Reports of Enlarged Session of the
Seminar of I. Vekua Institute
of Applied Mathematics
Volume 24, 2010

TRANSLATE IMPERATIVE CODE INTO LISP CODE FOR THEIR VERIFICATION

Archvadze N., Silagadze G., Pkhovelishvili M., Shetsiruli L.

Abstract. We briefly describe an approach to translation of imperative programs into functional ones. In particular, we focus on mapping the loop operators of C to LISP recursive functions. The translation is a step in the process of using functional program verification techniques for imperative code.

Keywords and phrases: Verification, programming languages, programming paradigm, functional programming.

AMS subject classification: 68Q60, 65G20, 68N15, 68N18.

1. Programming paradigm. Four major paradigms are determined within the programming: imperative, functional, logical and object-oriented. Generally, concept of "Programming Paradigm is referred to be the set of attitudes, methods, strategies, ideas and concepts, which determines the program writing style" [1]. The term "paradigm" was firstly used by Thomas Kun in his book "Structure of the Scientific Revolutions" [2]. Th. Kun deems paradigm to be the applicable system of the scientific visions, under the frames of which the research is being implemented. The term of "programming paradigm" was initially mentioned by Robert W. Floyd, the laureate of the Turing Prize in 1979. Floyd mentioned that the events similar to Kun paradigms can be observed while the programming; however, these events distinguish from Kun paradigms since they are not mutually exclusive [3].

The functional programming, being based on Lambda () computations, secures the opportunity of the mathematical research, namely, we can formally confirm the program features. Our aim is to translate the functional programming verification opportunity to the imperative languages, having high practical importance, considering that the formal confirmation for them is too sophisticated issue.

The idea of the paper is to translate imperative code into LISP code hoping that the latter can be verified easier that the former one. Possibility of such an imperative-to-functional code translation has been pointed our already in the 60'ies by McCarthy. It has been exploited for verification purposes by various people [4,5].

2. Two forms of the recursive functions for verification. Two forms of the recursive functions have been developed for the functional language *Lisp*, those being applied for presentation of the recursive functions within the functions verification and automated synthesis tasks. They have the following form at the *Lisp* :

```
< 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 >

At [6,7] the Lisp programs transfer on these forms is presented. The correctness for these forms is formally confirmed through application of the structural induction and special methods.

3. Universal approach to verification. On approach to verification of imperative programs suggests to translate them into functional ones and try to verify the obtained functional code. Proponents of this approach claim that, in this way, one can verify imperative programs by inductive methods that have been successfully applied to functional program verification. This method also helps to reduce the need of generating loop invariants for imperative program [4,5].

Thus we are tasked to create the translator that will translate C program into the List program form.

Initially, we would suggest that C program contains only arithmetical and logical expressions, operations $+$, $-$, $\wedge, \star$, $/$, $++$, $--$, $==$, $>=$, $<=$, assigning, intake-outtake operators, functions, functions assigns and operators *return, for, do, while, if.*

As an example, we present the C cycle operators through the Lisp functions and recursive functions.

4. Obtained results. We have developed programs to translate C code into Lisp. They consist of a pre-processor, the actual translator, and some auxiliary functions.

The pre-processor takes a C code, separates lines in it by spaces and put the processes C program between parentheses. The result, that we call an S-image, is then passed to the translator, which identifies C constructions within it and translates them into LISP expressions with the help of auxiliary functions. The obtained LISP code is saved in a file. After translation is finished, code corresponding to the *main* () function call is added to the LISP program and the file is closed. Now it can be loaded from a LISP session by the standard load function.

For the example, we present the translation of C while cycle operator to the recursive function. C while function translates C while cycle into recursive *Gnnnn*() function the body of which is the optional operator. Whether the option is not met, the output is done through nil; in contrary, (*cond*(t sequence is formed for the while internal operators and in the end the recursion inference on the new function is being stipulated. The function determination is laid within the first file, and the direction on this function is inserted into the second file; hence, the scheme is as follows:

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:

while $<$ option $><$ body $>$: transfers
(*defunGnnnn*(); function with unique name
(*if*(*not* $<$ *optiononLisp* $>$; recursion completion by nil
(*cond*(*t*; the body is completed and the recursive direction is being implemented
$<$ *body on Lisp* $>$
(*Gnnnn*))))); determination will be stipulated in the first file
(*Gnnnn*)

Below you can find the c while supplementary functions program text:
$(defun\ cwhile\ (opr\ ff);$ ff is the file name to be stipulated$ou\,0$
$(let((fn(gensym)));$ fn unique name
$(terpri\ \ ou0)(princ"(defun"ou0);$ $(defun$
$(princ\ fn\ \ ou0);$ Gnnnn recursive function
$(princ"()(if(not"\ \ ou\ 0);$ ()and body beginning
$(princ\ (pzap\ (cadr\ opr))\ ou0);$ while next option
$(princ")\ \ nil"\ \ ou0)\ ;)$ – stipulation and recursion will be completed by nil
$(terpri\ ou0);$ while internal operator is being launched
$(let\ ((opr1\ \ (cddr\ opr)));$ while internal operator
$(princ"(cond(t\ "ou0)(terpri\ ou\ 0);(cond(t - blockahead(inou0)$
$(if\ \ (atom\ (car\ opr1));$ simple operator or block?
$(opdam\ opr1\ \ ou0);$ operator processing(in$ou0$)
$(bldam\ (car\ opr1)\ ou0)\ );$ processing of blocks(in $ou0)if)$
$(princ"("ou0)\ (princ\ fn\ ou1);$recursion direction on the body end Gnnnn
$(princ"))\ )))"\ \ ou0)t);t)cond)\ if)\ defun)$ stipulation
$(terpri\ ou1);$ from the new line
$(princ"("\ \ ou1);$ direction on recursive function(
$(princ\ fn\ ou1)(princ")"ou1);$ $Gnnnn)$ is stipulated in$ou1$
$t)\ );$ $t -$ result and let)
$defun)$

Upon completion of the translation, first of all "*load*" < first file > function should be implemented (the cycle-respective recursive functions are determined); after that, the "*load*" < second file > should be implemented (the actions indicated in C++ program should be executed).

The obtained Lisp program is q special case of one of the general forms given in Section 2 above. Since these forms can be verified, one can obtain correctness of simple C programs from there. Therefore, verification of imperative programs in special cases can be through verification of functional programs.

R E F E R E N C E S

1. Floyd R.W. The paradigms of programming. *Communications of ACM.*, **22** (1979), 455-460.
2. Kuhn T.S. The Structure of Scientific Revolutions. *Univ. of Chicago Press*, 1970.
3. Archvadze N., Pkhovelishvili M., Shetsiruli L . Problems of verification of functional programs. *Bull. Georgian Acad. Sci.*, **3**, 3 (2009), 57-60. http://www.science.org.ge/moambe/3-3/Archvadze.pdf
4. Myreen M.O. Formal Verification of Machine-Code Programs. *PhD Thesis. University of Cambridge*, 2008.
5. Giesl J., Kuehnemann A., Voigtlaeder J. De accumulation techniques for improving provability. *Journal of Logic and Algebraic Programming*, **71**, 2 (2007), 79-113.
6. Archvadze N., Pkhovelishvili M., Shetsiruli L., Nizharadze. A recursion forms and their verification by using the undictive methods. *Comput. Computational Intelli. Proc. 3nd European Comput. Conference (ECC'09), Tbilisi*, (2009), 357-361. http://www.wseas.org/conferences/2009/tbilisi/Program.pdf

7. Archvadze N., Pkhovelishvili M., Shetsiruli L., Nizharadze M. Program recursive forms and programming automatization for functional languages. Wseas transactions on comput. **8**, (2009). ISSN: 1109-2750. 1256-1265. http://www.wseas.us/e-library/transactions/computers/2009/29-531.pdf

Received 4.06.2010; revised 19.10.2010; accepted 21.11.2010.

Authors' addresses:

N. Archvadze
Iv. Javakhishvili Tbilisi State University
2, University St., Tbilisi 0186
Georgia
E-mail: natarchvadze@yahoo.com

G. Silagadze
N. Muskhelishvili Computing Mathematic Institute
7, Akuri St., Tbilisi 0193
Georgia
E-mail: Givi.Silagadze@yahoo.com

M. Pkhovelishvili
N. Muskhelishvili Computing Mathematic Institute
7, Akuri St., Tbilisi 0193
Georgia
E-mail: merab5@list.ru

L. Shetsiruli
Shota Rustaveli State University
35, Ninoshvili St., Batumi 6010
Georgia
E-mail: lika77u@yahoo.com