An Investigation of Methods for Comparing Programming Languages

David Colgan

December 14, 2010

1 Introduction

Most computer science majors and software developers have used Java and C. According to the TIOBE Index [1], these two languages have consistently been among the most popular for general use. They are both established, well understood, and use the object oriented or imperative paradigm.

Most of the other top languages are fairly similar to Java and C. Languages like C#, PHP, Python, Perl, and Objective-C all use some combination of the procedural and object-oriented paradigm. But are procedural and object oriented languages the best computer science has to offer for creating reliable, high performing software on a budget?

Many lesser-known languages influenced by the functional paradigm claim increased programmer productivity, fewer errors, shorter programs, and greatly enhanced support for multicore processing.

2 Research Goals

This project seeks to investigate languages that have the potential to be compelling alternatives to the common procedural and object oriented languages most often used today in commercial environments.

These five languages are ones I deem interesting and have wanted to learn in the past. They are also all somewhat to very mind-bending, and several require a completely new approach to programming when compared to Java or C. The languages are:

- Clojure, a Lisp dialect on the Java Virtual Machine
- Forth, a stack-based language
- Erlang, a concurrency-oriented language
- Haskell, a lazy, purely functional language
- J, an array language similar to APL

If I somehow get done with these languages and have more time, I will add additional languages such as OCaml, Scala, Lua, or Groovy.

3 Literature Survey of Previous Work

Two kinds of research in the literature compare programming languages: feature-by-feature comparisons and comparisons of small programs.

3.1 Feature Comparisons

Feuer and Gehani [7] take a conceptual approach when they compare C and Pascal. They begin with a history of the languages and discuss design decisions, followed by a step-by-step walk through each language's major features. They also evaluate C and Pascal for different problem domains. This paper is more of an informative overview than a hard empirical evaluation. The only actual code they show is a single function implementing binary search.

In a comparison of Ada 95 and Java, Brosgol [6] takes a similar approach, going feature by feature through the two languages. He provides sample code snippets throughout. He arrives at a table of features, highlighting differences in syntax, program organization features, memory management, and OO features like inheritance, polymorphism, and encapsulation.

Nami [11] presents a factual comparison of Eiffel, C++, Java, and Smalltalk. He gives a brief introduction to each language, describing design decisions and history. He then classifies each language based on static vsdynamic typing, compiled vsinterpreted build methods, built-in quality assurance facilities, automatic documentation generators, multiple vssingle inheritance, and concludes with a brief discussion on each language's efficacy for building infrastructures.

Tang [13] does a similar comparison of Ada and C++ using many of the same methods as the other studies.

Many of these studies are a high level overview of various languages. They compare features, but do not give in-depth examples.

3.2 Program Comparisons

Perhaps more useful and interesting are those comparisons done by inspecting the same program written in different languages. These give concrete examples of the differences.

The method I follow for this project is closely related to the method used by Prechelt [12]. He compares C, C++, Java, Perl, Python, Rexx, and Tcl by having various computer science masters students and volunteers from newsgroups write the same small program in one of the languages. He then compared the resulting programs based on program runtime, memory consumption, lines of code, program reliability (based on whether the program crashes or not), the amount of time it took each programmer to write the program, and program

structure. The program he had the participants write was a simple string processing program that consisted of converting telephone numbers into sentences based on a large dictionary and mapping scheme. Most of the programs he received were fairly small, taking a median of 3.1 hours to write, and averaging 200-300 line of code.

Some of the more interesting results from Prechelt's study include the observation that in lower-level languages, a lot of code is dedicated to writing the data structures, while the higher-level languages, the programmer usually takes advantage of the language's built-in capabilities. He also found that the scripting languages (Perl, Python, Rexx, and Tcl) tend to require about twice as much memory as C and C++, with Java taking 3–4 times as much, and that C and C++ are about twice as fast as Java and several times faster still than scripting languages.

Prechelt discusses the validity of his evaluations. He acknowledges the potential problems of asking for self-reported data from the Internet, as well as potential differences in programmer ability and working conditions. He suggests that because 80 programmers contributed code, this large sample size balances out many of these problems. Though the results should not be trusted for small differences, he asserts that large differences are likely to be accurate.

Henderson and Zorn [10] perform a similar study. They compare C++, a well known language, with four lesser-known languages: Oberon-2, Modula-3, Sather, and Self. They also write a short program in each of the languages, a simple database for university personnel information. These are all object-oriented languages, and as such, the comparison is weighted specifically towards OO features. They compare each language's methods of implementing and capabilities for inheritance, dynamic dispatch, code reuse, and information hiding. In addition to OO features, they also compare execution time, lines of code, and compile time. Henderson and Zorn explicitly state that one of the goals of their survey is to increase programmer awareness of lesser-known languages.

In a less formal study, Floyd [8] compares C++, Smalltalk, Eiffel, Sather, Objective-C, Parasol, Beta, Turbo Pascal, C+@, Liana, Ada, and, Drool. He collects an implementation of a linked-list structure from various people and then summarizes the results in a table that compares garbage collection schemes, inheritance (single or multiple), binding time, compilation (compiled vsinterpreted), exception handling features, and lines of code. He simply enumerates the implementations and does not do further analysis.

4 My Work

I combine the two approaches discussed in section 3. The deliverables for my project, like Prechelt, include implementations of the same program in multiple languages. I also include high level feature comparisons among the languages.

The primary way I compare the languages is by writing the same program in each language. I also chose a system that is more complicated than that of Prechelt, as I felt the program he (and many of the others) used to compare the

languages lacked substance.

One day at family game night we played a dice game called Farkle [2]. The game is simple but has a fair amount of decision making. Therefore for each language I implement a system that both plays this dice game through a command line interface, as well as evolves an optimal AI using a genetic algorithm system. Such a system involves many different aspects that explore each language's potential and features.

Because GAs are well suited for parallelization, I have made that portion of the program work on multiple cores if the language has facilities for it. The GA therefore tests the concurrency capabilities of the languages, as well as their symbolic manipulation power.

4.1 Methodology

I first implemented the Farkle and GA system in Python, the language I know best, to serve as a basis for comparison. I then rewrote the system in Clojure. Clojure is a language I have not had experience with, although I have used other LISP dialects. In my past experience with other LISPs, I did not fully understand or apply the functional style. Having completed the Farkle and GA system for Clojure, I now have two versions of the same system written in different languages and different paradigms, but that do almost exactly the same thing. This allows me to make some interesting comparisons. I will now go through each language and discuss my findings.

4.2 Python

Since I already knew Python, there was no major learning involved in its implementation. It came in at around 900 lines of code excluding tests, and there are libraries available for concurrency, but the programmer must add it manually.

As I implement the system in other languages, I have noticed instances where I could have made the Python implementation shorter. Therefore the 900 lines could be reduced; Python had the disadvantage of going first.

I consider the Python implementation to be the standard for comparison. I give it a "normal" difficulty in terms of learning, shortness of programs, and concurrency.

4.3 Clojure

4.4 Learning Clojure

To begin the process of learning Clojure, I read the book Programming Clojure by Stuart Halloway [9]. After going through this book, I was a bit underwhelmed by my level of understanding. I got the basics, but I still didn't feel ready to begin implementing the Farkle system. A few of the other resources I used included a very well-written Clojure tutorial by RMark Volkmann [14], the official Clojure website [3], the PeepCode screencast on Clojure [4], the very

helpful community question and answer site Stack Overflow [5], and other pages found through Google.

4.5 Comparison of Lines of Code

The Clojure code weighs in at approximately 700 lines. This is a fair difference from the Python implementation, though it is more lines than I expected.

4.6 Ease of Learning

For those coming from a mostly procedural and object-oriented background, Clojure will definitely be a stretch. The most striking difference is the new functional paradigm. Clojure is not purely functional, allowing side effects like printing to the screen anywhere in the code. However, once a variable has been given a value, its value cannot be changed. All procedural programming involves changing state, so those without a functional background will have to completely adjust their thinking. Idiomatic Clojure makes heavy use of recursion and higher order functions. It should be noted, though, that because Clojure discourages side effects, unit testing is much easier.

Another major difference is Clojure's laziness. It does not evaluate expressions unless it has to, causing great speedups in some cases, and allowing for infinite data structures. This is a powerful feature, but it takes some getting used to.

The syntax of Clojure is also very different than most other languages. Its syntax of parentheses and brackets allows for powerful macros, but it is definitely not C-like, so programmers who have never seen syntax like this will have to make an adjustment.

Therefore, Clojure has three new major concepts to learn: the functional paradigm, laziness, and the new style of syntax.

4.7 Support for Concurrency

Clojure has excellent support for parallel processing. In the simplest case, a program can be made to perform parallel computations by replacing a call to `map` with a call to `pmap`. The `pmap` function has a larger overhead than `map`, so it is not effective for simple calculations. However, if processing one element of the list using `pmap` is free of side effects and computationally intensive, Clojure can utilize multiple cores automatically. This is much simpler than using a manual thread-based approach that one would need in Python, Java, C.

Note that this only works if the calculations pmap does are free from side effects and if each part of the calculation are independent. If the program needs to share state between threads or cores, Clojure has an extensive system known as Software Transactional Memory. Clojure implements a system similar to a database transaction that allows for easily sharing state in a thread-safe manner. I did not have to use this more complicated system, though, because the major calculation I performed were in the GA system, and the process of evaluating

and crossing over individuals is self contained and does not depend on any other individuals. Therefore my program got free multicore support using `pmap`.

4.8 Disadvantages of Clojure

Clojure is plagued by a few problems. One of the aspects of the language that is both a blessing and a curse is its close integration with Java. While this is good in that it can use Java libraries, it also inherits many of Java's problems. Clojure is not a very good language for scripts or anything that requires fast startup. Starting any Clojure program cold takes on the order of 10 seconds. For this reason, the developers recommend starting one running Clojure repl and continuously send commands to it. This mostly works, but I found myself having to start new instances more than was comfortable through the process of programming.

The second major problem with Clojure is its youth. The language is only a few years old, and it is rapidly evolving and getting better. However, even some core features of the language have been implemented just this year, such as protocols.

A bigger problem is that because the language is so young, the tools are not very mature either. Slime, the flagship editing environment for all kinds of LISPs built on Emacs does not support Clojure nearly as well as the more mature LISPs. A case in point is that the `read-line` function, the primary way of getting command line input, simply doesn't work in Slime. This created a problem for my command line program. The Slime debugger is also not very helpful when working with Clojure. Stack traces show 100 levels of Java method calls and a single location in my code where the error occured.

There is a system developed to write Clojure in Vim called VimClojure, but I could not figure out how to install it completely. After spending multiple hours on it, I eventually went back to Emacs.

There are also Eclipse and Netbeans plugins that I did not investigate.

Table 1: Summary of Language Features

Aspect	Python	Clojure
Ease of Learning	Easy	Hard
Ease of Programming	Easy	Hard
Approximate Line of Code	900	700
Support for Concurrency	Manual Threading	Automatic with `pmap`, Software Transactional Memory
Purity	Impure	Pure by default
Supported Paradigms	Procedural, Object Oriented, Functional	Functional, Object Oriented
Evaluation Strategy	Eager	Lazy
Execution Method	Interpreted	Byte-code Compiled

5 Conclusions

I was expecting a larger difference in the number of lines of code between Python and Clojure. One of the thing that the functional programming advocates push is how much shorter functional programs are than non-functional ones. The Clojure program was shorter in some parts, but in other parts it was almost the same size. I do expect that the C and Java implementations will be much longer than the Python implementation. Python, although not overly functional, is a very expressive language. I will be interested to see how many lines the Haskell implementation has.

Purely functional programs are much easier to test than those with side effects. I am very confident in the Clojure code that I wrote, and this experience encourages me to write in a style that uses as few side effects as possible even in procedural and object-oriented languages.

With concurrency being as easy to tack on as it is in Clojure, this makes me reluctant to try to ever implement a manually concurrent program in a language like Python. Since a language like Clojure can automatically parallelize my programs, if I ever need multicore support, I will be much more inclined to switch languages for that project.

6 Plans for Future Work

Since this is the midpoint of a year-long project, I still have much more to do. At first I thought it would be a good idea to implement the system quickly in Python as a prototype to aid in writing the system in other languages. However, the Python implementation ended up taking half of the semester to finish, even though it was not one of the original seven languages. The Clojure implementation has also taken another half of a semester. Hopefully future languages will not take nearly so long to learn now that I have refined my technique.

I still want to make sure that I get through Haskell and Erlang. I also need write the C implementation. C is a lot different than Clojure, and I will have more to talk about concerning the differences between those languages as opposed to Clojure and Python. Hopefully I will get to Forth and J, although those may need to be dropped in the interest of time.

The next language in the queue is Haskell. When I start this language, I will try to read more than one complete book or tutorial if the first one that I read is not sufficient. I think that may have slowed me down learning Clojure: the first book did not teach me all that I needed, but instead of reading another book, I went with a more ad-hoc approach.

Another thing I still need to do is to evaluate the relative performance of the implementations. Tentatively, the Clojure code should be a good deal faster than the Python code, but I do not know that at this time.

References

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[2] `http://en.wikipedia.org/wiki/Farkle/`.

[3] `http://www.clojure.org`.

[4] `http://peepcode.com/products/functional-programming-with-clojure`.

[5] `http://www.stackoverflow.com`.

[6] Benjamin M. Brosgol. A comparison of the object-oriented features of Ada 95 and Java. In *TRI-Ada '97: Proceedings of the conference on TRI-Ada '97*, pages 213–229, New York, NY, USA, 1997. ACM.

[7] Alan R. Feuer and Narain H. Gehani. Comparison of the programming languages C and Pascal. *ACM Comput. Surv.*, 14(1):73–92, 1982.

[8] Michael Floyd. Comparing object-oriented languages. *Dr. Dobb's J.*, 18(11):104–ff., 1993.

[9] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[10] Robert Henderson and Benjamin Zorn. A comparison of object-oriented programming in four modern languages. *Softw. Pract. Exper.*, 24(11):1077–1095, 1994.

[11] Mohammad Reza Nami. A comparison of object-oriented languages in software engineering. *SIGSOFT Softw. Eng. Notes*, 33(4):1–5, 2008.

[12] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[13] L. S. Tang. A comparison of Ada and C++. In *TRI-Ada '92: Proceedings of the conference on TRI-Ada '92*, pages 338–349, New York, NY, USA, 1992. ACM.

[14] R. Mark Volkmann. Clojure - functional programming for the jvm. `http://java.ociweb.com/mark/clojure/article.html`, nov 2010.

A Farkle Rules

Farkle requires six dice. On each turn, the player rolls all six dice and removes combinations that are worth points. The following combinations are worth points:

- One 5 - 50 points
- One 1 - 100 points

- Three 1s - 300 points
- Three 2s - 200 points
- Three 3s - 300 points
- Three 4s - 400 points
- Three 5s - 500 points
- Three 6s - 600 points
- Four of a kind - 1000 points
- 1-6 Straight - 1500 points
- Three pairs - 1500 points
- Five of a kind - 2000 points
- Two triples - 2500 points
- Six of a kind - 3000 points

To score, the combination must be removed all on the same turn. As long as the player can remove at least one die that scores, they can then continue rolling. If the player cannot remove at least one die, they "Farkle" and lose all points for that turn. The strategy in the game comes from knowing when to stop rolling and which dice to set aside.