Alexi Green

Dr. John Ramirez

CS1501

27 July 2020

W-Paper 5: W-Paper 2 Revision

Functional programming is a programming paradigm in which programs are constructed with applied mathematical functions. It is a declarative classification of programming languages, where the main focus of the style centers around solving problems through a system of expressions that each produce a value, as opposed more commonly used imperative style that simply uses statements to assign variables ("Functional Programming Paradigm"). This essay aims to examine the characteristics and history of functional programming and to briefly discuss some programming languages that use a functional approach for their implementations.

Several computational and mathematical concepts are unique to the functional paradigm. For example, higher-order functions are functions utilized in functional programming that can take another function as a parameter, produce a function as a return value, or both. As parameters, higher-order functions are comparable to "first-class citizens" because of their ability to support all of the operations available to other common entities within a regular programming language, such as being able to pass arguments, return values, and bind to identifiers. This characteristic allows programmers to write functional programs in a modularized, declarative style ("Functional Programmers: Why We Call Them First-Class Citizens?").

Higher-order functions can be implemented via currying, a technique of programming that transforms a function comprised of several arguments into several functions of a single argument. For example, a function in JavaScript such as f(a, b, c) can be transformed into a

Dr. John Ramirez

CS1501

27 July 2020

curried function such that it is now callable as f(a)(b)(c). This new function will return a function with a single argument, which returns another function of a single argument, and so on, until a it finally returns a value. A fully curried function will return either an ordinary result or another fully curried function ("What is Currying?"). The benefit in currying comes from the ability to partially configure functions. Multiple curried functions can act as building blocks for a functionality in the sense that more abstract functions become reusable for a variety of values and types. Additionally, currying allows for more expression through functions and can make a programming using functional programming more readable ("Why Curry Helps"). As a result, the combination of these advantages creates an ideal environment for implementing successor functions and recursive algorithms ("Higher Order Functions and Currying").

Pure functions are also unique to the functional programming paradigm. A pure function is a mathematical functions with two fundamental rules, the first being that it must return the same result every time it is called when passing the same set of arguments. The variables within a pure function are immutable and thus do not interfere with any other values beyond the local state, making the process of seeing dependencies of the function easier since the function can only access what is passed into it. The second rule for pure functions implores that a pure function does not have any impact on memory complexity or I/O, which can be valuable for optimization. These rules contribute to the idea of referential transparency within a functional program. Programs with referential transparency do not change the values of variables once they are defined ("3. Pure Functions, Laziness I/O, and Monads").

Dr. John Ramirez

CS1501

27 July 2020

Pure functions and referential transparency can be useful for memoization, a powerful technique for optimization that passes the results that an expensive function previously computed into a pure function to be retrieved later on, once the expensive function is called again ("Lecture 22: Memoization"). As a result, memoization lowers a function's time cost. Where one recursive function, such as the naïve implementation of the Fibonacci sequence, might run in exponential time, the same function with memoization implementation might run in linear time. However, a space-time tradeoff occurs during this process; in exchange for a faster algorithm, memoization will use a higher amount of memory, so the performance improvement will depend on the size of the problem ("On the Advantages of Memoization").

Recursion is another essential concept for functional programming, though it is not exclusive to the paradigm. Functional programming emphasizes the use of recursive function in order to accomplish iterative loops. Rather than recursive calls occurring before computation and freezing the state of the current computation until they finish evaluation the stack – which the functional paradigm refers to as "head recursion," and is common in languages that follow an imperative style – the recursive calls in the functioning programming occur only after computation. This form of recursion, called "tail recursion," can overstep the concern that head recursion creates when dealing with a very large set of numbers, therefore improving overhead in terms of time complexity ("Functional Programming: Recursion").

Functional programming languages are categorized by the type of evaluation strategy they use. An evaluation strategy details the way in which a program makes decisions on when to

Dr. John Ramirez

CS1501

27 July 2020

evaluate a function's arguments and what type of value should be passed into the function. The main type of evaluation strategy used by most functional programming languages is called "non-strict evaluation," more commonly referred to as "lazy evaluation." Lazy evaluation holds the evaluation of an expression until its value is explicitly called upon. This strategy comes to fruition within programming through tail recursion, which, as aforementioned, holds off from evaluating the stack until computation finishes and the result from the recursion is needed.

Some of the advantages of lazy evaluation include a reduction in the time complexity of an algorithm by discarding the sub-expressions and temporary computations and conditionals that can be expressed via some other implementation. Contrarily, an increase in space complexity acts as the most prominent disadvantage to lazy evaluation, since it can delay the creation of objects during the holding process ("Functional Programming – Lazy Evaluation"). The other kind of evaluation strategy, "strict evaluation," always evaluated argument functions before computation. Head recursion utilizes strict evaluation; although it suffers in time complexity for large values, it suffers much less than lazy evaluation in terms of its memory overhead.

Historically, functional programming languages evolved from Alonzo Church's work on the lambda calculus. The lambda calculus, being a higher-order functional, uses abstractions to define single-argument functions and perform computations using the reduction rule. Given Church's thesis, "effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus," the lambda calculus can computer anything that a

programming language can computer. Thus, it is widely acknowledged as one of the first functional languages (Gang).

Because of the lambda calculus' type-free nature, Church captured the idea that functions could be self-referential, which is now a foundational concept of the functional programming paradigm. Self-application allows modern functional programs to have the effect of recursion without explicitly defining a recursive definition (Hudak 363). The lambda calculus stands out from other mathematical theories by maintaining its self-referential principle without the occurrence of contradiction or paradox – unlike sets, which cannot contain the sets of themselves due to paradoxical restrictions – making it a solid mathematical system.

The first functional programming language took inspiration from Church's lambda calculus and was developed in the late 1950's for the IBM-700 series by John McCarthy at the Massachusetts Institute of Technology (MIT). LISP made strides beyond the then-current FORTRAN-Compiled List-Processing Language (FLPL), a compiled computer language created around the same time for the manipulation of symbolic expressions for simulation of the geometry theorem-proving machine on the IBM-704. While LISP used Church's lambda notation, it also used conditional expressions to control the base cases for recursive functions instead of higher-order functionals, which allowed for combining cases into a single formula (McCarthy 6).

Dr. John Ramirez

CS1501

27 July 2020

Since the development of LISP, many functional programming languages have emerged and become popular among many software developers, such as Python, JavaScript, Erlang, Haskell, and Clojure. Languages that utilize functional programming are categorized into two groups. The first group, "pure functional languages," includes languages that only support the aspects of the functional paradigm. For example, Haskell's features include being statically typed, being purely functional, type inferencing, concurrent in terms of primitives and abstractions, and using the lazy evaluation strategy (Haskell Language). Languages, like Python, fall into the other category, called "impure functional languages." These are programming languages that support the functional paradigm, but tend to maintain an imperative style of programming, using conditionals to control the program's flow ("Functional Programming – Introduction"). LISP, despite being deemed the first functional programming language, is considered to be an impure functional language because of its utilization of conditionals for controlling the base cases for its recursive functions.

In conclusion, functional programming is a programming paradigm where programs are constructed using mathematically pure functions in a declarative style. It possesses several unique mathematical concepts that differentiate itself from an imperative programming style, thanks to its origins in lambda calculus. These features give functional programming languages an edge in terms of the run-time performance of recursive algorithms when memory usage is less of a concern, hence the arising popularity in pure and impure functional programming languages.

Dr. John Ramirez

CS1501

27 July 2020

Works Cited

"Functional Programming - Lazy Evaluation." *Tutorialspoint*, www.tutorialspoint.com/functional_programming/functional_programming_lazy_evaluation.htm. Accessed 09 June 2020.

"Functional Programming - Introduction." *Tutorialspoint*, www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm. Accessed 10 June 2020.

Haskell Language, www.haskell.org/. Accessed 10 June 2020.

Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages ." *ACM Computing Surveys*, Standford, Sept. 1989, ccrma.stanford.edu/~jos/pdf/FunctionalProgramming-p359-hudak.pdf. Accessed 10 June 2020.

Jackson, Hugh. "Why Curry Helps." *Hugh FD Jackson*, hughfdjackson.com/javascript/why-curry-helps/. Accessed 27 July 2020.

"Lecture 22: Memoization." Cornell University, www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec22-memoization/lec22.html. Accessed 09 June 2020.

Dr. John Ramirez

CS1501

27 July 2020

McCarthy, John. *History of Lisp*. Artificial Intelligence Laboratory Standford University, 12 Feb. 1979, jmc.stanford.edu/articles/lisp/lisp.pdf. Access 10 June 2020.

Milewski, Bartosz. "3. Pure Functions, Laziness, I/O, and Monads." *School of Haskell*, 14 Dec. 2014, www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io. Accessed 08 June 2020.

Namc. "What Is Currying?" *DEV Community*, 25 Feb. 2018, dev.to/__namc/what-is-currying--3l2a. Accessed 27 July 2020.

Panadero, Christian. "Functional Programming: Recursion." *Dzone.com*, DZone, 31 Jan. 2018, dzone.com/articles/functional-programming-recursion. Accessed 09 June 2020.

prajawals. "Higher Order Functions and Currying." GeeksforGeeks, 27 Apr. 2018, www.geeksforgeeks.org/higher-order-functions-currying/. Accessed 08 June 2020.

Pranav. "Functional Programmers: Why We Call Them First-Class Citizens?" Busigence, 16 Oct. 2016, busigence.com/blog/functional-programmers-why-we-call-them-first-class-citizens. Accessed 08 June 2020.

Rosas, Olga. "On the Advantages of Memoization." *Medium*, 1 July 2020, medium.com/@olga.rosas/on-the-advantages-of-memoization-85806996ac67. Accessed 27 July 2020.

Dr. John Ramirez

CS1501

27 July 2020

Tan, Gang. *A Brief History of Functional Programming*, Penn State, 11 Mar. 2004, www.cse.psu.edu/~gxt29/historyOfFP/historyOfFP.html. Accessed 09 June 2020.

Vishalxviii, and KumariPoojaMandal. "Functional Programming Paradigm." *GeeksforGeeks*, 2 Jan. 2019, www.geeksforgeeks.org/functional-programming-paradigm/. Accessed 08 June 2020.