Functional vs. Object Oriented Programming: Productivity comparison

Lorenzo Corneo and Nikolaos-Ektoras Anestos
{corneo, anestos}@kth.se

October 31, 2015

Abstract

Human resources and time consumption are very important factors in software development. Both developers and managers try to maximize their productivity, in order to increase as much as possible the profits, or gain market advantage. We assume that choosing an appropriate programming paradigm is a critical factor that leads to a better allocation of human resources during software development.

The main object of investigation in this project are the lines of code needed to develop two different implementations of the same program and the level of reusability of existing code.

Functional Programming languages, have some fundamental benefits that increase programmer productivity over Object Oriented Programming languages. The purpose of this project is to demonstrate some of the benefits of functional programming over OOP, in order to persuade developers to adopt functional programming and use it in more and more cases.

From the evaluation of the gathered data it is possible to see that, averagely, the number of Lines Of Code in a functional programming language is smaller than in an Object Oriented Programming language. As a consequence, Functional Programming allows an increased productivity (fewer lines of code correspond to a time saving during development).

Keywords. Functional Programming, Object Oriented Programming, Comparison, Productivity, Code Reuse.

1 Introduction

The paper presents the results of a quantitative evaluation of code implemented in both Functional Programming and Object Oriented Programming. The main parameters that will be analyzed are the Line Of Code and the Code Reuse.

The main test-bed for the productivity analysis is based on the LOC of different implementations (both FP and OOP) of very well-known algorithms. The language pool is composed by Erlang, Haskell, Scala, F#, for the FP languages and by Java, C++, C#, Python for the OOP languages.

The CR is evaluated counting the difference in terms of LOC between adjacent releases of a program. Object of the investigation, as a case study, are the asynchronous web server Play framework[7], apache Flink[2] and Reactive X library[8] that are implemented both in Java and Scala.

In section 2, a summary of the studied literature will be presented while in section 3 the methods and the hypothesis will be discussed. The results of the investigation and the analysis are presented in section 5, in section 6 a discussion about the problems and finally we recognize the future work that needs to be done in section 7.

The aim of the research is to show the benefits of FP over OOP with the hope to persuade software developer to adopt FP as their main programming paradigm, for a better productivity and code reuse.

2 Literature study

OOP is the de facto standard programming paradigm for companies and widely adopted all around the world. This finds confirmation in the TIOBE Index[9]. In fact, in the first positions only OOP and Web programming languages are found, while the first FP language is Scala at the 24th position with a very low rating of 0.781%.

Nevertheless, functional programming gurus always try to persuade imperative programmers to switch to this paradigm promising incredible advantages. Amongst these benefits we found the lack of assignment statements (variables never change) and side effects, which is the major source of bugs in software development[6].

Furthermore, a thorough evaluation about the two programming paradigms has been performed by Harrison[4]. He defined parameters to be able to measure the quality of the software, including LOC and CR. Then, from the domain application of the image processing, he selected a developer with similar skills both in C++ and SML (functional language of the ML family) who developed a set of 12 algorithms in both languages. From the analysis of the results, it turned out that there are not many differences between the two paradigms.

Nonetheless, FP finds very important benefits in the application domain of the prototypes where a case study project is implemented in Haskell in 85 LOC against the 1105 of C++[5]. This is due to three main factors: (a) the syntax is simpler, (b) the use of higher order functions and (c) the standard list-manipulation primitives (i.e. map, fold, zip, ...).

3 Hypothesis

Human resources and time consumption are very important factors in software development, especially for companies that want to optimize profits and gain advantages over the market.

We strongly believe that, using functional programming languages, developers will be more productive and they will also produce higher quality code that will allow them to reuse much of it when updating their programs.

4 Methods

A quantitative method is used for this project because the authors have gathered data from many sources and they report in this paper. Afterward, the analysis of the gathered data is performed so that the initial hypothesis will be either accepted or rejected.

The approach used is deductive. The authors started this research stating their hypothesis on the basis of existing theories, for instance, regarding the benefits of FP and the advantages that developers can gain in software production.

Afterwards, when the data has been acquired, the authors perform observation and analysis of the gathered results. From that, it will be possible to accept or reject the initial hypothesis.

Static code analysis on the set of algorithms described below has been performed using the command line tool called Count Lines of Code (cloc) [1], which counts lines of code, blank lines and comments in multiple languages. There are several options on how to present the data using this tool, i.e. one can obtain an analysis of a language used inside a project (several files), or extract results for each file individuals etc.

5 Results and Analysis

The experimentation consists in static code analysis on a set of eight algorithms[3], namely:

- FASTA, generate and write random DNA sequences.
- k-nucleotide, repeatedly update hash tables and k-nucleotide strings.
- Mandelbrot, generate a Mandelbrot set and write a portable bitmap.
- n-body, perform an N-body simulation of the Jovian planets.
- pi-digits, calculate the digits of Pi with arbitrary-precision arithmetic.
- regex-dna, match DNA 8-mers and substitute nucleotides for IUB code.
- reverse-complement, read DNA sequences and write their reverse-complement.
- spectral-norm, calculate an eigenvalue using the power method.

This set of algorithms has been implemented in Haskell, Erlang, Scala, Clojure, F# and Python for the functional programming paradigm, in C++, C#, Go and Java for the object-oriented paradigm.

5.1 Lines of Code

In the experimentation we analyzed the LOC for each language per each algorithm and the results are shown in Table 1. In Fig. 1 is possible to see which language is the most efficient in terms of lines of code per each single algorithm.

An alternative way to visualize these results is to think in terms of language and display which algorithms is the best to implement in and to be more productive. Fig 2 shows this alternative representation.

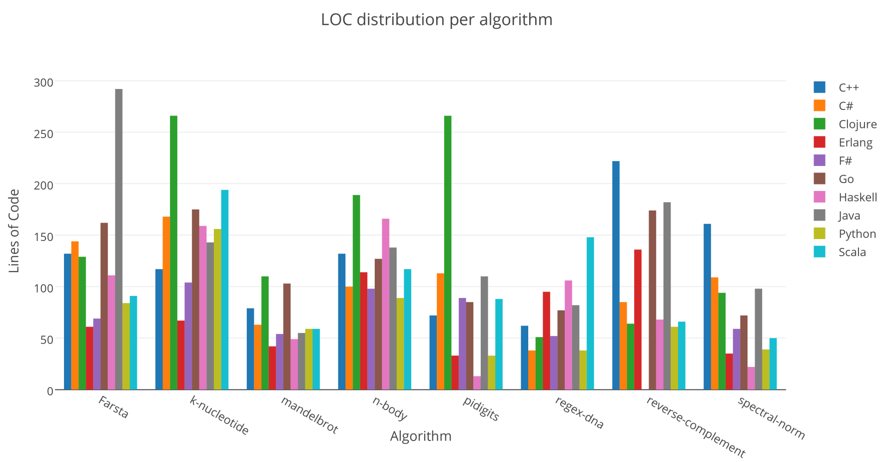

Figure 1: The graph shows the distribution of the LOC of different languages implementation for different algorithms.

Table 1: LOC for different implementations.

	Haskell	Erlang	Scala	Clojure	F#	Python	C++	C#	Go	Java
FASTA	111	61	91	129	69	84	132	144	162	292
k-nucleotide	159	67	194	266	104	156	117	168	175	143
Mandelbrot	49	42	59	110	54	59	79	63	103	55
n-body	166	114	117	189	98	89	132	100	127	138
Pi-digit	13	33	88	266	89	33	72	113	85	110
regex-dna	106	95	148	51	52	38	62	38	77	148
reverse-compl.	68	136	66	64	-	61	222	85	174	182
spectral-norm	22	35	50	94	59	39	161	109	72	98

Table 1 contains many measurements and it is not immediate to understand its meaning. Table 2 summarizes the aforementioned results in terms of FP and OOP, where the average of the LOC are the entries of the table for the two paradigms, and Fig. 3 is the graphical equivalent representation.

Table 2: LOC for the two paradigms.

	FP	OOP
FASTA	91	183
k-nucleotide	158	151
Mandelbrot	63	75
n-body	129	125
Pi-digit	87	95
regex-dna	82	65
reverse-compl.	66	166
spectral-norm	50	110

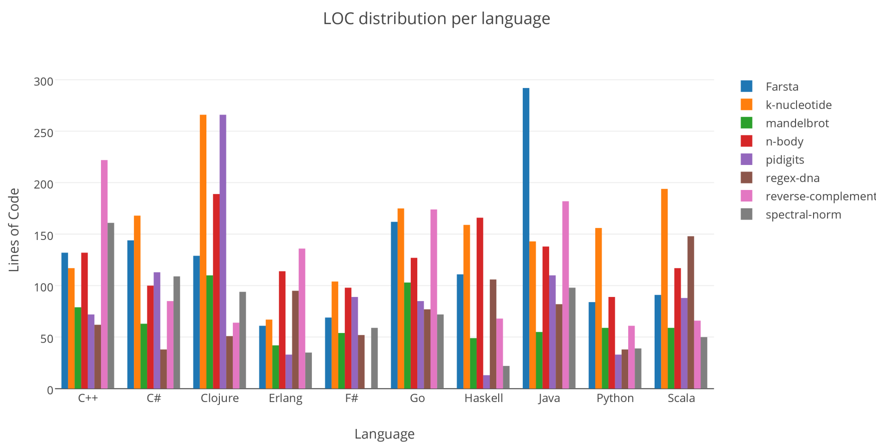

Figure 2: The graph shows the LOC used to solve each algorithm for the same language.

What we evince from the gathered data is that, normally, there is no big difference between the LOC for FP and OOP. Nevertheless, in algorithms like FASTA, revers-complement and spectral-norm, the LOC in FP implementations are less than the half of OOP implementation. This find explanation in [6] where the benefits of FP are discussed. Some of the most significant are: the syntax (which is simpler), higher order functions and the standard list-manipulation primitives (which offer better abstraction and code reuse).

5.2 Code Reuse

It has not been possible to succeed in the experimentation concerning the CR as we could not find a suitable case study[7, 2, 8]. For instance, many implementations are based on the JVM so that the developers reuse their Java

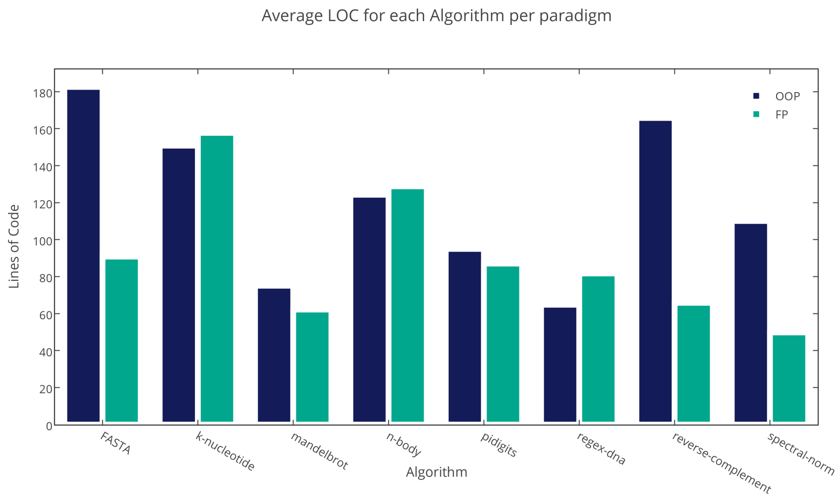

Figure 3: The graph represents the distribution of both FP and OOP paradigm in different algorithms.

implementation building an API layer for functional languages, like Scala and Clojure.

Also, in most cases, the features that the developers provide in each version are not exactly the same for the two paradigms. For example version 2.0 has features X, Y, Z in OOP but only feature X is implemented for the functional API. This makes it impossible to compare the two paradigms over this parameters, but we found that in the case where a feature is present for both paradigms, the FP API needs always less lines of code to call it, over the OOP counterpart.

6 Discussion

Static code analysis on the lines of code itself is not enough to have a global vision about the two paradigms. In fact, there are many additional factors that affect productivity such as the documentation, the comments, the development time, the application domain, the experience of the developer, the time to implement modifications[4, 5].

Nevertheless, we strongly believe that the LOC are directly involved in the productivity and our hypothesis finds confirmation in [5]. In fact, in Fig. 4 we notice that in (1), (2), (3), (10) the development time is proportional to the LOC and this fact support our hypothesis.

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Figure 4: The result of Hudak and Jones experimentation in[5].

7 Future work

We have just demonstrated with a quantitative approach that FP paradigm has real benefits that help the developers increase their productivity when writing code. Then, a question raises spontaneously: why FP languages are so unpopular, accordingly to TIOBE Index (October 2015)[9], if they have all of these benefits?

We strongly believe that the most important factor that cuts off from the market these languages is the lack of a consistent and agile Graphic User Interface library support. Nowadays, the user interface and the user experience have become an essential requirement for every kind software. Furthermore, many object oriented implementations, in complex software systems, are more verbose than functional programming because of the code concerning the GUI building. It is not a coincidence that Erlang, for example, runs in the back-end of Facebook, WhatsApp and many other software systems.

For future work, when also FP languages will have a valuable GUI support, we want to consider complex systems (not only algorithms) and investigate how the new results will differ from the ones presented here. As already mentioned in the previous section, in order to have a global view on the topic we want to analyze other meaningful parameters (i.e. documentation, the comments, the development time, the application domain, etc..) and see if FP paradigm keeps on being better than OOP paradigm.

Finally, one other aspect that managers and companies need to take into consideration when they want to hire programmers, is their cost. At the moment, the wide adoption of OOP languages over FP makes functional programmers less common which means that they are also more expensive. Our study so far, tries to show the advantages of FP in order to make new developers adopt it, reducing the aforementioned gap.

8 Conclusion

In this investigation we wanted to see whether the functional programming paradigm would affect positively the productivity and the code reuse during the software development process.

Unfortunately, we were not able to check whether it is easier for functional programmers to upgrade software in new releases due to difficulties in finding an appropriate case study. In fact, many frameworks has been taken as case studies but in none of them we could find a parallel branch. Usually, these framework has been early developed in Java and thanks to the JVM they could reuse the already existing Java code to feed an API layer in other JVM-based functional languages.

We were able to correlate the productivity through the analysis of the LOC and the development time. After the evaluation we can state that FP has real advantages over OOP as it is usually less verbose and this implies time saving during development.

Even though we could not demonstrate benefits in CR, our findings suggest that the functional programming paradigm might really help developers in terms of productivity in order to improve the allocation of human resources.

References

[1] AlDanial. Count lines of code. `https://github.com/AlDanial/cloc`. Accessed: 2015-10-06.

[2] Apache. Apache flink. `https://github.com/apache/flink/`. Accessed: 2015-10-06.

[3] debian.org. The computer language benchmark of computer game. `http://benchmarksgame.alioth.debian.org/`. Accessed: 2015-10-06.

[4] R. Harrison and Smaraweera ; Dobie ; Lewis. Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal (Volume:11 , Issue: 4)*, pages 247 – 254, 1996.

[5] P. Hudak and M.P. Jones. Haskell vs. ada vs. c++ vs. awk vs. ... an experiment in software prototyping productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, Oct 1994.

[6] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.

[7] Play. Play framework. `https://github.com/playframework/playframework`. Accessed: 2015-10-06.

[8] ReactiveX. Reactivex. `http://reactivex.io/`. Accessed: 2015-10-06.

[9] Tiobe. Tiobe index updated october 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed: 2015-10-06.