On the Superiority of Functional Programming

JOSHUA DATKO, Drexel University

1. INTRODUCTION

In 2011, C++ included powerful syntax into the language: the lambda expression. The shallow definition of a lambda expression is that it allows anonymous functions. However for a language to allow anonymous functions, it must be able to accept a function as a argument to other functions, which makes the receiving function a "higher order function." If a language can accept higher order functions, it is possible to write a function that accepts a number n and returns a function that adds n to its argument. Such a function is trivial in functional programming languages, whereas with imperative languages without support for lexical closures, this simply idea can't be as eloquently expressed.

Alonzo Church first introduced the idea of a lambda calculus in 1936. In 1958, John McCarthy's Lisp included support for lambda functions. Fifty-five years later, there is excitement that Java 8, the next Java version will include support for lambda expressions. Lambda expressions are but one idea from functional programming that today, popular languages are scrambling to adopt. The reason should be clear. Functional programming[1], the style of treating code modules as stateless functions and minimizing side-effects, is a more powerful and expressive paradigm.

2. LAZY EVALUATION

Lazy Evaluation is the technique of delaying computation until absolutely necessary. Some functional languages have deep support for this feature. For example, Scheme, has a concept of streams which are a powerful tool for creating delayed evaluation abstractions[Abelson et al. 1996]. In an interview, Simon Peyton Jones, a main designer for the Haskell programming language, recalls that lazy evaluation was a main factor in new functional language designs[Seibel 2009]. Specifically he mentions a program designed to calculate the number e to arbitrary precision. Simon says, "You just got given this list and you kept hauling on elements of the list [. . .] So that's not something that's very obvious to do if you're writing a C program [. . .] it's not a natural programming paradigm for C."

Other languages have adopted lazy evaluation as well, like iterators in Python. In designing Python, Guido van Rossum never considered Python to be heavily influenced by functional languages[van Rossum 2009]. But despite his dislike of the lambda keyword, the Python community clamored for functional features. For a non-functional language, Python shares many of its characteristics: first-class functions, lazy evaluation, lambda expressions, higher order functions, and an interactive interpreter. The adoption of these features by Python reveals the demand for the powerful and expressive syntax of functional programming.

3. BOTTOM-UP DESIGN

Functional programming encourages *bottom-up programming*, which is the practice of creating small modules, which are used by larger modules. To encourage this practice, functional languages support a

[1]In this paper I'm using the term *Functional Programming* as a style. This style incorporates several programming languages like Lisp, Scheme, Haskell, etc. . . I realize that Lisp is not a pure functional language, however it's influence on the style and popularity of functional programming can not be ignored.

Read-Eval-Print-Loop (REPL) and languages that mimic this functional behavior included an interactive interpreter. The REPL provides an interactive environment with immediate feedback to code. This paradigm is so fundamental different than the code-compile-test-deploy-repeat cycle that it can not be overstated. Paul Graham compares this paradigm to the revolution in painting in the fifteenth century. As painters discovered oil-based paints, they could now correct their mistakes on the fly whereas with *tempera*, mistakes were costly as the paint could not be blended or over-painted[Graham 1996]. The REPL is such a powerful concept that if a language does not have one by default, there is usually a shell replacement[Various 2013].

4. EXTENSIBLILITY FOR PARALLELISM

Since functional programming encourages stateless operations, immutable data structures are a natural result. Thus, operations may be easily split across independent computing units (cores or entire CPUs). The most famous paradigm that demonstrates this concept are the functions *Map* and *Reduce*. Map is function that takes a function as input and applies it to a list. Reduce is a function that accepts a function and a list as input and recursively applies the combination of the function to the elements of the list. These two higher-order functions provide the cornerstone of Google's *MapReduce* framework for processing large data sets.

5. ENCROACHMENT INTO JAVA

Perhaps the strongest example in support of the functional programming style is that several functional languages have been designed to use the Java Virtual Machine (JVM), specifically Scala and Clojure. Clojure, a Lisp dialect and Scala, a hybrid imperative-functional language include the functional advantages of easy parallelism, bottom-up design, and lazy evaluation. Additionally, they benefit from the vast existing libraries written for Java.

Scala takes a more inclusive approach than Clojure towards functional and object oriented languages. Dr. Martin Odersky, the creator of Scala, "wanted to show that the two paradigms can be unified and that something new and powerful could result from that combination.[Tate 2010]" Whereas Rich Hickey, the creator of Clojure, "wanted a predominantly functional, extensible, dynamic language, with a solid concurrency story, for the industry-standard platforms of the JVM [. . .] and didn't find one.[Tate 2010]" Both languages bring the power of functional programming to a wide-platform by leveraging the JVM.

6. CONCLUSION

The functional programming style is superior to the imperative model. Through lazy evaluation, computation is performed when needed allowing for processing of infinite streams without consuming infinite resources. A paradigm of bottom-up design allows for rapid-prototype and encourages modularity. With the need for system to scale, parallelism is critical and immutable data structures and higher order functions enable this behavior. The desire for functional programming has produced hybrid languages that run the JVM. For over fifty years, languages have been borrowing concepts from functional languages. Functional languages have been and will continue to be, the style of choice among all other programming languages.

REFERENCES

Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd editon ed.). MIT Press/McGraw-Hill, Cambridge.

Paul Graham. 1996. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA.

Peter Seibel. 2009. *Coders at Work: Reflections on the Craft of Programming*. Apress, New York.

Bruce A. Tate. 2010. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages* (1st ed.). Pragmatic Bookshelf, Dallas, TX, USA.

Guido van Rossum. 2009. Origins of Python's "Functional" Features. http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html. (April 2009).

Various. 2013. Wikipedia Entry on Read-Eval-Print-Loop. https://en.wikipedia.org/wiki/Read-eval-print_loop. (Oct. 2013).