Functional vs. Object Oriented Programming

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

October 10, 2015

Abstract

Human resources and time consumption are very important factors in software development. Both developers and managers try to maximise 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 saving of these resources.

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 has some fundamental benefits that increase programmer productivity over that when OOP languages. The purpose of this project is to demonstrate some of the benefits of functional programming over OOP, in order to persuade developers to switch to functional programming.

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).

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 analysed 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, is the asynchronous web server Play framework[6], that is 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 initial hypothesis and the results will be discussed.

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 find a confirmation in the TIOBE Index[8]. 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[5].

Furthermore, a thorough evaluation about the two programming paradigms has been performed by Harrison[3]. 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++[4]. 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 optimise profits and gain advantages from 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 releasing newer versions of their programs.

4 Methods

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

The approach used will be deductive because we started this research with our own hypothesis on the basis of existing theories, for instance, regarding the benefits of FP and the advantages that developers can gain in software production.

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

5 Results and Analysis

The experimentation consist in static code analysis on a set of eight algorithms[2], namely:

- FASTA
- k-nucleotide
- Mandelbrot
- n-body
- pi-digits
- regex-dna
- reverse-complement
- spectral-norm

The previous set of algorithms has been implemented in the following functional languages (both pure and not):

- Haskell
- Erlang
- Scala
- Clojure
- F#
- Python

The object-oriented languages these algorithm were implemented with are:

- C++
- C#
- Go
- Java

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

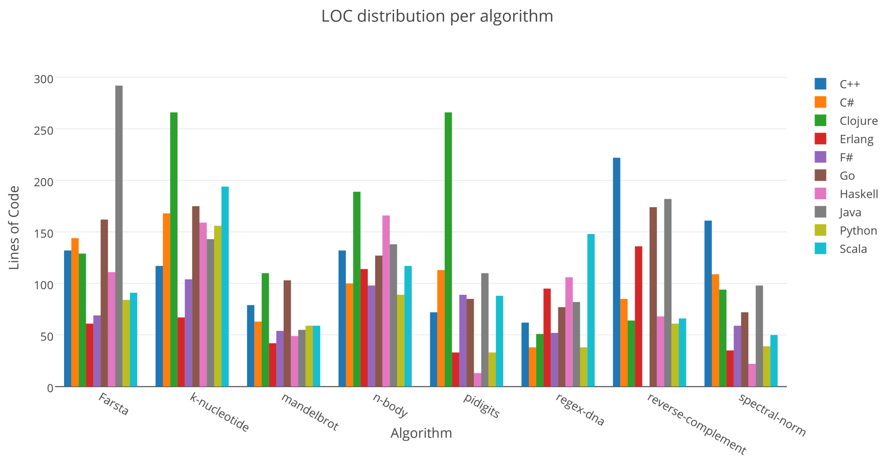


Figure 1: The graph shows the distribution of the LOC of different languages implementation for different algorithms.

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 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

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

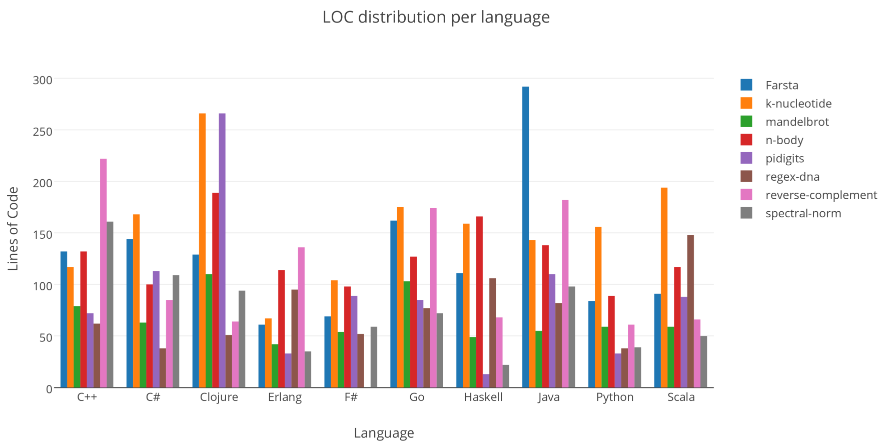


Figure 2: The graph shows the LOC used to solve each algorithm for the same language.

The above graphs have the disadvantages to present too much information so that the readers may have trouble understanding which paradigm is more efficient. For this reason we synthesised the gathered data in a graph that shows only the average LOC for each paradigm per each algorithm. This representation of the data is shown in Fig. 3.

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 [5]

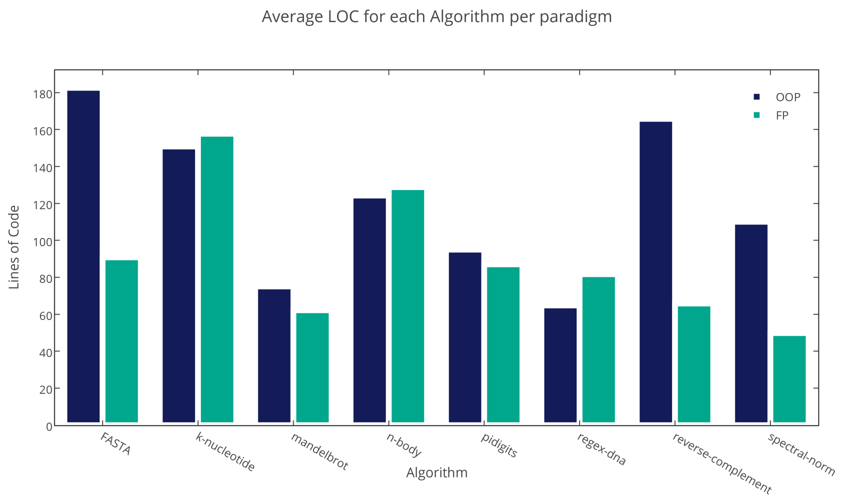


Figure 3: The graph represents the distribution of both FP and OOP paradigm in different algorithms.

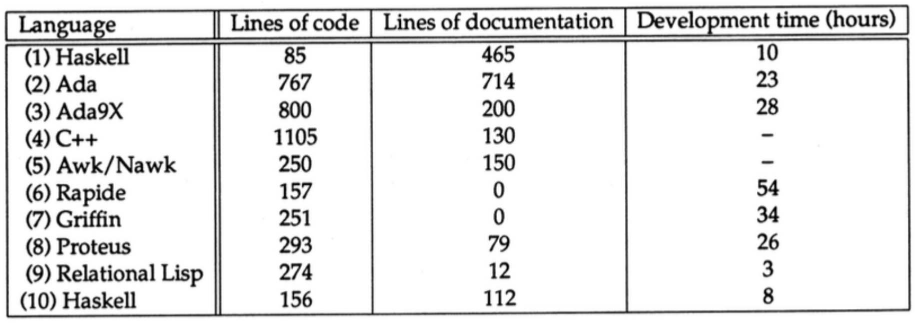
where the benefits of FP are discussed. Among the most significative there are: the syntax (which is simpler), higher order functions and the standard list-manipulation primitives.

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[3, 4].

For example, in Fig. 4 we notice that a very concise implementation may require a bigger documentation, like in the case of (1). Furthermore, we observe that in (1), (2), (3), (10) the development time is proportional to the LOC and this fact support our hypothesis.

It has not been possible to succeed in the experimentation concerning the CR as we could not find a suitable case study[6, 1, 7]. For instance, many implementations are based on the JVM so that the developers reuse their Java implementation building an API layer for functional languages, like Scala and Clojure.



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[4].

7 Conclusion

References

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

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

[3] 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.

[4] 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.

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

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

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

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