ISSN No. 0976-5697

Volume 4, No. 10, September-October 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Logical reconstruction of programming language paradigms

Davor Lauc
Faculty of Humanities and Social Sciences
University of Zagreb Zagreb, Croatia
dlauc@ffzg.hr

Abstract: The concept of programming language paradigms is a one of fundamental concepts of computing, but the present usage of the term is quite chaotic. Using method of logical reconstruction programming paradigms are modeled by original logical models of computation that are considered paradigms of programming language paradigms. The space of the programming languages is visualized as a prism, with edges of imperative, function and logical paradigms corresponding to basic models of computation, and depth axis as degree of modularity of languages. Actual programming languages are represented as occupying some space in such a model. Finally, the model is evaluated for completeness with regard to existing programming languages.

Keywords: programming language paradigms, models of computations, logical reconstruction

I. INTRODUCTION

The concept of programming languages paradigm is one of the most elusive concepts of computing. It was introduced by Floyd [1] when there already were a large number of programming languages and programming styles. The number of programming languages has now increased manifold with new languages designed almost every day. The need for precise and complete concept of programming language paradigms has become evident. However, as the original concept was not precise and well defined, the present usage of programming paradigms is quite chaotic. Most of the usage includes imperative and declarative paradigms; often, functional, logical, structural and object-oriented paradigms are also included. A variety of other paradigms like procedural, visual, modular, process-oriented, event-driven, automata-based, agent-oriented, concurrent and so on can be found in scientific, educational and industrial texts. Some of the usages are overlapping, some may even be synonymous, but most of them are partial, mentioning only some of the paradigms. Although a few interesting articles about programming paradigms have been published, including an outstanding analysis by van Roy and Hardi [2], there is no comprehensive analysis of the concept that is both simple and precise.

As for most fundamental concepts, it is next to impossible to provide some kind of informal or formal definition that is non-circular or even meaningful. However, this concept seems to be the perfect candidate for the "old fashion" method of philosophical analysis - logical reconstruction. Instead of taking the linguistic approach for programming languages and trying to enumerate the different usages of the term, and then relate them to the vast number of languages, it is more productive to construct a formal model that will demonstrate the main features of paradigms in a precise and complete manner.

Owing to the close connection between programming languages and logical models (formalizations) of computation, it is natural to try and reconstruct programming paradigms on the basis of models of computation. Although many models of computation are designed, they are outnumbered by computational languages in the order of magnitude. Also, models of computation are much more elegant and basic than programming languages that evolved facing complex real-world problems. Hence, foundational concepts, styles and features of programming languages should be easier to discover in the models of computation than in vast number of programming languages.

II. PARADIGMS OF PROGRAMMING LANGUAGES

The concept of paradigm of programming was introduced by Floyd in 1978 at his Turing award lecture [1]. Floyd applied the Kuhnian term paradigm to programming languages and practices of software development, entertaining the idea of similarity of activities of programming and scientific research. The structural paradigm is the only paradigm explicitly mentioned in his lecture that is still in use as a programming paradigm although implicitly functional paradigm was also mentioned. Floyd gives a few examples of paradigms like dynamic, branch-and-bound, divide-and-conquer, formula-manipulation, state-machine and so on. Most of them today would be characterized either as software development methodology or specific programming technique. There is no attempt to define the concept of paradigm or to distinguish it from similar concepts.

The analysis of Floyd's concept of paradigm should take into consideration that it was introduced in an informal style, aimed at the designers of programming languages and educators, and was not a formal introduction to a new concept. His usage of Kuhn's concept of paradigm should be taken metaphorically. In spite of the fact that Kuhn's paradigms do not have very precise meanings, it seems impossible to apply them directly to programming languages. Computer programming is hardly an empirical science and although there are communities connected with the specific programming paradigm that bears some similarities to communities formed by the empirical scientist, important features of paradigms as conceived by Thomas Kuhn are missing. The development of paradigm does not resemble the development of science, i.e., it would be implausible to search for periods of normal programming and "programming" revolutions. Although there are significant differences among programming paradigms, they are definitely not incommensurable. One of most evident proofs of this is the existence of multi-paradigm programming languages, which are present even in Floyd's analysis.

Regardless of the original concept, programming paradigms today represent the important concepts of programming and computing in general. The concept is mainly introduced in advanced programming textbooks or articles discussing programming languages in general. Due to various unsystematic usages of the concept, it is hard to trace the origins of the present usage of programming paradigms and nearly impossible to list them all, but the following list of paradigms should be good enough for the purpose of this paper:

a. Imperative paradigm - the most important and widespread programming paradigm, characterized by programming statements that change program state. It is often contrasted with declarative paradigm.
b. Declarative paradigm - often defined negatively as programming without describing imperative control flow. Alternatively, it is defined as programming without the possibility to use/change the program state. It may also be defined as programming that defines what the computation should do and not how. Sometimes it is used as synonymous with logic paradigm.
c. Procedural paradigm - sometimes synonymous with structural and sometimes even with imperative paradigm, characterized by organizing code around subroutines - procedures.
d. Functional paradigm - characterized by computation by valuation of functions, without changes of state - side effects. Sometimes included as a part of declarative paradigm.
e. Logic programming - characterized by use of logical statements for computing, stating relations and queering them without defining how queries are answered.
f. Object oriented paradigm - characterized by objects - data structures consisting of data and methods that change them, so that changing state of one object does not affect other objects.

There are other usages of the term paradigms in literature and programming practices, but most of them are excluded from this analysis. In order to obtain precision, every formal analysis of concept has to exclude some meanings of the analyzed concept, so we will not consider the following four groups of programming styles or languages as part of our definition of programming paradigm. The first group includes various software development methodologies, that is, ways or principles of developing programming systems. This includes parts of structural paradigm like top down method that defines how programmers should attack a problem. Likewise, newer methodologies like rapid application development or extreme programming are not considered paradigms according to our logic. Of course, software methodologies are related to programming paradigms as some paradigm support or even to enforce specific methodology, but it does not make them paradigms. The second excluded group is related to the way programmers interact with software development systems - so systems like visual programming or automated programming are not considered paradigms. The third group is connected with the interaction between program and machine (program environment) as well as between program and user. That excludes parallel programming and systems like event driven programming. The fourth excluded group includes languages that are not Turing complete, like standard SQL and domain specific languages.

Thus, our definition of programming paradigm should define general features and properties of programming languages that can be read/analyzed from written program code alone, regardless of the process by which the code is achieved or how it will be executed by the machine. In a way, this resembles famous positivistic distinction between context of discovery and context of justification. It seems plausible to try to find features of those paradigms in models of computation that can be seen as archetypical programming languages, or paradigms of programming language paradigms.

III. MODELS OF COMPUTATION

Like most other fundamental concepts, ideas of computation can be traced back to antiquity, and then from the bold imagination of Descartes, Pascal and Leibniz to the logicians and mathematicians like Charles Babbage, Leopold Kronecker and David Hilbert. The contemporary concept of computation owes its existence to the research in mathematical logic embarked upon by Kurt Gödel, Alonzo Church, Alan Turing and Emil Post in the 1930s. They have designed, by and large independently, four models of computation - (partial) recursive functions, lambda calculus, Turing machine and (Post) production systems. It was soon proved that those models are equivalent in computational power, in the sense that any computation that can be done by one of them can be done by all. The importance of this is best expressed by Gödel saying "...with this concept, one has, for the first time, succeeded in giving an absolute notion to an interesting epistemological notion, i.e., one not depending on the formalism chosen" [3].

However, the fact that those models have equal computational power does not make them equally convenient for solving different computational problems, something that is obvious to every student trying to use those models. Owing to differences in those models, they can be seen as "paradigms" of the models of computation, i.e., all other models of computation resemble some of them in the key features. Without examining those well-known models in too much detail, the following analysis describes some of their key features.

The most well-known model of computation is the Turing machine, designed by Alan Turing [5]. It is a model of abstract machine that manipulates symbols on (potentially) infinite tape. The key feature of a Turing machine is that computation is fully determined by a finite set of instructions that change the state of instructions and symbols on tape, which together can be considered as the state of machine. Which instruction will be executed depends solely on the state of the machine. So the whole process of computation can be seen as a step-by-step process that transforms the state of machine according to instructions, leaving the machine in a distinct determined state after every instruction. Computation is defined by the list of such state-changing instructions. Solving computational problems involves representing input and output symbols, and writing instructions that gradually transform input symbols to the desired output.

Turing machine can serve as a paradigm for other models of computation based on transforming its states like register machines, push-down automaton, random access machines and so on. The indication of closeness of those models is the relative triviality of equivalence proofs among them.

Recursive function is the model of computation with a substantially different approach from the Turing machines. Computation is done by combining a few primitive functions (zero, successor and projection) through composition, primitive recursion and minimalization. There is no instruction that changes states, and the computation is defined by a set of descriptions or definitions of functions that combine to form the main function and perform the desired computation. Computational problems are solved by defining functions that combined, mostly by composition or recursion, computes right values for given arguments.

Resembling the approach to computation is the lambda calculus, a very elegant model of computation that defines function as lambda terms combined by abstraction and application that are computed using (beta) reduction. Computation is done by designing lambda terms, which when applied to other lambda terms, results in desired values. In order to solve computational problems, values and arguments are represented by lambda terms and then define sets of lambda terms that are combined to perform the desired computation.

The other models of computation that are based on functions can be seen as same paradigms as recursive functions and lambda calculi.

The fourth archetypical model of computation designed in the 1930s is the post canonical system, today considered as the string-rewriting system. In this system, computation is done by rewriting a set of rules, which when applied to input strings, transforms them to give the desired results. Computational problems are solved by writing a set of rules that are not like recursive functions and lambda calculi combined into one "main" function that perform the computation; instead, all the transformations that are applicable are applied. The whole family of rewriting-systems, including type-0 grammar in Chomsky hierarchy and Markov algorithms, naturally belongs to this paradigm. However, models like logic programming and constrain programming that seem substantially different, share style and features of this paradigm.

IV. MODELS OF COMPUTATION AS PARADIGMS OF PROGRAMMING LANGUAGES PARADIGMS

Three computation paradigms could be identified from the analyzed models of computation. Using the terminology of programming paradigms, there are the imperative paradigms exemplified by the Turing machine and the functional paradigms with exemplars of lambda-calculus and recursive functions. The third paradigm could be called logical with exemplars of string-rewriting systems.

The main feature of pure imperative paradigm is the existence of an explicit machine or program state, and computation is performed by changing the state with the execution of instructions until the final state is achieved. On the other side of the spectrum, there are pure functional and logical paradigms that are stateless, and computation is done by describing or declaring the computational problem. Such paradigms are typically named declarative. 'Declarativeness' being a matter of degree can be imagined as one axis in the universe of programming languages. The differences between functional and logical paradigms can be measured by the organization of declarative statements that are considered in computational request. If there exists one main function that is executed, the paradigm is functional; if many statements, regardless of their order, are considered, the paradigm is logical. The three main paradigms can be visualized by the following diagrams:

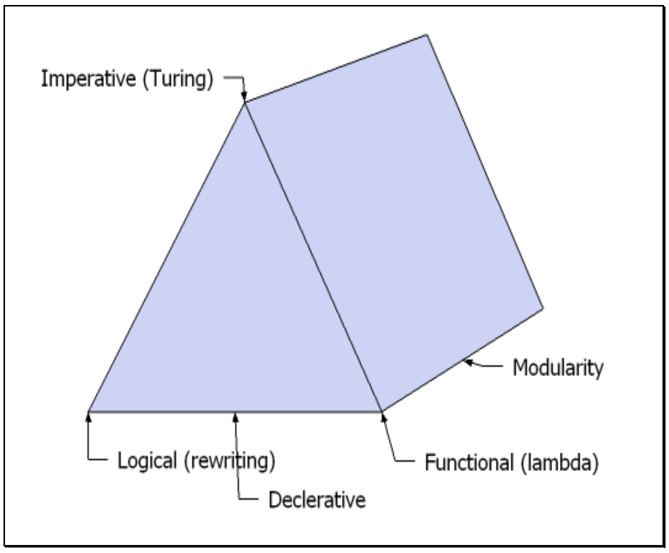

Figure: 1

Programming languages can be seen as occupying some areas of the diagram. They can occupy a small space close to the vertices that would make them pure exemplars of the corresponding paradigm, like Lispkit Lisp or Unlambda for functional paradigm. However, most of the real-life, complex programming languages occupies larger area that makes them more or less multi-paradigm languages.

The metrics for charting real programming languages on diagrams can be some standardized measurement of the length of proof of the Turing completeness in the computational models. The shorter and more elegant the proof, the closer is the language to the corresponding paradigm. The opposite is also true, and the implementation of some computational models is easier and more elegant in the programming language that belongs to the same paradigm. Another indicator of the closeness of some programming paradigms to the corresponding computational paradigms can be the formal semantics of the programming languages. Although all three major approaches can naturally provide meaning to all paradigms, it is the most elegant and straight forward to model the functional paradigm in denotational semantics, the imperative in operational semantics and the logical paradigm in axiomatic semantics.

There are two important and successful paradigms, namely structural and object-oriented paradigms that do not fit naturally into the above model. It is not surprising because they have evolved over a decade-long struggle of applying computation to various real-world problems, something models of computation were neither designed nor imagined for. However, important features of those and similar paradigms can be reduced to the concept of modularity. Modularity can be defined as the possibility to isolate one part of computation from another. Like the imperative-declarative axis, modularity is a matter of degree, so it can be visualized as a third dimension of the above diagram. Some programming languages are only modular, some completely non-modular, but most of the contemporary languages occupy larger spaces supporting, but not completely enforcing, modularity.

V. CONCLUSION

There are many properties by which the above model could be evaluated like precision, simplicity, fruitfulness and so on, but the property of completeness seems to be the most interesting. Completeness of the model means finding out whether every programming language, which exists now or will be designed later, can be naturally represented as occupying some of the space of the model. Two concepts of completeness could be identified, stronger and weaker completeness. Stronger completeness claims that there does not exist, nor will exist, a programming language that will exit the borders of the model or be completely outside of the model. The stronger completeness collapses to the famous Church-Turing thesis in the sense that every existing programming language is equivalent to the existing models of computation, as every future programming language will be.

The weaker completeness is a softer notion, meaning that the model represents all the main programming paradigms, in the meaning fixed above. Is there a paradigm or programming language that does not map naturally to the model? This paper claims that a model is complete in this sense for existing languages, but it does not mean that new paradigms will not emerge that would require modification of the model. This could either be the discovery of a completely new model of computation, which has not happened in the last 70 years, or the design of a new important feature like modularity that would dramatically change world of computing.

VI. REFERENCES

[1] Floyd, R.W.: ‘The paradigms of programming’, Journal of Personality Communications of the ACM archive, 22/8, pp. 455-460. 1976.

[2] van Roy, P. Haridi, S.: ‘Concepts, Techniques, and Models of Computer Programming’, MIT Press. 2004.

[3] Gödel, K.: ‘Remarks before the princeton bicentennial conference on problems in mathematics.’ In [4], pp. 84-88. 1946.

[4] Davis, M., editor. The Undecidable. Raven Press, Hewlett, New York. 1965.

[5] Turing, A. [1936]. On computable numbers with an application to the entscheidungsproblem. Proc. London Math. Soc., 42:230-265. In [4], pp. 116-154. 1965