Method of Paradigmatic Analysis of Programming Languages and Systems

Lidia Gorodnyaya[1,2][0000-0002-4639-9032]

[1] A.P. Ershov Institute of Informatics Systems (IIS), 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia

[2] Novosibirsk State University, Pirogov str., 2, Novosibirsk 630090, Russia
lidvas@gmail.com

Abstract. 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 proposed method can be used to assess the complexity of programming, especially if supplemented by dividing the requirements of 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.

Keywords: Definition of Programming Languages, Programming Paradigms, Definition Decomposition Criteria, Semantic Systems

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). 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 [4], 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, so the question arises about the belonging of new approaches in programming to the set of PP and the ordering of such a set. 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.

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). [1]

[1] The listings in Table 1 and in Table 2 are based on open sources such as Wikipedia and study guides. Lists can be replenished and updated by specialists.

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

[2]Year	PL	Predecessors	Used paradigms
2018	Dart	Java, JavaScript, CoffeeScript, Go	object-oriented web application framework script language imperative reflective functional
2014	Swift	Objective-C, C++, Java, Rust, Scala, Python, Ruby, Smalltalk, Groovy, D, LLVM	protocol-oriented object-oriented functional imperative
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 contractual generalized procedural
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).

[2] Saved vocabulary sources.

Table 2. PL - the founders of the basic programming paradigms [3]

Year	PL	Used paradigms	Sphere of influence
[4]1954 1958	Fortran, Algol-60	**imperative** parallel procedural modular structural procedural generalized object oriented	**IPP** 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** 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** A, A+, FP, J, K, LyaPAS, Nial, S, MATLAB, PPL, Wolfram Language
1962	Simula 67	**object oriented**	**OOP (1980)**
[5][6]1968	Forth	imperative **stack oriented**	Factor, RPL, REBOL, PostScript, Factor and other concatenative languages
[7]1968	Algol-68	**parallel** imperative	C, C++, Bourne shell, KornShell, Bash, Steelman, Ada, Python, Seed7, Mary, S3
1972	Prolog	declarative **logical**	**LP** Visual Prolog, Mercury, Oz, Erlang, Strand, KL0, KL1, Datalog
1970	Pascal	imperative **structural**	**SP** Ada, Component Pascal, Modula-2, Java, Go, Oberon, Object Pascal, Oxygene, Seed7, VHD, Structured text

[3] IPP – **imperative-procedural,** FP – **functional**, **PC – parallel calculation** OOP – **object oriented** , LP – **logical**, SP – **structural** programming.

[4] Algol-60 – it was from this language in our country that acquaintance with high-level languages began, its dominance was pushed back by the appearance of Fortran language implementations.

[5] Forth – a typical mechanism for implementing work with expressions in different Pls.

[6] The languages in which programs are built as concatenations of functions.

[7] Algol-68 represents the result of well-thought-out unification and orthogonalization of basic programming concepts.

The programming paradigm as a way of thinking is associated with a compromise between the features of the tasks being solved and the methods for solving them using programs. The most objective programming concepts are associated with architectural models, with methods for implementing a joint projects, and with the classification of problems to be solved. To show the features of software, it is convenient to single out conceptual monoparadigmal languages, models or sublanguages and provide criteria for the successful use of software with evaluating the results using examples of programs that was confirmed by programming practice. [5]. From the vast set, a small number of PLs can be distinguished, attracting attention with interesting combinations of visual means and semantics that affect the development of the main PPs.

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 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 V and the interrupt scale
E: evaluation	Operations on two or one value produce one or two values
M: memory	The correspondence between addresses from the set N and

	representations from the set D stored values at these addresses allows different methods for accessing memory elements, including replacing stored values, with the exception of address 0.
C: control	Comparing values with zero allows you 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

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 (See Tables 4-7). Data are addresses and stored values in IPP, stored methods and object signatures appear in the OOP, be binding with any value 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 PL there is also arbitrary arity. True values in LP include the special value “ESC”, which allows to distinguish normal predicate values from failure in calculations, and FP can use any value other than “NIL” as truth. Data structures in the IPP can not be considered as values processed by the basic means, and in the FP such structures are processed without special restrictions.

When preparing an imperative-procedural program, 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 (Table 4).

Table 4. Paradigmatic scale of IPP semantics.

Subsystem	Note
D: data	Values are limited by the size of their representations in memory registers at addresses from N. The interrupt vector is not represented.
E: evaluation	Operations differ on unary and binary with single result.
M: memory	Work with memory without emphasis on the zero, a methods variety for accessing memory and interrupts handling.
C: control	In addition to hardware comparison of values with zero, when controlling the course of calculations, operation priorities and parentheses in expressions are used along with transfers by labels, but without interrupt handling.
S: communications	You can design complex data and select its elements using the capabilities of in-memory indexing instructions

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 (Table 5).

Table 5. Paradigmatic scale of FP semantics.

Subsystem	Note
D: data	Representations of data are not limited in size and complexity
E: evaluation	Some operations can process any number of parameters and produce a series of values, if necessary combined into a complex given
M: memory	When processing complex data, the old values are not changed, and the new values are located in memory independently, and the correspondence between the associated data is stored in memory, allowing a change in association
C: control	Any calculation can be blocked or run. A program may contain branch points from an arbitrary number of branches selected by comparing the result with a “zero” (NIL), which is included in the set of values
S: communications	It is possible to construct arbitrarily complex, equitable with elementary, data, all elements of which are accessible using functions in any expression

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 (Table 6).

Table 6. Paradigmatic scale of LP semantics.

Subsystem	Note
D: data	Representations of facts or rules
E: evaluation	Attempts at a calculation that yields either a result or a signal of impracticability, which leads to further search for variants
M: memory	Some rules may have names with indicating the number of parameters, which allows them to be used as functions
C: control	Non-deterministic search for choosing a feasible variants giving a result other than “ESC”
S: communications	Comparison of complex data with a example allows you to select elements for choosing a branch

For object-oriented programming, 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 (Table 7).

Table 7. Paradigmatic scale of OOP semantics.

Subsystem	Note
D: data	Data represents not only values, but also methods for processing them.
E: evaluation	Any operations, as well as any calculations, can be overloaded by adding the processing of possible interrupts
M: memory	Dosed access to elements of objects is accompanied by mechanisms of implicit situation handlers, and addresses can be values
C: control	The feasibility of methods on objects is determined by checking their compatibility and access rights in the class hierarchy
S: communications	Object classes are adapted for additional definition and inheritance according to the class hierarchy

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.

4 Conclusions and Outlook

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 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.

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].

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

References

1. Diagramma, predstavlyayushchaya khronologiyu poyavleniya i nasledovaniya mnogikh YAP, https://www.levenez.com/lang/, . last accessed 2019/11/21.
2. Sayt s opisaniyami 171 yazyka i 31 paradigmy, http://progopedia.ru/, last accessed 2019/11/21.
3. Lavrov, S.S.: Metody zadaniya semantiki yazykov programmirovaniya. Programmirovaniye 6, pp. 3–10 (1978).
4. Bentley, D.: Zhemchuzhiny tvorchestva programmistov. Radio i svyaz', Moscow (1990).
5. Gorodnyaya, L.: On the Presentation of the Results of the Analysis of Programming Languages and Systems. In: CEUR Workshop Proceedings, vol. 2260, pp. 152-166 (2018).
6. Wegner, P.: Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess 1(1, August), pp. 7–87 (1990), http://dx.doi.org/10.1145/.
7. Lavrishcheva, E.M.: Programmnaya inzheneriya i tekhnologii programmirovaniya slozhnykh sistem. Moscow (2018).
8. Van Roy, P.: Diagramma s rezul'tatami sravneniya boleye 30-ti paradigm programmirovaniya, https://www.info.ucl.ac.be/~pvr/ paradigmsDIAGRAMeng108.pdf, last accessed 2019/11/21.