "\n opr1 " ou3)(princ opr1 ou3)
     (print"\n gam2 " ou3)(princ gam2 ou3)
      (print "\n opr3 " ou3)(princ opr3 ou3)
      (princ " )(if(not " ou1)                 ;ending list and starting body
      (princ(pzap gam2)ou1)                       ; transfer of the conditions
       (princ " )nil(progn " ou1)                       ; initial body progn
      (terpri ou1)                        ; the body starts with the new line
      (princ ″For operator processing is started″ ou3)
      (let((opr(cddr opr)))                  ; the internal operator for-a
      (princ opr ou3)                     ; printing the internal operator for-a
      (if(atom(car opr))                   ; processing the operator or block
       opdam opr)
      (bldam(car opr)))                                ;if completed
      (princ ″(Gnnnn j″ ou1)            ;  past for1 in the end of the body
      (princ ″) )))″ ou1))
      (terpri  ou1)
      (opdam opr1)(terpri ou1)      ;
      (princ ″(Gnnnn j)″ ou1);    ))t));
```

2. Translation of cycle "while"

The function "cwhile" while translates while operator language C into the recursion function of Lisp. The function body includes the optional operator. Whether the option is not met, the output is done through nil; in contrary, "prong" sequence from the internal operator "while" is formed and in the end the recursion inference on the new function is being stipulated. Upon determination of the functions, the access is being formed. Hence, the following scheme is developed: operator on C while <condition> <body>
is transferred to Lisp:

```
(defun whil() (if(not <condition on Lisp> )nil
       (progn  <body on Lisp> (whil )) ))
```

3. Cycle translation "do-while"

`do` <body > `while` <condition>

is transferred to form:

```
(defun do1() (progn <body on Lisp>  ))
```

While comparing these program-translators by the presented (1) recursive forms, we will witness the full conformity. For instance, whether we need to verify the functions:

`MEMBER (x y)= true`, whether `X` is the element of list `L`
`= false`, in other cases.

If compared to (1) form, we will receive:

$a \cong nil,\ g \cong equal,\ f \cong MEMBER,\ x \cong y.$

For verification of (1) form, the method of the structural induction is applied. The analysis confirm that the recursive activation for the argument x only is more simple then before (the length of the list is decreased). Thus we tackle the induction for this argument only. While the recursive activation, the argument was (`CDR X`) list containing one element less (on the high level) than `X`. Accordingly, the simple induction is used in the structural induction per the quantity of the elements. Therefore, we need to:

a) Confirm that for any list containing 0 element, `MEMBER` is the correct solution. Evidently, it is for the black list (`NIL`);
b) Confirm that for any integer `N`, `MEMBER`, working properly for all lists `L` consisting of `N` elements, will be proper and accurate for the lists containing `N+1` elements.
Suggest that `MEMBER` accurately works for list `L′`, containing `N`-elements:
`MEMBER (X,L′)= TRUE` whether `X` is the element of the list `L′`
`= FALSE` in contrary.

This is induction hypothesis. Suggest that `L` –list containing `N+1` elements. The situation is same as for `N+1>=1, L#NIL.`

We check the operations of the functions:

`MEMBER(X, L)= TRUE` whether `X = CAR(L)`
`= MEMBER(X,(CDR L)),` in other cases.

Whether `X=CAR (L), X` is the element of `L` and, accordingly, `TRUE` is the meaning actually required.

Whether `X#CAR (L),` than `X` will be element of `L` and only in such case it will be the element of `(CDR L).` But `(CDR L)` is the list containing `N`-elements and, pursuant to the hypothesis `MEMBER(X,(CDR L)), TRUE` and `FALSE` will be correctly calculated depending on the existence of `X` element in (`CDR L`) or not. Hence, whether `X#CDR (L),` than `MEMBER(X, L) = MEMBER(X (CDR L)).` The expression will calculate `TRUE` or `FALSE` depending on the existence of `X` element in `(CDR L)` as well as, accordingly, list `L` – the point we wanted to confirm.

As for the presented installed cycles, the view of the higher level recursions [6] can be applied. After that, the inductive methods of verification can be used for them as well.

References

1. D. Vudkok. Pervie shagi k resheniu problemi verifikacii program. (In Russian) – 2006 . – № 8. – С. 36-43.
2. 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.
3. 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
4. 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
5. N. Katlend. Vichislimost. Vvedenie k teoriu rekursivnix funkcii. (In Russian) Moscow, "Mir", 1983.
6. V. Berj. Metodi rekursivnix funkcii. (In Russian). Moscow,"Mashonostroenie", 1983.