საქართველოს მეცნიერებათა ეროვნული აკადემიის მოამბე, ტ. 3, №3, 2009
BULLETIN OF THE GEORGIAN NATIONAL ACADEMY OF SCIENCES, vol. 3, no. 3, 2009

Informatics

Problems of Verification of Functional Programs

Natela Archvadze*, Merab Pkhovelishvili, Lia Shetsiruli*****

* *I. Javakhishvili Tbilisi State University*
** *Niko Muskhelishvili Computer Mathematics Institute, Tbilisi*
*** *Shota Rustaveli State University, Batumi*

(Presented by Academy Member G. Gogichaishvili)

ABSTRACT. Program verification arguments are considered together with structural and transfinite induction methods. The nature of an abstract program for the functional programming language LISP and its mechanical verification problems are discussed. *© 2009 Bull. Georg. Natl. Acad. Sci.*

Key words: *functional programming, programs verification, recursive function, induction method.*

Programmers generally agree on the difficulty of writing a **correct program,** or of a program that would fulfill its purpose without errors. Large-volume programs frequently contain errors that can bring forth undesirable outcomes. Of course, there is a risk that software itself may not be error-free. Such errors are not, however, the theme of our discussion.

One of the methods of program proof is the testing of programs, although the possible options of data are so large and numerous that the testing process itself can not be guaranteed from errors.

In order to be fully aware of a program's accuracy, a method of **mathematical verification** is used, by which the program accuracy for all the possible data is being proved theoretically. A program realized in a rather precisely defined programming language has a clear-cut mathematical notation. Similarly, requirements to a program can be expressed in a mathematical language and logic, as an **exact specification**. Verification makes it possible to **strictly argue** the compliance of a program with its specification. Such formalism has a significant advantage over testing, for it states the program correctness once and forever, possibly for a whole class of programs.

Studies in verification are conducted rather for the program languages which are easier to "read" and whose texts can be processed in the same languages, for example functional language. Moreover, special languages are created to simplify verification of the programs written in them. At present, there are verifiers for problems of a limited area, or/and for specific structures of programs. We shall deal with the problems of verification of programs written in functional programming languages for such structures as recursion and, also, conditional statements. The reason of this is that the loop statements existing in the programming languages can easily be presented in them as recursions of various types.

Let us consider verification problems for individual cases.

I. Proof by induction is used for recursive functions, the arguments of which are numbers by which changes take place.

For example, the factorial function is defined for any positive number X:

F(X) ≅ IF (X=1) THEN 1
ELSE OTHERWISE X * F (X-1)

In [1] the proof of its accuracy is given by the traditional method of proof by induction. To consider the program operation, let us complete when the argument is *n*:

F(n) = n * F(n-1) =	Since condition n=1 is false, therefore F(n)=n*F(n-1). Now F(n-1) is to be computed, or a recursive reference to F.
= n *(n-1) * F(n- 2) =	Since in computing F (n-1) condition, n-1=1 is false, therefore F (n-1) = (n-1)*F (n- 2). etc.
=n *(n-1) *(n- 2) *… * F(1) =	Since condition 2=1 is false, therefore F (2) =2*F (1).
= n *(n-1) *(n- 2) *…* 2 * 1 =	Since condition 1=1 is true, therefore F (1) =1.
= n!	

II. Structural induction technique is used for such recursive functions, the arguments of which are not numbers but structures. Accuracy of such programs can be proved as follows:

1. Let us prove that the program operates correctly for the most important data (arguments of function).

2. Let us prove that the program operates correctly for more complex data (arguments of function), assuming that it is correctly operating for relatively simple data (arguments of function). For functional programming languages, for example for LISP, we assume that induction is fulfilled according to the list size.

For instance, let us refer to the LISP function MEMBER definition and its proof by the structural induction technique [Anderson]:

MEMBER (X, L) = IF L = NIL THEN FALSE
ELSE IF X = CAR (L) THEN TRUE
ELSE OTHERWISE MEMBER (X, CDR (L)).

The program analysis shows that reference to the second argument of the MEMBER function is simpler because CDR (L) contains one element less than the list L. The proof is in the following: 1. Let us prove that for any zero-element list the MEMBER function operates correctly. True, MEMBER (X, NIL) =FALSE, since X is not a member of NIL. 2. Let us prove that if the program MEMBER operates correctly for the N-element (upper level) list L^1, then it operates correctly for all the lists containing the N+1 element at the upper level. The induction hypothesis will be the following:

MEMBER (X, L1) = TRUE if X is an element from L^1,
= FALSE in other cases.

Assume L is the list containing the N+1 element. Since N+1≥1, therefore L≠NIL. Complete the function:

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

If X =CAR (L) (note that CAR (L) is defined because L≠NIL), then X is an element of the list L and hence the value TRUE is the required value. If X =CAR (L), then X will be an element of the list L only on condition that X is an element of CDR (L). In addition, CDR (L) is the N-element list and, according to the induction hypothesis, MEMBER(X, CDR (L)) will correctly compute the TRUE and FALSE values according to the condition that X is an element of CDR (L), the proof of which was required.

III. Transfinite induction technique is a proof technique that represents generalization of mathematical induction for non-numerical values of parameters. It consists in the following: say M is fully an ordered set and P(x), when x∈M is any proof. If P(y) is true, every y<x (any x∈M), it follows that P(x) is true, and if proof $P(x_0)$ is true, where x_0 is the minimal element of M, then P(x) is true for any x. It is noteworthy that under the Zermello's Theorem, any set can become fully ordered.

Mathematical induction is a private case of transfinite induction. True, when M is a set of natural numbers, then transfinite induction will consist in the following: if P(1) is true and from the following proofs P(1), P(2), …, proceeds the P(n-1) proof, then all P(n) will be true.

Consider the greatest common divisor finding the function gcd(m,n) for two natural *m* and *n* numbers:

<DE GCD (x y) (COND ((EQ y 0) x)
(T (GCD (y (x MOD y >

This function is used for any numbers and not only for positive arguments. The greatest common divisor notion should be extended. Introduce the correlation $'u|v'$ when $'u$ is divisor of v'. It includes a pair of integers u and v, for which $'v$ is the multiple of u', or is found the integer d when $v=du$. It is said that d is the greatest common divisor of x and y if:

- $d|x$ and $d|y$
- for any integer d', if $d'|x$ and $d'|y$, then $d'|d$.

Proof of the function GCD is accomplished by using transfinite induction. It is necessary that a relation to it be defined, so that further recursive calls would precede the former calls, which would guarantee termination of the argument. In general, definition of a relation is not easy; however, sometimes introduction of the size that will correlate the arguments to the respective numbers and reduce the numbers during recursion (to guarantee the recursion termination) is enough. Such size for the given function GCD is |Y|. In [2], theorems are given that prove that the operation of the function GCD terminates and its outcome is the greatest common divisor.

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

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

Therefore, cases with a mixed version represent special cases, being of considerable 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 of numbers. In this case, separate use of the proof induction and structural induction techniques can 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 the 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 is accomplished by using first the structural induction technique and then the proof-by-induction technique.

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

V. Special program verification technique

The program verification history is known for the cases when special languages used to be created for the purpose of simplifying 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 as accurate. It is necessary that the program needing verification be automatically transferred to the form of an abstract program. In such a case, it will not be subject to verification (because it will already be a particular form of a correct program).

The problem of automation of the program verification process in programming has retained its urgency to the present day. In respect of the functional language LISP, this problem consists in the reduction of any program to the abstract form and to its follow-up automated verification.

We offer the following verification algorithm:

a) To make a form of the so-called universal function (or functions) for the given programming language, allowing to represent the recursive function of any type written in this language. Following it, the universal function is to be verified. 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 in the frame 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 constructions. 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 turn, make it possible to use the given algorithm for other languages.

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

In [3], two forms of recursive functions are given, which provide for the list processing possibility. These

functions have the following form in the LISP:

<DE FUN(F F^0 L)
(COND((MEMBER NIL L)a)
(T(g(f(M F L))
(APPLY 'FUN(CONS 'F(CONS 'F^0(M F^0 L>

or <DE FUN (A F F^0 L)
COND ((MEMBER NIL L)A)
(T (FUN (g (f (M F Ø) a) (F F^0. (M F^0 L>

These forms were proposed for synthesis of functional programs, although they can be applied to verification of functional programs as well. We assume that the transition of the system into a dialog mode in this process could require additional information.

Thus, we have discussed for verification purposes such techniques as induction-by-proof, structural induction and transfinite induction to prove the composition accuracy. We have also dealt with the form of an abstract program for the programming language LISP and the problems of its computer-based verification. We would like to note that in the process when the program is reduced to the form of an abstract program, the arrangement of a dialog mode would be advisable.

ინფორმატიკა

ფუნქციონალური პროგრამების ვერიფიკაციის საკითხები

ნ. არჩვაძე*, მ. ფხოველიშვილი, ლ. შეწირული*****

** ი. ჯავახიშვილის სახ. თბილისის სახელმწიფო უნივერსიტეტი*
*** ნიკო მუსხელიშვილის გამოთვლითი მათემატიკის ინსტიტუტი, თბილისი*
**** შოთა რუსთაველის სახელმწიფო უნივერსიტეტი, ბათუმი*

(წარმოდგენილია აკადემიის წევრის გ. გოგიჩაიშვილის მიერ)

განხილულია პროგრამების ვერიფიკაციის მტკიცებულებათა, სტრუქტურული და ტრანსფინიტური ინდუქციის მეთოდები. განხილულია ფუნქციონალური პროგრამირების ენა LISP-სთვის აბსტრაქტული პროგრამის სახე და მისი ავტომატური ვერიფიკაციის საკითხები.

REFERENCES

1. *P. Anderson* (1982), Dokazatel'stvo pravil'nosti programm. Moscow (in Russian).
2. *M.Pkhovelishvili* (1984), Osnovnye formy rekursivnykh funktsii dlya obrabotki spiskov. Interaktivnye sistemy. Tbilisi (in Russian).
3. *N. Archvadze, M. Pkhovelishvili, L. Shetsiruli* (2009), Kompleks proizvodstva programm dlya funktsionalnykh yazykov. Kiev (in Russian).

Received June, 2009