Computer Science • 19(3) 2018 https://doi.org/10.7494/csci.2018.19.3.2863

Iurii Petrov
Andrey Alexeyenko
Galina Ivanova

USING ERLANG IN RESEARCH AND EDUCATION IN A TECHNICAL UNIVERSITY

Abstract *This paper addresses the problem of using functional programming (FP) languages for research and educational purposes. In order to identify the problems associated with the use of FP languages such as Erlang, an experiment consisting of two surveys was performed. The first survey was anonymous and aimed at establishing whether the participants prefer object-oriented or functional coding. The second one was a survey made after the students finished an Erlang course. The results of these two surveys demonstrate that functional programming is underrated with no apparent reasons. Possible steps to address this problem are suggested.*

Keywords parallel programming, distributed programming, teaching, graph theory, concurrency, recursion, survey, functional programming

Citation Computer Science 19(3) 2018: 333–343

1. Introduction

The BMSTU "Computer Systems and Networks" sub-department prepares bachelors and masters candidates of science in different disciplines: software engineering, hardware engineering, and hardware. Students study procedural (Pascal, C, Assembler) and object-oriented (Object Pascal/Delphi, C++, Ruby) languages. However, the functional paradigm is taught only in theory and not in practice. At the moment, object-oriented programming is commonly perceived as the only modern programming paradigm. Although this opinion is not so frequently found in the professional literature, it is still often expressed in online resources [10–13]. Students of most Russian university programs study Pascal/Delphi, C/C++/C#, Visual Basic, Java, JavaScript, and Ruby. It is claimed that procedural programming is suitable for embedded systems whereas functional programming is outdated and obsolete. In addition, one can refer to statistics of the most-mentioned books on StackOverflow [9]: when 40 million questions and answers were analyzed and a list of 5072 books was retrieved, 15 were solely dedicated to functional programming, while 8 out of the top 10 books contained tag "object" (out of the top 30 books containing tag "functional programming"). The authors would like to challenge this attitude by using our experience in research as well as our time teaching the course "Parallel programming" at BMSTU. We also refer to state-of-the-art worldwide experience.

2. Overview of approaches and paradigms

2.1. Asynchronous programming

Asynchronous programming is a paradigm that assumes the independent occurrence of events (for instance, a server process waiting for a signal) when it cannot be predicted which signal will be sent by client [4] and when.

In fact, the code behavior is event-driven; i.e., dependent on external signals such as messages. Many actors (independent processes) pass messages to each other, which makes them change states and/or compute. The Erlang programming language is an example of a concurrent asynchronous language; it possesses the following features:

- it is a managed language, since it uses a virtual machine for code execution,
- many thousands of processes can exist on one virtual machine,
- the processes are actors, so that they are completely unaware of other processes except for process identifiers or given names (and only when this information is explicitly given).

The latter feature is required because processes that do not know anything about each other cannot communicate, as message passing is the only possible type of communication between two actors.

2.2. Parallel programming

Parallel programming is a type of code execution when many calculations are performed simultaneously. Unlike concurrent programming, the code is run on different processors (or processor cores) without interruption. Apart from this, parallel programming is a programming paradigm that allows us to optimize the code (including data structures) for parallel execution.

2.3. Pattern matching

Pattern matching is a process of comparing a given sequence with predefined patterns. It is a very flexible tool in functional programming by which a value, type, or even structure can be checked immediately: do one action if X equals 1 but do another action if X is equal to 1.0. Also, patterns do not have to be fully predefined; free (or anonymous) variables are always used for successful matching. As an example, a third action should be performed if the given X equals to 2 or any other value. Patterns should be prioritized so that all patterns should be applicable in theory (an example of a strict pattern: X is equal to 1; a less strict pattern: X is an integer). Importantly, if a less strict pattern would be ranked higher in a matching list, then all integer X variables (including 1) will match it so that the code associated with the strict pattern will never be computed. Among others, pattern-matching is used in Markov algorithms.

2.4. Recursion

Recursion is a phenomenon when a piece of code or a data structure is defined throughout itself. An example of a recursive function is the factorial. If variable X given to the factorial function is 0 (base clause), then it returns 1. If X is greater than zero, then it returns $X * factorial(X - 1)$.

An example of a recursive data type is the list in Erlang. Each list can either be empty or contain a head and a tail. A tail is usually a list that can either be empty or contain a head and a tail. The recursion is widely used in pure (or close to pure) functional languages because of its immutable variables (which means a variable cannot be used in an iterative cycle). Indeed, cycles (for, while, until, etc.) do not exist in pure functional languages; therefore, each cycle has to be represented as a recursive function.

2.5. Distributed programming

Distributed programming could be described as a parallel or concurrent programming for distributed systems. A distributed system is a system with physically distributed components that can communicate with each other through a network using message passing. Commonly, all nodes in such systems are equivalent (homogeneous distributed systems), although their roles can differ in a certain way (heterogeneous distributed systems). Components of a distributed system are independent from the

point of view of reliability engineering. A system where the failure of one or more components would not lead to the denial of service (DOS) is called fault-tolerant or robust. The fault tolerance of computing nodes can be achieved at different levels: hardware, software, or both. In the case of hardware fault-tolerance, the duplication and reservation of the components is used.

In the case of software fault-tolerance, software is built layer by layer; the main principles are as follows:

- error isolation: an error from a lower level does not affect the processes on the higher levels or, in the best case, does not affect the processes on the same level or the lower levels,
- supervision: a relationship between processes when one of them is able to react to the unexpected termination of the other.

The Erlang programming language was created for distributed soft real-time fault-tolerant systems and offers to use both of the main principles of software fault-tolerance: the architecture of any software system should be hierarchical, and some processes from the higher levels (supervisors) have to intercept error messages from their child processes from the lower levels. Such a hierarchy is called a supervision tree. A supervision tree can be distributed when a process on one node can be a supervisor for the processes on other computing nodes (or on the same physical computing node but on a different Erlang virtual machine).

2.6. Functional programming

A functional programming paradigm relies on the use of mathematical functions. This paradigm is declarative; i.e., the programmer writes code of what to do rather than how to do it. Functional languages usually have fewer side effects, such as functions that change the internal states or do not have such effects at all (pure functional languages).

The immutability of the data is an important feature of functional programming: it is impossible to write $X = X + 1$ in most of the functional languages because the X variable is already bound to a value and cannot be modified in the local function. In addition, many functional languages do not have global variables. In a functional language, a function can be stored in a variable (lambda functions or fun-evaluations in Erlang).

In functional programming, an important role belongs to lambda functions (anonymous functions or fun-evaluations in Erlang) and higher-order functions (functions receiving other functions as arguments and/or returning functions as results). As mentioned above (see "Recursion"), many functional programming languages do not have iterative cycles and instead employ recursive functions for implementing cyclic operations.

3. Methodology

Two surveys were performed: a code-readability survey and a survey among BMSTU master students. Erlang was also used in the research.

3.1. Code-readability survey

The experiment regarding code readability was completed via Google Forms and involved 94 participants. Two anonymous code listings were offered, both implementing the same algorithm PageRank by Google [1]. The choice of PageRank was motivated by the fact that it is well-known and used in many scientific fields beyond IT [6]. The listings did not contain comments. The first listing contained 18 lines (with empty lines 14 without) of Google Pregel code, which represents object-oriented programming (OOP). The base code for this listing was taken from [7]. The second listing contained 21 lines of Erlang code (with empty lines – 16 without) for an earlier proposed model (see below). The original listings did not contain line numbers.

3.2. Student experience survey

The second experiment took place during and after the course "Parallel Programming" during April-May 2016. Masters students (35 in total) were asked about the difficulties they experienced while mastering Erlang. The answers were supposed to be given in a free form. The answers were analyzed, which included charting the most frequent problems and most popular explanations. Of note, the answers had to first be processed by generalizing them for better identification of the students preferences since they were in a free form.

3.3. Erlang in research

For a research example, a parallel asynchronous graph representation model had been suggested [8]. By now, the problem of graph representation in distributed heterogeneous systems is considered unsolved [5, 15]. In the case of a dynamically changing graph structure, the problem is complicated by the need of dynamic load balancing. Briefly, in [8], the author investigated and classified all existing graph representation models and identified their drawbacks. Based on this, a new model was suggested in order to alleviate or eliminate the drawbacks. For implementing the model, the Erlang programming language developed by the Ericsson company was chosen for distributed fault-tolerant soft real-time systems [2]. For the experiments, typical basic operations were chosen, such as a search for the maximum weight edge or a search for a path between two vertices in a De Bruijn graph [3,14]. The results of the study were presented at a number of events, including the international functional programming conference "Lambda Days 2015" in Krakow, Poland. The research demonstrated the superiority of the proposed model over a traditional representation as an adjacency matrix, implemented in C++ and OpenMP.

The code-readability survey was performed in April 2016. The survey method was described in the previous section. The respondents were asked the following questions:

- Which of these two listings is clearer? Respondents were supposed to choose one of the following answers: the first one, the second one, or both listings are equally clear and readable.
- Why? The answers to this question were given in a free form. Answers were generalized for finding common patterns.
- To which group does the respondent belong? The following groups were identified:
  - programmers (who were supposed to convey their professional opinion; however, a priori they were also likely to represent the widespread concept of the superiority of object-oriented languages and C-like syntax);
  - mathematicians and physicists (supposed to be more loyal to functional programming);
  - other professionals related to programming (such as system administrators), and finally;
  - professionals not related to programming.

The computer code listings are presented below. For Erlang, the listing does not contain full code but only meaningful functions equal to the Pregel listing.

Listing 1: Pregel

```
1  import pregel
2
3  class PageRankVertex(pregel.Vertex):
4
5          def update(self):
6          if self.superstep < 50:
7                  self.value=0.15*1/num_vertices+0.85*sum
                       (self.messages)
8                  self.messages= [(self.value/
                       num_adjacent_vertices)
9                            for each adjacent vertex]
10         else:
11                 self.active = False
12
13 num_vertices = 10
14 for each vertex:
15         vertex.value=1/num_vertices
16
17 run pregel
18 output values for all vertices
```

Listing 2: Erlang

```
receive
        {pagerank,N}-> pagerank:rank(N,LoC);

rank(0,LoC,Rank)-> Rank;

rank(N,LoC,Rank)-> Nrank=0.15/pagerank:number()+0,85*
    pagerank:sum(),
        lists:foreach(fun(V)-> V!Nrank/length(LoC) end,
            LoC),
        rank(N-1,LoC,Nrank).

number()-> master!number,
        receive
                Num-> Num
        end.

sum()-> sum(length(LoC),0).

sum(0,Sum)-> Sum;
sum(N,Sum)->
        receive
                {rank,Rank}-> sum(N-1,Sum+Rank)
        end.
```

The authors' expectations were as follows:

- Most of the programmers chose the Pregel option, while the "Both" option was second-most popular, and "Erlang" was chosen by the minority.
- While the majority of the "Related" group would still choose Pregel, the authors were expecting a growth of popularity in the "Both" option and possibly in the "Erlang" option.
- The majority of mathematicians would choose either "Both" or "Erlang."
- The behavior of the "Not related" group was difficult to anticipate. On the one hand, they would not be prejudiced against OOP nor FP; but on the other hand, they might lack the skills and/or training for making an informed choice.

4. Results

The results are presented in Figure 1 and Table 1. Unfortunately, the data about the mathematicians and physicists is non-representative, as only one respondent from this group took part in the survey. By accounting for all three variants of the answer,

the "Programmers" and "Not related" categories differed the most (Fisher's exact test p-value $= 0.0016$), while "Related" versus "Not related" differed to a lesser extent (p-value $= 0.006$).

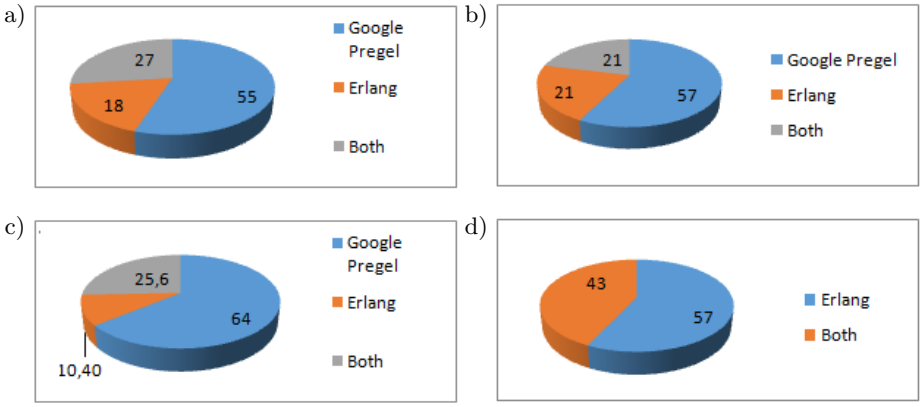

Figure 1. Results of survey: a) distribution of answers among all respondents; b) distribution of answers among programmers; c) distribution of answers among people related to programming; d) distribution of answers among people unrelated to programming

Table 1

Results of survey

Answer	Absolute number	Group's share [%]	All respondents' share [%]
Programmers			
Google Pregel	25	64.1	26.6
Erlang	4	10.3	4.3
Both	10	25.6	10.6
Total	39	100	41.5
Related to programming specialists			
Google Pregel	27	57.4	28.7
Erlang	10	21.3	10.6
Both	10	21.3	10.6
Total	47	100	50
Not related to programming specialists			
Erlang	3	42.9	3.2
Both	4	57.1	4.3
Total	7	100	7.4
Mathematicians/physicists			
Both	1	1	1.1
Total	1	100	1.1
Total	94	–	100

On the contrary, the difference between "Programmers" and "Related" was not detectable (p-value = 0.417). Thus, the "Related" category was in a somewhat intermediate position, likely because of being close to "Programmers" by training and background. The absence of such a background might be the main reason for the abundance of those who favored Erlang (including "Both") among "Not related": as many as 11 out of 14, or 78.5%. For comparison, this fraction among "Programmers" was 35.9%. As can be clearly seen from this survey, most of the respondents who chose Google Pregel did so because of syntax. Probably, most of them work or have experience with languages with C-like syntax.

The second survey was executed on a group of first-year master students who studied Erlang programming in their "Parallel Programming" course (in the "Systems without shared memory" section). During a crash course consisting of two lectures, two seminars, two laboratory assignments, and one homework assignment, the students had familiarized themselves with the basics of functional programming in general and Erlang in particular; these basics included pattern-matching (similar to Markov algorithms), variable single assignment, declarative programming, and the actor model.

The following problems were identified during the course:

1. Regarding pattern-matching, most (about 80%) of the students had problems with generalizing patterns even in simple tasks and often used if/case expressions; this made the code more difficult to read, debug, and refactor.
2. About 60% of the students pointed out that programming without shared memory and with immutable variables is difficult.
3. Regarding syntax, about 50% of the students mentioned that the Erlang syntax is very odd for them (Erlang has a slightly modified Prolog syntax). Most of them pointed out that a C-like syntax would be more convenient.

5. Discussion

Functional programming has been gaining popularity recently, which makes teaching it more important and in higher demand. Indeed, a number of well-known projects (e.g., WhatsApp) use functional programming languages such as Erlang. On the other hand, BMSTU wants to prepare specialists whose skills meet the requirements of the labor market, and Erlang definitely is not so popular as Java or Python.

The authors suggest the following steps for solving these problems:

1. It is necessary to enlighten first- and second-year students about programming paradigm diversity while urging them to follow the principle "each mastered paradigm makes me stronger" (from the universities of Oxford and Gothenburg). More information on functional programming should be added to general programming courses, emphasizing the FP features and advantages like recursion, lambda-calculations, etc.

2. During the study of a new programming language, it is important to distinguish between the language syntax, its paradigm, and its philosophy. The language philosophy is unique to each language and is in fact a "best practices" guide that helps programmers write efficient code. Even though some languages can share syntax and paradigms, their cores might be optimized for different use cases, making their philosophies differ (e.g., X10 and Java). Similar to many other university teachers, the authors encountered a problem of students being "addicted" to a certain syntax. They consider it necessary to "learn syntax" upon which they will be able to code efficiently in any C-like language. Such an attitude is counterproductive and makes students write and spread inefficient code. As an example, a student who was used to Java and was learning Erlang continued to practice the Java style instead of trying to rebuild his style for writing efficient Erlang code while writing programs in Erlang, thereby ignoring most of the FP aspects and Erlang's best practices.
3. Hence, it appears necessary to add a "Functional programming" course in any curriculum. Optional or mandatory courses in functional programming are found in many Computer Science/Theoretical Informatics programs (for example, at the universities of Oxford, Gothenburg, and La-Coruna as well as Munich Technical University). Haskell, Erlang, and LISP are examples of functional languages taught in the aforementioned universities.

References

[1] Brin S., Page L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine, *Computer Networks and ISDN Systems*, vol. 30, pp. 107–117. 1998.

[2] Cesarini F., Thompson S.: *Erlang Programming*. DMK, 2012.

[3] Compeau P., Pevzner P., Tesler G.: How to apply de Bruijn graphs to genome assembly, *Nature Biotechnology*, vol. 18, p. 987, 2011.

[4] Davies A.: *Async in C# 5.0*. O'Reilly Media, 2012.

[5] Gergel V.: *Theory and Practice of Parallel Computing*. BINOM, 2007.

[6] Guney E., Oliva B.: Exploiting Protein-Protein Interaction Networks for Genome-Wide Disease-Gene Prioritization. In: *PLoS ONE*, 2012.

[7] Nielsen M.: Pregel. `http://www.michaelnielsen.org/ddi/pregel/`, 2011. Accessed: 01.05.2017.

[8] Petrov I.: Implementing graph representation model for parallel and distributed systems using Erlang, *Computer Science*, vol. 17(1), pp. 99–120, 2016. `https://journals.agh.edu.pl/csci/article/view/1397`.

[9] Top mentioned books on stackoverflow.com. `http://www.dev-books.com/`, 2017. Accessed: 01.05.2017.

[10] What does object-oriented programming do better than functional programming, and why is it the most popular paradigm when everybody seems to say functional programming is superior? `https://www.quora.com/What-does-object-oriented-programming-do-better-than-functional-programming-and-why-is-it-the-most-popular-paradigm-when-everybody-seems-to-say-functional-programming-is-superior`, 2016. Accessed: 01.05.2017.

[11] What makes OOP "good"? `http://programmers.stackexchange.com/questions/198675/what-makes-oop-good`, 2013. Accessed: 01.05.2017.

[12] Which are the most used programming paradigms? `https://www.quora.com/Which-are-the-most-used-programming-paradigms-Which-are-good-coding-examples`, 2016. Accessed: 01.05.2017.

[13] Which programming paradigm should you start with? `https://www.quora.com/Which-programming-paradigm-should-you-start-with`, 2015. Accessed: 01.05.2017.

[14] Why use de Bruijn Graphs for Genome Assembly? `http://www.homolog.us/Tutorials/index.php?p=1.4&s=1`. Accessed: 01.05.2017.

[15] Yakobovskiy M.: *Introduction to the Parallel Methods of Problem Solving*. Publishing house of Moscow State University, 2013.

Affiliations

Iurii Petrov
Department of Microbiology, Tumor, and Cell Biology (MTC), Karolinska Institutet, Stockholm, Sweden; Science for Life Laboratory, Box 1031, 171 21, Solna, Sweden, iurii.petrov@scilifelab.se, ORCID ID: https://orcid.org/0000-0002-9227-7909

Andrey Alexeyenko
Department of Microbiology, Tumor, and Cell Biology (MTC), Karolinska Institutet, Stockholm, Sweden; Science for Life Laboratory, Box 1031, 171 21, Solna, Sweden, andrej.alekseenko@scilifelab.se, ORCID ID: https://orcid.org/0000-0001-8812-6481

Galina Ivanova
Department of Computer Systems and Networks, Bauman Moscow State Technical University, ul. Baumanskaya 2-ya, 5, Moscow gsivanova@bmstu.ru, ORCID ID: https://orcid.org/0000-0002-8926-2028

Received: 16.04.2018
Revised: 28.06.2018
Accepted: 29.06.2018