Computer Science Journal of Moldova, vol.**5**, no.3(15), 1997

On program correctness and teaching programming

M.Frenţiu

Abstract

First, a short survey on program correctness is given. Then, some consequences of this theory, given as important rules in the programming activity, are presented.

The most important property of a program is whether it accomplishes the intentions of its user, i.e. if it is correct. This means that it performs correctly for all inputs for which the input predicate $\varphi(X)$ is true, and in these cases the rezults Z are correct, i.e. the output predicate $\psi(X, Z)$ is true.

A method for proving partial correctness of a flowchart is due to Floyd [5]. An important step of this method consists in finding (guessing) the invariant predicate for each cutpoint. Usually these predicates reflect the meanings of variables.

But the students learn this theory some time after they know how to write a program. And they do not prove, usually, the correctness of their programs.

It is the time to teach programming in conformity with the theory of program correctness. As Naur has underlined in [20], "it is a deplorable consequence of the lack of influence of mathematical thinking on the way in which computer programming is being pursued". We think that teaching programming well is an important part of our tasks as teachers in the universities.

1 A short survey on program correctness

In a program P we distinguish three types of variables, grouped as three vectors X, Y and Z. The input vector $X = (x_1, x_2, \ldots, x_m)$ consists of the input variables, which do not change during computation. They denote the known data of the problem PP solved by the program P. The output vector $Z = (z_1, z_2, \ldots, z_m)$ consists of those variables which denote the results of the problem PP. The program vector Y consists of the auxiliary variables, which denotes various partialy results of the computation.

Two predicates are associated to the program P: an input predicate and an output predicate. **The input predicate** $\varphi(X)$ is TRUE for those values of X for each the problem may be solved. **The output predicate** $\psi(X, Z)$ is TRUE for those values a and b of the vectors X and Z for which the results of the problem are b when the initial input data is a.

The specification of the program is the pair formed from the input predicate $\varphi(X)$ and the output predicate $\psi(X, Z)$.

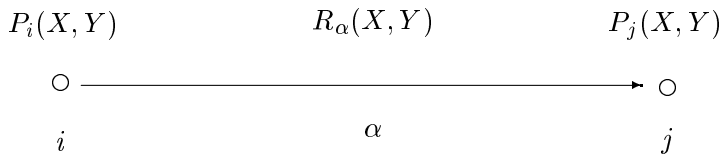
We say that the program P **terminates over the input predicate** $\varphi(X)$ if for each value $a = (a_1, a_2, \ldots, a_n)$ of the vector X for which the predicate φ is TRUE, the execution of P terminates. The program P is **partially correct** with respect to the specification if for the value a for which $\varphi(a)$ is TRUE and the execution terminates with the results $b = P(a)$ then $\psi(a, b)$ is TRUE. The program P **is totally correct** with respect to $\varphi(X)$ and $\psi(X, Z)$ if the program P terminates over $\varphi(X)$ and it is partiallly correct with respect to $\varphi(X)$ and $\psi(X, Z)$.

A method for proving partial correctness of a flowchart program is due to Floyd [5] and it uses a set of cutpoints. This is a set of points on the arcs of the flowchart such that every loop includes at least one such cutpoint. Also, there is a cutpoint on the arc leading from the START box, and there is a cutpoint on the arc leading to the HALT box.

To each cutpoint i of the flowchart, a predicate $P_i(X, Y)$ is associated. This predicate, called **inductive assertion**, is invariantly true

for the current values of X and Y in this cutpoint, i.e. it characterizes the relation that must exist between variables at this point. At the START cutpoint the corresponding inductive assertion is $\varphi(X)$ and at the HALT cutpoint the inductive assertion is $\psi(X, Z)$.

The set of cutpoints defines the paths that must be verified. Let α be a path leading from the cutpoint i to the cutpoint j, with no intermediate cutpoints (there can be more such paths). To this path we associate a predicate $R_\alpha(X, Y)$ which gives the condition for the path α to be traversed, and a function $r_\alpha(X, Y)$ such that if Y are the intermediate values in the cutpoint i then, when the path is traversed, $Y' = r_\alpha(X, Y)$ are the values in the cutpoint j. We represent this as follows:



Next, **a verification condition is associated to the path** α. This condition is:

$$\forall X\, \forall Y\, (P_i(X, Y) \wedge R_\alpha(X, Y) \rightarrow P_j(X, r_\alpha(X, Y)))$$

Floyd [5] proved that if all the verification conditions are true then the program is partially correct with respect to $\varphi(X)$ and $\psi(X, Z)$.

Floyd also suggested a method for proving termination using well-founded sets. A well-founded set M is a partially ordered set, without infinite decreasing sequences. For each path α from i to j, a **termination condition** is formed. It uses a function

$$u_i : D_X \times D_Y \rightarrow M.$$

The termination condition for the path α is:

$$\forall X\, \forall Y\, (\varphi(X) \wedge R_\alpha(X, Y) \rightarrow (u_i(X, Y) > u_j(X, r_\alpha(X, Y))))$$

If the partial correctness was proved and $P_j(X,Y)$ was the invariant predicate for the cutpoint i, then the termination condition for the path α may be:

$$\forall X\, \forall Y\, (P_i(X,Y) \wedge R_\alpha(X,Y) \rightarrow (u_i(X,Y) > u_j(X, r_\alpha(X,Y))))$$

If all termination conditions are proved then the program P terminates over $\varphi(X)$.

The ideas of Floyd were formalized by **Hoare** [11] who introduced an axiomatic method for proving the partial correctness of a program. His article changed the programming activity from an art into a science.

Some early papers on program correctness [10, 17, 18] have proved the correctness of some concrete algorithms. Just Hoare has made a significant move from a posteriori proof of an existing program [11, in 1969] to a program design method [10, in 1971].

Today there are hundreds of papers dealing with Program Correctness. There are also various aspects of analysing this problem. We can prove the correctness of an existing program, or we can try to build a program for doing it. For a given program we are interesting in an automatic proving the correctness, or in the synthesis of loop predicates [13, 14, 22].

Certainly, we can try to prove that a given Program is correct with respect to its specification. But we have learned from Gries [9] that it is more important to write correct programs from the beginning. As he said: "A program and its proof should be developed hand-in-hand, with the proof usually leading rhe way". The idea that the algorithm construction and its proof should proceed hand in hand originated with Dijkstra [2].

2 The consequences on teaching programming

It is not only possible, but necessary, to teach explicitly the methods and principles for good programming.

Certainly, the correctness is not the only quality that a good program must have. Robustness, extendibility, reusability are three other

external qualities [19]; they can be achieved through some internal qualities of programs: readability and modularity, simplicity and clarity.

Some of the most important rules considered important for programming well, which are consequences of the above mentioned theory, are given and commented below.

2.1. Define the problem completely (i.e., write the predicates $\varphi(X)$ and $\psi(X, Y)$) [4, 9, 16, 21].

This rule seems obvious and easy to respect. But too often the students (and just the old programmers) begin to work before they know the correct specification of the problem.

2.2. Think first, program later [3, 16].

Do not start writing the program before its design. Respect the required steps for a good programming. As the above rule asks, write first the program specification and think to its correctness. Then design the algorithms and prove their correctness. Program later, when the possibility of commiting logical errors was removed.

2.3. Use Top-Down Design [6, 16, 21] (Also known as **"Stepwise refinement"** [23], or **"Divide and Conquer"** [9]).

This is one of the most important programming method. Its main characteristics are [16]:

- correct specification;
- language independent design;
- Design in levels;
- Postponement of the Details to Lower Levels;
- Proof of correctness at each Level.

The accent is on the correctness at each step in the program construction. Again the first rule is underlined; the problem must be clearly and completely defined. Then think to the problem, not to a specific programming language. Think to the method of solving the problem,

and decompose it in easier subproblems. Design in levels; at each level solve the important actions of that moment. But turn back and refine the postponed activities (nonstandard sentences).

2.4. Write and use modules as much as possible [3, 16, 21].

A module (a procedure, a function, or a Turbo-Pascal Unit, a module of Modula, or an Ada package) has a well defined task. It may solve a subproblem, which is clearly, simpler than the entire problem. It is much easier to prove the correctness of small procedures. First of all, the number of paths between the cutpoints is reasonable. The tested modules may be organised in libraries and reused.

2.5. Use Structured Programming [1, 6, 21].

We do not try to define this method here. But we are convinced it has played an important role in programming. It asks for the clarity and the presence of structure in the texts of programs. Many of the present rules are connected to Structured Programming.

2.6. Prove the correctness of algorithms during their design [2, 9].

The errors may be discovered much early than running the program on a computer. As Gries [9] said: "A program and its proof should be developed hand-in-hand, with the proof usually leading the way".

2.7. Decide which are the necessary program variables, and what are their meanings. Write invariants for these variables and insert them as comments in the program [9, 16].

Use auxiliary variables only if it is necessary, if there is a reason to use them. Each variable plays its role in the program. It must receive correct values. Define rigorously the meaning of a variable by an invariant predicate and use it in proving the program correctness, and testing the program.

2.8. Choose suitable and meaningful names for variables [16].

During the maintenance phase, when other programmers have to work on the program, the clarity, and the readability of the program are very important, and good names improve this clarity.

2.9. For each variable of a program, make sure that it is declared, initialised and properly used [3, 20].

A very frequent error made by beginners is the utilization of noninitialised variables. This fact certainly causes errors. But a variable may appear in aprogram accidentally. A line of a program may be omitted, and some variables are not declared or initialised (for this, rule 2.18 is very important).

2.10. Verify the value of a variable immediately it was obtained [4, 6].

Each variable must be protected from wrong values. Especially during testing, if it is known that a variable must be in $[a, b]$ check this, and report if this is not true (see also, rule 2.14).

2.11. Use comments to document the program [16].

The text of a program must be understood easily by all persons who have to read it. For this purpose, comments can be very useful, illuminating the logical structure of the program. Each module must contain comments which give the module specification, the meaning of the variables, the invariants used in the proof of correctness, and any useful information for the reader.

2.12. Do not overlook the special cases of the problem [20].

The program must give correct answers for all inputs. From the programmers experience, it was observed that some special cases of the problem are omitted from programs. Also, one of the qualities of a program must be its robustness [19].

2.13. Define and use Abstract Data Types (ADT) [3, 21].

An ADT can be viewed as an independent module that define a data structure and operations on this structure. Due to its independence of the context where it was used, the reusability of this module is assured. Also, this ADT may be changed, adding new operations on the defined structure, or modifying some operations.

2.14. Design input-output routines for each ADT [4, 21].

These routines must check if the transferred values are correct, ensuring the fulfilment of 2.10. Also, these operations are important for the independence of the defined ADT.

2.15. Verify each part of a program as soon as possible [4].

A very important rule! The specification of the program may be incorrect; check it from the beginning. Then prove the correctness of problem decomposition, and of each subalgorithm.

Nevertheless, do not forget to **test the modules and the program** [7, 8]. The proof may me wrong, and, more probably, the program may not be a correct translation of the designed algorithms (a wrong translation, an omitted line, or some misspeled errors).

2.16. Use symbolic names for all entities (constants, types, variables, procedures and functions) [3].

It is considered that all properties of a type, and the meaning of the entity are concentrated in its name. Also, using names, the clarity of the program is improved, and the modifiability is easier.

2.17. Avoid to use global variables [4, 21].

It is difficult to analyse a program that uses global variables. And it is impossible to prove the correctness of a module that uses global variables independent of the context where it is used.

2.18. Hand-check the program before running it [15, 16].

First of all, it is important to have introduced in the computer the conceived program. The various errors (mispelling, typing errors, omitted lines) are very probable. It is not difficult to check for and then to eliminate these errors. But the rule asks for a human execution of the program, before an automated execution. We may be very surprised by the errors we have perpetrated. Esspecially for the beginners, for the first year students this own program execution may be very useful.

2.19. Write the documentation of the program simultaneously with its building [21].

The maintenance activities need information about all levels of the program development. Also, for the reusability of some parts of the program, this documentation is very useful. This rule asks to write the documentation during program development. It must contain documents for each phase of program building: specification, design, coding, and testing. The program itself must be selfdocumented by comments.

2.20. Strive for continuing invention and elaboration of new paradigms to the set of your own ones [6].

This, and many paradigms were discussed by Floyd [6] in his Turing award Lecture. We need to known the existing methods, and to permanently aquire new paradigms.

References

[1] Dahl O.J., E.W.Dijkstra and C.A.R.Hoare. Structured programming. Academic Press, New York, 1972

[2] E.W.Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Comm. A.C.M., 18 (1975), 8, pp.453–457

[3] M.Frenţiu, B.Pârv. Programming Proverbs Revisited. Studia Univ. Babeş-Bolyai, Mathematica, XXXVIII (1993), 3, pp.49–58

[4] M.Frenţiu, B.Pârv. Elaborarea Programelor. Metode şi Tehnici moderne. Ed. Promedia, Cluj-Napoca, 1994

[5] Floyd R.W. Asssigning meanings to programs. Proc. Symposium in Applied Mathematics, 19, AMS, 1967, pp.19–32

[6] Floyd R.W. The paradigms of Programming. Comm. A.C.M., 22 (1979), 8, pp.455–460

[7] Geller M. Test data as an aid in proving program correctness. Comm. A.C.M., 21 (1978), 5, pp.368–375

[8] Gelperin D. and B.Hetzel. The Growth of Software Testing. Comm. A.C.M., 31 (1988), 6, pp.687–695

[9] Gries D. The Science of Programming. Springer-Verlag, Berlin, 1981

[10] Hoare C.A.R. Proof of a program: FIND. Comm. A.C.M., 14 (1972), pp.39–45

[11] Hoare C.A.R. An axiomatic approach to computer programming. Comm. A.C.M., 12 (1969), pp.576–580

[12] Hogger. Derivation of Logic Programs. Journal ACM, 20 (1981), 2, pp.372–392

[13] Igarashi S., London R.L., Luckham D.C. Automatic Program Verification I: a logical basis and its implementation. Acta Informatica, 4 (1975), pp.145–182

[14] Katz and Manna. Logical analysis of programs. Comm. A.C.M., 19 (1976), 4, pp.188–206

[15] King J.C. Symbolic Execution and Program Testing. Comm. A.C.M., 19 (1976), 7, pp.385-394

[16] Ledgard H.F. Programming Proverbs for Fortran Programers. Hayden Book Company, Inc., New Jersey, 1975

[17] London R.L. Proving Programs Correctness. Some Techniques and Examples. BIT, 10 (1970), pp.168–182

[18] London R.L. Proof of Algorithms. A new kind of certification. Comm. A.C.M., 13 (1970), pp.371–373

[19] Meyer B. Object Oriented Software Construction. Prentice-Hall, Englewood Cliffs, 1988

[20] Naur. Proof of Algorithms by general snapshots. BIT, 6 (1966), pp.310–316

[21] Schach S.R. Software Engineeering. IRWIN, Boston, 1990

[22] Wegbreit. The synthesis of loop predicates. Comm. A.C.M., 17 (1974), pp.102–112

[23] Wirth N. Program development by stepwise refinement. Comm. A.C.M., 14 (1971), 4, pp.221–227

Militon Frenţiu,

Received October 13, 1996

"Babeş–Bolyai" University,
Faculty of Mathematics and Informatics,
1, M.Kogălniceanu str., Cluj–Napoca
3400, Romania.
phone: 40-64-194315
e–mail: *mfrentiu@cs.ubbcluj.ro*