MIPRO 2016, May 30 - June 3, 2016, Opatija, Croatia

Comparative analysis of functional and object-oriented programming

Dino Alic*, Samir Omanovic* and Vaidas Giedrimas**
* University of Sarajevo/Faculty of Electrical Engineering, Sarajevo, Bosnia and Herzegovina
** Siauliai University/Institute of Informatics, Siauliai, Lithuania
alic.dino@gmail.com, samir.omanovic@etf.unsa.ba, vaigie@mi.su.lt

Abstract - The choice of the first programming language and the corresponding programming paradigm is an important part of the software development process. Knowing the advantages and constraints of individual programming paradigms is important as it can be crucial for successful software implementation. In this paper we conduct an empirical comparison of functional and object-oriented programming languages using analog examples in C#, F#, Haskell, and Java. Three algorithms were implemented: algorithm for solving N queens problem, algorithm for generating n-th left-truncatable prime and merge sort algorithm in C#, F#, Haskell and Java programming languages. An overview of programming languages efficiency is given by measuring two basic parameters: number of lines of code and program execution speed. Also, system resource usage is monitored during execution. Limited experiments showed that the programming language Java is faster than the other three languages whose performances were measured. Java was surprisingly fast on these problems that are more suitable for functional programming languages. Haskell was less memory intensive (up to two times less than Java) with similar execution times, while .NET languages were slower up to four times in comparison to Java. Object-oriented languages C# and Java had significantly more lines of code for all three algorithms when compared to functional programming language Haskell and the hybrid one F#.

I. INTRODUCTION

As with the many things in programming languages history, there is an always present debate about which programming paradigm is better and when a specific paradigm should be used [1] [2]. Even the classification of paradigms itself is often cause for a debate. However, programming paradigms are often classified in six fundamental paradigms: imperative, declarative, object-oriented, functional, symbolic, and logic programming paradigms. Declarative and symbolic paradigms are sometimes left out leaving only four fundamental paradigms [3].

It is important to note that most of the mainstream programming languages in wide use today support multiple paradigms (multi-paradigm programming languages [4]). For example, C++ has native support for seven programming paradigms with additional libraries extending the support for six additional programming paradigms.

In this paper we focus on two fundamental programming paradigms: object-oriented and functional paradigm. Object-oriented paradigm is in wide spread use in industry [5] today with functional paradigm used mainly by the scientific community.

However, there is a notable trend of object-oriented programming languages introducing some basic principles of functional programming in new versions. In 2008 lambda expressions were introduced in .NET framework version 3.5 [6], primarily to support LINQ (Language Integrated Query). Lambda expressions were also introduced in Java 8 [7], allowing the developers to express instances of single-method classes more compactly. Introduction of functional programming principles in object-oriented languages shows that functional and object-oriented programming languages are not mutually exclusive and that combining the two principles can be effective [1].

II. FUNCTIONAL VS OBJECT-ORIENTED PROGRAMMING

Despite the always present debate over choosing the programming paradigm, there is reasoning behind choosing the programming paradigm which will be used to solve the problem. Object-oriented languages are good when there is a fixed set of operations on things, and as the code evolves, primarily new things are added [8], like adding new classes which implement existing methods (existing classes are left alone). Functional languages are good when there is a fixed set of things, and as the code evolves, primarily new operations are added upon existing things [8], like adding new functions that work with existing data types (existing functions are left alone). This is known as an expression problem [9], a term coined by Philip Wadler.

It is important to note that programmer can do functional programming in object-oriented languages. To accomplish this all program object (data) must have constant state and all methods and operations must depend only on its arguments and the context available at definition time.

A. Object-Oriented Programming Paradigm

Object-oriented paradigm is based on the concept of *objects* – data structures containing data known as attributes and corresponding code known as *methods*. The basic idea of object-oriented languages is combining *data* and the *functions and methods that operate on that data* into a single component.

Object-oriented programming languages implement three basic principles of object-oriented approach [5]:

- *encapsulation* – limiting the access to the object from other pieces of code;
- *inheritance* – using the existing object or class as a base for new ones;
- *polymorphism* – provision of a single interface to entities of different types (object can take on several forms while the program is running).

Object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the overall organization of the program.

B. Functional Programming Paradigm

Functional programming is at its core based on the mathematical apparatus. The base for all functional programming languages is *lambda calculus*. Lambda calculus has three basic rules [10]:

$$\begin{aligned} &(\alpha\text{ conversion}) && \lambda x.A \underset{\alpha}{\rightarrow} \lambda y.[y/x]A \\ &(\beta\text{ reduction}) && (\lambda x.A)B \underset{\beta}{\rightarrow} [B/x]A \\ &(\eta\text{ conversion}) && \lambda x.Ax \underset{\eta}{\rightarrow} A \quad \text{if x not free in A} \end{aligned} \qquad (1)$$

It can be argued that functional programming languages only implement the lambda calculus while adding constants and new data types. Lambda calculus treats functions as expressions which are gradually transformed until final solution is found – function actually defines the algorithm. Lambda calculus is in wide use in mathematics, philosophy, linguistics, and computer science.

In contrast to imperative programming, order of execution in functional programming languages is not defined by the programmer. Programmer only tells the program what needs to be done without explicitly defining how it should be done. Compiler/interpreter will determine order of execution. Functions in functional programming languages are evaluated only when the result they return is needed. This approach is known as *lazy evaluation* [11].

C. Differences Between Functional and Object-Oriented Programming

One of the fundamental differences between the two paradigms is the interrelationship between data and operations. Basic rule of object-oriented programming is that data and operations affecting them are tightly linked. Objects owns their data and control all actions (functions and methods) modifying them while hiding them from other objects using the interface. This implies that the abstraction model in object-oriented programming is the *data* itself. In functional programming data is tightly related with functions. It is possible do implement different operations over the same data structure. However, basic abstraction is a *function*, not the data structure. Functions hide their own implementation.

Recursion is used in both programming paradigms, but object-oriented programmers tend to avoid it. While recursion reduces the code length it can have a negative effect on code readability. In addition, iterative approach is more performance wise in comparison to recursive approach. In object-oriented programming there is always an iterative equivalent to any recursive algorithm. On the other hand, recursion is essential part of the functional programming since it is based on the lambda calculus where “everything is a function”. The essential build blocks of object-oriented programs are *statements* while functional programs are composed mostly from *expressions*.

Another fundamental difference between the two programming paradigms is the *program state*. The main limitation in functional paradigm and purely functional programming languages is the absence of program state. Purely functional programming languages are strictly *stateless* and work with *immutable* data. Functional program can be viewed as independent on the machine meaning that there is no state as we do not have the notion of memory. Conversely, object-oriented programs work with *mutable* data, storing the data in variables and defining statements which can change program state. As a result of stateless approach in functional programming, program execution has no side effects. This implies that the order of statement execution is not important. The program will, for identical inputs, return identical outputs regardless of the order of statement execution. This is known as *referential transparency*, with *referential opacity* being the opposite. Referential opacity is a side effect of working with mutable data. Eliminating side effects can make it simpler to help understand as well as predict behavior of the program [12].

III. Programming Languages in Comparative Analysis

Four programming languages were chosen for this comparative analysis: Haskell, C#, F# and Java. Haskell is a purely functional programming language [13], C# and Java are object-oriented languages with built in support for additional programming paradigms [14] [15], while F# is a hybrid of functional and object-oriented programming [16]. Here are presented some of the most important characteristics of four programming languages used in the analysis.

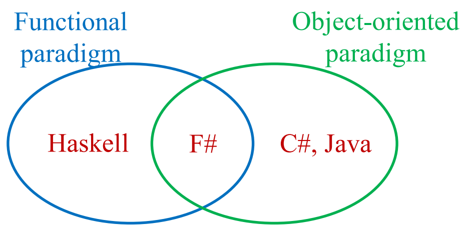

Figure 1. Principle programming paradigms in chosen programming languages

All chosen programming languages are *statically typed* which means that all data types are known at the compile time which leads to *type-safe code*. However, while Haskell and F# are statically type-checked, compiler automatically infers a precise type for all values. In Java, manners of evading the static type system are controlled by the Java virtual machine's type system. C# is similar to Java in this respect but it allows disabling dynamic type checking by explicitly placing code segments in *an unsafe context*. F# and C# are based on .NET platform. All .NET languages are compiled to Common Intermediate Language (CIL) meaning that, in theory, programs implementing the same functionality written in F# and C# will be translated into the same CIL code.

TABLE I. OVERVIEW OF PROGRAMMING PARADIGMS SUPPORT IN PROGRAMMING LANGUAGES IN ANALYSIS

Programming language	**Supported paradigms**
C#	structural, imperative, object-oriented, event driven, functional, generic, reflective, concurrent
F#	functional, imperative, object-oriented, metaprogramming, concurrent
Java	structural, imperative, object-oriented, functional, generic, reflective, concurrent
Haskell	functional, lazy/non-strict, modular

All four programming languages are standardized, general purpose programming languages. Haskell and Java are cross – platform languages while C# and F# are designed for Windows platform only, sharing the CLI platform.

A. *Haskell*

Haskell is a purely functional programming language with no attributes of imperative programming languages. It is a *higher order* programming language – functions can receive functions as arguments and return functions as a result. By using this feature it is possible to define new function as a composition of two existing functions [17][18].

Conditional expressions can be nested and they must have an *else* branch. There are no loops in Haskell with recursion as a programming equivalent. Functions can be defined using lambda expressions which create a frame for each argument and function body which will calculate result based on the arguments provided but they do not name the function. In other words, lambda expressions are nameless functions in Haskell [19].

Algorithms in Haskell consist of *base cases* and recursive cases. Determining the base cases is crucial for generating the correct output and avoiding the infinite loops [17]. For example, to calculate factorial of an integer provided as an input, the base case is 0 when the result is generated immediately and recursive case is used for all other inputs.

B. *F#*

F# is a programming language supporting both object-oriented and functional programming paradigms [16]. Object-oriented approach allows the programmer to write code using classes and objects. On the other hand with functional approach programmer can tell the program what needs to be done without explicitly defining how it should be done. F# uses *greedy evaluation* techniques, a distinctive feature of imperative programming languages. When using greedy evaluation, an expression is evaluated as soon as it is linked to a variable.

In object-oriented programming all variables are mutable unless explicitly declared as static. In F# all variables are static unless explicitly declared as mutable. *Null* values are not supported and *NullReference* exceptions are not possible [20]. Another important difference between most object-oriented languages and F# is that F# programs do not force the programmer to declare data types. Instead, type is determined based on the value provided (C# has the keyword *var* providing this feature).

C. *Java*

Java is a general purpose programming language. Java applications are compiled to *bytecode* which can then run on Java virtual machine. Important advantage of using Java is that the same code can be run on all platforms that support Java without the need for recompiling the code. The existence of the intermediate had significant impact on the performance until the just in time compilation was implemented, partially resolving the performance hit. Just in time compilation is a form of dynamic compilation which allows the code compilation during run-time opposed to compile and run approach. However, this can cause a small delay when starting the application.

Java 8 added lambda expressions – a key concept in functional programming. Lambda expression in Java is a block of code that can be passed around and executed later multiple times (*deferred execution*). Java lambda expressions are especially useful in GUI applications [21]. According to TIOBE index Java is the most popular programming language in use today [22]. That is why comparison with other languages is important to find his strengths.

D. *C#*

C# is a general purpose object-oriented language. C# supports more data types than Java and implements most of the features that Java possesses, with *checked expressions* being the notable exception. C# 3.0 added support for lambda expressions and expression trees as an incremental step for steadily expanding support for functional programming in C# [23]. A lambda expression in C# is an anonymous function that can be used to create delegates or expression tree types. By using the lambda expressions programmer can write local functions and then pass them as an argument or return them as a result. Expression lambda is a lambda expression with an expression on the right side of the operator and is used to create expression trees. Expression trees are a code in a tree-like data structure, where each node is an expression. Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework.

IV. OWERVIEW OF EXECUTION ENVIRONMENT AND DEFINITION OF PROBLEMS FOR ANALYSIS

For the purpose of comparative analysis three different algorithms where implemented in all four chosen languages. The implemented algorithms are:

- **Algorithm for solving the N queens problem.** Chessboard queens can attack horizontally, vertically, and diagonally. The N-queens problem asks: "*How can N queens be placed on an NxN chessboard so that no two of them attack each other?*". N-queens problem complexity is roughly exponential. Naïve approach to solving the problem consists of generating all possible configurations of queens on board and finding those that satisfy the requirement. Backtracking algorithm significantly reduces the number of variations checked by eliminating variations without explicitly checking them as they will not be part of the solution. Backtracking approach in solving the problem was used in order to show limitations of functional programming as this approach requires mutable data to work with. However, purely functional programming languages only work with immutable data. Algorithm was used to solve the N queens problem for N = 12, 13 and 14.
- **Algorithm for generating n-th left truncatable prime.** Left-truncatable prime is prime numbers which, in a given base, contains no zeroes, and if the leading left digit is successively removed, then all resulting numbers are prime numbers. There are only 4260 known n-th left trucatable primes in base 10 with the largest currently being the 24-digit number 357 686 312 646 216 567 629 13 [24]. Algorithm performance was measured when generating 1300th, 1700th and 1800th left trucatable prime.
- **Merge sort algorithm.** Merge sort is a divide and conquer algorithm for sorting. It works by dividing the unsorted lists into sub-lists until each sub-list contains one element only. Then the merge phase commences. Sub-lists are repeatedly merged producing new sub-lists until there is only one sub-list remaining. This is the final sorted list. Text files containing 10 000, 30 000, 50 000 and 100 000 random numbers were generated. Program flow consists of importing the text file and then sorting the numbers found in it by using the merge sort algorithm.

Object-oriented approach was used for implementing algorithms in C# and Java, functional approach for implementing algorithms in Haskell and a combination of functional and object-oriented approaches when implementing algorithms in F#. Algorithms were not implemented to support any kind of parallelism in code. It is important to note that Glasgow Haskell Compiler (GHC) was used. GHC is itself written in Haskell based on a technique known as bootstrapping. The runtime system for Haskell (used to run programs) is written in C.

A. N Queens Alghorithm with Backtracking

Haskell program had the least number of lines of code with C# having the most. Java program has only two lines less than the C# version. C# had twice as much lines of code comparing to F# but F# had more almost three times as much lines of code compared to Haskell.

TABLE II. OVERVIEW OF THE EXECUTION ENVIRONMENT

Processor	Intel Core i5-3210M 2.5 GHz
RAM	6 GB DDR3-1600
GPU	Optimus - Intel HD Graphics 4000, 2 GB
GHC version	7.10.3
Java version	8.66
.NET version	4.6.1
Operating system	Windows 7 Ultimate Service pack 1, 64bit

When it comes to program run time Java program was the notably faster than all other programs. C# was four times slower in average when compared to Java. Haskell and F# programs were significantly slower in generating results in comparison to object-oriented Java and C#. This can be attributed to the inability to take full advantage of backtracking approach.

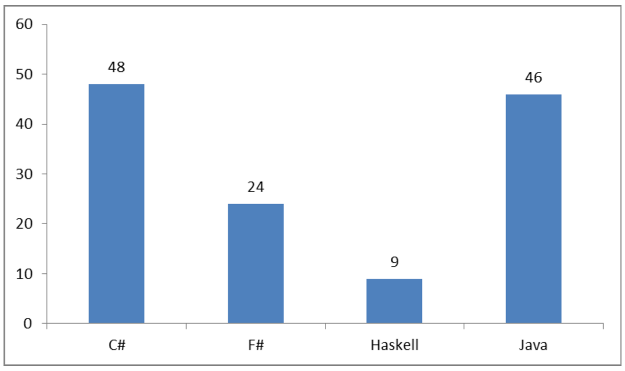

Figure 2. Number of lines of code in algorithm for solving the N queens problem

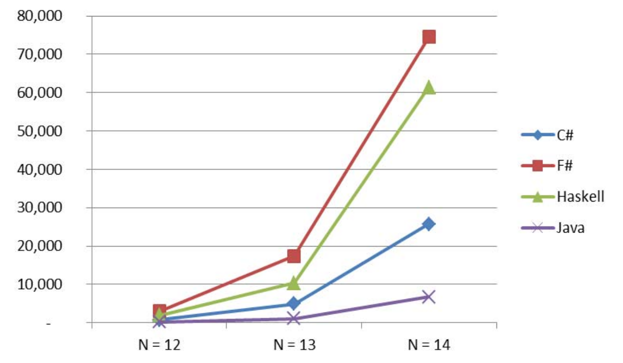

Figure 3. Average run time of N queens algorithm [seconds]

B. Generating n-th left trucatable prime

As with the N queens algorithm Haskell program had the least amount of number of lines of code. However, this time F# had only nine lines more than Haskell. Java and C# were almost tied again. Java and C# programs had almost three times as much lines of code when compared to Haskell and more than twice when compared to F#.

In comparison to N queens program, Haskell and C# changed places with Haskell program coming in second place. Once again, Java program was the fastest, C# and F# were behind the Java and Haskell with a notable difference.

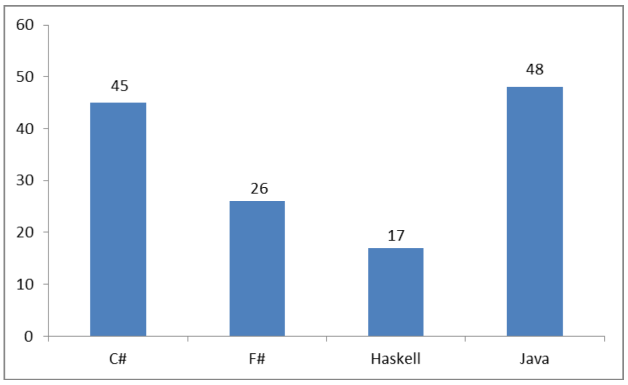

Figure 4. Number of lines of code in algorithm for generating n-th left trucatable prime

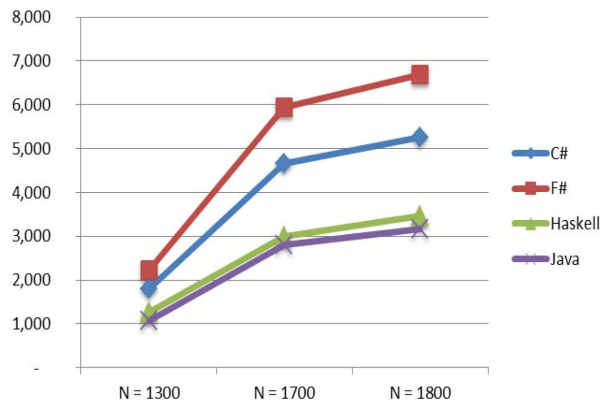

Figure 5. Average run time of algorithm for generating n-th truncatable prime [seconds]

C. *Merge sort*

As it was the case with the previous two algorithms, Haskell program had the least amount of lines of code. F# had somewhat more lines of code in comparison to Haskell but still notably less when compared to C# and Java. In merge sort algorithm a significant difference in number of code lines between Java and C# exists, which was not the case in the previously discussed algorithms.

Java program run time was fastest once again. However, Haskell program was second with an almost negligible difference which was reduced as the N increased. For N = 100 000, run time for Haskell and Java programs were almost identical. C# had the third longest execution time, notably longer than those of Java and Haskell. F# had slightly longer execution time than C#, coming in last.

Merge sort implementation in all programming languages was based on recursive approach, where we would expect that functional programming languages (purely functional in particular) show better performance. However, Java program was slightly faster than Haskel even in this algorithm.

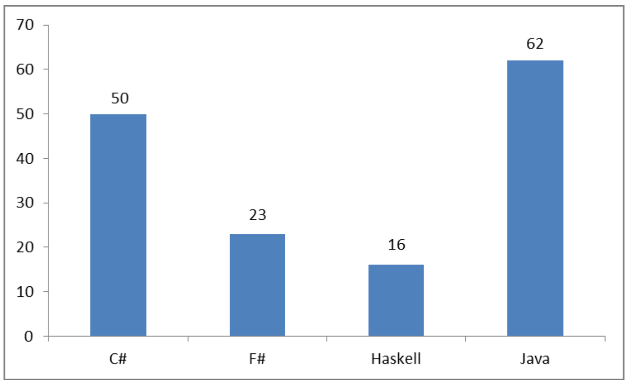

Figure 6. Number of lines of code in merge sort algorithm

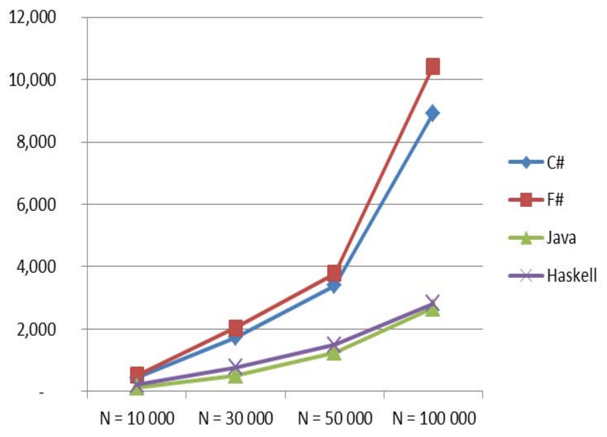

Figure 7. Average run time of merge sort algorithm [seconds]

V. DISCUSSION

Java programs run times were fastest in all three algorithms analyzed. Haskell programs had fewer lines of code in all three analyzed algorithms. Experiments showed that .NET programming languages were significantly slower solving the presented problems when compared to Haskell and Java.

Object-oriented languages were significantly faster when using the backtracking algorithm, being able to utilize all the advantages of backtracking approach.

Processor usage was almost the same for all programs with a negligible difference of +-1%. However, purely functional Haskell programs had a considerably lower memory load, being in average two times less memory intensive than all other programs. This can be explained with stateless approach being used in functional programming (working with immutable data only) which consumes much less memory in comparison to mutable data.

As a rule, functional programming languages tend to have less lines of code. This does not imply that they have better code readability as it will depend on the implementation and preferences of the programmer. Object-oriented languages, C# and Java, had only small differences in number of lines of code between the two. Haskell had, in average, three times less program code in comparison to object-oriented languages. Hybrid programming language F#, combining both functional and object - oriented paradigms proved to be significantly slower than all others. This goes to show that combining the two programming paradigms might not result in a better solution (in this case we have quite an opposite situation). When combining these two paradigms a programmer should have in mind all the advantages and disadvantages of each programming paradigm and combine them only in an extent necessary to solve the actual problem.

It would be interesting to compare the two paradigms when using the multithreading and concurrent approach. Mutable variables can result in many issues in object-oriented languages. Working with purely functional languages and using the immutable data eliminates most of these issues. This way, programmer does not have to worry about side effects of multithreading and concurrent approach.

VI. Conlusion

Functional paradigm is preferable choice for implementing algorithms described above. That is partially confirmed by the results obtained in the experiments, having in mind number of lines of code, memory usage and speed. On the other side, Java is surprisingly fast comparing to Haskell. C# implementation showed mainly expected results while F# showed disappointing results.

Experiments confirmed once more that it is hard to make general conclusion which programming paradigm is better, but it is obvious that they should be combined to increase execution efficiency. For the software engineering industry combination of paradigms is the best choice. Besides that, selection of the programming language is also important. Results support conclusion that Java is the best choice among languages used in these experiments, having in mind execution efficiency. The results also explain Java's popularity.

Comparation of programming paradigms and programming languages is always interesting topic if we want to estimate influence of technology changes to the observed concepts. In that sense our future work will be directed toward comparing Java and Haskell, as the best choice from these experiments, with some languages that are not pure functional programming languages but support functional programming (Swift programming language, for example) and having in mind some execution platform limitations (mobile phone, for example).

References

[1] B. Meyer, "SOFTWARE ARCHITECTURE: OBJECT-ORIENTED VS FUNCTIONAL," chapter 13 of *Beautiful Architecture, edited by D. Spinellis and G. Cousios*, O'Reilly, 2009, pp. 315-348.

[2] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, Piscataway, NJ, USA, 2015, pp. 778–788.

[3] K. Normark, "Functional Programming in Scheme." [Online]. Available: http://people.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html. [Accessed: 02-Feb-2016].

[4] J. O. Coplien, "Multi-Paradigm Design," Ph.D. dissertation, Informatics Dept., Vrije Univ., Brussel, Belgium, 2000.

[5] D. Donko and S. Omanovic, Objektno orijentirana analiza i dizajn primjenom UML notacije, Elektrotehnicki fakultet, Sarajevo, 2009.

[6] Microsoft, ".NET Framework Versions and Dependencies." [Online]. Available: https://msdn.microsoft.com/en-us/library/bb822049(v=vs.110).aspx. [Accessed: 02-Feb-2016].

[7] Oracle, "Information about Java 8." [Online]. Available: https://java.com/en/download/faq/java8.xml. [Accessed: 02-Feb-2016].

[8] A. Scott, "Functional Programming vs. Object-Oriented Programming." [Online]. Available: http://functionspace.com/topic/1467/Functional-Programming-vs--Object-Oriented-Programming. [Accessed: 02-Feb-2016].

[9] D. Wampler and A. Payne, Programming Scala: Scalability = Functional Programming + Objects. O'Reilly Media, 2014.

[10] D. A. Turner, "Some History of Functional Programming Languages" - *an invited lecture given at TFP12, St Andrews University*, 12 June 2012.

[11] T. Kuhne, A Functional Pattern System for Object-Oriented Design, Verlag Dr. Kovac, Hamburg, Germany, 1999.

[12] Basili, V.R.; Selby, R.W., "Comparing the Effectiveness of Software Testing Strategies," in *Software Engineering, IEEE Transactions on* , vol.SE-13, no.12, pp.1278-1296, Dec. 1987

[13] Haskell, "Haskell Language." [Online]. Available: https://www.haskell.org/. [Accessed: 03-Feb-2016].

[14] Microsoft, "Visual C# Resources." [Online]. Available: https://msdn.microsoft.com/en-us/vstudio/hh341490.aspx. [Accessed: 03-Feb-2016].

[15] Oracle, "Java 8 Central." [Online]. Available: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html. [Accessed: 03-Feb-2016].

[16] Microsoft, "Visual F#." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd233154.aspx. [Accessed: 03-Feb-2016].

[17] G. Hutton, Programming in Haskell, Cambridge University Press, 2007.

[18] M. Leucker, T. Noll, P. Stevens, and M. Weber, "M.: Functional programming languages for verification tools: Experiences with ML and Haskell," in In: Proceedings of the Scottish Functional Programming Workshop, 2001.

[19] B. O'Sullivan, J. Goerzen, D.B. Stewart, Real World Haskell, O'Reilly Media, 2008.

[20] T. Liu, F# for C# Developers, Microsoft Press, 2013.

[21] Oracle, "Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects)." [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Accessed: 03-Feb-2016].

[22] TIOBE, "TIOBE Software: Tiobe Index." [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html [Accessed: 03-Feb-2016].

[23] Microsoft, "Lambda Expressions (C# Programming Guide)." [Online]. Available: https://msdn.microsoft.com/en-us/library/bb397687.aspx. [Accessed: 03-Feb-2016].

[24] O. Angell and H. J. Godwin, "On Truncatable Primes," in *Mathematics of computation*, AMS, vol.31, No 177, pp. 265-267, Jan 1977