Alexi Green

Dr. John Ramirez

CS 1501

10 June 2020

W-Paper 2

Functional programming is a programming paradigm where programs are constructed with applied mathematical functions. It is a declarative classification of programming languages, in which its main focus centers around solving problems through a system of expressions that each produced a value, whereas an imperative style uses statements the simply assign variables ("Functional Programming Paradigm"). This paper will examine the characteristics and history of functional programming, and in addition briefly discuss some programming languages that follow a functional approach.

Several computational and mathematical concepts are unique to the functional paradigm. One such concept, higher-order functions, is where functions can take a function as a parameter, produce a function as a return value, or both. As arguments, higher-order functions are often described as "first-class citizens" because they can support all of the operations available to other common entities within a programming language, such as passing arguments, returning values, and being bound to identifiers. This characteristic allows for functional programs to be written in a modularized, declarative style ("Functional Programmers: Why We Call Them First-Class Citizens?"). Higher-order functions also enable currying, a methodology for translating a function evaluation by taking multiple arguments into evaluating a sequence of functions with their own, single argument. This creates an ideal environment for implementing successor functions and recursive algorithms ("Higher Order Functions and Currying").

Dr. John Ramirez

CS 1501

10 June 2020

The concept of pure functions is also unique to the functional paradigm. A pure function is a mathematical function with two fundamental properties, the first being that is must return the same result for every time is it called when using the same set of arguments. Its variable are immutable and they do not interfere with any values beyond the local state, making it easy to see the dependencies of the function since it can only access what is passed to it. The second property states that a pure function has no side effects on memory complexity or I/O, which is useful for optimization ("3. Pure Functions, Laziness, I/O, and Monads").

The properties of pure functions contribute to the idea of referential transparency within a functional program. For referential transparency, variables do not change their values throughout the program once they are defined. A practical use for such characteristics might be memoization, a powerful optimization technique that passes the previously computed results of an expensive function into a pure function to be retrieved later once the expensive function is called again. This utilization of pure functions saves on memory usage when implementing a complex algorithm ("Lecture 22: Memoization").

Recursion is another concept that is essential to functional programming, but it is not exclusive to the paradigm. Functional programming places emphasis on utilizing recursion in order to accomplish iterative loops. However, instead of having recursive calls occur before computation and holding the state of the current computation until it finishes evaluating the stack –which the functional paradigm refers to as "head recursion–" functional programming

implements recursion such that recursive calls occur only after computation. This version of recursion, called "tail recursion," can overstep the concern that head recursion demonstrates when dealing with very large sets of numbers, improving over head recursion in terms of time complexity ("Functional Programming: Recursion").

Functional programming languages can be categorized by the type of evaluation strategy they use, involving the way in which they make decisions on when to evaluate the arguments of a function and what type of value should be passed to a function. The main kind of evaluation strategy that is used by most functional programming languages is non-strict evaluation, more commonly called "lazy evaluation." Lazy evaluation holds the evaluation of an expression until its value is called upon. This strategy comes to fruition within tail recursion, which, as previously discussed, holds off from evaluating the stack until computation finishes and the result from the recursion is needed.

Some advantages to lazy evaluation include a reduction in the time complexity of an algorithm by discarding sub-expressions and temporary computations and conditionals that can all be expressed through some other implementation. Contrarily, an increase in space complexity acts as the biggest disadvantage to lazy evaluation, since it can create delayed object when holding the evaluation until it is required in the final return value ("Functional Programming - Lazy Evaluation"). The other type of evaluation strategy, strict evaluation, always evaluates the arguments to a function before computation. Head recursion uses this

Dr. John Ramirez

CS 1501

10 June 2020

strategy, and although it suffers in time complexity for large values, it suffers less than lazy evaluation in terms of memory overhead.

Historically, functional programming languages evolved from the work of Alonzo Church on lambda calculus. Lambda calculus can use abstractions to define single-argument functions and perform computations using a reduction rule, as it is a higher-order functional. Given Church's thesis that, "Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus," lambda calculus can compute anything a programming language can compute, and thus it is widely acknowledged as one of the first functional languages (Gang).

Because of lambda calculus' type-free nature, Church was able to capture the idea that functions could be self-referential, a foundational concept of the functional paradigm. The ability of self-application is what allows modern functional programs to have the effect of recursion without explicitly stating a recursive definition (Hudak 363). Lambda calculus defines itself from other mathematical theories by being self-referential without contradiction or paradox, unlike sets which cannot contain the sets of themselves due to paradoxical restrictions, and thus is 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 stride beyond the current FORTRAN-

Dr. John Ramirez

CS 1501

10 June 2020

Compiled List-Processing Language (FLPL), a compiled computer language created around the same time for the manipulation of symbolic expressions for simulation of geometry theorem-proving machine on the IBM 704. While LISP used the lambda notation of Church, it used conditional expressions to control the base cases for recursive functions instead higher-order functionals, which allowed for combining cases into a single formula (McCarthy 6).

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

Dr. John Ramirez

CS 1501

10 June 2020

several unique mathematical concepts that differentiate it from an imperative programming style thanks to its origins in lambda calculus. These features give functional programming languages an edge when it comes to the performance of recursive algorithms in terms of time complexity when memory usage is less of a concern, hence the arising popularity in programming languages that both partially and fully implement the functional paradigm.

Dr. John Ramirez

CS 1501

10 June 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.

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

Dr. John Ramirez

CS 1501

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.

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.

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.