A New Approach to the Metatheory of Correct Programming

Rationale

Valerie Novitzká,
University of Technology in Košice,
Slovakia
novitzka@tuke.sk

Abstract

In this paper we are giving arguments for the first attempt of constructing a new metatheory of correct programming. The following papers [25, 26, 27] contain more details for this argumentation.

1 Introduction

The correct solving of problems of persons, families, communities, societies and the whole mankind is the most important driving power of the evolution of the culture in the world. But what is generally the correct problem solving? George Pólya has written two excelent books [29, 30] about problem solving in mathematics, but this approximation was psychological or at most philosophically anthropological. A more logical method of correct problem solving in experimental sciences was written by Karl Popper in [31]. But the general and correct problem solving was not an object of research nor in philosophical anthropology, nor in logic, mathematics and metamathematics. Why, this question will be already the object of research in the next future.

In the half of 20th century it has been developed an excelent tool for general and correct problem solving - the computer, in that time of von Neumann architecture. Programming of such a computer was really a problem solving. But, and it is interesting, this aspect of computer programming was investigated only a little. The only book dealing with human aspects of problem solving by computers was [40]. Up to now, computer programming is not interpreted as a problem solving process, but as a possible economically effective writing of less or more long sequences of instructions; at the begining in the machine language, now in some kind of programming languages, i.e. procedural, logical or functional.

A programmer writes his programs respecting several not precise commands. Such commands are: make from a repeated sequences of instructions cycles, macros, functions or procedures; possibly do not use any kind of goto statements; form some kind of intelectually managable program units as modules; create large scale programs from modules with methods of top-down or bottom-up. These commands form actually some *deontology of programming*, while the *teleology of programming* is the reliability of programs. Also the contents of this notion is not precisely defined. Summary, we can say, that the *theory of programming* now consists of deontology and teleology of programming as mentioned above. We have to note that from philosophical standpoint such a theory of programming is not a true theory, only some "cryptotheory". For instance, a similar cryptotheory was physics before Newton.

Also practical programmers already need a systematic, consistent and complete theory of programming, which consists of a logical language for precise formulating fundamental principles as axioms and deduction rules for proving theorems about programming process. These theorems form a true deontology of programming, which programmers can applicate in their work according to need. Finally, the programmers need to prove the correctness of programs as an appropriate teleology of programming. Such a theory of programming has to be formulated in philosophical anthropology, logics, mathematics and metamathematics [32, 33].

In this paper we will deal only with the language, in which we can formulate the scientifically right theory of programming. We call this language *metatheory of correct programming.* In this Rationale we will state the reason of our decisions which we make in selection of mathematical and metamathematical elements of our metatheory of correct programming.

2 About correctness of computer programs

We have already stated that the teleology of computer programming is the correctness of programs. We note, that this correctness is not a simple teleological property. In that case the teleological correctness is essential for computer programs, because it makes from a sequence of declarations, expressions and statements a true program. We note, that program which is not correct is not a true program. A sequence of declarations, expressions and statements, which are not proved as correct, is not a program solving a human problem but only a guess and already Platon has proved that the guess has no meaning for human knowledge.

What is correctness of computer programs? Usually, one says that a computer program is correct when it trully solves the problem formulated by specification. The specification is *what* the program has to do. The program itself is the description of the method *how* to solve specified problem. The program trully solves the specified problem when it is itself in some manner identical with the specified problem. But how to specify the problem to be solved by computer? How to program the method of problem solving? And how to insure, that the program trully solves the specified problem? We note, that the last question is the most difficult one. The main result of our following paper [25] is mainly the mathematical formulation of the answer for this last question.

In the seventies of the 20th century it has been recognized that the a posteriori verification of a ready Pascal program, by help of axiomatic semantics, is practically too difficult [36]. Moreover, nobody in that time investigated the correctness of compilers generally and Pascal compilers especially. Therefore, our startpoint could not be a verification of a written, i.e. ready programs, the so called a posteriori verification.

The project *Correct Implementation Process, CIP* [4] and mainly the ESPRIT project *PROgram Development by SPECification and TRAnsformation, PROSPECTRA* [14] have begun investigate a new method: the derivation of programs from their specifications by mathematically correct transformations. The project PROSPECTRA was a very original and excelent idea, however the procedural implementation language Ada [12] and the corresponding simple specification language ANNA (ANNotated Ada) [16] have been not able to achieve the aim of the project and of course, also the correctness of Ada compilers in this case was not investigated at all. A similar method of correctness proof was investigated in [19, 15], where the programming language was not a pure functional language Standard ML and the specification language Extended ML was a simple extension of the kernel language SML. In this approach some simple derivation steps from an EML specification to a SML implementation were proved, but also in this case it was absent the correctness proof of the SML compiler. The starting point of our approach were the experiences with PROSPECTRA and SML/EML projects.

The main problem of our approach was the formulation of correctness criterion of the derivation process, i.e. the process of deriving program from its specification. According to corresponding literature [39, 37, 9, 3, 1] we have tried to find this correctness criterion by help of transliteration some already known results of multisorted universal algebras, first-order predicate logic and algebraic category theory. Now, we explain the reasons why we have chosen especially these concepts and facts for the foundations of our metatheory as we present in [25].

We refuse to use for problem specification any specification language with already defined concrete syntax. The reason of this decision we explain in the following section. Here we only emphasize that in our metatheory a specification has for us pure mathematical form. A specification is an ordered pair $Spec=(\Sigma, \Psi_\Sigma)$, where Σ is a signature containing lists of names for data sets (sorts), names for partial and total functions with their profiles, and names for predicates with their arities. It is clear, that to every such defined signature we can construct many Σ-algebras containing sorted sets of data, partial and total functions with domains and counter domains from these sorted sets, and predicates as general relations between elements of these sorted sets. Σ-formulas from Ψ_Σ are uniquely constructed from variables, (partial and total) function names, and predicate names from the signature Σ. The unique manner of Σ-formulas construction shall ensure that by substituting variables by data of corresponding sorts, function and predicate names by corresponding functions and predicates according function profiles and predicate arities in this formula we get a sentence, which must be valid at least in one Σ-algebra. We say, that a class of such Σ-algebras in which at least one Σ-formula from Ψ_Σ is valid is the *meaning* of specification, i.e. the *semantics* of specification. This concept of specification and semantics is not fully new and was publicated in [2, 10, 41, 34]; but our formulation is more precise and convenient for practical development of large scale programs.

In our last paper [27] of this series we define specification morphism, but in the first of them [25] we introduce firstly signature morphism which can be regarded as a special case of specification morphism if the set of Σ-formulas is empty. Specification morphism establishes relation between corresponding entities of two signatures. We extend known concept of signature morphisms by establishing also a relation between corresponding formulas from two specifications.

As we already mentioned, in derivation programs from their specifications we have to preserve some kind of 'invariancy' of the specification semantics and program semantics. It is a very difficult question, how to do it with a mathematically correct and intelectually managable manner. We find that a change of the concept of reduct can serve for our purposes. Our change of this notion really preserves every important property of semantics of a specification during a derivation step. Our definition of reduct with respect to signature morphism is really new transliteration and expansion of the original reduct definition in algebra. In [27] we explain examples of derivation steps, in which our concept of reduct really works in definition of correctness criterion of a derivation step and from here in the whole derivation process. We know that our correctness criterion only formally establishes but not constructs in every aspect the semantics of a specification after a derivation step. This construction has to be studied further and our idea is to use type theory [11, 17] in this investigation.

To be able precisely define, for our point of view, very important concept of institution, for continuous readability of text we introduce some auxiliary notions from algebraic theory of categories. These concepts were only a little more precisely reformulated for using in formal methods of computer science. Such needed notions are category, pushout in category and functor.

During the process of derivation programs from their specifications by well-defined derivation steps, we need to examine logical formulas in rather different algebras. This examination is enabled for us by using the concept of institution formulated by [13] and investigated by [7, 38]. We extend this concept only with more precise method of writing formulas.

In our following paper [26] of this series we develop abstract syntax and semantics of a specification language, which enables to specify very difficult problems to be solved by large scale programs. To enable the investigation of such specifications from the standpoint of their mathematical and metamathematical properties, we need consequently define such an instantiation of institution, in which the logical system is described with every detail. We have chosen the most simple and the most known logical system, the first-order predicate logic and we construct in details this instantiation, which can be applicated in investigations of mathematical and metamathematical properties of specifications during derivation process.

We suppose, that in the next future it will be elaborated also other instantiations with new logical systems, from which the best will be the logical system for problem solving.

3 Specifications

There are many specification languages; and there are still more programming languages. But only a few groups of programmers are writing specifications and mostly for only trivial problems. Every such group uses, from some precisely undetermined point of view, 'sympathetic' specification language. Why? There is no exactly founded answer to this question.

The situation is still worse for programming languages. Still among practical programmers of large scale programs there is no generally accepted programming language. Warning example of a well-prepared and still not generally used programming language is Ada. Many specialists prepared the requirements for this language (the last was Steelmann requirements). Many groups of specialists tried to satisfy these requirements with many own proposals of such a language. From these proposals was selected the ground version of Ada for which it was written a very detailed Rationale. This first version was further elaborated by many experts to a so called best procedural (now already object-oriented) language. Also it was worked out a substantial part of denotational semantics, which, however, did not contain the semantics of tasks. A specification language ANNA, ANNotated Ada was also constructed to this language, but the newest Ada is used only by a little group of people and ANNA was already forgotten. We do not try to explain the causes of this situation in programming practice. We note only, that the concrete syntax of specification languages and also the concrete syntax of programming languages were not formulated from the standpoint of systematic, complete and consistent theory of programming, but only from too subjective anthropological (mainly psychological) view on some constructions of artificial formal languages, in that case specification and programming ones.

We do not want to repeat mistakes of our respected predecessors in the definition of specification (and also programming) languages. It is the reason why we do not formulate until now a concrete syntax of our specification and programming language. In our paper [26] we accepted the results of the ESPRIT Working Group CoFI (Common Framework Initiative) [35, 8, 20, 21], but we tried to formulate an abstract syntax in such a manner, that we have to be able to formulate also the semantics of the language frame so, that this semantics satisfies our requirements: to preserve the correctness during transformation process from specifications to programs. In our paper [26] we have only a little to generalize the abstract syntax of CoFI language CASL, to enable a more flexible formulation of production rules of new language items. But we do not fix the manner of this generalisation, because the formulation of concrete syntax and also a specific requirements of correct programming steps of large scale programs may need some new formulation of production rules, which we yet now cannot predict.

In [26] we really work out a fully new method of defining the semantics of abstract syntax items. This description of mathematical semantics enables also to investigate every specification in the whole derivation process of programs from their specifications from mathematics and metamathematics point of view, because we need to know in every correct derivation step also the important mathematical properties of the original and final specifications and their corresponding semantics just in the framework of the instantiation of institution. So we founded an intelectually managable method of constructing a correct derivation step during the process of programming. A programmer can apply precise theorems and their logical consequences on the specification and its semantics.

4 Transformations

Already Cartesius in [6] understood, that a difficult problem cannot be solved at once as a whole problem. It has to be divided into many subproblems; these subproblems have to be solved independently; and finally, the answers for subproblems have to be synthesised into an answer of the whole problem. The design of our specification language respects these methodical rules as it is clear from the definition of abstract syntax and semantics presented in [26].

The transformation of a such hierarchical and structured specification into corresponding program cannot be accomplished by one step. For this reason we think about a transformation as a sequence of correct derivation steps, in which it is constructed a correct target specification with its semantics from a source specification with its own semantics. We suppose that the source and target specifications and their semantics are different in some point of view. So, in [27] we firstly define correctness criterion for a derivation step and in this definition we use results from [25, 26]. After this definition we express mathematically some frequently used practical actions of real programmers of large scale program systems. These steps were chosen only from practical reasons and they have no deep theoretical foundations. Simply, the programmers use connecting, renaming, hiding, extending, etc. as we express in [27] as examples of possibilities of our approach. In the process of further development of our metatheory and theory of correct programming we hope to deduct some theorems, which will be already exactly founded on the theoretically important and also practically reasonable derivation steps.

We now should like to point to an interesting problem. It is clear, that at the begining of derivation process we have an initial specification, from which we define a new, target specification in one derivation step. In the second step, we repeat this derivation, but the result of the first derivation step becomes the source specification and we get in this second step another new target specification. But when does finish derivation of a target specification from a source one and when does begin derivation of a target program from a source specification? This problem is in the literature not yet recognized. We hypothetically suppose, that the first target program in derivation process will be written in a language corresponding to the typed λ-calculus; and from this program will be refined step by step the executable final program written in a language of λ-calculus as a paradigma of functional languages [5]. But as we already mentioned, we will formulate the concrete syntax for such a language only when our metatheory and theory of correct programming will be work out enough.

5 Conclusion

We have no opportunity completely describe what a man has to know and what he/she has to understand, that he/she is able to identify a real problem in our reality, that is in the nature and in the culture. But it is clear that the identified problem a man has to describe also by using the language of mathematics. We want to say, that everybody needs a minimal knowledge of mathematics which enables him/her to describe and solve the problems of our world. It was the reason that the excelent matematician Rózsa Péter has written a marvelous booklet [28] for her artist friends about important problems of modern mathematics and metamathematics. To solve more difficult modern problems one needs not only language of mathematics and metamathematics, but also a computer; and also he/she needs to program the problem solving for this computer correctly. The theory of mathematics already exists; it exists also a mathematical theory about mathematics: metamathematics. So everybody can study such amount of mathematics and metamathematics what he/she needs. But there is no theory of computer programming. Therefore, who want to solve his/her non trivial problem on computers by a correct program, he/she has no opportunity to learn the correct programming theory and its metatheory in a concise and well-formed manner. Therefore only a little part of mankind uses computers for solving their important and realistic problems. Our intention to work out the metatheory of correct programming and to start working out a theory of correct programming is to enable for every educated man to learn the correct programming and so to solve the real problems by computers.

We have in our mind, that our work is only modest start of the whole task of working out this metatheory and theory. In this paper we were thinking only about the programming of computers of von Neumann achitectures. Already now it is clear, that such a conception of a theory has to be rather generalised, e.g. for neural computers. This is the task of a near future and we hope that we also begin to work it out.

References

[1] J.Adámek: *Mathematical Structures and Categories*, NTL, Praha, 1982

[2] E.Astesiano,H.-J.Kreowski,B.Krieg-Brückner,(Eds.): *Algebraic Foundations of System Specifications*, Springer, 1999

[3] M.Barr,Ch.Wells: *Category Theory for Computing Science*, Prentice-Hall, 1990

[4] F.L.Bauer,H.Ehler,B.Möller: *The Munich Project CIP*, Springer, LNCS 292, 1987

[5] E.Denney: *A Theory of Program Refinement*, PhD. Thesis, University of Edinburgh, 1998

[6] R.Descartes: *Disours de la méthode*, Oeuvres de Descartes IV, Paris, 1908

[7] M.Cerioli: *Relationships between Logical Formalisms*, PhD.Thesis, Universita di Pisa-Genova-Udine, 1993

[8] CoFI Task Group on Language Design. CASL - The CoFI Algebraic Specification Language - Summary, *www.brics.dk/Projects/CoFI/Documents/CASL/Summary*, 1999

[9] D.van Dalen: *Logic and Structure*, Springer, Berlin, 1994

[10] H.Ehrig,B.Mahr: *Fundamentals of Algebraic Specifications 1*, EATCS Monographs on Theoretical Computer Science, Springer, 1985

[11] P.Gardner: *Representing Logics in Type Theory*, PhD.Thesis, University of Edinburgh, 1992

[12] G.Goos,J.Hartmanis, (Eds.): *The Programming Language Ada. Reference Manual*, Springer, LNCS 155, 1983

[13] J.Goguen, R.Burstall: Introducing institutions, In: E.Clarke and D.Kozen, (eds.): *Logics of Programs Workshop*, LNCS 164, Springer, 1984, pp.221-256

[14] M.Hoffmann.B.Krieg-Brückner: *PROgram Development by SPECification and TRAnsformation: Methodology - Language Family - System*, Springer, LNCS 680, 1993

[15] S.Kahrs,D.T.Sannella,A.Tarlecki: The Definition of Extended ML, *Theoretical Computer Science*, 1996, pp.445-484

[16] D.C.Luckham,F.W.von Henke, B.Krieg-Brückner, O.Owe: *ANNA, a Language for Annotating Ada Programs*, Springer, LNCS 260, 1987

[17] J.H.McKinna: *Deliverables: A Categorical Approach to Program Development in Type Theory*, PhD. Thesis, University of Edinburgh, 1992

[18] S.MacLane: *Categories for the Working Mathematician*, Springer, 1971

[19] R.Milner, M.Tofte, R.Harper: *The Definition of Standard ML*, MIT Press, 1990

[20] P.Mosses: CoFI: The Common Framework Initiative for algebraic specification and development. *Proc. 7th. Intern. Joint Conference on Theory and Practice of Software Development*, Lille, Springer, LNCS 1214, 1997, pp.115-137

[21] P.Mosses: CASL: A guided tour of its design, Techn.Rep., University of Aarhus, 1997

[22] V.Novitzká: *Formal Foundations of Correct Programming*, Academic Press Elfa, Košice, 1999

[23] V.Novitzká: Some ideas about the model theory of correct programming, *Proceedings of the 34th Spring International Conference Modelling and Simulation of Systems, MOSIS'2000, Section Information Systems Modelling*, Rožnov pod Radhoštěm, May 2-4, 2000, pp.197-202

[24] V.Novitzká: *Foundations of Correct programming*, Academic Press Elfa, Košice, 1999

[25] V.Novitzká: Foundations of the metatheory of correct programming. A new approach to the metatheory of correct programming I., (submitted for publication), 2001

[26] V.Novitzká: On specifications in the metatheory of correct programming. A new approach to the metatheory of correct programming II., (submitted for publication), 2001

[27] V.Novitzká: From specification to program in the metatheory of correct programming. A new approach to the metatheory of correct programming III., (submitted for publication), 2001

[28] R.Péter: Játek a végtelennel, Budapest, 1974

[29] G.Pólya: *How to Solve It?*, Princeton University Press, 1946

[30] G.Pólya: *Mathematics and Plausible Reasoning*, Princeton University Press, 1954

[31] K.Popper: *The Logic of Scientific Discovery*, Routledge, London, 1994

[32] H.Rasiowa,R.Sikorski: *The Mathematics of Metamathematics*, Warsawa, 1963

[33] A.Robinson: *On the Metamathematics of Algebra*, North-Holland, Amsterdam, 1951

[34] D.T.Sannella,A.Tarlecki: Essential concepts of algebraic specification and program development, *Formal Aspects of Computing*, 9, 1997, pp.229-269

[35] D.T.Sannella: The Common Framework Initiative for algebraic specification and development of software, *Proc. 3rd Intern. Conference of System Informatics*, Springer, 1999

[36] N.Suzuki: *Stanford Pascal Verifier*, PhD. Thesis, Stanford University, 1975

[37] P.Štěpánek: *Mathematical Logic*, SPN, Praha, 1982

[38] A.Tarlecki: Moving between logical systems, *Proceedings of ADT'96*, Oslo, 1996

[39] W.Wechler: *Universal Algebra for Computer Scientists*, Springer, 1991

[40] G.M.Weinberg: *The Psychology of Computer Programming*, Van Nostrad Reinhold Company, New York, 1971

[41] M.Wirsing: Structured algebraic specifications: a kernel language, *Theoretical Computer Science*, 42, 1986, pp.123-249