UDC 004.43

Method of paradigmatic analysis of programming languages

Gorodnyaya L.V. (Institute of Informatics Systems SB RAS, Novosibirsk State University)

The purpose of the article is to describe the method of comparison of programming languages, convenient for assessing the expressive power of languages and the complexity of the programming systems. The method is adapted to substantiate practical, objective criteria of program decomposition, which can be considered as an approach to solving the problem of factorization of very complicated definitions of programming languages and their support systems. In addition, the article presents the results of the analysis of the most well-known programming paradigms and outlines an approach to navigation in the modern expanding space of programming languages, based on the classification of paradigms on the peculiarities of problem statements and semantic characteristics of programming languages and systems with an emphasis on the criteria for the quality of programs and priorities in decision-making in their implementation. The concept of "programming paradigm" is manifested as the way of thinking in the programming process.

The author thanks the organizers and participants of the conferences "Scientific Service in the Internet Environment" (http://agora.guru.ru/display.php?conf=abrau2020&page=subjects&PHPSESSID=qbn3kbhgnk8b6a9g21qi1nkkq2), discussions with which made it possible to understand the main provisions of this article.

This work was partially supported by the Russian Foundation for Basic Research, project No. 18-07-01048-a.

Keywords: *paradigm models, autonomously developed components, teaching system programming, concise definitions.*

1. Introduction

Descriptions of modern programming languages (PL) usually contain a list of 5-10 predecessors and a number of programming paradigms (PP) supported by the language [1,2]. In this article the method of representation of paradigms features of PL definition at the level of semantic systems is considered [3]. Using the method of paradigms analysis it is possible to build a space of constructions supported in the definitions of programming languages and systems (PLS). [4] This space can be the source structure in the selection of criteria of decomposition programs based on the development of statements of problems in the programming process of their solutions, a variety of types of semantic systems of PL and their extensions in the implementation of programming systems (PS) [5]. The technique is shown on the material of four classical programming paradigms without an excursion into the wider space of paradigms, especially new ones, which have not yet received support in well-known programming languages and recognition in the form of examples of debugged programs. The analysis of DSL—languages, which it makes sense to consider as a new meta-level in the field of programming linguistics, is left for the future. The concept of "programming paradigm" does not have a strict definition [4], so the question arises about the belonging of new approaches in programming to the set of PP and the ordering of such a set [6].

Programming paradigm is manifested as the way of thinking associated with the compromise between the characteristics of tasks, methods of their solution in the form of programs, quality criteria of programs adopted in PP and decision-making priorities in the programming process. Such feature of PP allows to understand a paradigm choice as process of acceptance, representation and debugging of decisions at statement of different tasks therefore it is natural to carry out systematization of PP on comparison with priorities and variations of schemes of statement of tasks and methods of their decision.

The most clear systematization of PP now allows to allocate the basic and derivative PPs supplemented by combined, auxiliary and system-forming or perspective-strategic. It should be noted that academician Andrei Petrovich Ershov was focused on strategic PPs, including fundamental, educational and technological. The set of basic PPs can be divided into basic, instrumentally expanding and unlimited depending on the content of semantic systems of computing organization, memory management, computation management and construction of complex data. The classification of the software can depend on 1) the class and the degree of study of the problems being solved and 2) the potential of the used hardware. .

2. Results of paradigm analysis

Analysis and comparison of a large number of PL of different levels allow to identify the most significant characteristics for the expression of paradigm specificity of a wide class of PL (Table 1).

Table 1. PL twenty-first century (all multi-paradigm)

Year	PL	Predecessors	Used paradigms
2018	Dart	Java, JavaScript, CoffeeScript, Go	object-oriented web application framework script language imperative, reflective, functional
2012	Rust	Alef, C++, Camlp4, Common Lisp, Erlang, Haskell, Hermes, Limbo, Napier, Napier88, Scheme, Newsqueak, NIL, Sather, Ocaml, Standard ML, Cyclone, Swift, C#, Ruby	parallel functional imperative structural systemic procedural free software

2005	F#	OCaml, C#, Haskell	functional object-oriented generalized, imperative
2003	Scala	Java, Haskell, Erlang, Lisp, Standard ML, OCaml, Smalltalk, Scheme, Algol68	functional object-oriented imperative
2001	D	C, C++, C#, Python, Ruby, Java, Eiffel	Imperative, object-oriented, functional, procedural, contractual, generalized
2000	C#	C++, Java, Delphi, Modula-3, Smalltalk	object-oriented generalized procedural functional event-driven, reflective

The multiparadigmality of long-lived and new PLs shows the need for more precise detailing of the dependencies between old and new ones. (Table 2.).

Table 2. PL - the founders of the basic programming paradigms

Year	PL	Used paradigms	Sphere of influence
1954 1958	Fortran, Algol-60	**imperative** parallel procedural modular structural procedural **generalized** object oriented	**IPP** — imperative-procedural ALGOL 58, BASIC, C, Chapel, CMS-2, Fortress, PL/I, PACT I, MUMPS, IDL, Ratfor
1958	Lisp	experimental **functional** object oriented procedural **reflective** **metaprogramming**	**FP** — functional CLIPS, Common lisp, CLOS, Clu, Dylan, Forth, Scheme, Erlang, Haskell, Logo, Lua, Perl, POP-2, Python, Ruby, Cmucl, Scala, ML, Swift, Smalltalk, Factor, Clojure, Emacs Lisp, Eulisp, ISLISP, Wolfram Language
1960	APL	**vector** functional structural **modular**	**PC** — parallel calculation A, A+, FP, J, K, LyaPAS, Nial, S, Wolfram language, MATLAB, PPL,
1962	Simula 67	**object oriented**	**OOP (1980)** — object oriented
1968	Forth	imperative **stack oriented**	Factor, RPL, REBOL, PostScript, Factor and other concatenative languages
1968	Algol-68	**parallel** imperative	C, C++, Bourne shell, KornShell, Bash, Steelman, Ada, Python, Seed7, Mary, S3
1972	Prolog	declarative **logical**	**LP** — logical Visual Prolog, Mercury, Oz, Erlang, Strand, KL0, KL1, Datalog
1970	Pascal	imperative **structural**	**SP** — structural programming Ada, Component Pascal, Modula-2, Java, Go, VHD, Oberon, Object Pascal, Oxygene, Seed7, Structured text

3. Semantic systems of basic paradigms

Considering the systematization of the paradigmatic features of the definition of PL at the level of semantic systems [3], it is convenient to classify language concepts by statement of tasks and language tools used to solve them. Even in last times, Nicholas Wirth noted the importance of matching the problem statement and the tools used to solve it, especially if you can catch the likeness of the processed data structures and their processing algorithms, which is now called homoiconicity. Based on this correspondence, it is possible to build a space of constructions supported in the definitions of PSL and compared with the complexity of the formulations of successfully solved problems. The resulting space can be the initial structure when choosing criteria for decomposing programs, taking into account the peculiarities of the development of problem statements in the process of programming their solutions [4], expanding the semantic systems of PL and their refinement when implementing PS [5].

When considering any semantic systems, it is important to do noted the difference in the nature of the performance of the functions of such systems in different complexes. So, for any data set D representing values from the set V of arbitrary nature, function schemes F are realistically distinguishable for calculation methods, memory access tools M, control features of computing C and communication, or reversible complexation and structuring of data S. This leads to an idea of the main categories of semantic systems for differently implemented types of functions. Historically, at the hardware level, such categories of semantic systems have had a cumulative effects in the “DEMCS” order - the representation of numbers, an arithmometer, a calculator with registers, an analog analizer with control system, a computer. Each hardware subsystem can interact with each other. (Table 3).

Table 3. A number of categories of semantic systems of hardware level.

Subsystem	Note
D: data	Data from set D represents values from the set V and the interrupt scale
E: evaluation	Operations on two or one data produce one or two datum
M: memory	The correspondence between addresses from the set N and representations from the set D stored datum at these addresses allows different methods for accessing memory elements, including replacing stored datum, with the exception of address 0
C: control	Comparing data with zero allows to control the progress of calculations along with go by labels and interrupt handling, not counting the transition in order
S: communications	The construction of complex data takes into account the capabilities of addressing commands in memory

The classification of programming paradigms can depend on 1) the space and degree of knowledge of the problems being solved and 2) the potential of technical devices that support programming paradigms, which reflects the operational and implementation pragmatics of the PL that support these paradigms. As new problems appear, new PPs are formed, the recognition of which by specialists requires a significant time, usually 10-20 years after the emergence of tools. The operational pragmatics of a programming language strongly depends on the space of the problems being solved and the practicality of their solutions. The implementation pragmatics of programming systems depends on the range of hardware that can be controlled in PL terms (processor, files and peripherals, networks and servers). In terms of the degree of study, the following spaces of problem statements differ significantly, affecting the choice of methods for solving problems and the complexity of their programming:

- new;
- research;
- practical;
- exact.

New problem statements are characterized by the absence of an accessible precedent for solving the problem, the novelty of the means used, or the inexperience of the performers.

Research problem statements are usually complicated by the requirements of originality and versatility, which can be demonstrated in a computer experiment.

Practical problem setting is aimed at relevance and convenience of multiple use of ready-made solutions.

Exact problem statements include testing the limiting capabilities of the tools used, associated with the degree of organization of the created program and the rank of the implemented solutions. The spread of labor intensity, depending on the degree of novelty or knowledge of the problem statement, is usually about 1 to 8. An underestimation of such a spread usually leads to systematic errors in forecasting the forthcoming labor costs.

4. Differences in programming paradigms

Programming paradigms can be distinguished by the priorities of the categories of semantic systems in the programming process, noting the paradigm differences in the general concepts in each category. Data are addresses and stored values representation in IPP, stored methods and object signatures appear in the OOP, be binding with any data in the FP instead of addresses in memory, and to the identifier in the LP. In IPP and OOP, operations are mostly unary or binary, and in FP and LP there is also arbitrary arity. True datum in LP include the special data “ESC”, which allows to distinguish normal predicate values from failure in calculations, and FP can use any data other than “NIL” as truth. Data structures in the IPP can not be considered as values representation processed by the basic means, and in the FP such structures are processed without special restrictions.

When preparing program an IPP, the most important are the means of working with memory in which data and the results of their processing are placed. Data processing is considered as a change in memory states when performing calculations. If necessary, data structures can be organized.

The focus of FP is the organization of calculations on symbolic representations of the entities of a given subject area. Working with memory in this case may not require binding to physical addresses, but rather confine itself to the representation of an associating function over data pairs of any nature. The control of the computation process can be considered as a function of program fragments. The construction of complex objects is free from the of elements neighborhood.

In the case of LP, the logic of non-deterministic search for feasible solutions dominates. Variants of possible solutions are being choose. Fragments with a fixed number of parameters are named. As structures, samples are used to control the choice of variants.

For OOP, it all starts with defining a hierarchy of classes of objects placed at fixed addresses in memory. The management of the data processing process uses a comparison of classes and valid methods, labeled with access rights from different parts of the program. Computations occur only upon successful matching and matching of access rights to objects. A detailed analysis of the semantics of OOP was performed in [5] and was accompanied by comparison with other software and partial formalization of the main mechanisms.

Thus, in addition to preferences on the features of the problem statements, one can see differences in the schemes for determining functions for different categories of semantic systems depending on the software. It should be noted that the transition from PL to PS is usually accompanied by an increase in the number of supported PPs, which, when defining the Haskell language, led to the concept of “monad”, which allows any PL to achieve practicality, which is usually done with the help of library modules.

Description of derivative PPs can be made relative and, therefore, more concise, expressing the difference with the basic paradigm. We can say that the derivative paradigm is a projection of the basic paradigm on the features of the problem statements of a certain application area. Usually in the projection the most important elements of paradigms are modified. Variations of the models of semantic systems that support derivative paradigms can be used as objective parameters in factorizing the definitions of languages and programming systems and decomposing programs, starting with taking into account the peculiarities of problem statements.

For practice, it is useful to describe the derivatives of PP relative, expressing the difference with the base PP. So, IPP derivatives distinguish different methods of representing data in memory and organizing sequential processes generated by the program, OOP derivatives give various concretizations of the concept of “class of objects”, FP derivatives represent variations in the methods of organizing calculations, and LP derivatives may use different approaches to mitigate the dependence of obtaining results on excessive or insufficient determinism.

In addition to the relatively clear classical basic paradigms of programming, there is reason to single out the main expanding system-instrumental paradigms aimed at the preparation and design of programs, operating systems and databases, support for working with files and various device configurations, as well as providing feedback when executing any programs. All expanding paradigms, some of them have not yet received their names, work with much more complex elements that have their own lives, which can be included in many systems and configurations in which their state can be changed. Data representations, in addition to complex data structures, formal definitions and codes, include processes, devices, roles of participants and complexes. The methods of processing elements and their interactions are subject to more stringent requirements of correctness, which entails supporting the improvement of elements in parts, that is, targeted development as errors are identified or the need for increased efficiency. There is a division of labor according to skill level and responsibility.

No less noticeable is the group of unlimited communication interface paradigms supporting large data processing (bigdata, sematic-web, rdf), remote work in networks, service-oriented programming based on markup and rewriting languages (html, XML, PHP), parallel, vector-oriented for processing arrays (APL) or supporting multiple theoretical insertion mechanisms, including dynamic insertion substitutions (SETL) and high-performance computing on supercomputers (OpenMP, mpC) and mobile devices.

There is a noticeable number of combined PPs that combine the advantages of a pair of PPs for solving different types of subproblems, which also are supported by multi-paradigmal PLs (Lisp 1.5, Planner, Merlin, F #, C #, Scala, etc.). There are rejected PPs that have not received recognition by the programming community, and esoteric PPs, the invention of which can be considered as a study the possibilities to represent and recognize information in the style of creating and decoding puzzles.

Any programming paradigm can be supplemented with additional forms, such as declarativeness, abstractions, specification languages, etc., mainly solving problems such as “scaffolding,” that is, the aim of these forms is not an alternative or opposed representation of programming tools and methods, but temporary structure which used to support setting the boundaries of the behavior of programs, highlighting the processes that are convenient for practice.

5. Conclusions and Outlook

The Most long-lived programming languages support the practical paradigms of impertive-procedural, object-oriented, and functional programming. A growing number of languages support declarative, reflexive and meta-programming. New languages complement these paradigms with separate networking tools, leading to the formation of paradigms for remote access, parallel processes, supercomputer computing, and large data visualization. The complexity of multi-paradigm PLs can be overcome through the style of separate description of the paradigms supported in them [8].

The proposed methodology can be used to assess the complexity and complexity of programming, especially if supplemented by dividing the requirements for setting tasks in the fields of application into academic and industrial, and by the level of knowledge into clear, developed and complicated difficult to certify requirements.

Basic programming paradigms can be distinguished by ordering the main categories of semantic systems, and derivatives - by the difference between individual categories of semantic systems from the basic paradigm. Any programming paradigm can be enriched with additional paradigms for representing restrictive conditions for the functioning of programs. For this reason, they cannot be opposed to the actual PP. The classification of programming paradigms can depend on the degree of knowledge of the expanding space of tasks to be solved and the progress of available technical means that occurs within the framework of a stable class of tasks to increase efficiency.

The statements of the problems of parallel computing take into account that the speed of obtaining results on the available programs for solving a specific problem is insufficient. Paradigms of this direction are in the process of formation. Given the diversity of theoretical models in this area, it is natural to assume that there will be many such paradigms. There is reason to single out software aimed at providing feedback when working with devices and networks, on the surface-interface style of IT, and on supporting supercomputer processes.

In recent years, reasons have been discovered and understood for conditioning program verification by formalizing the programming paradigms used. Programming projects should be accompanied by a justification for the choice of not only software tools, but also paradigms in order to avoid inter-paradigm conflicts, fraught with subtle errors associated with changing and developing the functioning environment of long-lived programmable components. It is necessary to consider the difference between theoretical and practical programming, similar to the difference between elementary music theory and the performing skill of a musician.

DSL languages deserve special consideration as a new level of languages creation. If in ordinary PLs, the accumulation of programming experience is performed in the form of procedures, then DSL is the mechanism for accumulating experience in the form of languages.

The works of E. M. Lavrishcheva [7], Peter Van Roy [8] and Peter Wegner [6] should be mentioned as related works. E.M. Lavrishcheva presented a fairly complete overview of programming paradigms that is relevant for programming technologies [7], P. Van Roy analyzed more than 30 paradigms, mainly combined, and presented a diagram of their interconnections cited in Wikipedia [8], and P. Wegner performed a very serious analysis of OOP, methods for supporting this paradigm, and its comparison with other classical PPs [6].

References

1. https://www.levenez.com/lang/ — Computer Languages History.
2. http://progopedia.ru/ — Encyclopedia of programming languages (171 languages and 31 paradigms) *(In Russian).*
3. Lavrov, S.S. Methods for definition the semantics of programming languages (Metody zadaniya semantiki yazykov programmirovaniya) // Programmirovaniye, 1978. № 6. S. 3–10 *(In Russian).*
4. Floyd, R. W. (1979). The paradigms Programming .-- Communications of the ACM 22 (8): 455
5. Gorodnyaya L. V. On the presentation of the results of the analysis of languages and programming systems (O predstavlenii rezul'tatov analiza yazykov i sistem programmirovaniya) // Nauchnyy servis v seti Internet: trudy XX Vserossiyskoy nauchnoy konferentsii (17–22 sentyabrya 2018 g., g. Novorossiysk). M.: IPM im. M.V. Keldysha, 2018. https://keldysh.ru/abrau/2018/theses/46.pdf *(In Russian)*
6. Peter Wegner. Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess. 1, 1 (August 1990), P. 7–87. https://pdfs.semanticscholar. DOI: http://dx.doi.org/10.1145/

7. Lavrishcheva E. M. Software engineering and technologies for programming complex systems. Textbook for universities. (Programmnaya inzheneriya i tekhnologii programmirovaniya slozhnykh sistem. Uchebnik dlya vuzov). M., 2018. 432 s. *(In Russian)*
8. Peter Van Roy. Classification of the principal programming paradigms. url: https://www.info.ucl.ac.be/~pvr/paradigms.html

1. https://www.info.ucl.ac.be/~pvr/paradigms.htmlhttps://www.info.ucl.ac.be/~pvr/paradigms.html