A Brief Overview of Functional Programming Languages

Jagatheesan Kunasaikaran[1], Azlan Iqbal[2]
[1]ZALORA Malaysia, Jalan Dua, Chan Sow Lin, Kuala Lumpur, Malaysia
e-mail: jagatheesan@my.zalora.com
[2]College of Computer Science and Information Technology, Universiti Tenaga Nasional, Putrajaya Campus, Selangor, Malaysia
e-mail: azlan@uniten.edu.my

***Abstract* – Functional programming is an important programming paradigm. It is based on a branch of mathematics known as lambda calculus. In this article, we provide a brief overview, aimed at those new to the field, and explain the progress of functional programming since its inception. A selection of functional languages are provided as examples. We also suggest some improvements and speculate on the potential future directions of this paradigm.**

Keywords* – *functional, programming languages, LISP, Python; Javascript,Java, Elm, Haskell

I. Introduction

Programming languages can be classified into the style of programming each language supports. There are multiple styles of programming which are generally known as programming paradigms.

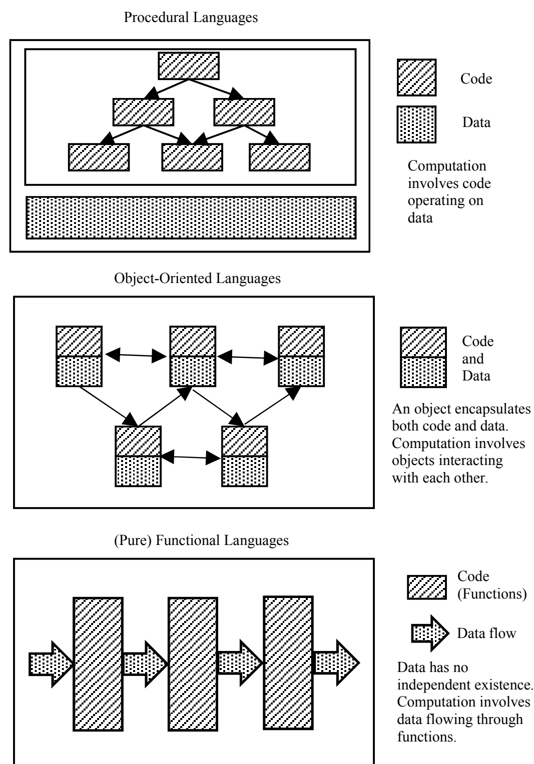

Figure 1: Differences between Procedural, Object-oriented and Functional Programming Paradigms

Common programming paradigms are procedural, object-oriented and functional. Figure 1 illustrates conceptually the differences between the common programming language paradigms [1]. Procedural and object-oriented paradigms mutate or alter the data along program execution. On the other hand, in pure functional style, the data does not exist by itself or independently. A composition of function calls with a set of arguments generates the final result that is expected. Each function is 'atomic' as it only executes operations defined in it to the input data and returns the result of the computation to the 'callee'.

II. Functional languages

Functional programming is based on mathematical logic. Lambda calculus forms the basis for modern functional programming languages. Lambda calculus was proposed by Alonzo Church in the 1930s and is based on function abstractions. Names and function applications are used to generalize expressions and these are evaluated by giving the names a value. There are a few properties of lambda calculus that make it suitable to be used as a descriptor for a programming language.

Firstly, in lambda calculus, only abstraction and application is needed to describe a programming language. Lambda calculus is independent of the order to evaluate the expression. This makes it a useful tool to investigate the effect of order independent evaluation in other programming languages. Besides, lambda calculus is based on strong proof techniques which make it applicable to the description of lambda calculus in other languages. It is also easy to implement because it is easy to understand [2].

There are a few key concepts in functional programming [3, 4]. Immutability being one of them states that data and the data structures managing the data do not change the data once the data structure is created. A mutable data structure can be changed along a program's execution cycle. This makes it harder to keep track of the state of a program. Immutability makes it easier to implement concurrent programming as the core data structure can be shared freely among the routines without having to consider the possibility of the data being shared changing during the execution of a routine [5].

Side effects is another key concept whereby in functional programming languages there are no side-effects to a function call. This means that there is no mutation of any global state during the function call. In some functional programming languages, the lack of side-effects is further strengthened by the fact that there are no variables or

assignments. Since no variables exist, there is no possibility of side effects.

The concept of 'purity' is also heavily explored in a functional programming language. A pure function only accepts a value and returns a value. Pure functions do not rely on any global states. As a direct consequence of functions being side-effect free and pure, a repeated call to a function with the same arguments returns the same value and this is known as referential transparency.

Referential transparency makes proofing and analyzing a program's execution to be easier and more understandable. Lazy evaluation is a concept where the execution of a piece of code is deferred until it is necessary to be computed. This avoids unnecessary computation and allows the creation of a theoretically infinite data structure. In a purely functional programming language, programs are also made from the composition of function applications. The result of a function application becomes the parameter of another function application.

The concept of composition where multiple functions are combined to create a new function is also widely used in functional programming languages. Functions can be combined in different orders to create functions that yield a different result. 'Currying' is also implemented in many functional programming languages. Currying is a technique of transforming functions that take multiple arguments into a sequence of functions that accepts one argument at a time [6]. This behavior allows for code implementations where functions are given fewer arguments than the total number of arguments that it accepts.

A. LISP

LISP, one of the earliest programming languages included functional programming elements in it. LISP was invented by the late John McCarthy in 1958. John McCarthy who was at the time at Massachusetts Institute of Technology published the design of the language in a paper entitled "*Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*" in the monthly journal of Association for Computer Machinery (ACM), Communications of the ACM. LISP has its fair share of criticism as a functional language. In his book, "Let Over Lambda – 50 Years of Lisp", Doug Hoyte argued that LISP is not functional [7].

The root of misconception according to him was the fact that programmers began to associate LISP's procedures with functions, disregarding the mathematical definition of a function which states that a function is a map from input values to output values. Being one of the earliest programming languages to implement procedures which looked like functions, LISP was taken to be a programming language that is functional. A LISP procedure can return different values for each call to the procedure with the same value for its arguments. This breaks away from the characteristic of functional languages that is to be free of side effects.

B. ML

The ML programming language was created in 1973. ML stands for MetaLanguage. A few other functional languages were also created around this time. Rob Burstall and John Darlington developed the 'New Programming Language' (NPL) [8, 9]. ML further developed into a few dialects namely Standard ML and Categorical abstract machine language (Caml).

C. FP

In 1977, FP was presented by John Backus in his lecture "*Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*". In his lecture, Backus compared functional programs with programs based on the von Neumann style. The von Neumann style program dominated the development of programs during the time. In programming languages based on Von Neumann's architecture, the state of the program is changed throughout its execution. This change in state is known as mutation. Backus argued that this concept created complex systems. Complex systems are bigger in size which results in higher maintenance costs so he put forwarded FP as an alternative [10].

D. Miranda

David Turner created the St. Andrews Static Language (SASL) programming language at University of St. Andrews in 1972. It formed the basis for the programming language, Miranda, created also by him in 1985. Miranda runs on the UNIX operating system and its goal was to be a commercial supported standard non-strict purely functional language [11]. Its source code is not available freely as it was commercially licensed by Research Software Limited of England.

E. Haskell

Haskell was formed by the mutual consensus of attendees of the 1987 Functional Programming Languages and Computer Architecture (FPCA) meeting. In this conference, it was widely accepted that a common standard for the development of a general purpose functional programming language was needed. At the time, Miranda created by David Turner was the closest to a full-fledged functional programming language. However, Turner did not want another dialect of the language. Hence, the committee formed at the conference started designing the language with the adoption of language features of Miranda that they felt to be a good fit inside Haskell [12].

Haskell integrated various functional programming concepts in it. Most notably, Haskell integrated the concept of type classes and monads. Type classes were first introduced inside Haskell. It is one of the most valuable contributions of Haskell to the general programming language design. Type classes allow for the definition of generic functions that operate on different data types. This is known as function composition. Monads can be visualized as descriptions of computations that are composable. Monads are usually used in Haskell to encapsulate I/O operations which are considered impure.

III. Functional concepts in non-functional languages

The contribution from Haskell paved the way to modern non-functional languages to adopt the functional programming paradigm in their language core. Functional features in non-functional languages usually do not strictly impose that functions should be pure. They do allow side-effects inside functions while giving and interface for functional style programming. Python, Javascript, and Lua supported first class functions since their inception.

A. Python

Functions such as “lambda”, “map”, “reduce” and “filter” was introduced in Python in 1994 through the work of Amrit Prem [13]. The lack of closures in Python initially caused a problem when it came to referencing values of variables outside of the lambda expression. Users that came from pure functional programming languages perceived this as a shortcoming in Python’s implementation as they believed a lambda expression should behave as in pure functional programming languages. Closures were introduced inside Python in version 2.2 to address this problem.

Closures allow inner functions a reference of values to variables surrounding it. Closures form the basis for the implementation of syntaxes to support the decorator software design pattern in Python [14]. Software design patterns are reusable solutions that can be applied to solve common programming problems. Decorator patterns are essentially wrappers around functions that extend the behavior of the function without changing the code inside the functions. Python supports decorator patterns by using the ‘@’ symbol as the syntax [15].

B. Lua

Lua is another non-functional programming language that has implemented functional concepts in it. It was created in 1993 and functional programming concepts were implemented in Lua 3.1 in 1998 [16]. Lua supports first-class functions and higher order functions. First class functions mean functions can be treated as any other data type. Functions can be passed as an argument to another function and be the return value also. Higher order functions are functions that can accept functions as arguments and return functions as the return value. The concept of higher order functions enables composability in the language.

C. JavaScript

JavaScript is a web scripting language that is also widely used in a non-browser environment which offers multiple programming paradigm styles. It follows the ECMAScript (ES) standardization. The language itself has first-class functions and higher order functions. The built-in array data type implements many functions that are functional. For example, ‘map’, ‘filter’ and ‘reduce’ are functions implemented for the array data type which does not mutate the internal structure of the array. They return a new array on each call.

Progress has been done in extending the language through libraries. JavaScript has many libraries built to support functional-style programming. For example, underscore.js, lodash, and ramda are libraries written in JavaScript that support the functional programming style. The former two extends JavaScript’s existing features to provide a functional flavored library to ease development.

Lodash has a separate module to support the functional programming style. This module makes the arguments sent to a function immutable. The original value of the argument is not altered by calling the function. Functions are automatically curried by using this module. The order of arguments to a method is also changed to be ‘iteratee’ first and data last [17]. Lodash gives the ability by using a separate module to modify the behavior of existing functions in it.

Ramda is focused on being close to the functional programming style as the core library itself is built to support this style by avoiding side-effects and mutability of data [18]. Ramda introduces two major features. Functions in Ramda are automatically curried. This means new functions can be created by not supplying the final parameters. Currying enables functions to be composed into well-connected logic.

D. Java

Java is a general-purpose, concurrent, strongly-typed and class-based object-oriented language [19]. Java 8 was released in 2014 [20]. This version of Java brought with it an upgrade to the existing Java programming model. Functional interfaces were introduced in Java 8. Functional interfaces are interfaces that only have one method that needs to be implemented. Lambda expressions which are also known as closures or anonymous methods were introduced in Java 8. Lambda expressions implement the functional interface and can be used to substitute code blocks that expect anonymous inner class.

Functional interfaces can also be used for higher order programming which includes function composition and currying [21]. However, a lambda expression in Java does not have the full characteristics of a lambda expression implemented in a purely functional language. It is inspired by the functional programming style while maintaining Java’s behavior of nominative typing intact [22]. ‘Stream’ is another API introduced in Java 8. A stream is essentially a consumable data structure that is passed from one operation to another [23]. To visualize, a stream can be seen as a pipeline where the data structure is passed from one function to another.

This is akin to a functional programming construct where the output from a function becomes an input to another function. A stream in Java has two types of operations; it can be either an intermediate or terminal operation. An intermediate operation produces a new stream that is used by the next operation in the pipeline. A terminal operation is the last operation in the pipeline which may produce an output. Java’s Collection API has also been upgraded with new methods such as map, filter, sorted, match, count and reduce to support stream operation [24].

E. Elm

Elm is a functional language that compiles to JavaScript. Elm was designed by Evan Czaplicki as his senior thesis work. The first release of Elm was done in 2012 [25]. Elm brings functional language constructs into web programming. This provides multiple benefits. One is that the risk of runtime errors is greatly reduced as Elm uses type inference during compilation whereby the data type of an expression is determined during compilation time itself. Elm has data structures such as maybe, result, and 'task' to handle errors. Task is an application of functional programming inspired error handling to web application problems. It can handle gracefully the problem of external services such as the HTTP API not responding.

IV. Possible weaknesses

Functional languages have their own disadvantages [26]. The flow of input and output is harder in a purely functional language because of the purity of functions enforced by functional languages. Interactive applications are harder to develop as most interactive applications rely on using the request and response method. In a purely functional language like Haskell, a computation that performs I/O is considered impure and needs constructs such as monads to isolate these computations which may incur side-effects.

Programs that need to be run over a long time may be challenging in a purely functional language as this usually needs an unending or sentinel-controlled loop. Most applications nowadays revolve around data which is retrieved from a data source such as database or REST API. This data is more easily represented in an object-oriented language as objects than in functional languages. The learning curve of functional programming languages is also generally higher compared to imperative programming languages.

V. Possible improvements

Functional programming is gaining more visibility among developers over time. Improvements are needed to integrate functional programming concepts into existing languages and frameworks. Better tooling is also needed to grow the Haskell community. Tooling in the form of an integrated development environment (IDE), plugins and command-line tools need to be developed further to support the growing ecosystem. In many companies the codebase is created using imperative programming languages such as Java. Changing these codebases to use a functional programming language such as Haskell is difficult.

However, there are concepts in functional programming languages that can be ported to these imperative languages to allow developers to write more readable code that produces fewer bugs. Functional programming languages also need to be ported into more environments such as embedded devices as these critical devices may benefit from the features of functional languages which encourage the writing of code that have fewer errors. Emphasis should be given to teaching functional programming concepts and languages in educational institutions. A firm grasp of these concepts will add to a student's knowledge of the available methods to solve programming problems.

VI. Future directions

In the years to come, functional programming languages will play a more important role in software development. More work will be done to integrate functional programming concepts into existing programming languages. This will be mostly driven in an open source manner. The work will be focused on either developing a module inside the existing programming language or complementing the language via libraries as seen in Ramda.

Advances in bringing functional programming to various platforms will also happen as seen in the case of Elm which introduces functional programming in a web development environment. Existing functional programming languages such as Haskell which is mostly used in the academic domain is expected to make further headways in industrial applications.

Acknowledgement

I would like to thank my co-author, Dr. Mohammed Azlan Bin Mohamed Iqbal for his guidance that has been of immense help in the writing of this paper.

References

[1] R. C. Bjork, "LISP." [Online]. Available: http://www.math-cs.gordon.edu/courses/cs323/LISP/lisp.html. [Accessed: 04-Sep-2016].

[2] G. Michaelson, *An introduction to functional programming through lambda calculus*. Courier Corporation, 2011.

[3] Aleksandar, "Functional programming - HaskellWiki." [Online]. Available: https://wiki.haskell.org/index.php?title=Functional_programming&oldid=59163. [Accessed: 04-Sep-2016].

[4] Sebastián Peyrott, "Introduction to Immutable.js and Functional Programming Concepts," 2016. [Online]. Available: https://auth0.com/blog/intro-to-immutable-js/. [Accessed: 04-Sep-2016].

[5] "Clojure - Concurrent Programming." [Online]. Available: http://clojure.org/about/concurrent_programming. [Accessed: 03-Sep-2016].

[6] "Currying - HaskellWiki," 2016. [Online]. Available: https://wiki.haskell.org/index.php?title=Currying&oldid=60510. [Accessed: 19-Sep-2016].

[7] D. Hoyte, *Let Over Lambda : 50 years of lisp*. Lulu.com, 2008.

[8] R. M. Burstall, "Design considerations for a functional programming language," *Softw. Revolut.*, pp. 45–57, 1977.

[9] H. W. Loidl and R. Peña, *Trends in Functional Programming: 13th International Symposium, TFP 2012, St Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Springer Berlin Heidelberg, 2013.

[10] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[11] "The Miranda Programming Language." [Online]. Available: http://groups.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html. [Accessed: 04-Sep-2016].

[12] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy With Class," *Proc. third ACM SIGPLAN Conf. Hist. Program. Lang.*, pp. 12–55, 2007.

[13] van R. Guido, “The History of Python: Origins of Python’s,” 2009. [Online]. Available: http://python-history.blogspot.my/2009/04/origins-of-pythons-functional-features.html. [Accessed: 10-Aug-2016].

[14] Ayman Farhat, “A guide to Python’s function decorators | The Code Ship,” 2014. [Online]. Available: http://thecodeship.com/patterns/guide-to-python-function-decorators/. [Accessed: 19-Sep-2016].

[15] “PythonDecorators.” [Online]. Available: https://wiki.python.org/moin/PythonDecorators. [Accessed: 22-Sep-2016].

[16] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “The Evolution of Lua,” in *Proceedings of the 3rd ACM SIGPLAN conference on History of programming languages*, 2007, pp. 2–1--2–26.

[17] “FP Guide.” [Online]. Available: https://github.com/lodash/lodash/wiki/FP-Guide. [Accessed: 03-Sep-2016].

[18] “Ramda Documentation.” [Online]. Available: http://ramdajs.com/0.22.1/index.html. [Accessed: 16-Aug-2016].

[19] “Chapter 1. Introduction.” [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-1.html. [Accessed: 04-Aug-2017].

[20] “Java 7 and Java 8 Releases by Date.” [Online]. Available: https://www.java.com/en/download/faq/release_dates.xml. [Accessed: 04-Sep-2016].

[21] Edwin Dalorzo, “Functional Programming with Java 8 Functions - DZone Java,” 2014. [Online]. Available: https://dzone.com/articles/functional-programming-java-8. [Accessed: 04-Sep-2016].

[22] Ben Evans, “How Functional is Java 8?,” 2014. [Online]. Available: https://www.infoq.com/articles/How-Functional-is-Java-8. [Accessed: 04-Sep-2016].

[23] Lucas Jellema, “Java 8 - Collection enhancements leveraging Lambda Expressions - or: How Java emulates SQL - AMIS Oracle and Java Blog,” 2013. [Online]. Available: https://technology.amis.nl/2013/10/05/java-8-collection-enhancements-leveraging-lambda-expressions-or-how-java-emulates-sql/. [Accessed: 04-Sep-2016].

[24] Benjamin Winterberg, “Java 8 Tutorial - Benjamin Winterberg,” 2014. [Online]. Available: http://winterbe.com/posts/2014/03/16/java-8-tutorial/. [Accessed: 04-Sep-2016].

[25] “Release Notes.” [Online]. Available: http://elm-lang.org/blog. [Accessed: 04-Sep-2016].

[26] J. Hunt, *A Beginner’s Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing, 2014.